AD _____

ARO 14752.1-A-EL

FINAL TECHNICAL REPORT
GIT–ICS–79/09
For Period Covering 25 November 1976 – 30 June 1979

PORTABILITY OF LARGE COBOL PROGRAMS: THE COBOL PROGRAMMERS WORKBENCH

By
Philip H. Enslow, Jr.

Prepared for
U. S. ARMY RESEARCH OFFICE
P. O. BOX 12211
RESEARCH TRIANGLE PARK, N. C. 27709
Under
Grant No. DAAG29–77–G–0045

ARO Project No. P–14752–A–EL
GIT Project No. G36–618

GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMPUTER SCIENCE
ATLANTA, GEORGIA 30332

1979

12

PORTABILITY OF LARGE COBOL PROGRAMS ---
THE COBOL PROGRAMMER'S WORKBENCH

FINAL TECHNICAL REPORT

GIT-ICS-79/09

25 November 1976 - 30 June 1979

Philip H. Enslow, Jr.

September, 1979

U. S. ARMY RESEARCH OFFICE

Grant Number DAAG29-77-G-0045
ARO Project Number P-14752-A-EL
GIT Project Number G36-618

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

THE VIEW, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1 REPORT NUMBER	2 GOVT ACCESSION NO. / 3 RECIPIENT'S CATALOG NUMBER
4 TITLE (and Subtitle): Portability of Large COBOL Programs: The COBOL Programmer's Workbench,	5 TYPE OF REPORT & PERIOD COVERED: Final Technical Report, 25 Nov 76-30 June 79 / 6 PERFORMING ORG. REPORT NUMBER: GIT-ICS-79/09
7 AUTHOR(s): Philip H. Enslow, Jr.	8 CONTRACT OR GRANT NUMBER(s): DAAG29-77-G-0045
9 PERFORMING ORGANIZATION NAME AND ADDRESS: School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332	10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11 CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, P. O. Box 12211, Research Triangle Park, NC 27709	12 REPORT DATE: September 1979 / 13 NUMBER OF PAGES: 273 + xiii
14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same as Item 11	15 SECURITY CLASS (of this report): Unclassified / 15a DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16 DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17 DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

N/A

18 SUPPLEMENTARY NOTES

The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

COBOL
Management of Programming
Portability of computer programmers
Programmer productivity
Programming Tools
Software Tools

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

The COBOL Programmer's Workbench is a fully integrated collection of automated software tools designed to substantially aid in the design, implementation, test, and maintenance of COBOL data processing systems, especially those that must run on a variety of target host operating environments. The Workbench also assists in the preparation and maintenance of all supporting documentation. One of the most important capabilities of the Workbench is the automatic preparation of a set of equivalent but compiler-unique versions of a baseline program that has been written in Workbench COBOL (COBOL-plus).

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410 044

20. Abstract cont.

The research ~~effort reported on~~ here included an investigation of the problems of converting a baseline program into compiler-unique versions, an initial study of the use of reuseable modules in-line COBOL code, a limited feasibility demonstration of these capabilities, and a preliminary study for the design of COBOL.wbc.

ABSTRACT

The COBOL Programmer's Workbench is a fully integrated collection of automated software tools designed to substantially aid in the design, implementation, test, and maintenance of COBOL data processing systems, especially those that must run on a variety of target host operating environments. The Workbench also assists in the preparation and maintenance of all supporting documentation. One of the most important capabilities of the Workbench is the automatic preparation of a set of equivalent but compiler-unique versions of a baseline program that has been written in Workbench COBOL (COBOL.wbc).

The research effort reported on here included an investigation of the problems of converting a baseline program into compiler-unique versions, an initial study of the use of reusable modules to produce in-line COBOL code, a limited feasibility demonstration of these capabilities, and a preliminary study for the design of COBOL.wbc.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SECTION 1

INTRODUCTION

1.1 SCOPE OF THIS PROJECT

This project was initiated originally as a broad-based examination of the problems and difficulties encountered in transporting very large COBOL programming systems and the development of tools that would assist in such activities. As the project progressed, it became apparent that there were a number of other issues that should be addressed, such as the maintenance of transportable programs as well as the quick development of breadboard programs to examine specific concepts. These concepts have been integrated into the original project and this report covers the most important output of this project, the COBOL Programmer's Workbench which is the collection of tools and other supporting software for the design, implementation, and maintenance of baseline programs that can be easily transported to a wide variety of target operating environments as well as for the support of the quick development of breadboard programs.

1.1.1 Goals

The specific goals established for the final portion of this project were:

1) Examine the problems involved in establishing a baseline program that could be easily transported to a wide variety of target operating environments.
2) Establish techniques for the design and implementation of such baseline programs utilizing the maximum assistance possible from automated software programming aids.
3) Develop a plan for the maintenance of the system documentation required for support of the programs as well as user documents.
4) Examine the feasability of integrating all of these tools into a single programmer support environment.

1.1.2 Accomplishments

The project has been successful in accomplishing the specific goals set out at the beginning.

1) The concept of being able to develop a single baseline program that can automatically be transported to various operating target environments has been successfully demonstrated.
2) The preliminary characteristics for the COBOL

programmer's environment have been established.

3) All of the software aids have been integrated into a single operating environment known as the COBOL Programmer's Workbench.

1.1.3 Organization of this Report

Following this introductory section, which establishes the environment applying to the need and value of transportable baseline programs, Section 2 presents an overview of the facilities and capabilities that would be provided by a complete COBOL Programmer's Workbench. Section 3 describes the functional components that constitute the Workbench. Section 4 discusses how the Workbench would be utilized to accomplish the various tasks required during the design, implementation, and maintenance of large programs and systems. Section 5 presents some details on the implementation of the demonstration version of the COBOL Programmer's Workbench that was implemented at the Georgia Institute of Technology. Following a summary section, the appendices provide more detail on topics such as Workbench COBOL (the transportable subset of the language), techniques for document control, a description of the major components of the Georgia Tech Software Tools Subsystem which was utilized extensively in the implementation of the Workbench, a glossary, and, perhaps of most interest, an annotated example of the operation of the Demonstration Workbench.

1.2 MOTIVATIONS FOR THE COBOL WORKBENCH

There are a number of objectives motivating the design and implementation of a facility such as the COBOL Programmer's Workbench. A few of these are outlined below:

1) Reduce cost and increase productivity.
2) Avoid reprogramming.
   a) Support the multiple use of program modules.
   b) Reduce the problems of system maintenance.
   c) Support easy transportation of a large program to a number of operating environments without requiring reprogramming.
3) Maintain control of the baseline program.
   a) Code.
   b) Documentation.
4) Support the quick and low-cost development of breadboard systems that will allow the examination of a system or program concept without incurring the full cost of complete development.

1.3 OPERATING ENVIRONMENT OF INTEREST

Although almost any programming environment would benefit from the ideas and facilities that are incorporated into the COBOL Programmer's Workbench, the research on the Workbench was focused primarily on an environment having the following characteristics.

1) A single organization is responsible for the design, implementation, and maintenance of a large number of data processing systems.
2) The data processing systems are written in COBOL.
3) The systems are quite large, usually consisting of more than 100,000 COBOL statements and often much larger.
4) The data processing systems have a long lifetime - five to eight years - and they must be maintained by the preparing organization throughout their entire lifetime.
5) The personnel in the maintenance group of the programming organization may work on more than one system.
6) The individuals that develop the system are not necessarily available for for its maintenance; most often, they will not be available.
7) The data processing systems will be executed on a variety of hardware configurations which

may not utilize COBOL systems identical to that utilized by the development agency.

8) The design cycle for a new system or a major modification to an existing system is extremely long and costly.

9) The time pressures involved for the delivery of a new system usually preclude being able to produce a trial design.

Figure 1.3-1 illustrates the meanings of the terms used in this report to describe the various programs existing in the operating environment --- RELEASES, VERSIONS, and MODIFICATIONS.

1.3.1 Multiple Execution Environments

An extremely important characteristic of the environment projected for the utilization of the COBOL Programmer's Workbench is the requirement that the COBOL data processing systems developed be capable of being executed on a "wide" variety of computer systems. The variety of execution environments may include totally different systems supplied by different hardware manufacturers as well as variations in the operating systems under which identical hardware systems may be executing. The different environments may also exhibit large variations in the operational load placed on the execution system since the total processing loads may be quite different.

knowledge, information, and understanding required to transport a program exceeds even that required to maintain the program. (We at Georgia Tech did not consider our transporting activity to be totally successful; however, the knowledge that we gained about the problems of transporting programs was extremely valuable.)

There is no avoiding the fact that transporting large COBOL programs from one operating environment to another is an extremely costly and time consuming activity. There are two approaches to this problem and both have applicability to specific instances.

The first approach might be characterized as a linear technique. The program is originally written on machine A to run only on that system. Then, as the need arises, the program is transported from machine A to machine B, and, then, later from machine B to machine C and so on. If there is only a single operating environment required at any one time and if the machine conversions do not occur too frequently, then the linear approach to transporting the programs is probably the best and cheapest since the only problems that must be addressed are those arising directly from the incompatability of the two machines involved.

The other approach to transporting programs can be characterized as the "star technique." The program is originally written in a language applicable to the development machine.

This program then becomes the baseline program which is controlled to insure that the program continues to meet the operating requirements of the system. In this environment, whenever a new target operating environment is required, the program is transported from the central development machine directly to the target environment. It should be recognized that in this environment, the initial cost of developing a program on the central machine that can be transported to a wide variety of target operating environments without any recesion is higher than the cost for developing a program on a single machine that will then later be transported to only another single machine.

SECTION 2

CAPABILITIES OF THE COBOL PROGRAMMER'S WORKBENCH

2.1 GENERAL OVERVIEW

The primary goal of the COBOL Programmer's Workbench is to assist and support the development of COBOL programs in an efficient, cost effective, and speedy manner. This goal is to be accomplished by the development of an integrated environment of powerful software tools easily accessible and usable by the programmer. The general objective is to make maximum use of the data processing capabilities of the Workbench in order to increase the productivity of the COBOL programmer.

The COBOL Workbench provides a wide variety of capabilities to the systems analyst and programming staff. As can be seen from the list of these capabilities given in Table 2.1-1, it is envisioned that automation support be provided for all phases of program development, maintenance, and documentation.

The central concept of the programming subsystem is the preparation of a baseline program that can be easily and accurately transported to a large variety of target operating environments. The baseline program is written and

maintained in "Workbench" COBOL and is processed by one of the components of the Workbench to produce a "compiler-unique version" of the baseline program. The Workbench also supports the testing of the baseline program. It will also provide some assistance in the testing of compiler-unique versions of the baseline program; however, it is obvious that the final testing of these programs must be done on the target environment. Both the baseline RELEASES of the program as well as its compiler-unique VERSIONS will be maintained on the Workbench. The principal problem here is not so much one of programming in COBOL as it is a problem of document maintenance and control. Therefore, the program maintenance function is almost totally supported by the capabilities of the documentation subsystems.

The documentation subsystem is designed to handle both program text and supporting documentation. The preparation of programs is supported by the programming subsystem while the preparation of supporting documentation is supported by the documentation subsystem. However, both of these types of documents fall under a common document control procedure. In a situation where programs are under continual maintenance and modification, several versions of both the code as well as the support documentation must be maintained. It is the function of document control to provide programmers and systems analysts with the ability to recall any specific versions of a program or its supporting documentation that they wish to utilize. The documentation

subsystem also provides the capabilities needed for producing final documentation in a format suitable for publication and distribution.

Another important capability of the Workbench is the ability to produce and utilize reusable COBOL modules. There are a number of COBOL programming support systems in existence that facilitate and support the preparation of COBOL programs in modular form so that a program may be divided among a team of programmers. However, those systems support the preparation of modules for utilization in a **single, compiler-and-application-dependent** COBOL program. In other words, the final use and the detailed specifications of each module are dictated by the specific program under development. The goal of the Workbench is to give programmers the ability to develop modules that can be used in several different programs. These modules will have "standard" interfaces and will be capable of being parameterized internally to match the specific requirements of the COBOL program under development. The purpose of providing this capability is to facilitate the rapid and low-cost preparation of prototype COBOL systems providing a specified capability and functionality without the expense and time required for the design of a completely new COBOL system.

TABLE 2.1-1

CAPABILITIES OF THE COBOL PROGRAMMER'S WORKBENCH

I. Programming Subsystem
  A. Program Preparation
    1. Programmer's Environment
      a. Special COBOL Screen Editor
      b. Standards Enforcement
      c. Library Support
        - Program Preparation Skeletons
        - Utilization of Reusable Modules
    2. Transportable Baseline Program
      a. COBOL.wbc
    3. Program Processing
      a. Expansion
      b. Library Support
      c. Compiler-Unique VERSIONS
  B. Program Testing
    1. Test "Harnesses"
    2. Test-Data Generation
    3. Test Libraries
  C. Program Maintenance
    1. Incremental Changes
    2. Support of Multiple RELEASES/VERSIONS
II. Documentation Subsystem
  A. Document preparation
    1. Originals
    2. Updating
  B. Document Control
    1. Support Multiple RELEASES/VERSIONS
    2. Audit Trail of Changes
    3. Linkage of Comments
  C. Document Production
    1. Programs
    2. Supporting Documentation

2.2 PROGRAM PREPARATION

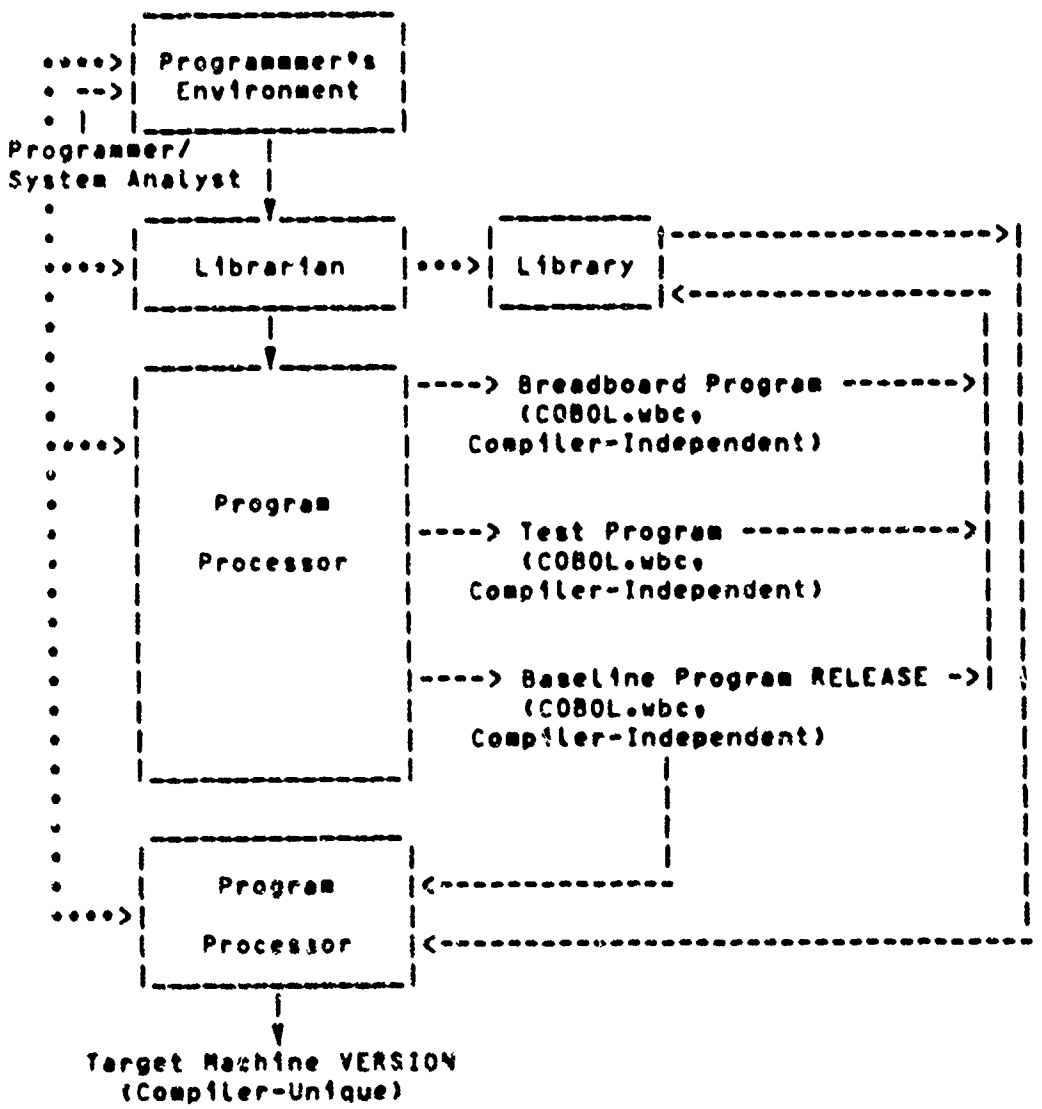
Program preparation is the first major capability provided by the Programming Subsystem. It consists primarily of the capabilities to produce the Baseline Program RELEASE and the Compiler-Unique VERSIONS of that RELEASE. The Programming Subsystem also provides the capability for quickly and easily implementing a "breadboard design" of a programming system utilizing reusable modules and other facilities of the Workbench.

2.2.1 Overview of Program Preparation

One of the goals of the Workbench is to provide a programming environment which will assist the programmer throughout the entire development, implementation, operation, and maintenance processes. This implies the development of a number of software tools. Another goal is the concurrent development of programs for different target machines. The approach taken in the Workbench is to write in an environment-independent language, Workbench COBOL (COBOL.wbc), and to convert that language to a particular COBOL dialect only when preparing the delivery of the complete program to the installation (see Figure 2.2.1-1). A third goal is to provide the capability to make use of reusable code modules whenever a task or function can be identified that is frequently used.

FIGURE 2.2.1-1

PROGRAM PREPARATION



2.2.2 Programmer's Environment

The programmer's environment may be described as the interface between the human and the machine. It consists of those tools which are provided to assist directly in the task of preparing code. Depending on the particular system, the programmer may be faced with a "hostile" or a "friendly/hospitable" environment. A hostile environment may complicate the programming task in many ways: primitive file system, restrictive naming conventions, obscure error messages, etc. A system may also make programming difficult by providing the programmer with only a few tools of limited capabilities.

The Workbench contains a number of very powerful, yet easy to use software tools to simplify the programmer's task and to permit a more efficient interface between the human and the machine. These tools include a special editor for entering COBOL text, a standards enforcer to verify the clerical correctness of code before further processing, skeleton programs to aid in code preparation, test routines to aid in the test process, and a well-structured file system to allow all of these to work together efficiently and productively.

2.2.2.1 Special COBOL Editor

The purpose of the special editor is to assist the programmer in the entering, maintenance, and updating of files containing COBOL code. It is designed to support the interactive preparation of COBOL code utilizing a CRT terminal.

The special editor should have the following capabilities:

1) Provide the programmer the capability to enter COBOL code easily.
2) Provide the capability to modify or extend existing code files.
3) Organize and display the code entered in a standard and easy to read format.
4) Provide the ability to refer to code by line numbers as well as paragraph names.
5) Provide the ability to recall and easily modify modules of code such as file definitions or a series of almost identical repetitive calculations.

2.2.2.2 Standards Enforcer

The purpose of the standards enforcer is to verify that the programmer has adhered to the COBOL.wbc syntax, names, and format standards.

The standards enforcer must check that the following requirements are met:

1) Code is written in a machine- and compiler-unique form. In the Workbench, this is accomplished by verifying that all statements are acceptable within COBOL.wbc syntax.
2) Naming conventions are followed correctly. At the least, it is possible to determine that a variable used in several code segments maintains the same data type throughout its use.
3) A check is made for clerical errors, such as the misspelling of reserved words or of variable names. Following this, the code is processed through a "pretty-print" program to guarantee clerical uniformity.

2.2.2.3 Library Support

There are a number of different "libraries" utilized by the Workbench, and several of these collections of files may be utilized by the same Workbench activity. A complete discussion of the various libraries and their components will be given later after more of their uses have been covered. As far as supporting the COBOL Programmer's Environment is concerned, it should be obvious that some of the files that should be available are

1) Blocks of code providing definitions of

characteristics applicable to a specific program under development such as the File Definition statements.

2) Blocks of code providing a framework for the development of a special or tailored version of a commonly encountered function such as the File Definitions, an edit routine, a repetitive calculation, etc.

3) Larger blocks of code that are treated as "open subroutines" and inserted in the program under development as in-line code.

4) etc.

These files might have been developed by a programmer for his own use and kept in his private or Personal Library, or they may have been developed for wider usage and placed in the General Library or one applicable to a specific application area or project under development. Since similar files may exist in all of these various libraries, the order in which they are searched is very important. This topic is also discussed below.

2.2.2.3.1 Reusable Modules

An example of the use of reusable modules may be seen in the very large class of problems in which a transaction file is processed against a master file to produce a report. The mainline processing may be performed by the following module:

```
OPEN INPUT MASTER-FILE
           TRANS-FILE
     OUTPUT REPORT-FILE

PERFORM PROCESS-TRANSACTIONS
     UNTIL NO-MORE-TRANS

CLOSE MASTER-FILE
      TRANS-FILE
      REPORT-FILE.

STOP RUN.
```

Once this module of code is thoroughly tested, it can be copied from the appropriate library for use without modification in similar programs. Since this module is logically quite simple and relatively short, it may appear easier to write it again for each new program than to expend the additional effort required to check out the code and catalog it in a library. The major advantage of considering this short program segment as a reusable module is that these nine lines of code can be included in some larger module by means of a simple macro call. Thus many errors, even simple typographical ones, are avoided.

Continuing the same example, other reusable modules may be developed. The main-line module performs a paragraph PROCESS-TRANSACTIONS which is invariant.

The module

```
PROCESS-TRANSACTIONS.
     MOVE 'NO' TO VALID-TRANSACTION.
     PERFORM GET-VALID-TRANSACTION
          UNTIL VALID-TRANS
          OR NO-MORE-TRANS.
     IF VALID-TRANS
          THEN PERFORM PREPARE-RESPONSE.
```

will call the reusable module GET-VALID-TRANSACTION and con-

ditionally call the module which will actually process the transaction. The paragraph PREPARE-RESPONSE will typically not be in the form of a reusable module since each application program using this general model will have different requirements for the processing of a transaction.

The essential feature of the two paragraphs given in the example above is that they apply to a large class of application programs and therefore may serve as a model for the development of reusable modules for a commonly encountered class of problems.

2.2.2.3.2 Program Development Skeletons

As contrasted to reusable modules, which provide the complete code required to accomplish a specific task or automatically generate such code after suitable parameters are provided in the reference to the module, a program development skeleton is primarily a framework and guide for the development of a program which might vary quite considerably from anything that has been developed previously. A program development skeleton is much more generally applicable than a reusable module. An example of a skeleton is the partial description of the file definition section of the program in which those characteristics which have been established as local standards for the programming organization are already specified and the skeleton indicates where further information should be provided by the programmer utilizing it in the preparation of a specific program. There can also be program development skeletons

which handle much larger tasks. There are even instances of skeletons being utilized for the development of complete programs.

2.2.2.4 Use of Libraries

Different types of modules and skeleton programs are provided on different libraries to permit ease of reference and to prevent improper access. The libraries defined are described in section 2.2.4.1 and include libraries for the following:

1) General purpose modules.
2) Modules for a specific application area.
3) Modules under development for a particular project.
4) Programmer-developed modules for personal use.
5) Test modules.
6) Compiler-unique modules.

2.2.3 Transportable Baseline Programs

The development and maintenance of a complete COBOL program that can be easily transported to a large number of target computer operating environments is certainly one of the most important products produced by the COBOL Programmer's Workbench. There is certainly a large monetary value that can be associated with the other capabilities of the Workbench such as the ability to quickly develop prototype systems

utilizing reusable program modules and the machine assistance provided in the development and production of supporting documentation. However, in the environments of interest in this study (those that were detailed in section 1) the major economic payoff appears to be vested in the ability to produce a program that can be automatically transported to a variety of target computer operating environments without extensive human intervention in the transportation process. This transportation capability is provided primarily through the utilization of Workbench COBOL or COBOL.wbc as the programming language.

2.2.4 Workbench COBOL

The concept of Workbench COBOL is certainly one of the most important central ideas in the COBOL Programmer's Workbench. The objective of Workbench COBOL is to provide a language the programmer may use to prepare programs that can be easily transported to a variety of target environment systems. A secondary objective of Workbench COBOL is to include facilities for the use of reusable COBOL modules and other program productivity aids. There have been, since the very beginning of COBOL, a number of demonstrations of the ability to transport programs from one system to another. This has been reinforced in recent years by the establishment of national standards for COBOL; however, it is a well recognized fact that it is extremely difficult, if not impossible, to execute any but the most trivial COBOL programs on different systems without making some changes to the

code. Some differences are quite obvious, such as the situation where statements required by one compiler must be absent in another or prepared in quite a different form, e.g., the file label statement. Other problems of transportation are much more difficult to handle, such as the effects of word size when the computational form for variables is utilized. It is a goal of the design of Workbench COBOL to overcome all of these difficulties.

The basic objectives in the design of the Workbench COBOL language are as follows:

1) COBOL.wbc should be as close as possible to standard COBOL in syntax and semantics. This will greatly assist in the training of programmers to utilize the Workbench.

2) COBOL.wbc should be capable of unambiguous translation to a wide variety of compiler-unique target environment dialects. For this reason, there will be features and statement types included in Workbench COBOL that appear identical to those in the standard; however, it may be necessary that these statements be processed by the Workbench in order to produce the exact form required for a particular target dialect.

3) COBOL.wbc should be "rich" enough for easy use by programmers in the development of complex programs. It appears that there is no

real need to include all of standard COBOL in COBOL.wbc. What is more desirable is to insure that those features of COBOL that are frequently used by programmers are available in Workbench COBOL. The basis for the design of COBOL.wbc should be a thorough examination of the dialects of standard COBOL that are encountered on the target environments as well as a study of the types of statements that are actually utilized by COBOL programmers. (Initial work in this area is documented in Appendix 9.)

The solution to the "transportation problem" seems simple. If programmers are restricted to implementing their programs using only the features and format of "standard COBOL," those programs should compile and execute on any machine. Unfortunately, it is not that easy. Despite the existence of two COBOL "Standards" (68 and 74), there is no real agreement on the features that should be included in the language. Even if a subset of language features could be identified which were provided by all COBOL compilers, statement syntax varies between compilers. Even in those cases where the syntax used to express a particular language element by two different compilers is the same, the language features of the two compilers may have different interpretations. The next possible solution is to select one of the two COBOL Standards as the "Standard Standard"

and to develop a compiler for the language which that standard defines. However, compilers are developed to meet the needs of a particular machine environment and interpretation of certain features is ultimately a decision of the compiler designer. Many operations, such as input and output, must depend on the host operating system. Since different machine environments provide many different features, a standard compiler is not sufficient to eliminate the transportation problem.

Perhaps a solution can be found by investigating the method in which COBOL programs are generally transported now. The general procedure is to locate those features in the grammar of the source compiler which are not the same as in the grammar of the target compiler. All occurrences of these statements must then be located in the source code and translated into the corresponding format for the target system. This step may be carried out with the aid of a text editor. The next step involves locating those language features which are provided by the source compiler, but are not available from the compiler to which the program is to be transported. (Note that the reverse step is not necessary, but may be desired due to the possible increase in program efficiency.) Those sections of code which rely on such features must be identified and recoded using only features available from the target compiler. If the source program was developed for a compiler which accepts a very rich dialect of COBOL and the target compiler accepts only a

minimum, this process may be very long and difficult. As this cannot be automated, the possibility of introducing errors to the translation which were not present in the original is very high. In some cases, transportation may be found to be impossible.

The above procedure only describes the transportation of a COBOL program from one particular compiler to another particular compiler. The entire process is repeated every time the source program is to be moved to a new environment. Also, since no compiler is accepted as the "source" compiler, programs may have to be transported between any pair of COBOL compilers.

In an easily transported COBOL program, only a subset of COBOL is used, consisting of those features which are available in some form from all COBOL compilers. Where this is not possible, as in the case of certain machine dependent features such as occur in input and output operations, these compiler-unique features are isolated in easily identified sections of code which may be changed without requiring the recoding of other portions of the program. The remaining differences, due to differing syntax between the two compilers, may be easily eliminated with the aid of a text editor.

The process described above permits easier transportation of COBOL programs between differing machine environments but

still requires the efforts of a programmer who is cognisant with both the source and target systems, as well as being familar with the desired behavior of the program to be transported. Since programmers are an expensive resource, it would be desirable to automate this procedure.

A necessary first step in attempting this automation is the reduction of the number of pairs of dialects between which programs are to be transported. As new dialects are continuously being developed, the scope of application of the resulting system would be unduly limited were the choice of target languages to be restricted in any manner. Less difficulties arise in limiting the number of possible source dialects. No single existing dialect contains enough of the necessary information to permit easy translation into all possible target dialects. As a result it is necessary to develop a dialect of COBOL particular to the Workbench system (COBOL.wbc). COBOL.wbc will provide the user with those COBOL features that can be supported in a "portable" programming environment. Some features which rely on specific hardware or operating system capabilities cannot be implemented on all target machines and thus will not be available in COBOL.wbc. COBOL.wbc can be translated into any of a number of target dialects (COBOL.k). Much of the translation process will be automated; however, because of variations in computer architectures and organizations, some human intervention will still be required to translate a program from Workbench COBOL into the COBOL dialect that

will execute on a specific machine.

2.2.5 Program Processing

There are two primary products from the Program Processing Subsystem of the Workbench. The first of these are "compiler-independent" versions of the program or module that are written in COBOL.wbc. The second type or group of products are the "compiler-unique" versions of the program written in a specific dialect of COBOL (COBOL.x) and ready for furthering processing (translation) and execution by a target operating environment. The first class of products are the result of expansion of the code entered by the programmer while the second group results from translation of COBOL.wbc into a target COBOL dialect.

2.2.5.1 Expansion

The purpose of program expansion is to produce compiler-independent program modules meeting the standards of COBOL.wbc and capable of being collected together with other modules to produce complete, compiler-independent programs.

Input to the program expansion process consists of programmer specification of the COBOL.wbc text for the module, the name of the module, and the name of the library in which the module is to be found. The output is a program module, or complete program, meeting the standards of COBOL.wbc.

Expansion may be performed relative to one library at a time, or to any combination of libraries.

2.2.5.1.1 Expansion Utilizing the General Library

The General Library will be available to programmers on all projects and in all application areas and will include general-purpose modules which are frequently used. These modules might include the following:

- data input editing
- file structure definition
- report generation

Use of these general purpose modules will eliminate much repetitive coding. The possibility for error will also be reduced greatly through two causes. Mistakes caused by clerical error will be less frequent, since fewer lines of code will be entered. More significantly, the General Library will include only modules that have been thoroughly tested, reducing the possiblity of incorrect code.

2.2.5.1.2 Expansion Utilizing the Application Library

The Application Library contains modules developed for an applications area such as inventory or payroll. These modules will be thoroughly tested before addition to the application library. The application libraries might include the following:

- a collection of payroll modules
- a collection of modules developed for inventory control

Each applications area will possess its own library, APPLIB-name (e.g. APPLIB-PAYROLL or APPLIB-INVEN), which will contain this type of module. Projects will be permitted to make use of other application libraries as well to encourage standardization and to decrease maintenance costs. One example of this potential savings is a business maintaining both a payroll program and a financial report generator. If both of the programs call the same module to calculate withholding tax, changes in the tax laws will cause less modification to the currently running programs than if each figures the tax separately. Standardization also decreases the errors that occur when different heuristics are used for the same function by different programmers.

2.2.5.1.3 Expansion from the Project Library

The Project Library contains those modules currently under development specifically for a particular programming project. Each project will have a library, PROJLIB-name, containing these modules. This library differs from an application library primarily in including modules tailored to a specific project.

2.2.5.1.4 Expansion from the Personal Library

The programmer may develop a personal "shorthand" notation to avoid repetitive programming when the same task is to be performed a number of times. Personally developed modules

give the programmer the ability to enter code into the machine in a convenient form. Calls to modules contained in a Personal Library must be expanded into standard COBOL.wbc as soon as the code containing those non-standard module calls is made available to other members of the programming project (included in the Project Library).

2.2.5.1.5 Expansion from the Test Library

The Test Library, TESTLIB, will be available to programmers on all projects and in all application areas to assist in module and program testing at all stages of project development. The Test Library contains modules for the collection and output of test data, dummy modules to be used in testing calling sequences, and skeleton programs for the testing of module behavior in isolation and as a result of different calling sequences.

2.2.5.2 Build Programs

In order to build a program or system, the following input is needed:

1) A collection of COBOL.wbc modules selected by the programmer in order to produce a desired overall functional capability in the output program.
2) Special Instructions required by the Program Preparation Subsystem.

The following output is generated:

A complete, but non-executable, COBOL.wbc program which can be expanded with the appropriate choice of target machine macro library into an executable COBOL program in a form which is acceptable to any one of a number of target COBOL compilers.

Specifically, in order to build complete COBOL.wbc programs, the Workbench must have the following capabilities. The Workbench must be able to

1) Collect and link together all of the appropriate interface components of the input COBOL.wbc modules producing an output program in COBOL.wbc. These interface components include but are not limited to data definitions and file specifications.
2) Resolve or identify unresolvable naming inconsistencies.
3) Establish consistency in data and file definitions or identify possible problems that might result from inconsistencies.

2.2.5.3 Transportion

One of the most important capabilities of the Workbench is the **development** and **transportation** of **portable** COBOL programs. This goal is achieved by the development of baseline COBOL programs in a compiler-independent form (in COBOL.wbc) and the completely automatic translation of these programs into the COBOL dialect of a particular target compiler system.

This translation could have been accomplished by the development of a new COBOL dialect and of a new compiler to accept this language. This was not the chosen solution. A new COBOL compiler and a new dialect would only add to the problem. While it would be possible, assuming programmer and industry acceptance of the new dialect, to develop new programs which could run on any target machine, the addition of new environments to the original set of target machines would be an extremely difficult proceeding and would probably result in the same chaotic situation presently existing with the "standard COBOLs." Worse, only new systems would be portable. Systems would be able to include previously written software only after extensive rewriting to translate the old program into the new language. Finally, development of yet another COBOL compiler does nothing to address the problem of developing reusable code.

The chosen solution is to express those functions and statements for which a compiler-dependent implementation is necessary as special statements in a compiler-independent representation of the program, and then to "expand" these statements separately, relative to each target compiler.

2.2.6 Breadboard Programs

A major expense in the development of automated information systems has been the high cost for the redesign of the system after the initial prototype has been implemented. This is especially true when a separate organization is preparing the program for another organization which is the proponent or motivator for the system to be designed. One approach that has been proposed to lower this initial design cost is the utilization of a breadboard program, or system, that provides the basic functionality of the desired system without the high expenses associated with detailed and complete development. This breadboard program or prototype could then be demonstrated to the proponent organization and feedback on the design would be obtained at a much earlier stage and at a much lower cost without complete development being required.

The characteristics of a breadboard program/prototype system would be that most of the desired functionality would be present in the system; however, the performance of the system would certainly not be optimized or even "well

engineered." To the maximum extent possible, the breadboard system would be assembled out of reusable modules of code which provide the functionality desired, or close approximations thereto. The overall objectives of breadboarding are to provide quick and low-cost prototypes of systems under design so that meaningful feedback may be obtained at the earliest date possible.

2.3 PROGRAM TESTING

The modular approach to the preparation of complex systems significantly aids in the development of quality programs. While it is true that in most large systems absolute program accuracy can only be approximated, the separate testing and verification of the "correctness" of individual modules, before the modules are combined into a single program unit, will lead to the development of quality programs in an economic fashion.

Program testing is aided by a collection of utility functions and modules which are used by the programmer to test both modules and programs. Besides providing the opportunity for functional testing, utilities will be provided to aid in the measurement of performance. Specific utilities which might be included for this purpose include:

1) main programs to invoke the module in question and print input and output ("test harnesses")
2) routines to generate random input data meeting a specified COBOL format
3) routines to generate random input files meeting the conventions of a particular compiler and machine
4) routines to collect performance data, including time and space usage

These utilities are placed in a special test library which

contains tools used for testing modules and programs. Some of the essential tools are:

1) Test Control System
2) Test Harness
3) Automatic Test Data Generators
4) Automatic Verification System
5) Object-Time Monitors

2.3.1 Goals for Testing Support

Program testing capabilities are included in the Workbench to provide for the testing of COBOL.wbc modules produced using the Workbench Module Preparation Subsystem prior to the inclusion of these modules in COBOL.wbc programs. The following goals must be met:

1) Provide a test environment covering both language tests as well as I/O and data tests to easily check-out the modules as they are produced by the above process.
2) Permit the collection of data about module performance **before** the inclusion of the module in a finished program.
3) Test module behavior as it will perform on each target machine. This will involve converting the original COBOL.wbc module into each member of the related family of COBOL.i modules, providing COBOL.i drivers, and executing each of the resultant programs on the appropriate machine.

4) Provide utility routines to aid the programmer in the generation of test data meeting particular program requirements as specified by the appropriate data declaration division.

5) (Optional) Provide utility routines to enable the programmer to determine if the set of test cases is adequate to properly test the modular logic.

2.3.2 Test Harness

One of the major tools to be used in program (or module) testing is the test harness. This main or driver program serves as a framework which supplies appropriate inputs to a module. It also records the outputs produced by the module (along with the associated inputs) in such a manner that the programmer can determine whether the output produced is in accordance with the specifications established by the system designer.

2.3.3 Test Control System

The Test Control System causes the tests to be executed. Through the Test Control System, the appropriate test data is retrieved, the driver or test harness is executed using the prescribed test data, and any postprocessing of the output from the test harness is performed. The Test Control System automatically modifies the source code as necessary when performance evaluations are conducted. In the case of

the testing of modules which perform certain file processing activities internally, the Test Control System will automatically compensate for differences between the way in which files are stored and accessed during the test and the way in which files are to be managed under the eventual production system.

It is through the Test Control System that most of the tools used in the program testing phase are sequenced, monitored, and controlled. The Test Control System can also gather, record, and report in an appropriate form information to be used in the management of program testing. Pertinent information regarding each test (who tested what module, what data was used in the test, when was the module last tested, etc.) are logged by the Test Control System.

2.3.4 Test Data Generation

While the testing of individual modules or components of a large system may significantly aid in error prevention, each individual module must be exercised for the purpose of error detection. The primary goal of the testing process is to make mistakes happen (if errors do in fact exist) with results that can be easily recognized as incorrect and which provide enough information about program behavior that the errors can be found quickly and corrected. In order to accomplish this task, enough inputs must be tested to exercise the program or module under consideration completely.

The development of sufficient test data is an exacting task. It is not enough that a large number of inputs are tested; the variety of inputs tested is also important since each control path in the code must be examined. If the test data is prepared manually, control paths may be missed, causing errors to pass undetected until the system is in production, and any errors found will require expensive correction. The preparation of "good" test data by manual means is an extremely difficult task which requires extensive analysis of branching and control flow. The Workbench must therefore provide a mechanism to facilitate the generation of a sufficient number and variety of test cases for each module to be tested. This facility must provide for the automatic generation of test data using (at a minimum) as input the DATA DIVISION of the test harness and the test definitions given by the programmer in some procedural notation. It will also permit the generation of test data by combining records from existing files of test material and integrating these records into a comprehensive set of test data. This latter capability may be used to insure that all modules in a particular program receive final testing over an identical data set produced as the union of test input generated for the individual modules.

2.3.5 Automatic Verification System

A number of tools will be available for the analysis of output resulting from execution of the test harness. An output analyser would permit comparision of outputs with corresponding inputs, with previously generated outputs, or with predetermined results presented to the analyser in some form, and would report the discrepancies on an exception basis. Standard statistical library routines would be used to interpret the output from the object-time monitoring facility.

2.3.6 Object-Time Monitors

Options provided within the Test Control System will permit the "instrumentation" of the module being tested in such a way that program execution efficiency can be evaluated. The areas within a given module which will receive closest monitoring will be determined by usage statistics provided at the source program paragraph level.

2.4 PROGRAM MAINTENANCE

The maintenance of programs that have already been developed and placed into operation is widely acknowledged as the most expensive part of the life-cycle cost of such programs. The COBOL Programmer's Workbench fully acknowledges this fact and provides complete support for maintenance.

2.4.1 Nature of Maintenance Activities

Referring to Figure 1.3-1, it can be seen that there are two dimensions to the maintenance activities supported by the Workbench. The first of these is the maintenance that must be performed on compiler-unique VERSIONS of the program in a "quick-fix" manner to correct logical errors in the development of the baseline program or Workbench errors in the translation of that program to the compiler-unique environments. Each VERSION of the program presents a separate maintenance problem and MODIFICATIONS of VERSIONS are generated individually as required.

The other dimension of maintenance is the preparation of a new release of the baseline program. In this instance, there is only one program being maintained.

2.4.2 Maintenance Support

The support for program maintenance provided by the COBOL Programmer's Workbench consists of several capabilities.

1) The concept of a single, controlled baseline program reduces the amount of maintenance programming required in a multiple-target operating environment.

2) The programmer's environment provided by the Workbench for the preparation of programs also greatly facilitates the preparation of modifications to previously developed programs.

3) The on-line documentation system and the capabilities it provides for linking comments to specific portions of the documentation text greatly facilitate what is perhaps the most time consuming activity in maintenance operations - the correlation of all of the information applicable to the original program and the changes to be made.

4) The Documentation Control System also provides for the control of multiple versions of both code and text so that the maintenance programmer can easily check out a new version of a program without disturbing the old one.

2.5 DOCUMENTATION PREPARATION

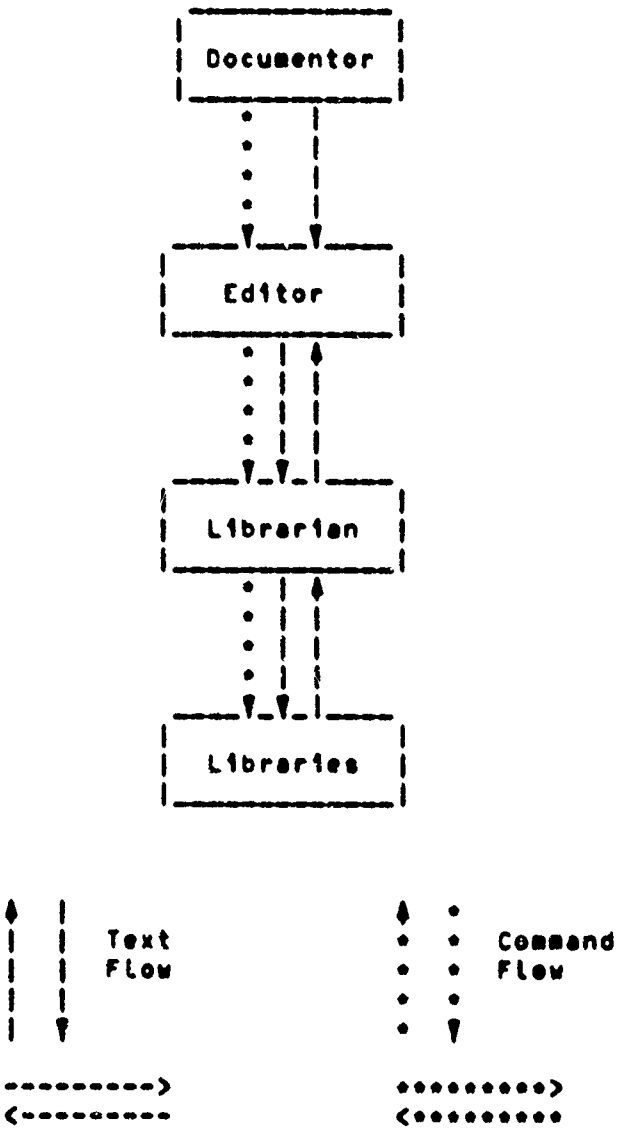
One of the major tasks faced by the designers of software systems is the production and maintenance of complete documentation including specifications, designs, user and reference manuals, and reports as well as the program code. To encourage the production and maintenance of current and complete documentation, the Workbench will be equipped with a broad range of tools which are designed to assist in the preparation of all of the system documentation required for the support of prototype systems, the base-line system, and those versions of the system produced to run on specific target machines. The documentation tools provided by the Workbench will be simple to use, will interface with each other, and will provide additional capabilities which make the documentation task less burdensome to the documentor and increase his productivity.

2.5.1 Overview of the Documentation Sub-system

Figure 2.5.1-1 depicts the flow of text and commands that will occur during the preparation of system documentation. The documentor will be able to both enter an original document as well as modify existing documents. The documents will be stored in a series of libraries. In addition, it will not be necessary for a document to be entered in a final finished format; instead, a separate

FIGURE 2.5.1-1

PREPARATION AND MODIFICATION OF SUPPORTING DOCUMENTATION



process, the Formatting Process, will be available to perform these formatting duties (see section 2.7.2). By making formatting a separate procedure, one will only have to save and revise the unformatted text of documents. This will make the task of modifying documents easier since it will be possible to make changes without worrying about the effect of those changes on the format of the final output.

2.5.2 On-Line Document Handling

One of the essential requirements for the Documentation Subsystem is that it be available for on-line use. This will enable the documentor to compose and enter documentation into the system in a single step. In addition, he will be able to extract text from other documents for use in another document that is being prepared or modified.

2.5.2.1 Preparing Documents

The documentor will be able to enter documents in an on-line fashion. Intermingled with the normal text of these documents will be formatting commands which will be interpreted at a later stage by a Formatting Process. These commands will allow the documentor to control the format of the final document (see section 2.7.2). For example, one will be able to underline and bold-face desired words in the text by simply inserting the appropriate commands in the proper position of the unformatted text.

The documentor will be able to easily edit the input text during the document preparation phase. He will have the capability to insert, delete, or modify as well as to reorder parts of the text. The documentation process often involves copying text from other sources, for example, other documentation or actual program text. The latter situation often arises when one wants to present an example within the documentation or when one wants to discuss the text of the program. Thus, an important editing feature of the Documentation System will allow text from any source to be inserted into any document while it is being prepared or modified. Since the copy capability is included as part of the editing capabilities, one will be able to obtain text from other sources and then trim and modify the text to suit his needs.

2.5.2.2 Updating Documents

The capabilities provided by the Workbench for originally preparing documents will also be available for updating documents. The documentor will be given editing capabilities which enable him to insert, delete, reorder, and modify text. As mentioned above, he will also be able to copy text from a variety of sources, including program text, and will be able to perform this while entering or modifying the original document.

2.6 DOCUMENT CONTROL

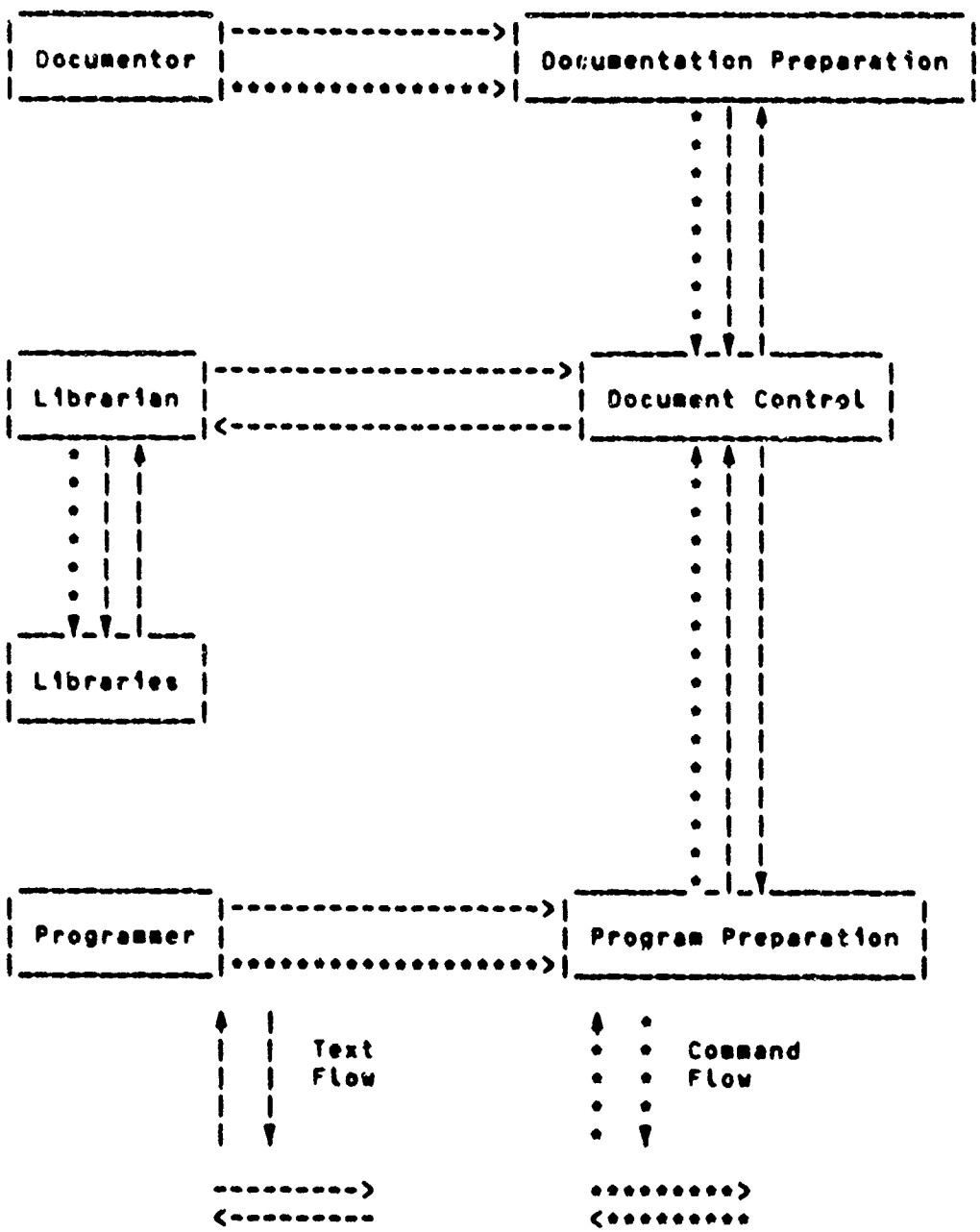
Programs and supporting documentation often exist simultaneously in several different forms. For example, a program may be in production use while at the same time improvements are being developed for it. A form of document control that can selectively retrieve any specified version of a document is required. The facility providing document control must provide the user access to any revision, allow the user to create new revisions, and insure that no change made to a document is lost. The last capability will help to insure against an accidental loss of information.

2.6.1 Overview of Document Control

The Document Control System will handle all programs and supporting documentation. Any user or process needing a specific revision of any document (the term document refers to both programs and documentation) will request that revision from the Document Control Process. Thus, both the Documentation Preparation System and the Program Preparation System will have to interact with the Document Control System in order to obtain documents. The Document Control System will store all revisions of all documents in various libraries.

FIGURE 2.4.1-1

DOCUMENT CONTROL OF ALL DOCUMENTATION



2.6.2 Audit Trail of Changes

The Document Control System will automatically maintain an audit trail of all changes made to a document (see Appendix 3). This will allow the programmer or systems analyst to identify changes made to a document in a chronological manner. Thus, corrections need not be considered permanent because they can always be recovered. It should be noted, though, that the Document Control System will never discard any change; rather, if the change is not desired, it simply will not be applied during the construction of a particular revision.

2.6.3 Ability to Maintain Different Releases/Versions

The Document Control System will provide the capability to maintain separate RELEASES of all documents. The need for separate RELEASES occurs when, for example, one RELEASE of a program is in production while a newer RELEASE is in development. In addition to maintaining separate RELEASES, one must be able to maintain the different VERSIONS of all RELEASES. Thus, the Document Control System will allow one to obtain or create any VERSION of any RELEASE of a document.

2.6.4 Linkage of Comments on Documents

To encourage as much feedback as possible on programs and supporting documentation, the Document Control System provides the ability for user comments about documents to be dated and linked to the document to which they refer. Thus, users can comment on any errors which they believe they have discovered or make suggestions for changes to the documents. The systems analyst responsible for a document will be able to examine all comments pertaining to the document. He can then act upon them and report back to the originator of the comment as to the status of the document or the point raised by the user.

2.7 DOCUMENT PRODUCTION

2.7.1 Program Production

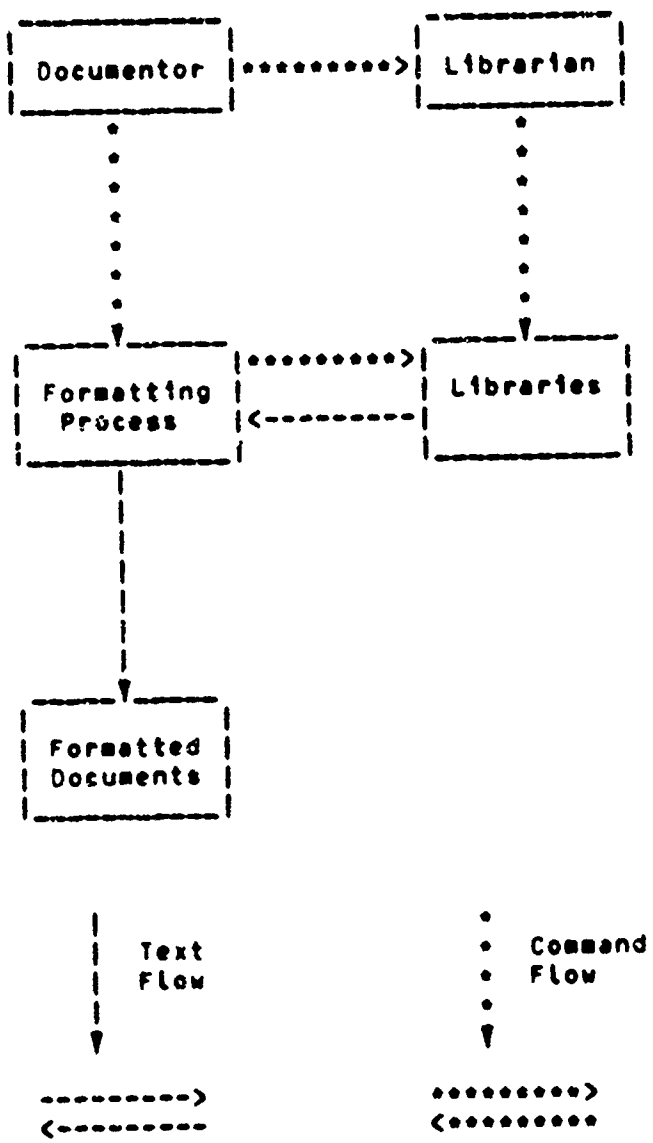
The Workbench has the capability to produce record or documentation copies of the various VERSIONS of the source code; however, it cannot produce the object code version of the program. That has to be produced in the target-compiler environment.

2.7.2 Documentation Production

The Workbench will be able to process the documents as they are stored in the various libraries and produce printed reports which are formatted according to appropriate standards. The Workbench will provide a Formatting Program which will accept commands imbedded in the text of the documents. These commands will inform the Formatter as to how the text should be printed: for example, what words should be underlined or bold-faced, what text should be centered, how many lines should be skipped between sections of the text, etc. In addition, the Workbench will provide the means for defining documentation standards and aiding documentors in following these standards.

FIGURE 2.7.2-1

PRODUCTION OF DOCUMENTATION



2.7.2.1 Formatting

The Workbench will contain a Formatting Process which will print documentation according to the users specification or a standard set of specifications stored in the General Library. The user will imbed commands in the documentation text stored in libraries which tell the Formatting Process what form of output is desired. There will be default settings for parameters not specified by the user. Thus, the Formatter will produce a formatted document according to the commands specified within the text utilizing default settings for those options not specified by the user.

2.7.2.2 Documentation Format Standards

The Workbench will also provide the capability to define documentation standards and aid documentors in following these standards. Standards may vary from project to project or from one type of document to another within a single project. The Workbench will provide a uniform technique to locate definitions of the various standards and the means to automatically produce documentation meeting those standards.

SECTION 3

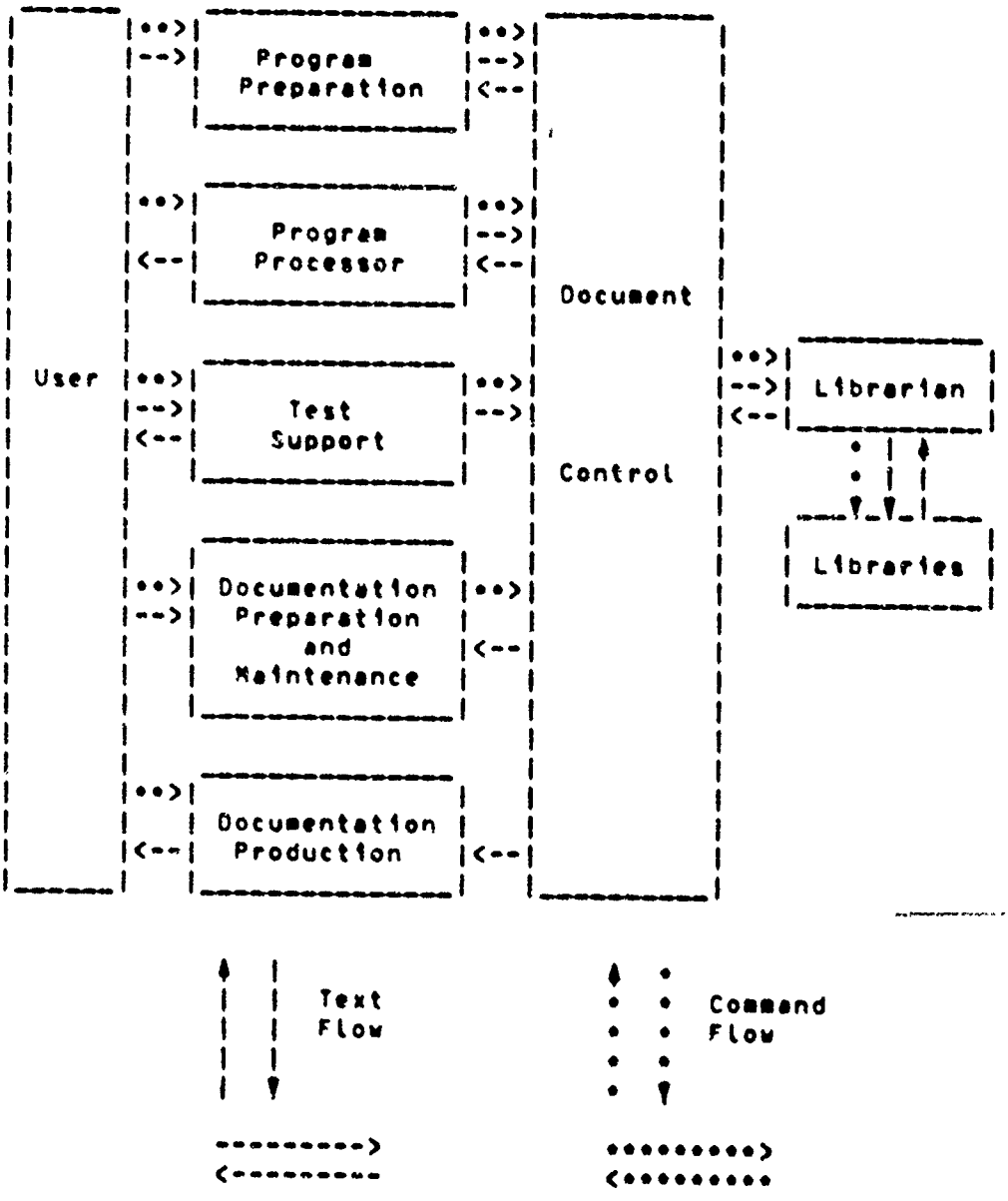
FUNCTIONAL COMPONENTS OF THE COBOL PROGRAMMER'S WORKBENCH

3.1 OVERVIEW OF WORKBENCH COMPONENTS AND ORGANIZATION

The user sees the workbench as a collection of capabilities and functions that provide him with support in accomplishing his programming and documentation tasks. Functionally, these capabilities can be divided into several major subsystems or components as shown in Figure 3.1-1. Since the major activity of the workbench is the preparation and processing of documents, whether they are program modules or supporting documentation, the subsystem providing control of those documents is certainly a central component of the Workbench. The Document Control Subsystem as shown in Figure 3.1-1 might be compared to the file system of a regular computer; however, as pointed out in Section 2 of this report, the Workbench envisions the maintenance of multiple versions of the same document, and the support provided by the Document Control Subsystem is much more extensive than that provided by normal file systems.

FIGURE 3.1-1

MAJOR FUNCTIONAL COMPONENTS OF THE COBOL WORKBENCH



3.2 PROGRAMMING SUBSYSTEM

3.2.1 Program Preparation

3.2.1.1 Workbench COBOL --- COBOL.wbc

Workbench COBOL (COBOL.wbc) is the portable version of COBOL. It has been designed to provide the user with as many of the capabilities of full COBOL as may be supported in a "portable" programming environment. COBOL.wbc was developed from standard COBOL with the explicit goal of producing a COBOL dialect which would permit a COBOL program developed on one machine to be executed on another without tedious and error-producing hand-modification. However, COBOL.wbc is not the smallest common subset of COBOL applicable to all of the target computers selected; instead, it consists of the common subset of COBOL applicable to all of the selected target computers (COBOL.ccs) plus those those features heavily used by COBOL programmers even if these features are not directly portable. The translation of this latter group of features into compiler-unique code is a function of the Program Processor.

3.2.1.2 COBOL.demo

COBOL.demo is this project's first approximation to COBOL.wbc. COBOL.wbc consists of those elements of COBOL which are common to all COBOL compilers plus a number of other capabilities which cannot be directly implemented in the same format by all compilers. The nature and type of these capabilities which must be provided in order to permit the development of useful COBOL programs is not yet known. A number of COBOL capabilities (such as data declaration) are assumed to be necessary; these capabilities will be included in COBOL.demo. Other capabilities will be added to COBOL.demo as their usefulness is demonstrated. The eventual product of this process will be COBOL.wbc.

For the purposes of this report, only a casual distinction is made between COBOL.wbc and COBOL.demo. COBOL.wbc is used whenever either COBOL.demo or COBOL.wbc might be meant. (For example, at different stages in the Workbench's development). COBOL.demo is used only in reference to developmental activities which will be over when the Workbench system is complete.

3.2.1.3 COBOL Screen Editor

In addition to the standard features available in a comprehensive screen editor (line-at-a-time deletions, additions, search, scan and substitution, and full screen

display of adjacent text), the screen editor utilized in the programmer's environment would also contain features that are specifically tailored to the preparation of COBOL programs. These features would be capabilities such as automatic formatting, automatic searching for sections of the program by name, etc.

3.2.1.4 Standards Enforcer

The most important standards enforcer is obviously the one for COBOL.wbc. This would function in a manner very similar to presently existing standards enforcers. It would operate as a preprocessor on the source code ensuring that programs contained only statements taken from COBOL.wbc or properly formatted references to reusable modules.

If it is desired that the Workbench support COBOL programming in a specific dialect (COBOL.k), then a different standards enforcer would have to be provided for that dialect of COBOL.

3.2.2 Program Processing

3.2.2.1 Library Support

A number of support libraries must be made available to the program designer. The Workbench provides the capability for each installation to develop and maintain a useful and meaningful collection of macro and module libraries tailored

to the needs of the individual installation. These libraries are described in detail in Section 3.5.1.

A COBOL.wbc program containing references to reusable modules may be expanded into an equivalent COBOL.wbc program relative to any one or more of the module libraries. For example, the reusable module references used by an individual programmer as abbreviations may be removed while leaving module calls from the Project, Application, and General Libraries. Translation of a COBOL.wbc program into the equivalent COBOL program for a particular target machine (CCBOL.k) should occur only after all reusable module references have been expanded.

The macro processor uses a search rule to locate the macro definition corresponding to a specific module call in much the same way that a loader searches for load modules. Unless otherwise specified the search rule used is:

1) Personal Library
2) Project Library
3) Test Library (utilized only when specifically requested)
4) Application Library
5) General Library
6) Compiler-Unique Macro Library (utilized only when transporting a COBOL.wbc program to a specific compiler)

In this way, improvements to modules already included in

standard libraries may be tested without recopying the standard library or modifying programs to use new names for modules.

3.2.2.2 Program Processor

Using the tools described in the previous sections, the programmer develops a COBOL.wbc program. Before this program can be executed on some target machine, processing will be necessary to convert the COBOL.wbc code into compiler-acceptable code.

The nature and extent of processing needed depends on the nature of the task. The Workbench is a flexible tool, suitable for use in a number of ways, ranging from the most simple to the most complex applications. In its simplest mode, the Workbench serves as a text editor and librarian. In its most complex mode, the Workbench provides the programmer with all of the tools necessary to produce COBOL code which is both portable and reusable.

Basically, the Program Processor in the Workbench can be used to **expand** calls to reusable modules into in-line code and to **translate** programs written in COBOL.wbc into programs which will execute correctly on a particular compiler/machine-environment.

3.3 DOCUMENTATION SUBSYSTEM

As contrasted to the various steps involved in programming, the three aspects of documentation --- preparation, control, and production --- do have rather clear divisions and can be discussed separately.

3.4 DOCUMENTATION PREPARATION

To encourage the production of complete and up-to-date documentation, the Workbench provides an environment well suited to the documentor. This environment includes tools, libraries, and a set of documentation format standards. A major portion of this environment is provided by the Text Editor which serves as the user's primary means for creating and modifying documentation.

3.4.1 Documentor's Environment

The Workbench provides an environment for the documentor which is designed to make the documentation task as easy as possible. This encourages the documentor to maintain complete documentation and to keep this documentation up-to-date. The documentor's environment consists of tools, libraries, and documentation format standards. The basic tool for documentation is the Text Editor. The libraries contain text which can be used during the creation of new documents and macros which can be used during the formatting stage to make the documentation task simpler (see section 3.6.2.1). The final aspect of the documentor's environment is a set of documentation format standards and aids to help produce documentation fitting the standard.

Figure 3.4.1-1 depicts the Documer'ation Preparation System. The documentor interacts with the Text Editor to create and

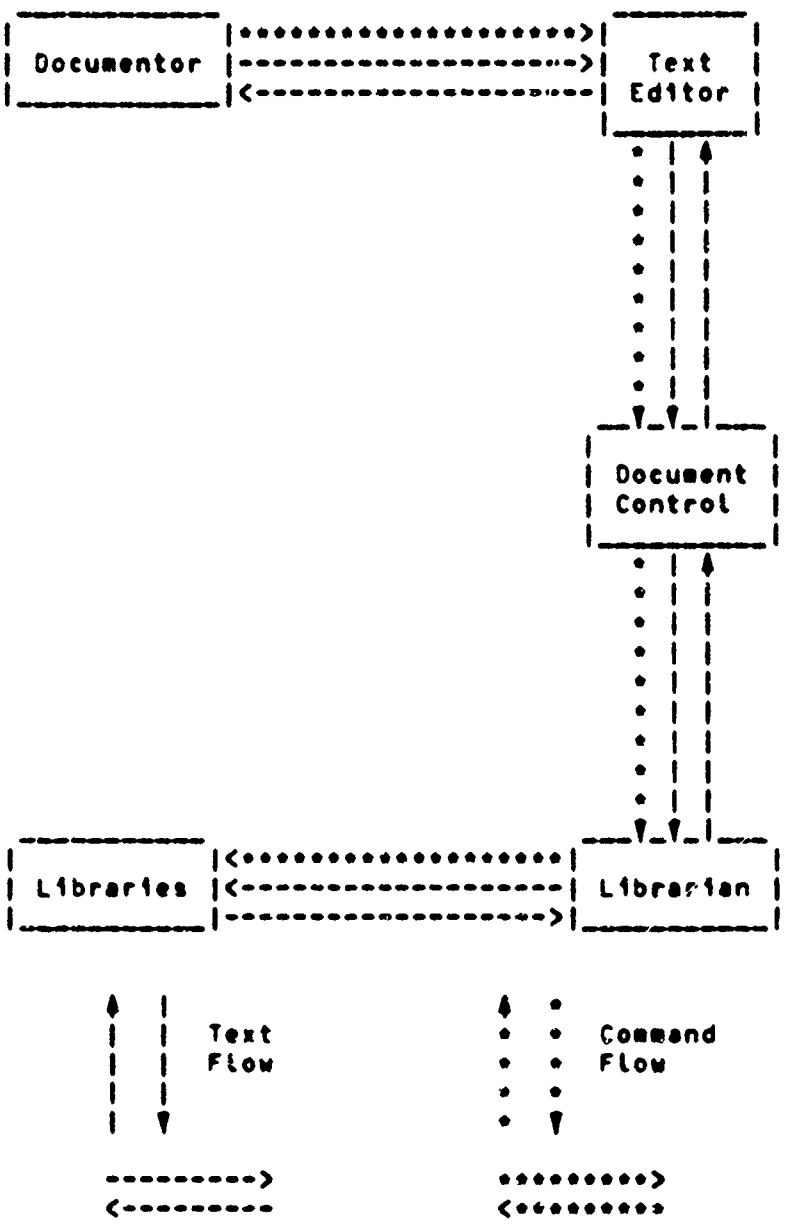
modify documents which are maintained in various libraries. To maintain order among the documents, a Document Control System (see section 3.5) stands as an interface between the Text Editor and the librarian.

3.4.1.1 Tools and Libraries

To encourage the practice of good documentation, the documentor needs a good set of tools. Chief among these tools is a Text Editor (see section 3.4.2) which allows the user to create and modify text in an on-line fashion. In addition to the Text Editor, other simpler tools are also useful. Examples of these are a file copier, a cut and paste tool, a pattern searching tool, a tool for comparing the similarity of two files and identifying their differences, etc. These tools are designed to provide the documentor the ability to perform his task with the expenditure of as little time and effort as possible. To meet this goal the tools must be simple to use, easy to learn, flexible, consistent with respect to each other, have default values assigned to options which reflect the common usage of the options, and finally they must naturally interface with each other so that several simple tools can be connected together to perform complex tasks.

FIGURE 3.4.1-1

PREPARATION OF SUPPORTING DOCUMENTATION



The documentor's environment also includes a series of libraries. The libraries contain among other things text which may be used by a documentor in the construction of a document. This text can be either other documentation or programs. The libraries also contain commonly used macros for the Formatter (see section 3.6.2.1) which help to produce standard formats of items within a document (e.g., tables, charts, lists, figures, etc.).

3.4.1.2 Documentation Format Standards

Another aspect of the documentor's environment is a set of documentation standards. These are designed to ensure that all documents of a particular type or belonging to a particular organization use the same set of formatting conventions. The definitions of the standards are maintained in libraries and are thus readily available for consultation during the documentation preparation process.

Since the ultimate goal of documentation preparation is a finished document, the documentor will have to consider the format of the finished product during the preparation stage of the documentation process. This means that the documentor will have to insert formatting commands within the text in order to produce the desired results when the text is processed by the Formatting Program. These imbedded formatting commands are also the key to achieving standard formats within the documentation. By constructing a set of formatting macros (see section 3.6.2.1), and storing them in

libraries, one can provide the necessary aids to help the documentor easily construct documents conforming to a set of format standards.

This technique was used to provide the section headings in this report. Five macros were available (h0, h1, h2, h3, h4) depending on the level of the section being written. The parameters required by each macro consisted of the section number and the title for the section. The macros provided for the consistent spacing around the headings and consistent use of underlining and **boldfacing**.

3.4.2 Text Editor

The Text Editor allows the documentor to create, update, and modify unformatted documents in an on-line manner. In addition, it allows him to copy text from one document while creating or editing another. Thus, the documentor can copy text from other documents and trim and modify the copied text to fit his needs.

The Text Editor possesses a powerful set of commands which obey a simple and consistent syntax. Complex pattern matching is also available and allows matching patterns within a line of text as well as locating lines contains the pattern. Printed responses by the Editor are terse resulting in a minimum of delay to the user.

The Text Editor also provides users possessing fast CRT terminals the ability to view a window of their text while editing. This is a powerful feature which not only aids the user in finding portions of text but also makes the construction and modification of text easier. In addition to being able to view a window of text, one has the ability to directly change text displayed in the window. To enhance this capability, the user is also given the ability to position the cursor to the left, to the right, up, down, or to a particular character on a line.

In case the user forgets one of the simple commands of the Editor, he can ask the Editor for help. Depending upon how the help is requested, the user will get either general information about the Editor or information about the use of a particular command. The help feature is designed to provide complete information for the beginner and specific information for the more experienced user of the Text Editor.

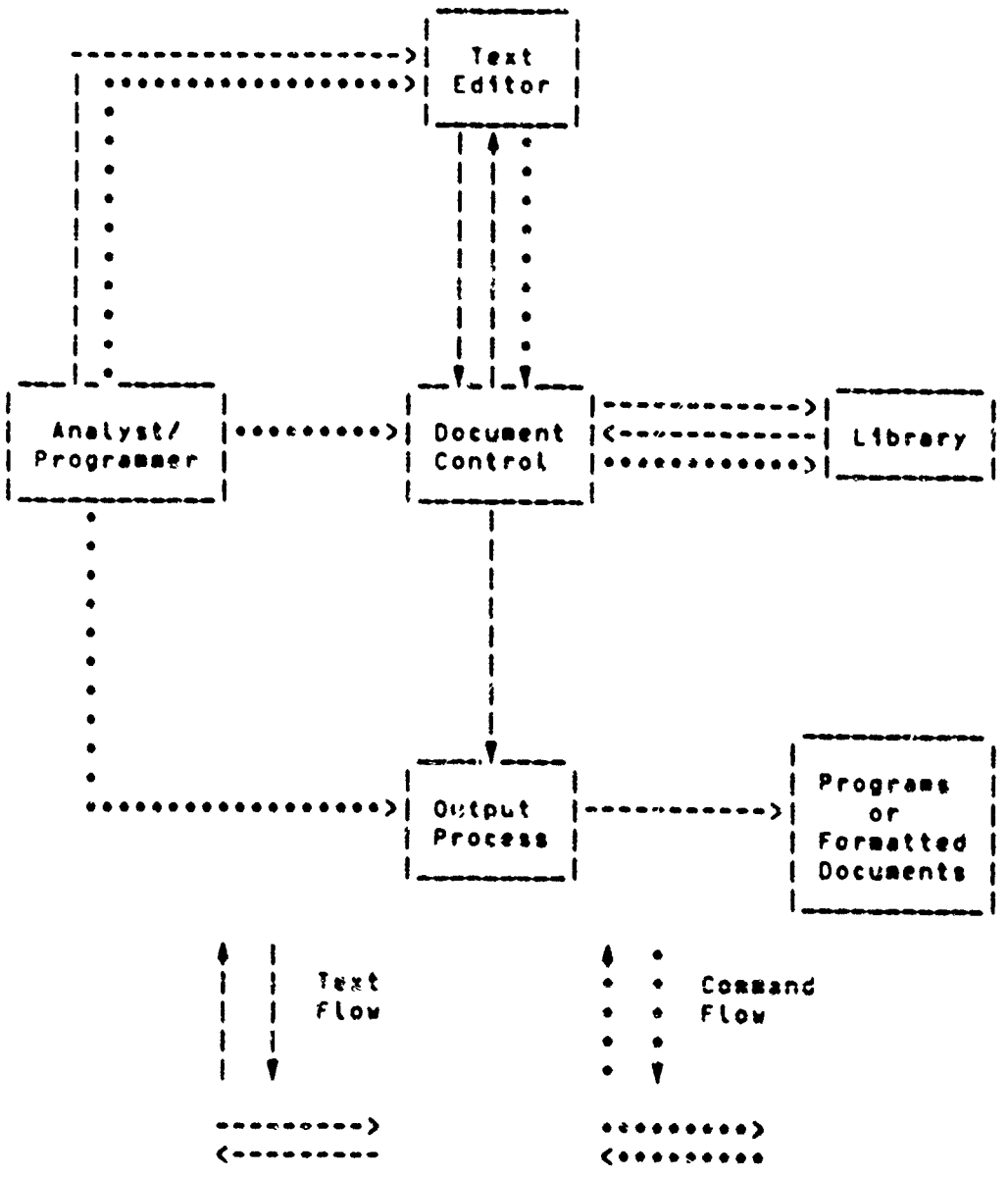
3.5 DOCUMENT CONTROL

The development of software systems is an evolutionary process. Thus, there will usually be multiple versions of a program or document in existence simultaneously. For example, a particular program may be in production while an updated release of the same program is in development. These constitute two different versions of the same program. In addition to these two versions of the program, there will also be two versions of the documentation supporting the programs. To help maintain order among multiple versions of the same program or document, the Workbench provides a Document Control System.

The most promising strategy for providing document control is that of maintaining a list of changes corresponding to a particular revision of a document. These changes will reflect the differences between a particular revision and its previous revisions. A more detailed explanation of this technique can be found in Appendix 3. The revisions of all documents will reside in a set of libraries (see section 3.5.1).

FIGURE 3.5-1

PREPARATION AND CONTROL OF PROGRAMS AND DOCUMENTATION



3.5.1 Libraries

An important part of document control is library management. As mentioned earlier, there are a number of libraries required to support all the activities of the Workbench, and many of these libraries contain similar component files or, at least, component files that could be applied to the same use and might even be identified by the same name. The two important aspects then of library management are

1) The management of the contents of the various libraries in a consistent manner, and
2) The management of the use of the various libraries in the support of the various activities provided by the Workbench.

The various libraries in the Workbench and their contents are depicted in Table 3.5.1-1. The uses of these libraries are listed in Table 3.5.1-2.

3.5.1.1 General Library

The General Library, GENLIB, contains the compiler-independent text of a collection of general-purpose reusable modules which will be of use to all programmers. These modules might include the following:

- data input editing
- file structure definition
- report generation

TABLE 3.5.1-1

WORKBENCH LIBRARIES AND THEIR CONTENTS

		LIBRARIES						
		General	Application	Project	Release	Personal	Test	C.U. Macro
	Definition of Compiler-Unique Functions					D		T
	Reusable Modules	T	T	T		D		
CONTENTS	Test Harnesses			T		D	T	
	Programming Skeletons		T	T		D		
	Baseline Programs			T	T	D		
	Program RELEASES & VERSIONS				T			
	Notational Shortands					D/T		
	Personal Programs					D		
	Breadboard Programs			"T"		D		

T: Tested D: In Development

TABLE 3.5.1-2

WORKBENCH LIBRARIES AND THEIR USES

Library	Uses
General	Supports work on a large number of projects and application areas.
Application	Primarily to support work in a specific application area, e.g., payroll, inventory, etc.
Project	The development files for a specific project.
Release	The documentation actually released to users.
Personal	Anything that is not ready for public use, e.g., has not been tested or is not intended for public use.
Test	Test harnesses.
Compiler-Unique Macro	Compiler unique macros which are used to replace macro calls of COBOL.wbc in order to produce code for a particular compiler (COBOL.k).

GENLIB contains only tested code. New module definitions may be added to GENLIB by installation personnel only after complete testing.

The modules are written in COBOL.wbc and contain no compiler-dependent references. These modules may contain calls to macros as well as calls to other modules.

3.5.1.2 Application Libraries

The Application Libraries are special purpose module libraries containing modules developed for special applications. Examples might include a package of payroll modules or a package of modules developed for inventory control. Every application will possess its own library, APPLIB-name (e.g. APPLIB-PAYROLL or APPLIB-INVEN). Application libraries contain only tested code.

3.5.1.2.1 Reusable Modules

Modules are used to isolate any frequently performed function or task in a single piece of code, to eliminate redundant coding of such tasks, and to simplify maintenance. The modules included in the Application Libraries are developed specifically for a particular application but may be used by other projects as the occasion requires. They are written in COBOL.wbc and must be both tested and complete.

3.5.1.2.2 Programming Skeletons

Tested Programming Skeletons which are useful to a particular application and are written in COBOL.wbc are kept in the Application Library for the particular application. These skeletons may refer to reusable modules stored in other libraries.

3.5.1.3 Project Libraries

Project Libraries contain reusable test harnesses, programming skeletons, baseline programs, and breadboard programs, all of which are developed specifically for a particular programming project. The libraries are named "PROJLIB-proj" where "proj" is the name of the particular project which is served by the library.

3.5.1.3.1 Reusable Modules

The reusable modules contained in the Project Libraries serve the same purposes as the reusable modules kept in the Application Libraries. They are developed specifically for a particular project under development and should not be used by any other project since the changing needs of the development team may modify module calling usage or behavior or eliminate some modules altogether. The modules included in the Project Library are written in COBOL.wbc and are tested.

3.5.1.3.2 Test Harnesses

Test harnesses to be used in the testing of the modules of a particular project are housed in that project's Project Library. These modules are written in COBOL.wbc and have been tested.

3.5.1.3.3 Programming Skeletons

Programming Skeletons that are applicable only to a particular project are kept in the project's Project Library. They are written in COBOL.wbc and have been tested. They may call modules which are housed in other libraries.

3.5.1.3.4 Baseline Program

The baseline program as contained in the Project Library is a tested program. It is written in COBOL.wbc and may contain calls to modules developed especially for the project and maintained on the project library, as well as calls to tested modules contained in the other libraries described in this section.

3.5.1.3.5 Breadboard Programs

Breadboard programs that are developed for a particular project are kept in the Project Library for the project. These programs are written in COBOL.wbc. They have not been rigorously tested, but are still believed to be correct.

3.5.1.4 Release Library

The Release Library contains those programs which have been released for general use. This includes tested Baseline Programs written in COBOL.wbc and the compiler-specific RELEASES and VERSIONS of the Baseline Programs which are written in the particular dialect of COBOL (COBOL.i) provided by the target machine on which the program is to run.

3.5.1.5 Personal Libraries

The Personal Libraries are used by the programmer to store any material he is developing. They contain code in COBOL.wbc or COBOL.k and will usually be untested. After a program or module is completed and has been satisfactorily tested, it will normally be moved to one of the other libraries (e.g., General Library, Application Libraries, Project Libraries, etc.).

The individual programmer may choose to make use of the Workbench facilities to permit notational shorthand for things like long variable names or frequently occurring statements. These notational shorthands are stored in the Personal Libraries. These individual abbreviations should be removed from the completed code by expanding such module calls into the corresponding code.

3.5.1.6 Test Library

The Test Library, TESTLIB, consists of test harnesses into which the programmer may insert a newly designed module for test purposes, routines for the generation of random data meeting specified format restrictions, and general-purpose output routines. Other TESTLIB functions might provide the capability to collect performance data. These modules are provided in order to assist the programmer in the development of new modules.

TESTLIB contains only tested code. New module definitions or routines may be added to TESTLIB by installation personnel only after complete testing.

3.5.1.7 Compiler-Unique Macro Libraries

The workbench contains a series of libraries, MACLIB.n, for each target compiler n. These contain the text for the compiler-dependent macro expansions for the particular compiler n.

The macro library will serve as one of the inputs to the macro expansion processor which will translate the compiler-independent program text into COBOL code for compilation by a particular compiler. Macros may call other macros, but may not call modules or contain COBOL.wbc statements not included in the compiler for the given target machine.

The Macro Library contains the text which the Macro Processor uses to expand macro calls into compiler-unique notation. It is necessary that strict naming conventions be used, as each MACLIB.i must contain definitions for all of the same macro calls. Due to the nature of this process, it is likely that the majority of macro definitions stored in any given MACLIB.k will be empty.

The necessary MACLIBs will be provided along with the Workbench and should not be modified by installation personnel. MACLIBs contain only tested code.

3.5.2 Control of Source Code

Within the Workbench environment, a single module may be maintained in a number of different versions. The source code comprising the module may be maintained in multiple copies, each including (possibly minor) changes from the original. This approach is not desirable for three major reasons.

1) Storage is used inefficiently to maintain multiple copies of (essentially) duplicate information.
2) A programmer making a modification may not be modifying the latest version.
3) It is very difficult to remove a modification other than the latest.

A better approach is to store the original module source

code only once and to record all modifications as insertions, deletions, or replacements to the original file. To simplify operation by avoiding the need for special creation facilities, the original Module Source File may be regarded as empty, with the first update consisting of a series of insertions. The Workbench Source Control Facility is suggested to provide these features.

3.5.2.1 Structure

The Source Control Facility maintains a project's source code in a number of data files, hierarchically organized. Each module is maintained as an individual file which may be modified individually. Larger program fragments, while stored internally as a collection of module source files, may be treated logically as a single file and modified as a unit. This process of combination of modular subunits to produce larger units is carried upwards throughout the project source library. The PRIMOS file system (see Appendix B) facilitates the use of this kind of hierarchical structure.

3.5.2.2 Features

Features of the Source Control Facility include

1) Creation of Module Source File.
2) Updating of Module Source File by inserting, deleting, and restoring source lines according to information stored in the file or

provided as part of the correction set.

3) Ability to completely and permanently remove correction sets from the Module Source File.

4) Generation of a module version corresponding to the modifications desired by the programmer and expressed in the correction set.

5) Generation of a new, updated Module Source File.

6) Comprehensive listings noting any changes made to the Module Source File, as well as the status of all source lines contained in the file (inserted, deleted, modified).

7) The ability to group modules together into larger units and to perform the above functions on these larger units.

In order to implement these features, it is necessary to assign each source line a unique identifier.

3.5.2.3 Status

The Source Control Facility is not yet fully implemented; however, the facility parallels a number of commercially available tools, such as CDC's programs UPDATE and MODIFY.

3.5.3 User Documents

Among the documents maintained by the Document Control System are user documents. Examples of this group of documentation are user manuals, reference manuals, instruc-

tions for data preparation and entry, reference material to support feedback from users, etc.

3.5.3.1 Original Document

An original document is identified as version one of release one of the document. Since it is the first revision of the document, the list of changes corresponding to it will consist only of insertions. Treating the original document in this manner will make it consistent with the other revisions.

3.5.3.2 Released Modifications

Modifications to a document are obtained by creating new VERSIONS for a particular release or creating a new RELEASE (see Figure 1.3-1). Changes to the baseline program result in the creation of a new RELEASE while changes to the compiler-unique VERSIONS of the baseline program result in new VERSIONS. The Document Control System provides the ability to mark revisions as to whether or not they are available for release. In addition, the user can upon request obtain a list of those revisions approved for release.

3.5.3.3 Latest Version of a Release

The Document Control System allows one to obtain any VERSION of any RELEASE by simply specifying the specific version number and release number. Users can also obtain the latest VERSION of a RELEASE by simply specifying the desired release or the latest RELEASE by specifying the name of the document.

3.5.3.4 Maintaining Comments on Documents

To obtain a document which is correct and satisfies the needs of the user community, a means of obtaining feedback from the user community must be available. This service is provided with the help of the Document Control System. User comments, which may be either notes on possible errors or suggestions for changes to a document, are dated, identified by the user making the comment, and linked to the document to which the comment refers. The one responsible for the document can obtain the comments from the Document Control System, act upon them, and reply to the originator of the comment.

3.5.3.5 Working Documents

Documents in the working state can be identified as such by the one responsible for the document. The Document Control System will then be able to detect that the document is a working document and will not give that document to a user requesting a released document.

3.5.4 Management Documents

Management documents are maintained by the Document Control System in the same manner as user documents. Examples of this group of documents are personnel/team organization and task breakdown, project budgets, cost and time performance, project status, test plans, etc.

3.5.4.1 Original Requirements

Among the documents maintained by Document Control are those specifying the requirements for the software systems. The original requirements for a system are found in version one of release one of the document containing the requirements.

3.5.4.2 Current Requirements

As the requirements for a system change, the document containing these requirements will evolve new versions and possibly new releases. To obtain the latest requirements, one need only specify the name of the document containing the requirements and the Document Control System will obtain the correct release and version. It should be noted that the treatment of these documents is the same as the user documents mentioned above.

3.5.4.3 Change Requests

Change requests received from either users or the proponent organization for whom the system was designed are reviewed by the project management staff and then entered into the documentation data base so that they will be available for consideration when the next major release of the program is prepared.

3.5.4.4 Approved Modifications

Based upon change requests from users or based upon changes in the functions that must be provided by the data processing system, the proponent or the proponent organization or "owner" of the system in consultation with the developing organization will review and approve modifications to be made to the system the next time a major release is prepared. These modifications will be stored in the documentation data base appropriately linked to those portions to which they apply so that they are readily available and apparent to the project team when the new release is prepared.

3.5.4.5 Maintaining Comments on Documents

The technique for maintaining comments on management documents is identical to that for user documents (see section 3.5.3.4).

3.5.4.6 Working Documents

The techniques used to handle working management documents is identical to that for user documents (see section 3.5.3).

3.5.4.7 Project Status and Control

Since the Document Control System maintains all revisions of all documents and all comments pertaining to those revisions, one should be able to easily obtain the status of a project. Since the Document Control System possesses the necessary information to report on the status of a project, it is responsible for providing this service. The user need only specify the documents to be analyzed, and the Document Control System will provide information indicating the revisions available and the change requests which have been acted upon.

3.6 DOCUMENT PRODUCTION

3.6.1 Program Production

Program production is a function provided by the COBOL compiler and programming system applicable to the target environment. The Workbench tools provide the capability to produce a compiler-unique VERSION of a baseline program RELEASE; however, the only dialect of COBOL that can be translated into an object program by the Workbench is COBOL.wbc.

3.6.2 Documentation Production

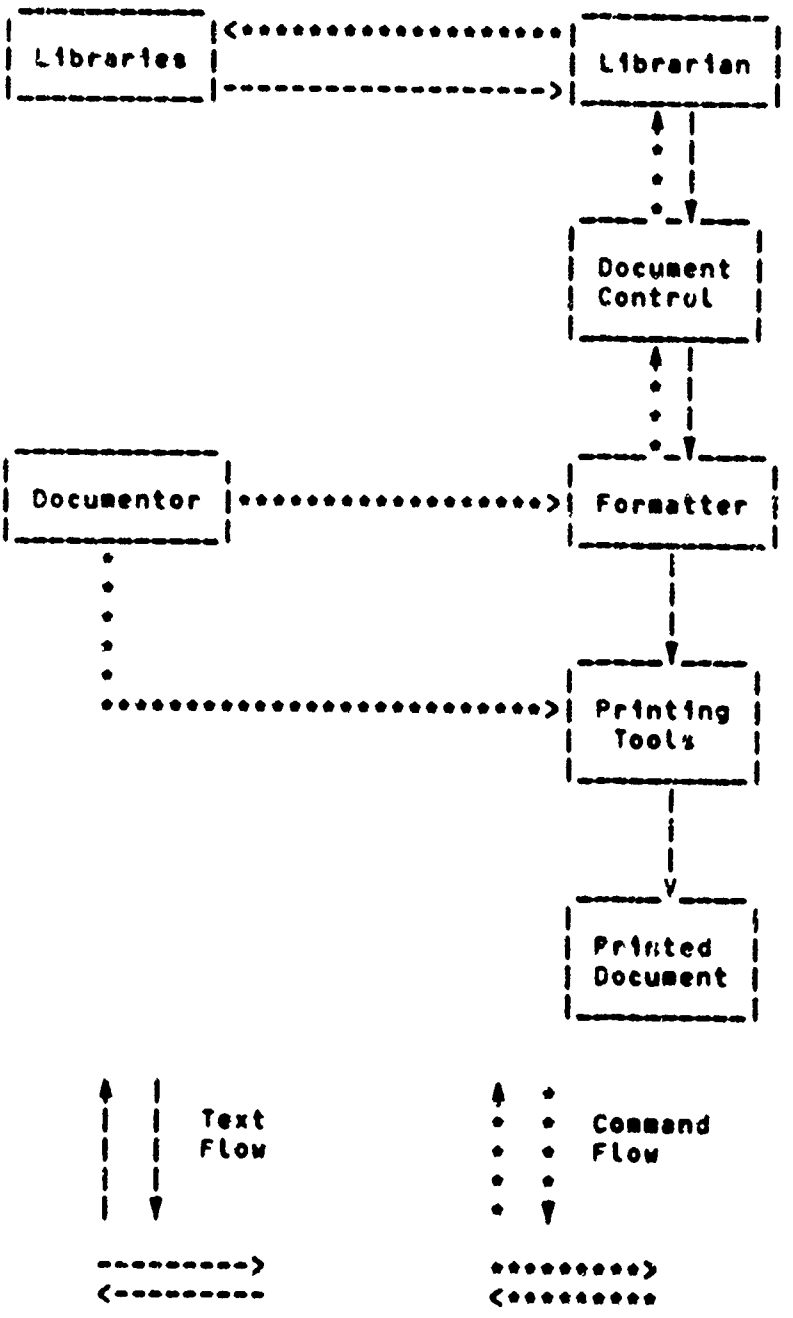
To produce printed documentation, the documentor must direct the Text Formatter and the desired printing tools. The text to be formatted is obtained via the chain from Document Control to the librarian to the libraries and passed along through the Formatter and printing tools to the device which is to print the documentation.

3.6.2.1 Text Formatter

Documents in their finished forms are obtained by running them through the Text Formatter. This formats the text according to commands intermingled with the text and default settings for those options not specified by commands. The Text Formatter allows the user to make either the left or

FIGURE 3.6.2-1

DOCUMENTATION PRODUCTION



right margin even, control the size of the left and right margins as well as the top and bottom margins, and specify headers or footers to be printed on each page along with numbering of the page. The user can control the line spacing and specify the centering of specific lines. In addition, specific words or phrases can be highlighted by **boldfacing** or <u>underlining</u>.

The ability to define and use macros is also available in the Text Formatter. This includes the ability to pass parameters when calling a macro. This capability gives one the ability to provide a set of standards to a group of documentors. This is done by creating macros to perform the operations which are to be made standard and providing these macros to the documentors through specific libraries.

Thus, the Text Formatter provides neatly formatted output from very unstructured input. By delaying formatting to a later stage, one is able to enjoy a considerable amount of flexibility during the entering and modification stages of document preparation. This allows a document to grow and change with a minimum of effort.

3.6.2.2 Printing Tools

In addition to the text formatter, other tools directed specifically at the production of printed material are needed in order to produce documentation. Among these are tools to select the range of pages and number of copies of

the document to be printed. The documents may be printed on a variety of devices including line printers and hard copy terminals. Thus, there must be tools to take the output from the text formatter and convert it to a form appropriate for the device on which it is to be printed. For example, documents to be printed on a line printer need to convert all back spaces into separate line outputs with a forms control that does not advance the paper. These tools must interface with the other tools available on the Workbench.

SECTION 4

UTILIZATION OF THE COBOL PROGRAMMER'S WORKBENCH

4.1 GENERAL

As is perhaps evident from Section 3, the Workbench consists of only a few "general-purpose" tools that are utilized in "specialized" ways to provide the capabilities outlined in Section 2. This flexibility is provided by the very powerful and versatile Command Interpreter that controls the operation of the Georgia Tech Software Tools Subsystem.

The paragraphs below are not meant to be a user manual for the Workbench. Rather, they provide some insight into how the system can be utilized to provide the capabilities desired.

4.2 PROGRAM PREPARATION

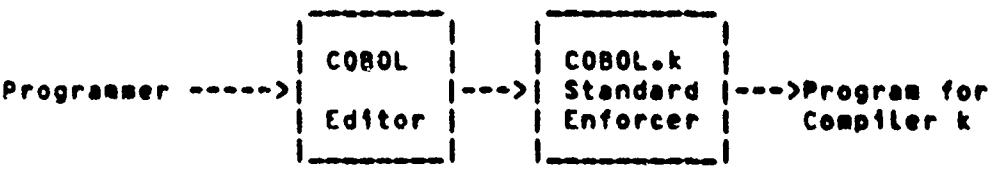
The variety of options that are possible in the use of the Program Preparation Subsystem are almost limitless. The paragraphs below provide some examples of typical or representative uses of the capabilities of the Workbench.

4.2.1 Producing a COBOL Module or Program

In the simplest situation, the Workbench is used to produce code for a particular compiler k. (Note that it is not necessary that k be a COBOL compiler.)

FIGURE 4.2.1-1

PRODUCTION OF A PROGRAM IN COBOL.k



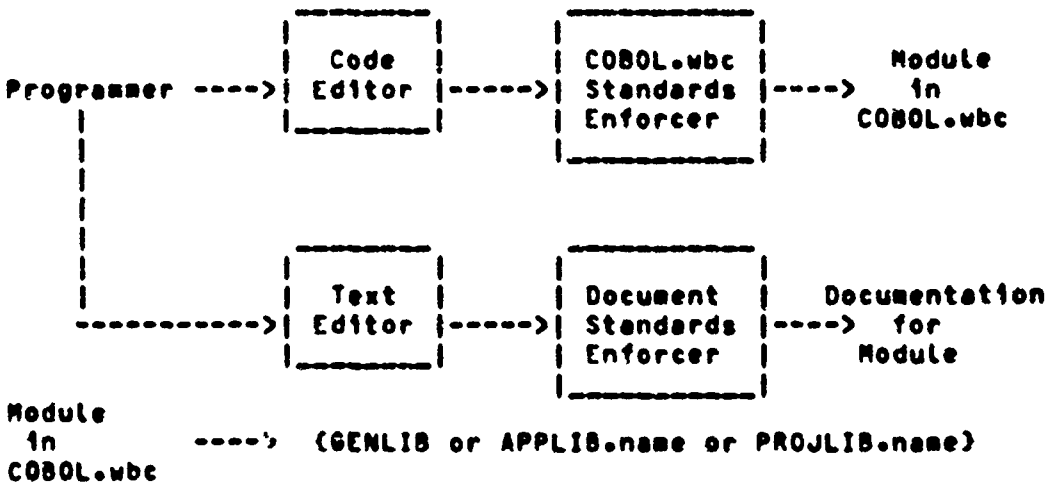
4.2.2 Preparing a Reusable Module

A reusable module is prepared very similarly to any other code module. The critical aspect of a reusable module is its interface to other modules. The production of a reusable module differs from the production of a complete system primarily in that the interface behaviour must be clearly and exactly defined and must meet the standards

established for reusable modules. Use of the standards enforcer is required for consistency between modules developed by different programmers. New modules should be transferred from the programmer's personal library to the project library only after passing the standards enforcer and undergoing thorough testing. Modules developed by the project should be added to the application library or other general-access libraries only after complete test. Once a module has been transferred to a permanent library, any further modification must maintain interface behaviour.

FIGURE 4.2.2-1

PRODUCTION OF REUSABLE MODULES AND THEIR DOCUMENTATION

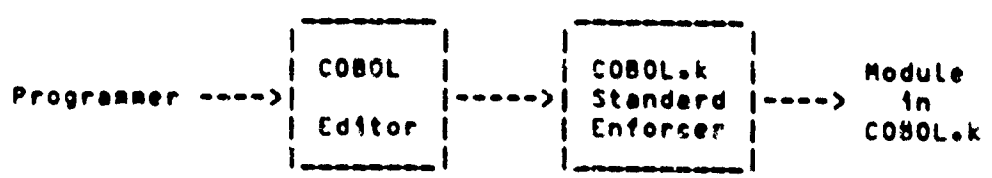


Documentation should be produced in parallel, and should include information about the status of testing and a record of any subsequent modifications performed or requested.

The Workbench may also be used to produce reusable code for a particular compiler k.

FIGURE 4.2.2-2

PRODUCTION OF REUSABLE MODULES IN COBOL.k



4.2.3 Producing a Baseline Program

Perhaps the most important factor in the preparation of a baseline program is that the resultant COBOL.wbc program must be transportable to all of the target operating environments of interest. The exact sequence of steps to be followed and the depth of nesting of calls to code modules and/or reusable modules is really a matter of personal programming style. The Workbench will support almost any technique possible.

The basic form of the baseline program that is retained for maintenance can contain references to reusable modules that are available for public use (defined in the project, application, or general libraries).

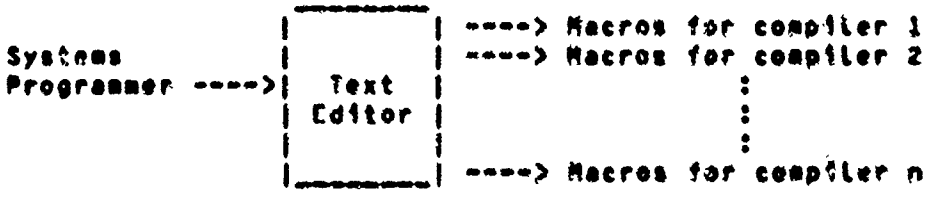
4.2.4 Preparing the Compiler-Unique Macro Libraries

There must be a complete macro library prepared for each target operating environment whether they are distinguished by hardware, software, or other differences. As can be seen from the example in Appendix 2, macro libraries can become quite lengthy since they each must contain a macro definition for every environmentally-unique feature even if that uniqueness applies to only one environment. As can be seen from the example, macro statements can be utilized to accomplish several actions

1) Selection of one (or more) of several statements in the original program based on the name of the target environment.
2) Expansion of a statement in the original program which may pass parameters to the macro.
3) Elimination of a statement in the original program.
4) Do nothing (copy the original statement).

FIGURE 4.2.4-1

PREPARATION OF COMPILER-UNIQUE MACROS



It can be seen from Figure 4.2.4-1 that it is anticipated that a systems programmer will be required to prepare compiler-unique macros.

4.2.5 Producing a "Breadboard" Program

A Breadboard Program is prepared in a manner similar to any other program or module with two exceptions:

1) The emphasis is on preparing a prototype program proviging a close approximation to system functionality desired, not on meeting all user requirements or on performance and efficiency.
2) The breadboard program is not meant to be portable to a number of operating environments.

As a result of the two factors cited above, the developer of a breadboard program will make much heavier use of reusable modules even if they do not provide exactly the detailed functionality desired and even if they are compiler-unique. The steps in the preparation and processing of a breadboard program are basically the same as those required for a baseline program.

4.2.6 Maintaining the Baseline Program

As noted above, the baseline program is prepared utilizing only COBOL.wbc which contains statements that act as macro calls to handle compiler-unique features and reusable module "statements" which are also expanded using a macro call procedure. It is important that the baseline program retain these features in order to maintain its integrity. Any changes or modifications to the baseline program must follow these same standards.

Maintenance of the baseline program is usually performed for one of two purposes:

1) To modify or increase the functionality of the system as directed by the system proponents, or
2) To correct errors in the program resulting from logical errors in the design of the baseline program or "mechanical" errors caused by the expansion and/or transportation phases of program processing.

The first situation is a fairly standard situation and merely requires reworking the code in the baseline program; however, the latter circumstance, which will certainly occur quite frequently in a dynamic world of changing target environments, modifications of target system software, etc., will necessitate changes in the definitions of the reusable module and compiler-unique macro libraries.

4.2.7 Maintaining RELEASES and VERSIONS

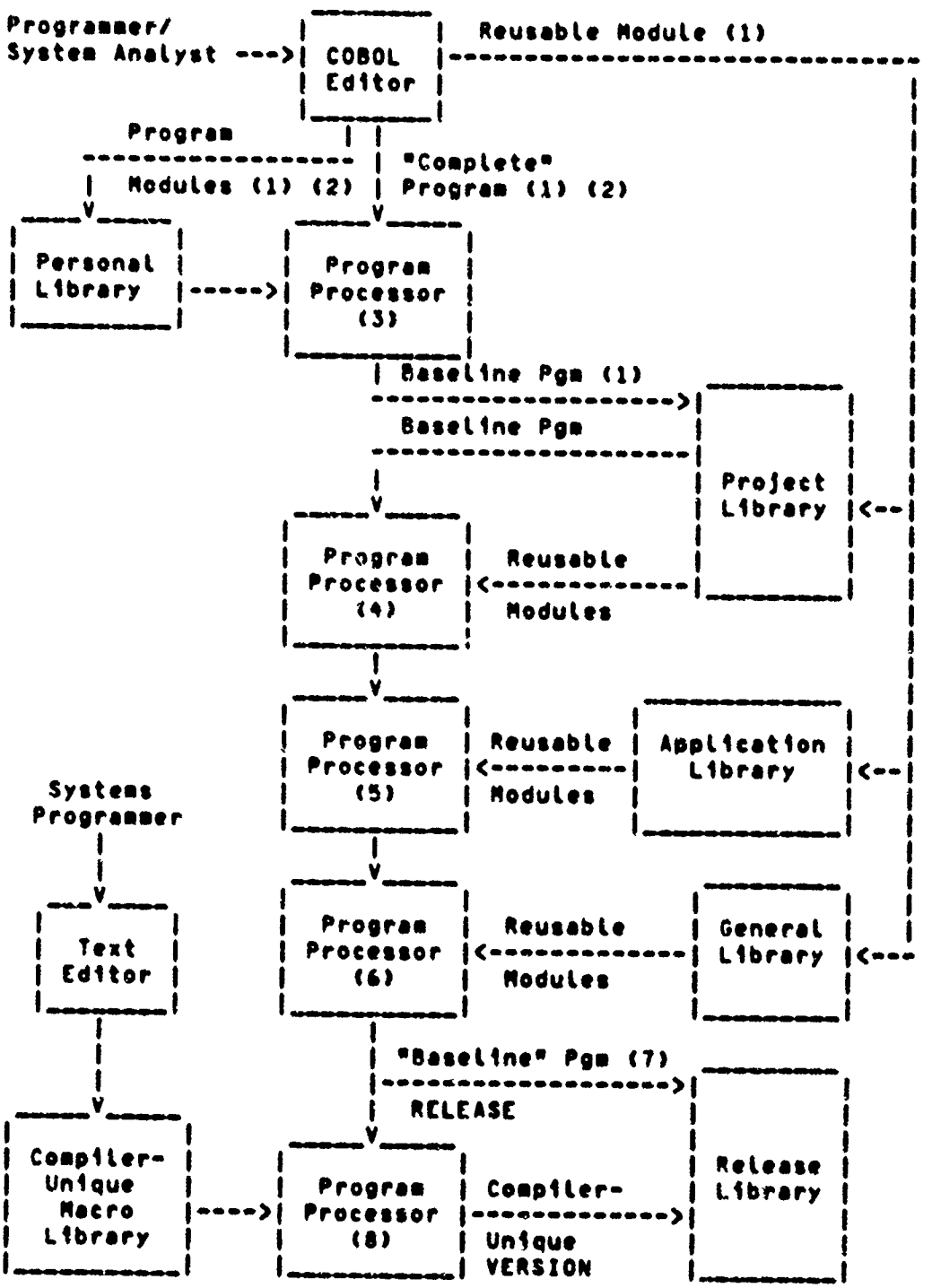
There is actually no need to maintain a program RELEASE other than the activities described above for the maintenance of the Baseline Program. However, there will be continual maintenance on compiler-unique VERSIONS of the program to provide immediate correction of errors in the design of the program as well as in its automatic processing by the Workbench. Such changes will be made directly on the distribution VERSION of the program in COBOL.k and will produce MODIFICATIONS of VERSIONS.

4.2.8 Summary

The best summary possible is a figure relating the various activities described above. This information is provided in Figure 4.2.9-1.

FIGURE 4.2.8-1

UTILIZATION OF THE COBOL PROGRAMMER'S WORKBENCH
(Notes on following pages)



Definitions of Various Code Modules

Reusable Modules

Written in COBOL.wbc (or, possibly in special instances, COBOL.k). May contain references to other reusable modules and/or compiler unique features in COBOL.wbc.

Compiler-Unique Macros

Macro definitions that provide the information required to translate references to compiler unique features into fully expanded compiler unique code.

Compiler-Unique Program VERSION

Contains no references to reusable modules or compiler unique features. Has been processed so that it is a source program written entirely in COBOL.k (ready for processing by compiler "k" on target machine K).

Baseline Program

Written in COBOL.wbc. Contains references to compiler-unique features and reusable modules (written in COBOL.wbc).

Breadboard Program

Written in COBOL.wbc (or, in special instances, COBOL.k). Makes heavy use of reusable modules written in COBOL.wbc (or COBOL.k). If no compiler exists for

COBOL.wbc, the breadboard program can be processed to produce a ".k" version for execution of the prototype system.

Notes for Figure 4.2.8-1

1) May contain references to Workbench Reusable Modules and/or compiler-unique features.
2) May also contain references to Personal Reusable Modules.
3) Processes and removes references to Personal Reusable Modules.
4) Processes and removes references to Project Reusable Modules.
5) Processes and removes references to Application Reusable Modules.
6) Processes and removes references to General Reusable Modules.
7) A copy of the Baseline Program will not contain references to any Reusable Modules. All such calls have been expanded fully into COBOL.wbc. It may still contain references to compiler-unique features.
8) Processes and removes references to compiler-unique features.

4.3 DOCUMENTATION PREPARATION

4.3.1 Prepare Original Documents

The preparation of original documents is accomplished by means of the Text Editor. Text can be either entered directly or copied from other documents which already exist within the system. Possible sources for copies include both program code and text documents. The documentor will normally only wish to copy parts of a program or a text document. This is accomplished by first copying the entire document and then deleting those portions which are not desired. Since these operations are all carried out by the Text Editor, one can easily trim and modify the copied text to suit the needs of the document being created.

4.3.2 Modifying Documents

The modification of documentation is also realized through the Text Editor. The operations available to the documentor include insertion, deletion, and modification of text. In addition, the documentor can copy, trim, and modify text from any desired source.

4.3.3 Annotating Documents

It is extremely likely that after a program or text document has been released for general use that an error will be discovered or a place for improvement will be noticed. If one wishes to point out an error or make a suggestion for improvement, one need only access the annotating feature of the Workbench, specify the document of interest, including the release and version numbers, and specify the comment concerning the document. The Workbench will append to the document the name of the user generating the comment and will then link this collection of information to the proper document. Those responsible for the document will be able to obtain the list of comments pertaining to any document, act upon these comments, and respond to those identified as responsible for the comment.

The annotating capability need not only be used on programs and documentation released for general use. It can also be used by the development groups to aid in coordinating all parts of a large system.

4.3.4 Producing Specific Versions

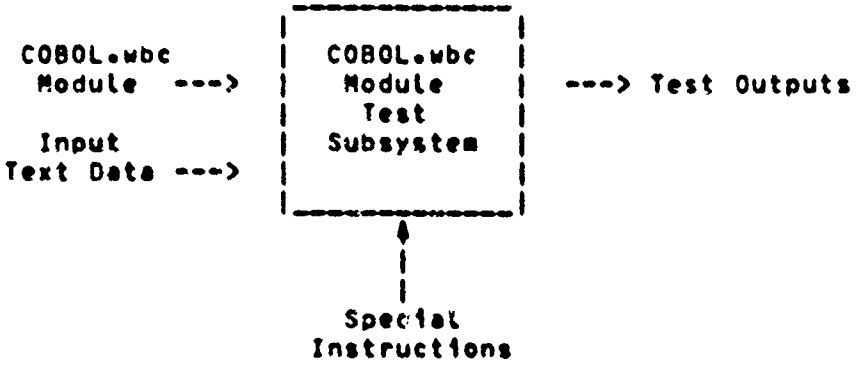
Specific versions of documentation can be maintained and produced through the document control facilities of the Workbench. Documentation is organized according to a release number and a version number within a release. To create a new version of a release, one must obtain the

latest version of the document from Document Control. Any changes, additions, and deletions can then be made to this document by means of the Text Editor. When all of the modifications have been made, Document Control is called upon to build a new version. If a new release is to be constructed, one must obtain the latest version of the latest release and proceed as described above. Finally, if the first version of the first release of a document is to be produced, one need only build the document with the Text Editor and inform Document Control that the first version of the first release is to be built. Further details concerning Document Control can be found in Appendix 3.

4.4 PROGRAM TESTING

FIGURE 4.4-1

TESTING A COBOL.wbc MODULE



4.4.1 Workbench Test Operation

4.4.1.1 Input to the Testing Process

1) COBOL.wbc Source Module

   The testing process will convert the COBOL.wbc module into successive COBOL.i modules and test each member of the resulting family of modules.

2) PACLIB.i, where i goes from 1 to n

necessary to permit conversion of the COBOL.woc module into the corresponding COBOL.i modules.

3) Special instructions (Modules provided on TESTLIB)

- To establish the test environment.
- To control the test sequencing, etc.
- To create test data meeting data declaration requirements.
- To create test files meeting the file format specifications of a given machine.

4.4.1.2 Output from the Testing Process

1) Numeric values from test runs.

Among the special instructions included on TESTLIB will be routines to print both the random data generated as input and the results of the test runs.

2) Diagnostic information.

Among the special instructions included on

TESTLIB will be routines to collect performance data and to trace execution of modules under test.

4.4.2 Specific Test Capabilities

1) Accept the COBOL.wbc Source Module.

It is strongly recommended that only those modules which have passed Standards Enforcer.wbc be accepted as input. This may be achieved automatically, by calling the standards enforcer as the first step in the testing process, or manually, by requesting project programmers to submit only approved modules for testing.

2) Using test harnesses (modules) provided on TESTLIB, produce an executable COBOL.wbc program.

What we mean by a test harness is a main program, expressed in the form of a module, which serves as a driver to execute an otherwise incomplete COBOL.wbc module. Specific capabilities and requirements of test harnesses will be discussed in a later section.

3) Translate the resulting COBOL.wbc program

into COBOL.k.

At the moment we make no attempt to execute COBOL.wbc. While it may in fact be possible to do so, and thus to test COBOL.wbc programs directly without conversion into specific dialects, this approach leads us into a very complex problem area which is beyond the scope of this project, namely the problem of verification. If we are to draw any useful conclusions from our testing of a COBOL.wbc program we must be able to verify that a particular COBOL.k program is functionally identical to the original COBOL.wbc program. All that we are able to show without dispute is that a particular COBOL.k program was derived (i.e., as a macro expansion) from the original COBOL.wbc program. This is not sufficient to demonstrate correctness.

4) Generate test data.

What we want here are a set of routines to generate random data meeting a given data specification. These routines

- could be implemented in any language (need not be written in COBOL)

- could be implemented either as modules (if written in COBOL.ubc) or as separate routines (in which case there would be at least one routine for each COBOL.i)
- would be stored in TESTLIB
- accept as input COBOL.ubc data declaration divisions
- produce as output random data meeting the input data declaration and the file structure requirements of the given target machine

5) Exercise the program logic.

In other words, execute the COBOL.i program developed in steps 2 and 3 above using the random data from step 4 as input. A trace routine to ensure that every logic branch is tried might be useful here. Another useful approach would be to make use of the idea of program "mutants" to ensure that the test cases generated by step 4 represent an adequate test of the program logic.

6) Derive statistics on performance of the module being tested.

TESTLIB includes COBOL.wbc modules designed for the collection of performance data. The appropriate modules are selected and called (macro calls) by the test harness. Performance data which might be collected includes

- CPU time used
- main memory used
- paging
- number of iterations

7) Repeat steps 3 - 6 above for each COBOL.i as i goes from 1 to n.

SECTION 5

IMPLEMENTATION OF THE DEMONSTRATION WORKBENCH

An important goal of this research project was empirical studies of the concept of the COBOL Programmer's Workbench and the implementation issues involved. There have been a large number of proposals to achieve program portability; however, very little of this work has been reduced to practice. It was felt essential that a practical demonstration of at least the general concepts be included in this research project. Although it was not possible to construct a complete version of the workbench, it is felt that the Demonstration Workbench that was produced clearly demonstrates the feasibility of the concepts and ideas presented as well as highlights the problems that must be addressed in the development of a complete workbench system.

5.1 GOALS OF DEMONSTRATION WORKBENCH

There were four basic objectives for the development of the demonstration COBOL Programmer's Workbench. These four were

1) Feasibility demonstration of the operation concepts proposed.
2) Identification of the major problems that would be encountered in the implementation of a complete Workbench.
3) Investigation of the desirable features of a

user environment for the Workbench.

4) Estimation of the value of the concepts of reusable program modules and maintenance of a single baseline program for multiple target machines.

What was attempted was a balance approach toward making some progress in reaching all of these goals. It was recognized that it would not be possible to implement a complete version of the COBOL Programmer's Workbench with the time and resources available nor did the research team feel that it had sufficient information to allow it to embark on a complete development project. It was felt that the effort could be much more usefully applied to laying solid ground work on which a further project aimed at the complete Workbench implementation could be based.

5.2 SYSTEM ENVIRONMENT AND TOOLS AVAILABLE

The research team was fortunate in having available for this project an extremely powerful hardware and software environment on which to implement the Demonstration Workbench. The computing system, a Prime 400, is one of the "large" or "mega" minicomputers which rivals mainframe systems as large as the IBM System 370/158 and 168 for computational power. A further advantage of this specific computer system was the extremely large virtual address space available. (However, there were limitations on the size of object programs that could be generated by the versions of the COBOL compiler that were available during the early phases of the project.) The hardware, however, provides only a portion of the environment in which the Demonstration Workbench was established.

The research systems environment available was also greatly enhanced by having available on the Prime 400 system the Georgia Tech Software Tools Subsystem [Kernighan 1976]. The Georgia Tech Software Tools Subsystem, which is described briefly below and in more detail in the appendix, provides not only an extremely hospitable environment for the users but also an extremely powerful environment for the construction of command files that support the use of the subsystem components in the implementation of the Demonstration Workbench. The availability of the Georgia Tech Software Tools Subsystem was invaluable in expediting the development of

the Demonstration Workbench.

5.2.1 Prime 400 Computer System

The Prime 400 computer system is a fairly recently introduced system (1976) that falls into the category of the large or mega-minicomputer size. The Prime 400 shows strong influences of the Multics System in both its hardware and software. It is an extremely powerful minicomputer providing an excellent environment for software development projects such as this.

5.2.1.1 Hardware

The Prime 400 is of the large mini or mega-mini class of computers. It possesses a powerful processor, large main memory, large virtual memory, and can handle up to 63 users. Shared code facilities and a multi-ring protection mechanism are also provided. The Prime 400 uses microprogrammed logic and possesses a writable control store.

5.2.1.1.1 Memory

The Prime 400 provides both a large real and virtual memory. The maximum size of physical memory is 8 million bytes. It is incrementally expandable by 256K byte boards. In addition to main memory, a high speed, bipolar cache memory of 2K bytes is provided.

Virtual memory is provided in terms of paging and segmentation. The maximum size of virtual memory is 512 million bytes, but each user program is limited to 32 million bytes. The page size is 1024 bytes; the segment size is 128K bytes; and the maximum number of segments is 4096.

5.2.1.1.2 Registers

The Prime 400 has 128 program addressable 32-bit registers. Of these, 4 are used as base registers, 2 are used as floating point accumulators, and 2 are used as field address and length registers. The remaining registers are used to control DMA channels and hold the machine states of active processes.

5.2.1.1.3 Instruction Set Hardware Support

The Prime 400 hardware provides a 32-bit arithmetic logic unit and 32-bit and 64-bit integer arithmetic. A floating point unit is provided by means of microcode. The following is a list of some of the instruction groups and how they are implemented:

decimal arithmetic	emulation
integer arithmetic	hardware
floating point	microcode
character string	emulation
conditional branches	hardware
logical operations	hardware
logical test and set	hardware
program control and jump	hardware

queue management	microcode
shifts and skips	hardware
data move	hardware

5.2.1.1.4 Process Exchange Facility

The Prime 400 process exchange facility is provided through firmware that automatically dispatches tasks for execution and reorders those which remain. This is accomplished with no software intervention. The process exchange facility also automatically handles the register switching that is needed as a result of a process exchange.

5.2.1.1.5 Input/Output

Four types of access modes for I/O are available:

1) Direct Memory Access (DMA)
2) Direct Memory Channels (DMC)
3) Direct Memory Transfer (DMT)
4) Direct Memory Queue (DMQ)

There are a maximum of 32 program assignable DMA channels. These are controlled by high speed channel address registers. They are used for high speed peripherals such as fast disk devices. The maximum transfer rate is 2.5 million bytes per second.

The DMC channels are controlled by channel address words in the first 4K bytes of main memory. Up to 2048 of these channels can be provided. Their use is mainly for medium speed I/O transfers such as data communications transfers. The maximum transfer rate is 960K bytes per second.

The DMI channels are used by high speed device controllers, e.g., controllers for moving head disks, that execute channel control programs. The maximum throughput rate is 2.5 million bytes per second.

The DRQ mode of operation provides a circular queue for handling communication devices. It reduces operating system overhead by eliminating interrupt handling on a character-by-character basis.

5.2.1.1.6 Program Environment

Programs on the Prime 400 operate in a multi-segment environment. This consists of a stack segment for local variables, a procedure segment, and a linkage segment for statically allocated variables and linkages to common data. Stack management is provided by means of hardware, and procedure calls are managed by microcode.

5.2.1.2 Standard System Software

The Standard Prime 400 Systems Software of interest in this project is the PRIMOS Operating System, the PRIMOS File System, and the Prime COBOL Programming System.

5.2.1.2.1 The PRIMOS Operating System

The operating system for the Prime 400 computer system, PRIMOS, provides interactive, batch, and real-time supervisory services within a single system. It can handle up to 63 concurrent processes including interactive users at local

or remote terminals, phantom users, and RJE processes. The Prime 400 system contains a segmented and paged virtual memory with a 32 megabyte address space and up to 8 megabytes of main memory. In addition, a 2K-byte bipolar cache memory is utilized. PRIMOS can maintain a disk capacity exceeding 2.4 billion bytes.

PRIMOS is embedded in the virtual address spaces of all processes. This results in providing access to any operating system resource in the same amount of time as it takes a user process to call a subroutine. The multiple ring protection system ensures that this feature does not result in the operating system being modified by the users.

The allocation of CPU time is provided by means of time slices, normally 1/3 second. Time slices are allocated on a priority basis with highly interactive processes receiving relatively high priorities and processor-bound processes being given lower priorities.

To minimize paging, multiple processes are able to use identical pages of a shared procedure segment. These shared procedures are reentrant and thus remain unaltered by the processes that use them. A shared procedure exists only once on disk and, when active, only once in main memory regardless of the number of processes using it.

Process exchange is handled by a hardware dispatcher (microcode) which manages the Ready List, a number of Wait Lists, Semaphores, and the Process Control Blocks. A process exchange is caused asynchronously by hardware generated interrupts, faults, and checks, and it is caused synchronously by a process executing either the WAIT or NOTIFY instructions. The dispatcher also manages the processor's five registers so that sets of the registers can be assigned to different processes, and thus the need to save and restore register contents is reduced. The Ready List identifies all processes which are ready to run and is ordered by priorities and then chronologically. Semaphores are associated with each event that can cause an exchange. They occupy two words of storage. One word contains either a counter which indicates how many times an event has occurred without being serviced by a process or the number of processes waiting for an event depending on the sign of the number. The second word contains a pointer to a Wait List of processes waiting for the event.

The virtual address space is organized as multiple segments each with sixty-four (64) 2K-byte pages. A virtual address consists of a segment number, a page number, and a word or displacement number. A physical address can be obtained by means of segment tables and page maps which are maintained in main memory. The system uses demand paging with the LRU replacement scheme. To speed access to memory there is a

segment table lookaside buffer which holds physical page addresses and a high speed cache memory which contains the most recent memory references. To satisfy a memory reference, the cache memory is addressed on the basis of the word number of the virtual address and the segment table lookaside buffer is consulted to determine if the word obtained from the cache is the desired one.

5.c.1.2.2 Prime COBOL

COBOL on the Prime 400 system is based on the ANSI COBOL X323-1974 standard. Programs up to 32 million bytes can be supported. All ANSI '74 COBOL files organized as INDEXED or RELATIVE are established as MIDAS (Multiple Index Data Access System) files. A MIDAS file can be accessed by multiple users in a sequential or random manner and can have locks specified at the data record level to resolve concurrent usage conflicts. Up to six keys or key synonyms can be used to perform a partial file search.

Prime COBOL implements, to a minimum of level 1, the nucleus; table handling; sequential, relative and indexed I/O; library; and interprogram communications modules. The debug module is implemented with the READY TRACE, RESET TRACE, and EXHIBIT NAME features supported in the procedure division. The level 2 features which are implemented are the following:

1) STRING/UNSTRING statements.

2) Conditional expressions:

   a) Conjunctive operators (and, or, not).

b) Relational operators (=, >, <, not =, not >, not <).

c) Full parenthesis support.

d) Implied subjects and relationships.

e) Nested IF statements.

3) OPEN with EXTEND.

4) INSPECT statement.

5) ALTER statement.

6) COMP-3 pack decimal.

7) COMPUTE with multiple receiving fields.

8) PERFORM VARYING with index.

9) SEARCH statement.

10) ALTERNATE RECORD KEYS WITH DUPLICATES for INDEXED files.

11) A series or range of values in level 88 condition.

5.2.1.2.3 The PRIMOS File System

The PRIMOS operating system for the Prime 400 computer supports as one of its services a flexible, hierarchical file system that provides users with the facility to maintain large quantities of data in an orderly, logical manner. A brief overview of the file system's capabilities and features is provided in Appendix B. It is somewhat tutorial in nature and does not attempt to cover all of the available features, nor to present the details of implementation.

5.2.2 The Georgia Tech Software Tools Subsystem

5.2.2.1 General

The Subsystem consists of a command interpreter that supports easy interconnection of programs, a large collection of software tools, and the capability for developing customized tools. It runs as a collection of user programs under standard versions of the PRIMOS IV and V operating systems, thus retaining all security provided by PRIMOS as well as allowing users to ignore the Subsystem if they so desire. Even so, the Subsystem is sufficiently comprehensive to permit use of the computer without a working knowledge of PRIMOS.

The concepts used in the Subsystem are derived from the text Software Tools [Kernighan 76]. The Georgia Tech Software Tools Subsystem, however, has undergone development that has greatly extended it beyond the starting point given by the book. For instance, the Text Editor has been expanded into a full-screen CRT editor, and the command language has become a superset of that provided by the UNIX operating system.

5.2.2.2 Major Components and Features

The major components and features of the Subsystem are as follows:

1) Command Language
   a) Uniform Invocation of Commands
   b) Dynamic Command Line Structures
   c) Command Files
   d) Control Structures
   e) Scoped Variables
   f) Networks of Cooperating Programs

2) Ratfor ("Rational FORTRAN" - a Preprocessor)
   a) Modern Control Structures
   b) Free Format Source Code
   c) Basic Macro Capability
   d) Long Variable Names
   e) Debugging and Performance Monitoring

3) Text Editor
   a) Full-Screen Version
   b) Line-Oriented
   c) In-line Editing Capability
   d) Similar to UNIX Text Editor

4) Text Formatter
   a) Automatic Hyphenation
   b) Macros
   c) In-Line Functions
   d) Diversions
   e) Number Registers

5) On-Line Documentation
   a) Access to Reference Manual
   b) Short Usage Summary Available

6) Assorted Tools
   a) Directory Maintenance
   b) Stream Editing and Selection
   c) Sort and Comparison
   d) Spelling Check

5.3 ORGANIZATION OF THE DEMONSTRATION WORKBENCH

Conceptually the organization of the Demonstration Workbench is quite simple. It consists of the following major components:

1) The PRIME 400 Computer System
2) The PRIMOS Operating System
3) The Georgia Tech Software Tools Subsystem
4) A collection of Software Tools command files

The last component above transforms the total environment from a general purpose support environment into a system which will provide the specific capabilities of the COBOL Programmer's Workbench. In this specific area, the real power of the Software Tools Subsystem proved invaluable to the project team.

A casual or non-systems programmer user of the Workbench does not need to see the individual components listed above. It is also highly unlikely in normal use that a casual user would need to examine or understand the contents of the command files or their operation. One of the extremely useful features of the Software Tools Subsystem is the ability to treat executable command files in the same manner as any other file, and if the user sitting at his terminal enters the name of that file, then it will be automatically executed. This allows us to define and utilize extremely simple and meaningful command names for utilization by the COBOL programmer. It is also the extreme flexibility

provided by the command interpreter that provides an environment that will react to any style of programming desired by the programmer.

5.4 OPERATION OF THE DEMONSTRATION WORKBENCH

Simplicity of operation by the user has been one of the primary objectives in the design of the workbench. This is one area in which we feel that the research project has been extremely successful.

As mentioned above, the workbench user invokes the service desired by merely giving a single name command to the command interpreter interface. However, at the same time, the programmer is able to change or modify the standard operation of the workbench by building personalized command files or entering single more primitive commands to the command interface.

One of the primary foundations for this flexibility is the standard file structure for all files created by the workbench. Regardless of whether a file is created and/or modified by a command file operation or by individual Software Tools commands, it retains its standard organization, and therefore can be easily integrated, concatinated, or operated on with other files in a uniform manner.

5.5 MAJOR PROBLEMS ENCOUNTERED

There were actually only two major problems encountered during this project and, unfortunately, neither one has been solved completely. One of these major problems pertains to Program Preparation, while the other concerns the Documentation Subsystem.

The concepts of the Program Preparation Subsystem have been implemented and amply demonstrated for both their power and usefulness. The major impediments to completeing an operational model of the programming subsystem are:

- The definition of Workbench COBOL (COBOL.wbc)
- The definition of the COBOL Programmer's Environment

COBOL programmers, as a group, have been very slow to utilize automatic programming aids, even those rudimentary ones presently available to them. During the course of this study, extensive conversations have been held with key personnel in several large COBOL programming groups. In almost all of these meetings the same conclusions were reached. Since COBOL programmers have had so little exposure to automated software tools, they are not able to provide any definitive guidance as to what would be useful for them or even what they would like to have available. On the other hand, all of these programmers admit that there is a high potential for increased productivity if such tools were

available. It appears that the only approach that is going to work in solving this problem is to develop some specific COBOL programmer's environment including specialized tools such as customized screen editors, copy-from-library capabilities, programming skeletons, fill-in-the-blanks techniques, etc., and to try out these specialized tools with COBOL programmers in an operation environment.

As has been well argued earlier in this report, there is no desire to design a new language for the use of programmers preparing transportable COBOL programs. Rather, what is desired is a language that is as close as possible to standard COBOL with deviations being made only to accomodate those absolutely essential requirements necessary for portability. Our studies during this project, however, indicate that it does not appear feasable to include the entire COBOL standard in Workbench COBOL. It does not appear to us that such an extensive language is at all necessary. The real goal in the design of Workbench COBOL is to provide a language powerful enough and rich enough so that experienced COBOL programmers can easily and efficiently implement operational COBOL data processing systems. There have been a number of studies prepared on the development of portable COBOL subsets. However, it should be emphasized that that is not the approach being taken here. What is being done here is preparing a "complete" COBOL language where those features that have to be tailored to a specific compiler-unique environment are handled by the program

processors in the Workbench. The initial studies that have been performed in obtaining this goal have focused on identifying what features of COBOL are utilized by programmers in the execution of their task. It apears that this is a valid approach to this problem, and all that remains to be done is to utilize a broader sample of programs and application areas in order to obtain high confidence in the validity for the design of Workbench COBOL.

The last major problem area that was not completely solved during this research project was the selection of a method to store and produce various versions of a document that is undergoing continual change and revision. The basic approaches to this problem are described in Appendix 3. This is another area in which it appears that more detailed knowledge of the environment in which the system is to be utilized will be required before a specific implementation technique can be selected.

SECTION 6

SUMMARY

This research project has been successful in achieving its goals in the study and development of the COBOL Programmer's Workbench. The concept of the Workbench and the capabilities it provides have been shown to be quite valid, and the Demonstration Workbench clearly illustrates the feasibility of implementing a complete, full-scale COBOL Programmer's Workbench. The Workbench similarly supports the development of breadboard, prototype systems; however, it should be noted that investigation of the value and feasibility of the use of breadboard systems was not within the scope of this project.

Although the concept and feasibility of the Workbench have been verified, a large amount of work remains to be done before a complete Workbench could be implemnted. The major areas in which work is still required are

- Completion of the definition of Workbench COBOL
- Completion of the definition and design of the COBOL programmer's environment
- Selection and implementation of the technique to be utilized to maintain and generate various versions of documentation.

One of the requirements necessary to support this further work is access to several major programming teams and cooperation from those teams in obtaining actual usage data

and information about the present program development and maintenance environments.

REFERENCES

ANSI, Draft Proposed Revised X3.23 American National Standard Specifications for COBOL, August, 1972.

Applied Data Research, The Librarian User Reference Manual, Pub. No. P111L, Princeton, New Jersey: Applied Data Research, Inc., 1971.

Applied Data Research, MetaCOBOL Macro Writing, vol. 1, Pub. No. P501M, Princeton, New Jersey: Applied Data Research, Inc., 1973.

Applied Data Research, MetaCOBOL Macro Writing, vol. 2, Pub. No. P550M, Princeton, New Jersey: Applied Data Research, Inc., 1973.

Applied Data Research, User Guide to Meta COBOL, Pub. No. P206M, Princeton, New Jersey: Applied Data Research, Inc., 1973.

Brown, P. J., ed., Software Portability, Cambridge, Massachusetts: Cambridge University Press, 1977.

Crafford, J. C., Jr., Curry, R. W., and DeCarli, P. A., Integrated Test Bed Support Software System Study Phase II, U.S. Army Computer Systems Command Technical Report USACSC-AT-7T-08, November, 1976.

Curry, R., Integrated Test Bed Support Software System Study Phase I, U.S. Army Computer Systems Command Technical Report USACSC-AT-7T-04, August, 1976.

Denike, T., Holland, A., Ward, T., and Desai, H., Software Portability Study - Conversion Procedures, U.S. Army Computer Systems Command Technical Report USACSC-AT-77-11, June, 1977.

Digital Equipment Corporation, PDP-11 COBOL User's Guide, Pub. No. AA-1757C-TC, Maynard, Massachusetts: Digital Equipment Corporation, April, 1977.

Hardy, I. Trotter, Leong-Hong, Belkis, and Fife, Dennis W., Software Tools: A Building Block Approach, National Bureau of Standards Special Publication 500-14, August, 1977.

Ivie, Evan L., "The Programmer's Workbench - A Machine for Software Development," Communications of the ACM 20 (October, 1977): 746-753.

Kernighan, Brian W., and Plauger, P. J., Software Tools, Reading, Massachusetts: Addison-Wesley, 1976.

PRIME Computer, Inc., PRIME COBOL Programmer's Guide, Pub. No. POR3056, Framingham, Massachusetts: Prime Computer, Inc., September, 1978.

Redwine, Samuel T., Jr., "Using COBOL Macroprocessing for Reliable Parameter Passing," ACM SIGPLAN NOTICES, 14:5, Sept, 1979, pp 59-60.

Rochkind, Marc J., "The Source Code Control System," IEEE Transactions on Software Engineering SE-1 (December, 1975): 364-369.

Sordillo, Donald A., The Programmer's ANSI COBOL Reference Manual, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.

Turner, Dennis J., "An Integrated System of Tools to Support the DOD Common Language," Second U. S. Army Software Symposium, 25-27 October, 1978, U. S. Army Computer Systems Command.

University of Florida and University of South Florida, Optimal COBOL Subset for Software Portability: Portable Standard COBOL (PSC) Specifications, U.S. Army Computer Systems Command Interim Technical Working Report, February, 1978.

Waite, W. M., "Hints on Distributing Portable Software," Software - Practice and Experience 5 (1975): 295-308.

APPENDIX 1

GLOSSARY

APPLIB-name: A library file containing macro definitions (in COBOL.wbc) of utility routines and modules commonly needed in a particular application area. Typical examples include APPLIB-PAYROLL and APPLIB-INVEN.

Base Line System: The COBOL.wbc program that defines the delivered COBOL.n programs for the various target machines.

Breadboard: A program developed primarily as an experimental model of a production program. Easily modifiable to permit both requirements changes and eventual optimization.
[See prototype]

Capability: A task, purpose, or function which may be carried out in some unspecified manner by a language or system. An example of a capability in COBOL is the ability to sort files according to the value of a given field; this might be performed by the use of a SORT verb, by a call to an external procedure, or by the inline inclusion of the appropriate COBOL commands.

COBOL Programmer's Workbench: A comprehensive collection of capabilities and facilities to assist the COBOL programmer in the development and preparation of complete COBOL systems supported by full documentation. Other capabilities of the Workbench are to facilitate the development of "prototype or breadboard" COBOL programs from a collection of reusable COBOL modules and to facilitate the preparation of COBOL programs that may be easily converted to a number of different target machines each having its own unique dialect of the COBOL language.
[See program preparation system; text documentation preparation; document control system]

COBOL.ccs: Common COBOL Subset. This consists of those elements of COBOL common to all COBOL compilers. COBOL.ccs is not sufficient for the development of useful programs. COBOL.ccs is included in both COBOL.demo and COBOL.wbc. COBOL.ccs is a proper subset of COBOL.

COBOL.demo: The initial approximation to COBOL.wbc. As COBOL.demo is expected to approach COBOL.wbc over time, distinction is made between the two languages only when necessary for a clear understanding of the difference between what should be included (COBOL.wbc) and what it has been able to implement thus far (COBOL.demo).

Eventually, COBOL.demo and COBOL.wbc will become identical. Note that COBOL.demo **is not** a proper subset of standard COBOL!

COBOL.i: Refers to any member of the group of COBOL dialects accepted by target machines.
[See COBOL.wbc; COBOL.k]

COBOL.i Program: A compiler- (and machine-) dependent COBOL program. Refers to any member of the family of programs defined by a particular COBOL.wbc program.
[See COBOL.k program]

COBOL.k: The dialect of COBOL associated with a given COBOL compiler k. Programs written in COBOL.k may not run on a different COBOL compiler without modification since the dialect may contain compiler-dependent features.
[See COBOL.wbc; COBOL.i]

COBOL.k Program: A program written in COBOL.k. A non-portable program.
[See COBOL.wbc program]

COBOL.wbc: A language for the development of portable COBOL programs. This consists of COBOL.ccs plus a collection of macro calls giving the programmer the ability to express compiler-dependent features in a compiler-independent manner. The Workbench expands COBOL.wbc into the COBOL.k of a target compiler k. Note that COBOL.wbc is <u>not</u> a proper subset of standard COBOL!
[See COBOL.i; COBOL.k; COBOL.demo] [See also Appendix 2; Appendix 7; Appendix ?]

COBOL.wbc Module: A macro which expands into COBOL.wbc code defining a commonly occurring function or action. This permits the programmer to develop <u>reusable</u> sections of code. COBOL.wbc module definitions are included in the following libraries: GENLIB, APPLIB-name, PROJLIB-name, TESTLIB.

COBOL.wbc Program: A program written in COBOL.wbc. A program which can be easily transported. Consists of COBOL.ccs statements, module calls, and macro calls.

Compiler-Dependent: Written in a particular COBOL dialect for compilation by a given compiler. Cannot necessarily be compiled by a compiler written for a different dialect. It is possible for a program to be compiler-dependent while being machine-independent.
[See nonportable; machine-dependent; COBOL.k; dialect]

Compiler-Independent: Written without reliance on language features unique to a given COBOL dialect. Can be transported easily. Usually designed with portability as a major goal. Must be machine-independent.
[See portable; machine-independent; COBOL.wbc]

Conversion: The modification of a program written in one COBOL dialect for execution on a machine accepting a different COBOL dialect.
[See portable; COBOL.wbc]

Delivered Program: The COBOL.k program as it will run on the target machine k in a production environment. The program as intended for use by the customer or other end user. At this point the program specifications are fixed until the original COBOL.wbc program is modified and new COBOL.k code is delivered.
[See revision]

Dialect: The particular COBOL syntax required by a given COBOL compiler. Also, those language features supported by that compiler.
[See COBOL.k; COBOL.wbc; machine-dependent; compiler-dependent]

Document Control System: Maintains each program and each revision or version thereof, as well as any related text documents, online in an easily accessible and easily modifiable form.

Element: A characteristic of the language as defined by the 1974 COBOL standard.

Feature: A characteristic of the language as implemented by a compiler on a specific machine.

Functional Testing: Testing to show the **functional** equivalence of each of the family of COBOL.1 modules or programs derived from a particular COBOL.wbc module or program.
[See version; program family]

GENLIB: A library file containing macro definitions which define utility functions and modules of general use. A typical module which might be included here would be an abbreviation of the program-id section, such as provided by META-COBOL. Should contain only <u>listing</u> macro definitions written in COBOL.wbc.

"Help" Command: On-line documentation provided for the Workbench user in the form of a guided tutorial.

Machine-Dependent: Written using hardware and operating system features unique to a particular machine. Cannot be executed on a machine with different features. Necessarily compiler-dependent as well due to the need for the compiler to allow access to the hardware and operating system features.
[See nonportable; compiler-dependent; COBOL.k; dialect]

Machine Environment: The combination of hardware and software features (including hardware capabilities, operating system features, and compiler dialect) which define the environment under which a program is to be run. In particular, those features which differ from machine to machine, compiler to compiler, and installation to installation, and make program transportation difficult.

Machine-Independent: written without reliance on hardware or operating system features unique to a given machine. Can be transported easily unless the program is compiler-dependent.
[See portable; compiler-independent; COBOL.wbc]

MACLIB.k: A library file containing those macro definitions which MACRO needs as input in order to convert COBOL.wbc into COBOL.k. (Macro calls must meet standards so that a given macro will be called the same way by COBOL.k and COBOL.j.) Should contain only <u>resident</u> macro definitions.

Macro: (In general use the term is used to refer to assembly-level programming. However, it may be applied to higher level language programming, as it is used here) A skeleton for an open subroutine which is completed by a macro generator in response to a call by a macro instruction during the process of assembling a program. The completed subroutine is passed to the assembler for incorporation into the program. However, the term may also apply to higher level language programming. A program unit which performs a single function and is expressed in a reusable form. Similar to a subroutine, although a macro is a purely <u>notational</u> device which is expanded <u>in-line</u> (like a Fortran statement function) to produce the appropriate code <u>each time</u> it is called while a subroutine call produces only a jump to a single copy of the required code regardless of the number of calls to the subroutine.
[See module] [See also Appendix 7]

Macro Processor: A program which accepts as its inputs text containing macro calls and a file containing macro definitions and produces as its output text in which the macro calls have been expanded into their related definitions.
[See macro; module] [See also Appendix 7]

Maintenance: The changes necessary to keep a delivered program executing correctly according to the original specifications despite discovery of errors and changes in machine environment.
[See modification]

Modification: The changes necessary to change the specified behavior of a delivered program.
[See maintenance; revision]

Module: A program unit which performs a single function and is expressed in a reusable form so that the same code may be used to perform the same function wherever it appears in the same program or other programs. Written in COBOL.wbc.
[See macro]

Module Interface: The standards governing module calls and interrelations between the segments resulting from modular expansion into the COBOL.wbc program. For the purposes of the Workbench, all modules are expressed as macros and module usage must meet the same standards as macro usage.

Nonportable: A program written to fit a specific machine environment and incapable of running without modification on all machine environments.
[See portable.]

Portable: A program written to run in any machine environment and on any compiler. Generally, this is accomplished by restricting the program to the use of those hardware features which are common to all of the machine environments on which the program will run and those language features which are implemented in compilers for each of these machine environments. This approach prohibits the programmer from making use of machine-dependent or compiler-dependent features for the sake of efficiency.
[See nonportable; COBOL.wbc; COBOL.k; MACLIB.k; COBOL.ccs.]

Production Environment: The system on which the customer or end user will run the delivered COBOL.k program. Performance requirements are assumed.
[See Workbench]

Program: The block of code necessary to perform some task without additional instructions. Usually compiled and loaded separately.

Program Family: The set of COBOL.i programs defined by a particular COBOL.wbc program and produced by macro expansion using MACLIB.i where i represents each of the target machines in sequence.
[See COBOL.wbc program; COBOL.k program]

Program Fragment: An incomplete section of code, usually one which performs an indentifible task. Cannot execute without additional instructions. An incomplete program.
[See Program; Module]

Program Preparation System: Provided to assist the programmer in developing the original baseline system, as well as in converting or transporting that system to various target machines.

PROJLIB-name: A library file containing macro definitions of modules developed specifically by the project team for use on their project.

Prototype System: The COBOL.wbc program that defines a family of COBOL.n programs during testing, before delivery. The first, as well as the successive, approximation towards the program which will be delivered.
[See breadboard]

Release: Refers to a family of documents which differ from predecessor or successor releases by virtue of modification for efficiency, expanded capability, or to remove errors. This concept allows several groups to work independently on a document existing in several stages or releases. For example, one release may be in production and another in development. Thus, one group can make corrections or modifications to the release in production with no effect on the release in development.
[See modification; revision; version]

Reusable: written in a flexible manner so that the code may be used in other program applications.
[See module; portable]

Revision: Refers to a document which differs significantly from predecessor or successor revisions by virtue of modification for efficiency, expanded capability, or to remove errors. A revision consists of a release of a document and the version of the release. Thus, revision 2.3 of a document is version 3 of release 2.
[See modification; release; version]

Software Tools Subsystem (SWT): An integretated set of automated software and text preparation and manipulation routines.
[see Section 5.2.2; Appendices 4 through 7]

Source Code Control System (SCCS): As defined in Bell Labs' Programmer's Workbench [Rochkind 75], this system maintains and provides access to all revisions of any document.

Standards Enforcer: A routine that operates as a "preprocessor" to verify that the standards for COBOL.wbc or COBOL.k are followed in both program text and documentation.

Test Harness: A main program which provides a framework into which a modular unit can be placed for the purpose of testing. A driver program, expressed as a macro and stored in TESTLIB.
[See functional testing]

Text Documents: All program documentation apart from comments internal to the code. Usually includes program specification, project memos, error reports, and a user's guide. Maintained on line.

Text Documentation Preparation System: This system is provided to assist the systems analyst in the preparation of all of the system documentation required for the support of the prototype systems, the base line system, and those unique varieties of the system that are produced to run on the various target machines.

TESTLIB: A library file containing macro definitions of utility functions and modules to aid the programmer in testing both modules and programs for both correctness and performance. Should contain only tested code.

Transportable: A program written to run in any machine environment and on any compiler. Generally, this is accomplished by restricting the program to the use of those hardware features which are common to all of the machine environments on which the program will run and those language features which are implemented in compilers for each of these machine environments. This approach prohibits the programmer from making use of machine-dependent or compiler-dependent features for the sake of efficiency. Use of certain machine-dependent or compiler-dependent features may be permitted, if any necessary modifications can be made automatically.
[See nonportable; COBOL.wbc; COBOL.k; MACLIB.k; COBOL.ccs.]

Transportation: The process involved in moving a program from one machine to another. If the two machines accept the same COBOL dialect, the process is mechanical; if the two dialects involved differ, extensive modification may be required.
[See conversion]

Version: Refers to a particular document within a family of documents defined by a release of a document. Version refers to a document resulting from specific modifications to another version within a release.
[See modification; release; revision]

Workbench: A comprehensive collection of capabilities and facilities to assist the COBOL programmer in the development and preparation of complete COBOL systems supported by full documentation. Other capabilities of

the Workbench are to facilitate the development of "prototype or breadboard" COBOL programs from a collection of reusable COBOL modules and to facilitate the preparation of COBOL programs that may be easily converted to a number of different target machines each having its own unique dialect of the COBOL language.

APPENDIX 2

AN ANNOTATED EXAMPLE OF TRANSPORTING A WORKBENCH COBOL PROGRAM

In this example, the "baseline" program is written in COBOL.demo, the demonstration version of COBOL.wbc. The two target operating environments are the Control Data Corporation CYBER 70/74 and the PRIME Computer P400 system. This example focuses on the differences in the form and content of the source program; however, the principal technique illustrated, the use of macro libraries, can also be applied to some of the other incompatibilities in the operating environment.

IMPORTANT NOTE: In the examples given below, lower case letters are used to indicate calls to reusable modules and compiler-unique features. Lower case is used here solely as an aid to the reader's comprehension and conveys no added information to the macro processor. All characters are treated by the macro processor as if they were upper case.

2.1. MACRO LIBRARIES

The macro libraries contain the definitions of the macros that transform the program from COBOL.wbc (demo) to COBOL.target. In this example, there are two macro libraries. Note that throughout this example calls or references to the functions in these libraries are identified by using lower-case letters; however, this is not required. It is done in the example to highlight the calls. Also note that "$1" and "$2" refer to the first and second arguments given in the macro call, respectively.

Macro Library for the PRIME

```
define(id,
  IDENTIFICATION DIVISION.
  PROGRAM-ID.
        $1.
)dnl
define(comp5,COMPUTATIONAL)dnl
define(reader,PF"S)dnl
define(printer,PF"S)dnl
define(file_id,VALUE OF FILE-ID IS "$1")dnl
define(cyber,)dnl
define(p400,$1)dnl
define(quote,"$1")dnl
```

Macro Library for the CYBER

```
define(id,
  IDENTIFICATION DIVISION.
  PROGRAM-ID.
          $1.
)dnl
define(comp3,COMPUTATIONAL)dnl
define(reader,INPUT-FZ)dnl
define(printer,OUTPUT-FZ)dnl
define(file_id,)dnl
define(cyber,$1)dnl
define(p400,)dnl
define(quote,"$1")dnl
```

2.2. PROJECT LIBRARY

A programmer using the Workbench will need to have access to code modules that are common to all those working on a given project. For this example, we assume that two file descriptions are to be shared, and that one COBOL code structure (a multi-level split on several keys in a sorted file) occurs often enough that it has been included in the project library.

General

The two File Description modules in the Project Library illustrate how two rather drastic differences between CYBER and PRIME COBOL are treated. These differences are

- LABEL RECORDS statement must say
  - -- STANDARD in PRIME
  - -- OMITTED in CYBER
- FILE ID must be
  - -- VALUE OF FILE-ID IS ____ in PRIME
  - -- omitted in CYBER

Examining the macro library definitions for "cyber" and "p400" in each library shows how the P400 library causes entries annotated "cyber" to be deleted and entries annotated "p400" to be copied into the transported program. The "cyber" library causes just the reverse action. The two definitions for "file-id" handle the second problem by generating the proper statement for the PRIME and ignoring it completely for the CYBER.

File Description Module for "SALES-FILE"

```
 FD SALES-FILE
      p400(LABEL RECORDS ARE STANDARD)
      cyber(LABEL RECORDS ARE OMITTED)
      file_id(SALES)
      DATA RECORD IS SALES-RECORD.
 01 SALES-RECORD.
      05  SALESMAN              PICTURE X(5).
      05  DISTRICT              PICTURE XXX.
      05  SALES-DOLLARS         PICTURE 9(5)V99.
```

File Description Module for "REPORT-File"

```
 FD REPORT-FILE
      p400(LABEL RECORDS ARE STANDARD)
      cyber(LABEL RECORDS ARE OMITTED)
      file_id(REPORT)
      DATA RECORD IS REPORT-RECORD.
 01 REPORT-RECORD.
      05  CARRIAGE-CONTROL      PICTURE X.
      05  SALESMAN-OUT          PICTURE X(5).
      05  FILLER                PICTURE XXX.
      05  SALESMAN-TOTAL-OUT    PICTURE $$$,$$$,$$9.99.
      05  FILLER                PICTURE X(8).
      05  DISTRICT-OUT          PICTURE XXX.
      05  FILLER                PICTURE XXX.
      05  DISTRICT-TOTAL-OUT    PICTURE $$$,$$$,$$9.99.
      05  FILLER                PICTURE X(8).
      05  FINAL-TOTAL-OUT       PICTURE $$$,$$$,$$9.99.
```

Reusable Code Modules Identified as "nlevel"

```
define(nlevel,
 MOVE $2 TO PREVIOUS-$2.
 PERFORM $1 UNTIL
[ifelse($2,,,                    $2 IS NOT EQUAL TO PREVIOUS-$2
)]dnl
[ifelse($3,,,                 OR $3 IS NOT EQUAL TO PREVIOUS-$3
)]dnl
[ifelse($4,,,                 OR $4 IS NOT EQUAL TO PREVIOUS-$4
)]dnl
                 OR NO-MORE-DATA.
)dnl
```

2.3. WORKBENCH COBOL PROGRAM

Now that the preliminaries are out of the way, the programmer can create a program in "Workbench Cobol," a combination of COBOL.ccs and macro calls:

Baseline Program

```
include(PROJLIB/NLEVEL)

ifdef(TWOLEVEL)
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT SALES-FILE ASSIGN TO reader.
    SELECT REPORT-FILE ASSIGN TO printer.

DATA DIVISION.
FILE SECTION.
include(PROJLIB/FD-SALES-FILE)
include(PROJLIB/FD-REPORT-FILE)

WORKING-STORAGE SECTION.

01 FLAGS.
    05  MORE-DATA-FLAG         PICTURE XXX VALUE quote(YES).
        88  MORE-DATA                      VALUE quote(YES).
        88  NO-MORE-DATA                   VALUE quote(NO ).

01 SAVE-ITEMS.
    05  PREVIOUS-SALESMAN      PICTURE X(5).
    05  PREVIOUS-DISTRICT      PICTURE XXX.

01 TOTALS comp3.
    05  SALESMAN-TOTAL         PICTURE S9(8)V99 VALUE ZERO.
    05  DISTRICT-TOTAL         PICTURE S9(8)V99 VALUE ZERO.
    05  FINAL-TOTAL            PICTURE S9(8)V99 VALUE ZERO.

PROCEDURE DIVISION.
PREPARE-SALES-REPORT.
    OPEN INPUT SALES-FILE,
        OUTPUT REPORT-FILE.
    READ SALES-FILE
        AT END MOVE quote(NO ) TO MORE-DATA-FLAG.
    MOVE ZERO TO FINAL-TOTAL.
    PERFORM DISTRICT-TOTAL-PROCESSING
        UNTIL NO-MORE-DATA.
    MOVE SPACES TO REPORT-RECORD.
    MOVE FINAL-TOTAL TO FINAL-TOTAL-OUT.
    WRITE REPORT-RECORD.
    CLOSE  SALES-FILE,
        REPORT-FILE.
    STOP RUN.
```

```
DISTRICT-TOTAL-PROCESSING.
    MOVE ZERO TO DISTRICT-TOTAL.

    nlevel(SALESMAN-TOTAL-PROCESSING,DISTRICT)

    MOVE SPACES TO REPORT-RECORD.
    MOVE PREVIOUS-DISTRICT TO DISTRICT-OUT.
    MOVE DISTRICT-TOTAL TO DISTRICT-TOTAL-OUT.
    WRITE REPORT-RECORD.
    ADD DISTRICT-TOTAL TO FINAL-TOTAL.

SALESMAN-TOTAL-PROCESSING.
    MOVE ZERO TO SALESMAN-TOTAL.

    nlevel(PROCESS-AND-READ,SALESMAN,DISTRICT)

    MOVE SPACES TO REPORT-RECORD.
    MOVE PREVIOUS-SALESMAN TO SALESMAN-OUT.
    MOVE SALESMAN-TOTAL TO SALESMAN-TOTAL-OUT.
    WRITE REPORT-RECORD.
    ADD SALESMAN-TOTAL TO DISTRICT-TOTAL.

PROCESS-AND-READ.
    ADD SALES-DOLLARS TO SALESMAN-TOTAL.
    READ SALES-FILE
        AT END MOVE quote(NO) TO MORE-DATA-FLAG.
```

Program Data

The data for this program is present in the file "sales":

```
41  10010032
41  10000300
41  10010005
52  10011000
69  10007403
59  10006062
18  20020769
37  20018560
36  20019415
19  30012140
92  30005180
```

2.4. PROCESSING

Using the workbench utility "convert," the initial program can be expanded utilizing the Project Library and then transported utilizing the Macro Libraries. In the examples below the intermediate, expanded versions of the program are not listed. Only the two compiler-unique forms of the program are listed completely; however, the results of the expansion are readily apparent in the file definitions and in the calls to "nlevel."

2.4.1 Prime 400 Version

Since the standards enforcer is not available, we will simply proceed to the compilation, linking, and execution of the P400 version of the program.

2.4.1.1 Prime 400 Program

```
 REV 14  COBOL   SOURCE FILE: EX.P400          08/09/77
 (0001)
 (0002)
 (0003)
 (0004) IDENTIFICATION DIVISION.
 (0005) PROGRAM-ID.
 (0006)   TWOLEVEL.
 (0007)
 (0008) ENVIRONMENT DIVISION.
 (0009) INPUT-OUTPUT SECTION.
 (0010) FILE-CONTROL.
 (0011)   SELECT SALES-FILE ASSIGN TO PFMS.
 (0012)   SELECT REPORT-FILE ASSIGN TO PFMS.
 (0013)
 (0014) DATA DIVISION.
 (0015) FILE SECTION.
 (0016)
 (0017) FD SALES-FILE
 (0018)   LABEL RECORDS ARE STANDARD
 (0019)
 (0020)   VALUE OF FILE-ID IS 'SALES'
 (0021)   DATA RECORD IS SALES-RECORD.
 (0022) 01  SALES-RECORD.
 (0023)   05  SALESMAN              PICTURE X(5).
 (0024)   05  DISTRICT              PICTURE XXX.
 (0025)   05  SALES-DOLLARS         PICTURE 9(5)V99.
 (0026)
 (0027)
 (0028) FD REPORT-FILE
 (0029)   LABEL RECORDS ARE STANDARD
 (0030)
 (0031)   VALUE OF FILE-ID IS 'REPORT'
 (0032)   DATA RECORD IS REPORT-RECORD.
 (0033) 01  REPORT-RECORD.
 (0034)   05  CARRIAGE-CONTROL      PICTURE X.
```

```
(0035)    05  SALESMAN-OUT           PICTURE X(5).
(0036)    05  FILLER                 PICTURE XXX.
(0037)    05  SALESMAN-TOTAL-OUT     PICTURE $$$,$$$,$$9.99.
(0038)    05  FILLER                 PICTURE X(8).
(0039)    05  DISTRICT-OUT           PICTURE XXX.
(0040)    05  FILLER                 PICTURE XXX.
(0041)    05  DISTRICT-TOTAL-OUT     PICTURE $$$,$$$,$$9.99.
(0042)    05  FILLER                 PICTURE X(8).
(0043)    05  FINAL-TOTAL-OUT        PICTURE $$$,$$$,$$9.99.
(0044)
(0045)
(0046)  WORKING-STORAGE SECTION.
(0047)
(0048)  01  FLAGS.
(0049)    05  MORE-DATA-FLAG         PICTURE XXX  VALUE 'YES'.
(0050)      88  MORE-DATA                       VALUE 'YES'.
(0051)      88  NO-MORE-DATA                    VALUE 'NO '.
(0052)
(0053)  01  SAVE-ITEMS.
(0054)    05  PREVIOUS-SALESMAN      PICTURE X(5).
(0055)    05  PREVIOUS-DISTRICT      PICTURE XXX.
(0056)
(0057)  01  TOTALS COMPUTATIONAL.
(0058)    05  SALESMAN-TOTAL     PICTURE S9(8)V99 VALUE ZERO.
(0059)    05  DISTRICT-TOTAL     PICTURE S9(8)V99 VALUE ZERO.
(0060)    05  FINAL-TOTAL        PICTURE S9(8)V99 VALUE ZERO.
(0061)
(0062)  PROCEDURE DIVISION.
(0063)  PREPARE-SALES-REPORT.
(0064)    OPEN   INPUT SALES-FILE,
(0065)      OUTPUT REPORT-FILE.
(0066)    READ SALES-FILE
(0067)      AT END MOVE 'NO ' TO MORE-DATA-FLAG.
(0068)    MOVE ZERO TO FINAL-TOTAL.
(0069)    PERFORM DISTRICT-TOTAL-PROCESSING
(0070)      UNTIL NO-MORE-DATA.
(0071)    MOVE SPACES TO REPORT-RECORD.
(0072)    MOVE FINAL-TOTAL TO FINAL-TOTAL-OUT.
(0073)    WRITE REPORT-RECORD.
(0074)    CLOSE  SALES-FILE,
(0075)      REPORT-FILE.
(0076)    STOP RUN.
(0077)
(0078)  DISTRICT-TOTAL-PROCESSING.
(0079)    MOVE ZERO TO DISTRICT-TOTAL.
(0080)
(0081)
(0082)    MOVE DISTRICT TO PREVIOUS-DISTRICT.
(0083)    PERFORM SALESMAN-TOTAL-PROCESSING UNTIL
(0084)      DISTRICT IS NOT EQUAL TO PREVIOUS-DISTRICT
(0085)      OR NO-MORE-DATA.
(0086)
(0087)
(0088)    MOVE SPACES TO REPORT-RECORD.
(0089)    MOVE PREVIOUS-DISTRICT TO DISTRICT-OUT.
(0090)    MOVE DISTRICT-TOTAL TO DISTRICT-TOTAL-OUT.
```

```
(0091)    WRITE REPORT-RECORD.
(0092)    ADD DISTRICT-TOTAL TO FINAL-TOTAL.
(0093)
(0094)  SALESMAN-TOTAL-PROCESSING.
(0095)    MOVE ZERO TO SALESMAN-TOTAL.
(0096)
(0097)
(0098)    MOVE SALESMAN TO PREVIOUS-SALESMAN.
(0099)    PERFORM PROCESS-AND-READ UNTIL
(0100)      SALESMAN IS NOT EQUAL TO PREVIOUS-SALESMAN
(0101)      OR DISTRICT IS NOT EQUAL TO PREVIOUS-DISTRICT
(0102)      OR NO-MORE-DATA.
(0103)
(0104)
(0105)    MOVE SPACES TO REPORT-RECORD.
(0106)    MOVE PREVIOUS-SALESMAN TO SALESMAN-OUT.
(0107)    MOVE SALESMAN-TOTAL TO SALESMAN-TOTAL-OUT.
(0108)    WRITE REPORT-RECORD.
(0109)    ADD SALESMAN-TOTAL TO DISTRICT-TOTAL.
(0110)
(0111)  PROCESS-AND-READ.
(0112)    ADD SALES-DOLLARS TO SALESMAN-TOTAL.
(0113)    READ SALES-FILE
(0114)      AT END MOVE "NO" TO MORE-DATA-FLAG.

58:/W/ "COMP" IGNORED FOR DECIMAL ITEM.
59:/W/ "COMP" IGNORED FOR DECIMAL ITEM.
61:/W/ "COMP" IGNORED FOR DECIMAL ITEM.

0000 ERRORS  0003 WARNINGS. P400/500 COBOL REV 14.0 <TWOLEV>
```

2.4.1.2 Prime 400 Output

The output of the P400 version of the program is:

```
41      $203.37
52      $110.00
69      $134.65
                        1       $448.02
18      $207.69
32      $185.60
                        2       $393.29
36      $194.15
39      $121.40
50       $51.80
                        3       $367.35
                                        $1,208.66
```

2.4.2 CYBER Version

The Cyber version of our sample program was written to tape on the Workbench (PRIME system) and then transferred to the Cyber, where its compilation and execution yielded the following results.

2.4.2.1 CYBER Program

AO 0112 COBOL

```
00001
00002
00003  IDENTIFICATION DIVISION.
00004  PROGRAM-ID.
00005    TWOLEVEL.
00006
00007  ENVIRONMENT DIVISION.
00008  INPUT-OUTPUT SECTION.
00009  FILE-CONTROL.
00010    SELECT SALES-FILE ASSIGN TO INPUT-FZ.
00011    SELECT REPORT-FILE ASSIGN TO OUTPUT-FZ.
00012
00013  DATA DIVISION.
00014  FILE SECTION.
00015
00016  FD SALES-FILE
00017
00018    LABEL RECORDS ARE OMITTED
00019
00020    DATA RECORD IS SALES-RECORD.
00021  01  SALES-RECORD.
00022      05  SALESMAN              PICTURE X(5).
00023      05  DISTRICT              PICTURE XXX.
00024      05  SALES-DOLLARS         PICTURE 9(5)V99.
00025
00026
00027  FD REPORT-FILE
00028
00029    LABEL RECORDS ARE OMITTED
00030
00031    DATA RECORD IS REPORT-RECORD.
00032  01  REPORT-RECORD.
00033      05  CARRIAGE-CONTROL      PICTURE X.
00034      05  SALESMAN-OUT          PICTURE X(5).
00035      05  FILLER                PICTURE XXX.
00036      05  SALESMAN-TOTAL-OUT    PICTURE $$$,$$$,$$9.99.
00037      05  FILLER                PICTURE X(8).
00038      05  DISTRICT-OUT          PICTURE XXX.
00039      05  FILLER                PICTURE XXX.
00040      05  DISTRICT-TOTAL-OUT    PICTURE $$$,$$$,$$9.99.
00041      05  FILLER                PICTURE X(8).
00042      05  FINAL-TOTAL-OUT       PICTURE $$$,$$$,$$9.99.
00043
```

```
00044
00045  WORKING-STORAGE SECTION.
00046
00047  01  FLAGS.
00048    05  MORE-DATA-FLAG          PICTURE XXX   VALUE "YES".
00049        88      MORE-DATA                      VALUE "YES".
00050        88      NO-MORE-DATA                   VALUE "NO ".
00051
00052  01  SAVE-ITEMS.
00053    05  PREVIOUS-SALESMAN       PICTURE X(5).
00054    05  PREVIOUS-DISTRICT       PICTURE XXX.
00055
00056  01  TOTALS COMPUTATIONAL.
00057    05  SALESMAN-TOTAL      PICTURE S9(8)V99 VALUE ZERO.
00058    05  DISTRICT-TOTAL      PICTURE S9(8)V99 VALUE ZERO.
00059    05  FINAL-TOTAL         PICTURE S9(8)V99 VALUE ZERO.

    TWOLEVEAO 0112                                  COBOL

00060

    TWOLEVEAO 0112                                  COBOL

00061  PROCEDURE DIVISION.
00062  PREPARE-SALES-REPORT.
00063    OPEN   INPUT SALES-FILE.
00064      OUTPUT REPORT-FILE.
00065    READ SALES-FILE
00066      AT END MOVE "NO " TO MORE-DATA-FLAG.
00067    MOVE ZERO TO FINAL-TOTAL.
00068    PERFORM DISTRICT-TOTAL-PROCESSING
00069      UNTIL NO-MORE-DATA.
00070    MOVE SPACES TO REPORT-RECORD.
00071    MOVE FINAL-TOTAL TO FINAL-TOTAL-OUT.
00072    WRITE REPORT-RECORD.
00073    CLOSE  SALES-FILE.
00074      REPORT-FILE.
00075    STOP RUN.
00076
00077  DISTRICT-TOTAL-PROCESSING.
00078    MOVE ZERO TO DISTRICT-TOTAL.
00079
00080
00081    MOVE DISTRICT TO PREVIOUS-DISTRICT.
00082    PERFORM SALESMAN-TOTAL-PROCESSING UNTIL
00083      DISTRICT IS NOT EQUAL TO PREVIOUS-DISTRICT
00084      OR NO-MORE-DATA.
00085
00086
00087    MOVE SPACES TO REPORT-RECORD.
00088    MOVE PREVIOUS-DISTRICT TO DISTRICT-OUT.
00089    MOVE DISTRICT-TOTAL TO DISTRICT-TOTAL-OUT.
```

```
00090      WRITE REPORT-RECORD.
00091      ADD DISTRICT-TOTAL TO FINAL-TOTAL.
00092
00093    SALESMAN-TOTAL-PROCESSING.
00094      MOVE ZERO TO SALESMAN-TOTAL.
00095
00096
00097      MOVE SALESMAN TO PREVIOUS-SALESMAN.
00098      PERFORM PROCESS-AND-READ UNTIL
00099        SALESMAN IS NOT EQUAL TO PREVIOUS-SALESMAN
00100        OR DISTRICT IS NOT EQUAL TO PREVIOUS-DISTRICT
00101        OR NO-MORE-DATA.
00102
00103
00104      MOVE SPACES TO REPORT-RECORD.
00105      MOVE PREVIOUS-SALESMAN TO SALESMAN-OUT.
00106      MOVE SALESMAN-TOTAL TO SALESMAN-TOTAL-OUT.
00107      WRITE REPORT-RECORD.
00108      ADD SALESMAN-TOTAL TO DISTRICT-TOTAL.
00109
00110    PROCESS-AND-READ.
00111      ADD SALES-DOLLARS TO SALESMAN-TOTAL.
00112      READ SALES-FILE
00113        AT END MOVE "NO" TO MORE-DATA-FLAG.
  TWOLEVE LENGTH IS  000347
    260200- SCM USED
```

2.4.2.2 CYBER Output

```
 41 $203.37
 57 $110.00
 69 $134.65
                          1 $446.02

 18 $207.69
 32 $185.60
                          2 $393.29

 36 $194.13
 39 $121.40
 50 $51.80
                          3 $367.35

                                    $1,208.66
```

APPENDIX 3

DOCUMENT CONTROL

The information contained in this appendix was extracted from a research paper by Timothy G. Saponas. That research paper served as partial fulfillment of the first year Ph.D. research requirement at Georgia Tech.

Document Control is the Workbench facility that provides for the generation, modification, and general management of all revisions of any document. Among the documents which are serviced by Document Control are COBOL modules, specifications, manuals, catalogs, and reports. Most of the discussion concerning Document Control will refer to COBOL modules, but it is equally applicable to any other document.

3.1. CAPABILITIES

A document, such as a program, may go through many revisions. At any time several of these revisions may be in use. For example, one group of people may be working with program A in production, another group may be correcting known bugs in program A, and still another group may be extending the capabilities of program A. This obviously could result in a very chaotic situation if all three groups are using the same program. Thus, it is necessary for Document Control to maintain and follow a document through all of its revisions. Document Control should be able to create any revision at any point in time. A protection mechanism should be provided by Document Control to protect against accidental tampering. Corrections should be selectively propagated to those revisions which need it, and users of revisions later than those changed should be alerted to the changes. Each revision should contain a release number, version number, date and time of creation, status (production, test, development), and history information (who made the modification, what it consisted of, where the modification occurred, and why the modification was made). Finally, the revision control program should possess useful audit information to aid management in following the progress of a project.

3.2. THE MECHANISM

A good candidate for the Document Control mechanism is described in [Rochkind 1975]. The philosophy of this method is that the text common to more than one revision will not be duplicated. The fundamental atom used to record information is called a delta. Deltas are identified by an ordered pair (i,j) where i refers to release i of a document and j refers to level j of the release. We will denote the delta

corresponding to version j of release i of a document by "Di.j." A delta consists of list a of changes that must be applied to the latest previous revision to obtain the current revision. For example, to obtain the current revision for the situation in figure a.3.2-1, deltas 1.1 through 1.4 must be applied in succession. In all cases delta 1.1 will represent the original document. All changes are expressed in terms of the primitives, insert and delete. Each of these primitives is performed on an entire line of text. For example, to represent the change of one character, one would delete the line containing the character and insert a new line containing the changed character. Experience has shown that the information which is lost by representing all changes as a combination of inserts and deletes is not really essential [Rochkind 1975]. Thus, there is no penalty in constructing a delta from a series of inserts and deletes.

FIGURE a.3.2-1

SEQUENCE OF FOUR DELTAS
INDICATING FOUR REVISIONS OF A MODULE

D1.1---D1.2---D1.3---D1.4

D1.2 represents the changes that must be applied to

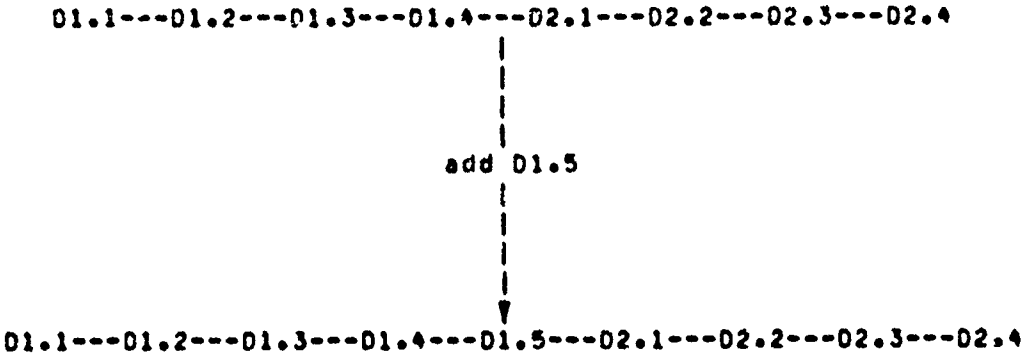
D1.1 to get release 1, version 2 of a document

3.2.1 Adding New Deltas

New deltas can only be applied after the last delta of a release. Thus, if one has a document that has two releases as is the case in the example depicted in figure a.3.2.1-1, and it is desired to make a modification to release 1, then the new delta must be added between delta 1.4 and delta 2.1. This situation could occur when release 1 is being tested by one group and release 2 is being developed by another group. If the first group finds an error in the first release, delta 1.5 must be added to correct the error. Delta 1.5 will not be used to obtain any version of a release after release 1 unless an optional delta as explained below is applied; instead, a warning message will be issued informing the user that a modification to the first release has occurred.

FIGURE a.3.2.1-1

THE RESULTS OF ADDING A REVISION TO RELEASE 1



3.2.2 Special Deltas

In addition to these deltas, one can specify two kinds of special deltas. The first is the optional delta. These are like the regular deltas except that associated with each of these deltas is an option letter. A delta of this type will only be applied if the option letter is presented along with the request for the specified document. This is useful in situations in which one wants to make a change to the code to provide for a special environment experienced by a subset of users but wants this change to be invisible to the rest of the users. The second kind of special delta is used to indicate whether or not previous deltas are to be included or excluded from the revision represented by the special delta. For example, if it is desired to remove delta 1.3, a special delta will be added which will indicate that delta 1.3 is to be excluded from any revision beyond this point in time. This method prevents the risk of accidentally removing code that may be later found desirable. One can also use this special delta to specify the inclusion of previous deltas. In the discussion concerning figure a.3.2.1-1, it was mentioned that delta 1.5 would not be applied to obtain any version in release 2. If the change represented by delta 1.5 is desired in a revision of release 2, one can add a special delta, delta 2.5, that requests the inclusion of delta 1.5.

3.2.3 The Storage of Deltas

All the deltas for a particular document are placed in a data structure which is designed to allow the parallel application of all deltas. Each document (i.e. set of deltas) is stored in a separate sequential file which consists of four parts.

3.2.3.1 Header Information

The first part is the header, which contains release locks, a list of programmers authorized to add deltas, an English description of the module, and any other information that one feels is appropriate for the header. Next in the file is the release table. This contains a count of the number of deltas in each release (this is useful in determining the configuration of storage for the processing of the body to be described below). The release table is followed by the delta table, which contains for each delta the release number, level number, option letter if one exists, date and time it was added, list of other deltas included or excluded by this delta, who created this delta, and why this delta was created. The last part of the file is the body in which the actual deltas are placed.

3.2.3.2 The Main Body

The body contains two types of records, text records, which contain the source code inserted by the deltas, and control records, which specify the effects of each delta. The control record uses three letters (I, D, E) to indicate the action to be taken (insert, delete, end of control). The release and version numbers for which the action is to be applied are also specified on the control record. An example of a body presented in [Rochkind 1975] is depicted in figure a.3.2.3.2-1.

FIGURE a.3.2.3.2-1

EXAMPLE BODY PART OF THE DELTA FILE OF A MODULE

I1.1

I1.4

text of 1.4

E1.4

text of 1.1

D1.2

more text of 1.1

E1.2

I1.2

text of 1.2

D1.3

more text of 1.2

E1.2

more text of 1.1

E1.3

more text of 1.1

E1.1

Notice the bracketing by (I, E) and (D, E) pairs.

3.2.4 Protection

Document Control must also include a method of protection against accidental tampering and change. As mentioned in the discussion above, no delta can be removed physically from the system; instead, a new delta, which indicates that the delta of interest is to be ignored in the production of a revision, must be constructed. Thus, one can be assured that no change can be lost, which implies that any accidental damage done to the module can ultimately be corrected. This then reduces the problem to one of preventing tampering. Obviously, it is impossible to be certain that the changes being made are valid. Thus, protection must be limited to screening the users trying to make changes to modules. This can be implemented by associating a bit matrix with each module [Rochkind 1975]. One dimension represents the names of the programmers allowed to modify the module, and the second dimension represents the revisions of the modules. Thus, each programmer can modify only certain modules and certain revisions within those modules.

3.2.5 Stamping an Identification on a Module

In addition to protection of modules, it is necessary to stamp an identification on the modules. This should include the release number, version number, date and time the module was last modified, the names of the programmers responsible for the module, and the status of the module (production, test, development). It has been suggested that this information be incorporated into the module in such a way that the load module will also contain this information [Rochkind 1975]. In COBOL, this information could be stored in the working storage section. In fact by using a standard macro, this task could be easily performed, and one could be insured of a uniform system of identification.

3.3. PERFORMANCE

This approach to the problem at first glance appears very costly because of the space required, but this anxiety is soon quenched by the following statistics taken from [Rochkind 1975]. The statistics were based on the largest user group on an IBM 370. The group included 100 programmers and 2964 modules. This resulted in 14,455 deltas which occupied 1,016,766 records. There was an average of five deltas per module, but 40% of the modules had only one delta. Of those with more than one delta, the average was 7.5 deltas per module. There were 126 modules that had more than 25 deltas. The number of lines of code resulting from all the modules taken at their latest revision was 740,719. This meant that there was 37% additional space occupied by the deltas, but one must remember that this was obtained by comparing the space occupied by only one revision of each

module. Thus, for 37% additional space, they were able to possess the capability of accessing any revision of any module.

APPENDIX 4

GEORGIA TECH SOFTWARE TOOLS SUBSYSTEM COMMAND INTERPRETER

The Software Tools Subsystem is a set of program development tools based on the book Software Tools by Brian W. Kernighan and P. J. Plauger. It was originally developed for use on the Prime 400 computer in 1977 and 1978 in the form of several cooperating user programs. The present Subsystem, the sixth version, is a powerful tool that aids in the effective use of computing resources.

The command interpreter, also referred to as the shell, is a vital part of the Subsystem. It is a program which accepts commands typed by the user on his terminal and converts them into more primitive directions to the computer itself. The user's instructions are expressed in a special medium called the command language. The next few sections will describe the command language and give examples of how it is used.

Three areas will be covered in the following pages. First, there is a tutorial on the use of the command language. Second, there is a summary of the syntax and semantics of the command language. Lastly, there is a selection of application notes. This section is a good source of useful techniques and samples of advanced usage.

4.1. TUTORIAL

4.1.1 Getting Started

After you have logged in to the computer, you must start up the Subsystem. To do this, type the command "swt":

```
OK, swt
]
```

The Subsystem fires up and the command interpreter prompts you for input by typing a right bracket.

4.1.2 Typographical Conventions

The rules for correcting typos under the Subsystem may vary from system to system. Usually, the Subsystem expects a backspace (control-h) to be used for deleting single characters that are in error, and a DEL (RUBOUT on some terminals) to delete entire lines that are in error.

In the next few sections, references will be made to input lines that are terminated with a "newline." You should use the "newline" key on a terminal only if the terminal lacks a "return" key. They do not necessarily have the same effect.

4.1.3 Commands

Input to the command interpreter consists of "commands." Commands, in turn, consist of a "command name," usually made up of letters and digits, and additional pieces of information, often called "parameters" or "arguments." (Note that a command may or may not have arguments, depending on its function.) The command name and any arguments are separated from each other by spaces.

For example:

```
] echo Hello world!
Hello world!
]
```

The command name is "echo". 'Echo' is not a very involved command; it simply types back its parameters, whatever they may be.

Here is another command, one that is a bit more useful:

```
] lf
adventure       cc              guide           m6800
shell           shell.doc       subsys          time_sheet
words           zunde
]
```

'Lf' is used to list the names of your files.

4.1.4 Special Characters and Quoting

Some characters have special meaning to the command interpreter. For example, try typing this command:

```
] echo Alas, poor Yorick
Alas
poor:  not found
]
```

This strange behavior is caused by the fact that the comma is used for dark mysterious purposes elsewhere in the command language. (The comma actually represents a null I/O connection between nodes of a network. See the section on pipes and networks for more information.) In fact, all of the following characters are potential troublemakers:

, ; # @ > | ( ) [ ] ( ) blank

The way to handle this problem is to use quotes. You may use either single or double quotes, but be sure to match each with another of the same kind. Try this command now:

```
] echo "Alas, poor Yorick; I knew him well."
Alas, poor Yorick; I knew him well.
]
```

You can use quotes to enclose other quotes:

```
] echo 'Quoth the raven: "Nevermore!"'
Quoth the raven: "Nevermore!"
]
```

A final word on quotations: Note that anything enclosed in quotes becomes a single argument. For example, the command

```
] echo "Can I use that in my book?"
```

has only one argument, but

```
] echo Can I use that in my book?
```

has seven.

4.1.5 Command Files

Suppose you have a task which must be done often enough that it is inconvenient to remember the necessary commands and type them in every time. For an example, let's say that you have to print the year-end financial reports for the last five years. If the "print" command is used to print files, your command might look like:

```
] print year73 year74 year75 year76 year77
```

If you use a text editor to make a file named "reports" that contains this command, you can then print your reports by typing

```
] reports
```

No special command is required to perform the operations in this "command file;" simply typing its name is sufficient.

Any number of commands may be placed in a command file. It is possible to set up groups of commands to be repeated or executed only if certain conditions occur. See the Applications Notes for examples.

It is one of the important features of the command interpreter that command files can be treated exactly like ordinary commands. As shown in later sections, they are actually programs written in the command language; many Subsystem commands ("e", "fos", and "rcl", for example) are implemented in this manner.

4.1.6 Doing Repetitive Tasks --- Iteration

Some commands can accept only a single argument. One example of this is the "fos" command. "Fos" stands for "format, overstrike, and spool." It is a shorthand command for printing "formatted" documents on the line printer. (A "formatted" document is one prepared with the help of a program called a "text formatter," which justifies right margins, indents paragraphs, etc. This document was prepared by the Software Tools text formatter, called "fmt.") If you have several documents to be prepared, it is very inconvenient to have to type the "fos" command for each one. A special technique called "iteration" allows you to "factor out" the repeated text. For example,

```
] fos (file1 file2 file3)
```

is equivalent to

```
] fos file1
] fos file2
] fos file3
```

The arguments inside the parentheses form an "iteration group." There may be more than one iteration group in a command, but they must all contain the same number of arguments. This is because each new command line produced by iteration must have one argument from each group. As an illustration of this, note that

```
] (echo print fos) file(1 2 3)
```

is equivalent to

```
] echo file1
] print file2
] fos file3
```

Iteration is performed by a simple text substitution; if there is no space between an argument and an iteration group in the original command, then there is none between the argument and group elements in the new commands. Thus,

```
file(1 2 3)
```

is equivalent to

```
file1
file2
file3
```

Iteration is most useful when combined with function calls, which will be discussed later.

4.1.7 Sources and Destinations of Data

Control of the sources and destinations of data is a very basic function of the command interpreter, yet one that deserves special attention. Every program and command in the Subsystem can gather data from certain well-known places and deliver it to other well-known places. Data sources are known as *standard input ports*; data destinations are called *standard output ports*. Programs are said to "read from standard input" and "write on standard output." The key point here is that programs need not take into account how the input data is made available or what happens to the output data when they are finished with it; the command interpreter is in complete control of the standard ports.

A command we will use frequently in this section is 'copy'. 'Copy' does exactly what its name implies; it copies data from one place to another. In fact, it copies data from its first standard input port to its first standard output port.

The first point to remember is that *by default, standard ports reference the terminal*. Try 'copy' now:

```
) copy
```

After you have entered this command, type some random text followed by a newline. 'Copy' will type the same text back to you. (When you tire of this game, type a control-C; this causes an end-of-file signal to be sent to 'copy', which then returns to the command interpreter. Typing control-C to cause end-of-file is a convention observed by all Subsystem programs.) Since you did not say otherwise, standard input and standard output referred to the terminal; input data was taken from the terminal (as you typed it) and output data was placed on the terminal (printed by 'copy').

Obviously, 'copy' would not be of much use if this was all it could do. Fortunately, the command interpreter can change the sources and destinations of data, thus making 'copy' less trivial.

Standard ports may be altered so as to refer to disk files by use of a *funnel*. The greater-than sign (>) is used to represent a funnel. Conventionally, the ">" points in the direction of data flow. For example, if you wished to copy the contents of file "ee" to file "old_ee", you could type

```
) ee> copy >old_ee
```

The greater-than sign must always be immediately next to its associated filename; no intervening blanks are allowed. At least one blank must separate the ">" from any command name or arguments. This restriction is necessary to insure that the command language can be interpreted unambiguously.

The construct "ee>" is read "from ee"; ">old_ee" is read "toward old_ee." Thus, the command above can be read "from ee copy toward old_ee," or, "copy from ee toward old_ee." The process of changing the file assignment of a standard port by use of a funnel is called "I/O redirection," or simply "redirection."

It is not necessary to redirect both standard input and standard output; either may be redirected independently of the other. For example,

```
) ee> copy
```

can be used to print the contents of file "ee" on the terminal. (Remember that standard output, since it was not specifically redirected, refers to the terminal.) Not surprisingly, the last variation of "copy",

```
) copy >old_ee
```

is also useful. This command causes input to be taken from the terminal (until an end-of-file is generated by typing a control-C) and placed on the file "old_ee". This is a quick way of creating a small file of text without using a text editor.

It is important to realize that all Subsystem programs behave uniformly with regard to redirection. It is as correct to redirect the output of, say, "lf"

```
) lf >file_list
```

as it is to redirect the output of "copy".

Although the discussion has been limited to one input port and one output port up to this point, more of each type are available. In the current implementation, there are a total of six: three for input and three for output. The highest-numbered output port is generally used for error messages, and is often called "errout;" you can "capture" error messages by redirecting this output port. For example, if any errors are detected by "lf" in this command

```
) lf 3>errors
```

then the resulting error messages will be placed on the file "errors."

Final words on redirection: there are two special-purpose redirection operators left. They are both represented by the characters ">>". The first operator is called "append:"

```
) lf >>list
```

causes a list of files to be placed at the end of (appended to) the file named "list". The second operator is called "from command input." It is represented as just >> with no file name, and causes standard input to refer to the current source of commands. It is useful for running programs like the text editor from "scripts" of instructions placed in a command file.

4.1.8 Pipes and Networks

The last section discussed I/O redirection, the process of making standard ports refer to disk files, rather than just to the terminal. This section will take that idea one step further. Frequently, the output of one program is placed on a file, only to be picked up again later and used by another program. The command interpreter simplifies this process by eliminating the intermediate file. The connection between programs that is so formed is called a pipe, and a linear array of programs communicating through pipes is called a pipeline.

Suppose that you maintain a large directory, containing drafts of various manuals. Each draft is in a file with a name of the form "MANxxxx.rr", where "xxxx" is the number of the manual and "rr" is the revision number. You are asked to produce a list of the numbers of all manuals at the first revision stage. The following command will do the job:

```
) lf -c | find .01
```

"lf -c" lists the names of all files in the current directory. The "pipe connection" (vertical bar) causes this listing to be passed to the "find" command, which selects those lines containing the string ".01" and prints them. Thus, the pipeline above will print all filenames matching the conventional form of a first-revision manual name.

The ability to build special purpose commands cheaply and quickly from available tools using pipes is one of the most valuable features of the command interpreter. With practice, surprisingly difficult problems can be solved with ease. For further examples of pipelines, see the Applications Notes.

Combinations of programs connected with pipes need not be linear. Since multiple standard ports are available, programs can be and often are connected in non-linear networks. (Some networks cannot be executed if the programs in the network are not executed concurrently. The command interpreter detects such networks, and prints a warning message if they cannot be performed.) Further information on networks can be found in sections 6.7.2.2, 6.7.2.3, and 6.7.3.1.

4.1.9 Compound Nodes

It is sometimes necessary to change the standard port environment of many commands at one time, for reasons of convenience or efficiency. The "compound node" (a set of networks surrounded by curly braces) can be used in these situations.

As an example of the first case, suppose that you wish to generate a list of manual names (see the last example) in either the first or the second stage of revision. One way to do this is to generate the list for the first revision stage, place it on a file using a funnel, then generate a list for the second revision stage and place it on the end of the same file using an "append" redirector. A compound node might simplify the procedure thusly:

```
] { lf -c | find .01|  lf -c | find .02 )  >list
```

The first network finds all manuals at the first revision stage, and the second finds all those at the second stage. The networks will execute left-to-right, with the output of each being placed on the file "list," thus generating the desired listing. With iteration, the command can be collapsed even farther:

```
] { lf -c | find .0(1 2) )  >list
```

This combination of iteration and compound nodes is often useful.

Efficiency becomes a consideration in cases where successive long streams of data are to be copied onto a file; if the "append" redirector is used each time, the file must be reopened and repositioned several times. Using a compound node, the output file need be opened only once:

```
] { (file1 file2 file3)> copy )  >all_files
```

This complex example copies the contents of files "file1," "file2," and "file3" into the file named "all_files."

4.1.10 Function Calls

Programs in the Subsystem receive information through their command-line arguments as well as their standard ports, so it is sometimes useful to deliver the output of a program to the command interpreter for further processing, rather than to a pipe. The "function call" mechanism is available for this purpose. For example, recall the situation illustrated in the section on pipes and networks; suppose it is necessary to actually print the manuals whose names were found. This is how the task could be done:

```
) print [lf -c | find .01]
```

The function call is composed of the network "lf -c | find .01" and the square brackets enclosing it. The output of the network within the brackets is passed to 'print' as a set of arguments, which it accesses in the usual manner. Specifically, all the lines of output from the network are combined into one set of arguments, with spaces provided where multiple lines have been collapsed into one line.

'Print' accepts multiple arguments; however, suppose it was necessary to use a program like 'fos', that accepts only one argument? Iteration can be combined with a function call to do the job:

```
) fos ([lf -c | find .01])
```

This command formats and prints all manuals in the current directory with revision numbers "01".

Function calls are frequently used in command files, particularly for accessing arguments passed to them. Since the sequence "lf -c | find pattern" occurs very frequently, it is a good candidate for replacement with a command file; it is only necessary to pass the pattern to be matched from the argument list of the command file to the 'find' command with a function call. The following command file, called 'files', will illustrate the process:

```
lf -c | find [arg 1]
```

"arg 1" retrieves the first command file argument. The function call then passes that argument to 'find' through its argument list. 'files' may then be used anywhere the original network was appropriate:

```
) files .01
) print [files .01]
) fos ([files .01])
```

4.1.11 Variables

It has been claimed that the command language is a programming language in its own right. One facet of this language that has not been discussed thus far is the use of its variables. The command interpreter allows the user to create variables, with scope, and assign values to them or reference the values stored in them.

Certain special variables are used by the command interpreter in its everyday operation. These variables have names that begin with the underline (_). One of these is "_prompt", which is the prompt string the command interpreter prints when requesting a command. If you object to "]" as a prompt, you can change it with the "set" command:

```
] set _prompt = "OK, "
OK, set _prompt = "% "
% set _prompt = "] "
]
```

You may create and use variables of your own. To create a variable in the current scope (level of command file execution), use the "declare" command:

```
] declare i j k sum
```

Values are assigned to variables with the "set" command. The command interpreter checks the current scope and all surrounding scopes for the variable to be set; if found, it is changed, otherwise it is declared in the current scope and assigned the specified value.

Variables behave like small programs that print their current values. Thus the value of a variable can be obtained by simply typing its name, or it can be used in a command line by enclosing it in brackets to form a function call. The following command file (which also illustrates the use of "if", "eval", and "goto") will count from 1 to the number given as its first argument:

```
declare i
set i = 1
:loop
   if [eval i ">" [arg 1]]
      goto exit
   fi
   i
   set i = [eval i + 1]
   goto loop
:exit
```

Note the use of the "eval" function, which treats its arguments as an arithmetic expression and returns the expression's value. This is required to insure that the string "i

• 1" is interpreted as an expression rather than as a character string. Also note that "fi" terminates the "if" command.

In the future, typed variables and better control structures will be added to the command interpreter.

4.1.12 Conclusion

This concludes the tutorial sections for the command interpreter. Despite the fact that a good deal of material has been presented, much detail has been omitted. The next next few sections include a complete summary of the capabilities of the command interpreter and some examples which may prove helpful.

4.2. SUMMARY OF SYNTAX AND SEMANTICS

This section is the definitive document for the syntax and corresponding semantics of the Software Tools Subsystem Command Interpreter. It is composed of several sub-sections, each covering some major area of command syntax, with discussions of the semantic consequences of employing particular constructs. It is not intended as a tutorial, nor is it intended to supply multitudinous examples; other sections in this document are provided to fill those needs.

4.2.1 Commands

<command> ::= [<net> (; <net>)] <newline>

The command is the basic unit of communication between the command interpreter and the user. It consists of any number of networks (described below) separated by semicolons and terminated by a newline. The networks are executed one at a time, left-to-right; should an error occur at any point in the parse or execution of a network, the remainder of the <command> is ignored. The null command is legal, and causes no action.

The command interpreter reads commands for interpretation from the command source. This is initially the user's terminal, although execution of a command file may change the assignment. Whenever the command source is the terminal, and the command interpreter is ready for input, it prompts the user with the string contained in the shell variable "_prompt". Since this variable may be altered by the user, the prompt string is selectable on a per-user basis.

4.2.2 Networks

<net> ::= <node>
(<node separator> (<node separator>) <node>)

<node separator> ::= , | <pipe connection>

<pipe connection> ::= [<port>] '|' [<node number>] [.<port>]

<port> ::= <integer>

<node number> ::= <integer> | $ | <label>

A <net> generates a block of (possibly concurrent) processes that are bound to one another by channels for the flow of data. Typically, each <node> corresponds to a single process. (<Node>s are described in more detail below.) There is no predefined "execution order" of the processes composing a <net>; the command interpreter will select any order it sees fit in order to satisfy the required input/output relations. In particular, the user is specifically enjoined not to assume a left-to-right serial execution, since some <net>s cannot be executed in this manner.

Input/output relations between <node>s are specified with the <node separator> construct. The following discussion may be useful in visualizing the data flows in a <net>, and clarifying the function of the components of the <node separator>.

The entire <net> may be represented as a directed graph with one vertex for each <node> (typically, equivalent to each process) in the net. Each vertex may have up to n arcs terminating at it (representing input data streams), and m arcs originating from it (representing output data streams). An arc between two vertices indicates a flow of data from one <node> to another, and is physically implemented by a pipe.

Each of the n possible input points on a <node> is assigned an identifier consisting of a unique integer in the range 1 to n. These identifiers are referred to as the port numbers for the standard input ports of the given <node>. Similarly, each of the m possible output points on a <node> is assigned a unique integer in the range 1 to m, referred to as the port numbers for the standard output ports of the given <node>.

Lastly, the <node>s themselves are numbered, starting at 1 and increasing by 1 from the left end of the <net> to the right.

Clearly, in order to specify any possible input/output connection between any two <node>s, it is sufficient to specify:

1. The number of the "source" <node>.

2. The number of the "destination" <node>.

3. The port number of the standard output port on the source <node> that is to be the source of the data.

4. The port number of the standard input port on the destination <node> that is to receive the data.

The syntax for <node separator> includes the specifications for the last three of these items. The source <node> is understood to be the node that immediately precedes the <node separator> under consideration. The special <node separator> "," is used to separate <node>s that do not participate in data sharing; it specifies a null connection. Thus, the <node separator> provides a means of establishing any possible connection between two <node>s of a given <net>.

The full flexibility of the <node separator> is rarely needed or desirable. In order to make effective use of the capabilities provided, suitable defaults have been designed into the syntax. The semantics associated with the defaults are as follows:

1. If the output port number (the one to the left of the vertical bar) is omitted, the next unassigned output port (in increasing numerical order) is implied. This default action takes place only after the entire <net> has been examined, and all non-defaulted output ports for the given node have been assigned. Thus, if the first <node separator> after a <node> has a defaulted output port number, port 1 will be assigned if and only if no other <node separator> attached to that <node> references output port 1. It is an error for two <node separator>s to reference the same output port. (This particular behavior may be changed in the future to allow "forking" output streams, which would be copied to more than one destination.)

2. If the destination <node> number is omitted, then the next node in the <net> (scanning from left to right) is implied. Frequently a null <node> is generated at the end of a

<net> because of the necessity for resolving such references.

3. If the destination <node>'s input port number is omitted, then the next unassigned input port (in increasing numerical order) is implied. As with the defaulted output port, this action takes place only after the entire <net> has been examined. The comments under (1) above also apply to defaulted input ports.

In addition to the defaults, specifying input/output connections between widely separated <node>s is aided by alternative means of giving <node> numbers. The last <node> in a <net> may be referred to by the <node number> $, and any <node> may be referred to by an alphanumeric <label>. (<Node> labelling is discussed in the section on <node> syntax, below.) If the first <node> of a <net> is labelled, the <net> may serve as a target for the 'goto' command; see the Applications Notes for examples.

As will be seen in the next section, further syntax is necessary to completely specify the input/output environment of a <node>; the reader should remember that <node separator>s control only those flows of data between processes.

A few examples of the syntax presented above may help to clarify some of the semantics. Since the syntax of <node> has not yet been discussed, <node>s will be represented by the string "node" followed by a digit, for uniqueness and as a key to <node number>s.

A simple linear <net> of three <node>s without defaults:

```
node1  1|2.1  node2  1|3.1  node3
```

(Data flows from output port 1 of node1 to input port 1 of node2 and output port 1 of node2 to input port 1 of node3.)

The same <net>, with defaults:

```
node1 | node2 | node3
```

(Note that the spaces around the vertical bars are mandatory, so that the lexical analysis routines of the command interpreter can parse the elements of the command unambiguously.)

A simple cycle:

```
node1 |1.2
```

(Data flows from output port 1 of node1 to input port 2 of node1. Other data flows are unspecified at this level.)

A branching <net> with resolution of defaults:

```
node1  |$  node2  |.1  node3
```

(Data flows from output port 1 of node1 to input port 2 (!) of node3 and output port 1 of node2 to input port 1 of node3.)

A.2.3 Nodes

```
<node> ::=  :<label> [ <simple node> | <comp.  d node> ] |
            ( <simple node> | <compound node>

<simple node> ::= ( <i/o redirector> ) <command name>
                  ( <i/o redirector> | <argument> )

<compound node> ::= ( <i/o redirector> )
                    *(* <net> ( <net separator> <net> ) *)*
                    ( <i/o redirector> )

<i/o redirector> ::= <file name> *>*  [ <port> ]    |
                     [ <port> ]  *>*  <file name>   |
                     [ <port> ]  *>>* <file name>   |
                                 *>>* [ <port> ]

<net separator> ::=  !

<command name> ::= <file name>

<label> ::= <identifier>
```

The <node> is the basic executable element of the command language. It consists of a label (string of letters, digits, and underscores, beginning with a letter), or an optional label followed by one of two additional structures.

The first option is the <simple node>. It specifies the name of a command to be performed, any arguments that command may require, and any <i/o redirector>s that will affect the data environment of the command in addition to <pipe connection>s in any <nets> containing the <simple node>. (<i/o redirectors will be discussed below.) The execution of a simple node normally involves the creation of a single process, which performs some function, then returns to the operating system.

The second option is the <compound node>. It specifies a <net> which is to be executed according to the usual rules of <net> evaluation (see the previous section), and any <i/o redirector>s that should affect the environment of the <net>. The <compound node> is provided for two reasons. One, it is occasionally useful to alter default port assig-

nments for an entire <net> with <i/o redirector>s, rather than supplying <i/o redirector>s for each <node>. Two, use of compound nodes containing more than one <net> gives the user some control over the order of execution of his processes. These abilities are discussed in more detail below.

Since it is the more basic construct, consider the <simple node>. It consists of a <command name> with <argument>s, intermixed with <i/o redirector>s. The <command name> must be a filename, usually specifying the name of an object code file to be loaded. The command interpreter locates the command to be performed by use of a user-specified search rule. The search rule resides in the shell variable "_search_rule", and consists of a series of comma-separated elements. Each element is either a template in which ampersands (&) are replaced by the <command name> or a flag instructing the command interpreter to search one of its internal tables. The flag "^int" indicates that the command interpreter's repertoire of internal commands is to be checked. (An internal command is implemented as a subroutine of the command interpreter, typically for speed or because of a need to access some private data base.) The flag "^var" causes a search of the user's shell variables (see below for further discussion of variables and functions). The following search rule will cause the command interpreter to search for a command among the internal commands, shell variables, and the directory /bin, in that order:

"^int,^var,/bin/&"

The purpose of the search rule is to allow optimization of command location for speed, and to admit the possibility of restricting some users from accessing "privileged" commands. (For example, the search rule

"^var,/project/library/&"

would restrict a user to accessing his variables and those commands in the directory /project/library. He could not alter this restriction, since he does not have access to the (internal) 'set' command; the "^int" flag is missing from his search rule.)

<Argument>s to be passed to the program being readied for execution are gathered by the command interpreter and placed in an area of memory accessed by the library routine 'getarg'. They may be arbitrary strings, separated from the command name and from each other by blanks. Quoting may be necessary if an <argument> could be interpreted as some other element of the command syntax. Either single or double quotes may be used. The appearance of two strings adjacent to one another without blanks implies concatenation. Thus,

"quoted "string

is equivalent to

"quoted string"

or to

quoted' string'

Single quotes may appear within strings delimited by double quotes, and vice versa; this is the only way to include quotes within a string. Example:

"'quoted string'"
'"Alas, poor Yorick!"'

Arguments are generally unprocessed by the command interpreter, and so may contain any information useful to the program being invoked.

In the previous section, it was shown that streams of data from "standard ports" could be piped from program to program through the use of the <pipe connection> syntax. It is also possible to redirect these data streams to files, or to use files as sources of data. The construct that makes this possible is the <i/o redirector>. The <i/o redirector> is composed of filenames, port numbers (as described in the last section), and one or more occurrences of the "funnel" (>).

The two simplest forms take input from a file to a standard port or output from a standard port to a file. In the case of delivering output to a file, the file is automatically created if it did not exist, and overwritten if it did. In the case of taking input from a file, the file is unmodified. Example:

documentation>1

causes the data on the file "documentation" to be passed to standard input port 1 of the node:

1>results

causes data written to standard output port 1 of the node to be placed on the file "results".

If no <i/o redirector> is present for a given port, then that port automatically refers to the user's terminal.

If port numbers are omitted, an assignment of defaults is made. The assignment rule is identical to that given above for <pipe connections>: the first available port after the entire <net> has been scanned is used. <I/O redirector>s are evaluated left-to-right, so leftmost defaulted redirectors are assigned to lower-numbered ports than those to their right. For example,

```
data> requests>  trans  2>summary 3>errors | spool
```

is the same as

```
data>1 requests>2 trans 2>summary 3>errors 1|2.1 spool
```

where all defaults have been elaborated. 'Trans' might be some sort of transaction processor, accepting data input and update requests, and producing a report (here printed off-line by being piped to a spooler program), a summary of transactions, and an error listing.

In addition to the <i/o redirector>s mentioned above, there are two lesser-used redirectors that are useful. The first appends output to a file, rather than overwriting the file. The syntax is identical to the other output redirector, with the exception that two funnels '>>' are used, rather than one. For example,

```
6>>stuff
```

causes the data written to output port 6 to be appended to the file "stuff". (Note the lack of spaces around the redirector: a redirector and its parameters are never separated from one another, but are always separated from surrounding arguments or other text. This restriction is necessary to insure unambiguous interpretation of the redirector.) The second redirector causes input to be taken from the current command source file. It is most useful in conjunction with command files. The syntax is similar to the input redirector mentioned above, but two funnels are used and no filename may be specified. As an example, the following segment of a command file uses the text editor to change all occurrences of "March" to "April" in a given file:

```
>> ed file
g/March/s//April/
w
q
```

When the editor is invoked, it will take input directly from the command file, and thus it will read the three commands placed there for it.

The "command source" and "append" redirectors are subject to the same resolution of defaults as the other redirectors and <pipe connection>s. Thus, in the example immediately above,

```
>> ed file
```

is equivalent to

```
>>1 ed file
```

Now that the syntax of <node> has been covered, just two further considerations remain. First, the nature of an executable program must be defined. Second, the problem of execution order must be clarified.

In the vast majority of cases, a <node> is executed by bringing an object program into memory and starting it. However, the <command name> may also specify an internal command, a shell variable, or a command file. Internal commands are executed within the command interpreter by the invocation of a subroutine. When a shell variable is used as a command, the net effect is to print the value of the variable on the first output port, followed by a newline. If the filename specified is a text file rather than an object file, the command interpreter "guesses" that the named file is a file of commands to be interpreted one at a time. In any case, command invocation is uniform, and any <i/o redirector> or <pipe connection> given will be honored. Thus, it is allowable to redirect the output of a command file, just as if it were an object program, or copy a shell variable to the line printer by connecting it to the spooler through a pipe.

As mentioned in the section on <net>s, the execution order of nodes in a <net> is undefined. That is, they may be executed serially in any order, concurrently, or even simultaneously. The exact method is left to the implementor of the command interpreter. In any case, the flows of data described by <pipe connection>s and <i/o redirector>s are guaranteed to be present. There are times when it would be preferable to know the order in which a <net> will be evaluated; to help with this situation, <compound node>s may be used to effect serialization of control flow within a network. <Net>s separated by semicolons or newlines are guaranteed to be executed serially, left-to-right, otherwise the command interpreter would exhibit unpredictable behavior as the user typed in his commands. Suppose it is necessary to operate four programs; three may proceed concurrently to make full use of the multiprogramming capability of the computer system, but the fourth must not be executed until the second of the three has terminated. For simplicity, we will assume there are no input/output connections between the programs. The following command line meets the requirements stated above:

```
program1, (program2; program4), program3
```

(Recall that the comma represents a null i/o connection.) Suppose that we have a slightly different problem: the fourth program must run after <u>all</u> of the other three had run to completion. This, too, can be expressed concisely:

```
program1, program2, program3; program4
```

Thus, the user has fairly complete control over the execu-

tion order of his <net>s. (The use of commas and semicolons in the command language parallels their use for collateral and serial elaboration in Algol 68.)

This completes the discussion of the core of the command language. The remainder of the features present in the command interpreter are provided by a built-in preprocessor, which handles function calls, iteration, and comments. The next few sections deal with the preprocessor's capabilities.

4.2.4 Comments

Any good command language should provide some means for the user to comment his code, particularly in command files that may be used by others. The command interpreter has a simple comment convention: Any text between an unquoted sharp sign (#) and the next newline is ignored. A comment may appear at the beginning of a line, like this:

```
# command file to preprocess, compile, and link edit
```

Or after a command, like this:

```
file.r> rf    # Ratfor's output goes to the terminal
```

Or even after a label, for identification of a loop:

```
:loop      # beginning of daily cycle
```

As far as implications in other areas of command syntax, the comment is functionally equivalent to a newline.

4.2.5 Variables

<variable> ::= <identifier>

<set command> ::= set [<variable>] = [<argument>]

<declare command> ::= declare (<variable [= <argument>])

<forget command> ::= forget <variable> (<variable>)

The command interpreter supports named string storage areas for miscellaneous user applications. These are called variables. Variables are identified by a name, consisting of letters of either case, digits, and underscores, not beginning with a digit. Variables have two attributes: value and scope. The value of a variable may be altered with the 'set' command, discussed below. The scope of a variable is fixed at the time of its creation; simply, variables declared during the time when the command

interpreter is taking input from a command file are active as long as that file is being used as the command source. Variables with global scope (those created when the command interpreter is reading commands from the terminal) are saved as part of the user's profile, and so are available from terminal session to terminal session. Other variables disappear when the execution of the command file in which they were declared terminates.

Variables may be created with the 'declare' command. 'Declare' creates variables with the given names at the current lexical level (within the scope of the current command file). The newly-created variables are assigned a null value, unless an initialization string is provided.

Variables may be destroyed prematurely with the 'forget' command. The named variables are removed from the command interpreter's symbol table and storage assigned to them is released to the system. Note that variables created by operations within a command file are automatically released when that command file ceases to execute. Also note that the only way to destroy variables at the global lexical level is to use the 'forget' command.

The value of a variable may be changed with the 'set' command. The first argument to 'set' is the name of the variable to be changed. If absent, the value that would have been assigned is printed on 'set's first standard output. The last argument to 'set' is the value to be assigned to the variable. It is uninterpreted, that is, treated as an arbitrary string of text. If missing, 'set' reads one line from its first standard input, and assigns the resulting string. If the variable named in the first argument has not been declared at any lexical level, 'set' declares it at the current lexical level.

Variables are accessed by name, as with any command. (Note that the user's search rule must contain the flag ""var" before variables will be evaluated.) The command interpreter prints the value of the variable on the first standard output. This behavior makes variables useful in function calls (discussed below). In addition, the user may obtain the value of a variable for checking simply by typing its name as a command.

4.2.6 Iteration

<iteration> ::= '(' <element> (<element>) ')'

Iteration is used to generate multiple command lines each differing by one or more substrings. Several iteration elements (collectively, an "iteration group") are placed in parentheses; the command interpreter will then generate one

command line for each element, with successive elements replacing the instance of iteration. Iteration takes place over the scope of one <net>; it will not extend over a <net separator>. (If iteration is applied to a <compound node>, it will, of course, apply to the entire <node>; not just to the first <net> within that <node>.)

Multiple iterations may be present on one command; each iteration group must have the same number of elements, since the command interpreter will pick one element from each group for each generated command line. (Cross-products over iteration groups are not implemented.)

An example of iteration:

```
] fos part(1 2 3)

   is equivalent to

] fos part1; fos part2; fos part3

   and

] cp (intro body summary) part(1 2 3)

   is equivalent to

] cp intro part1; cp body part2; cp summary part3
```

4.2.7 Function Calls

<function call> ::= '[' <net> (<net separator> <net>) ']'

Occasionally it is useful to be able to pass the output of a program along as arguments to another program, rather than to an input port. The function call makes this possible. The output appearing on each of the first standard output ports of the <net>s within the function call is copied into the command line in place of the function call itself. Line separators (newlines) present in the <net>'s output are replaced by blanks. No quoting of <net> output is performed, thus blank-separated tokens will be passed as separate arguments. (If quoting is desired, the filter 'quote' can be used or the shell variable "_quote_opt" may be set to the string "YES" to cause automatic quotation.)

A <net> may of course be any network; all the syntax described in this document is applicable. In particular, the name of a variable may appear with the brackets; thus, the value of a variable may be substituted into the command line.

4.2.8 Conclusion

This concludes the description of command syntax and semantics. The next section contains actual working examples of the full command syntax, along with suggested applications.

4.3. APPLICATION NOTES

The next few sections consist mostly of examples of current usage of the command interpreter. Extensive knowledge of some Subsystem programs may be necessary for complete understanding of these examples, but basic principles should be clear without this knowledge.

4.3.1 Basic Functions

In this section, some basic applications of the command language will be discussed. These applications are intended to give the user a "feel" for the flow of the language, without being explicitly pedagogical.

One commonly occurring task is the location of lines in a file that match a certain pattern. The 'find' command performs this function:

```
] file> find pattern >lines_found
```

Since the lines to be checked against the pattern are frequently a list of file names, the following sequence occurs often:

```
] lf -c directory | find pattern
```

Consequently, a command file named 'files' is available to abbreviate the sequence:

```
] cat files
lf -c [args 2] | find [arg 1]
```

('Cat' is used here only to print the contents of the command file.) The internal command 'arg' is used to fetch the first argument on the command line that invoked 'files'. Similarly, the internal command 'args' fetches the second through the last arguments on the command line. The command file gives the external appearance of a program 'files' such that

```
] files pattern
```

is equivalent to

```
] lf -c | find pattern
```

and

```
] files pattern directory
```

is equivalent to

```
] lf -c directory | find pattern
```

Once a list of file names is obtained, it is frequently processed further, as in this command to print Ratfor source files on the line printer:

```
] pr [files .r$ | sort]
```

'Files' produces a list of file names with the ".r" suffix, which is then sorted by 'sort'. 'Pr' then prints all the named files on the line printer.

One problem arises when the pattern to be matched contains command language metacharacters. When the pattern is substituted into the network within 'files', and the command interpreter parses the command, trouble of some kind is sure to arise. There are two solutions: One, the filter 'quote' can be used to supply a layer of quotes around the pattern:

```
lf -c [args 2] | find [arg 1 | quote]
```

Two, the shell variable '_quote_opt', which controls automatic function quotation by the command interpreter, can be set to the string "YES":

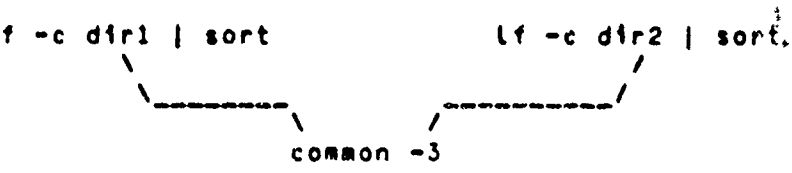
```
declare _quote_opt = YES
lf -c [args 2] | find [arg 1]
```

This latter solution works only because 'args' prints each argument on a separate line; the command interpreter always generates separate arguments from separate lines of function output. In practice, the first solution is favored, since the non-intuitive quoting is made more evident.

One common non-linear command structure is the so-called "Y" structure, where two streams of data join together to form a third (after some processing). This situation occurs because of the presence of dyadic operations (especially comparisons) in the tools available under the Subsystem. As an example, the following command compares the file names in two directories and lists those names that are present in both:

```
] lf -c dir1 | sort |$ lf -c dir2 | sort | common -3
```

Visualize the command in this way:



The two 'lf' and 'sort' pairs produce lists of file names that are compared by 'common', which produces a list of those names common to both input lists.

Command files tend to be used not only for often performed tasks but also to make life easier when typing long, complex commands. Quite often these long command lines make use of line continuation -- a newline preceeded immediately by an underscore is ignored. The following command file is used to create a keyword-in-context index from the heading lines of the Subsystem Reference Manual. Although it is not used frequently, it does a great deal of work and is illustrative of many of the features of the command interpreter.

```
* make_cmd.k --- build permuted index of commands
   files /doc_? -f commands_files _
      | change % "find %.hd " _
      | sh _
      | change '%.hd  *([^ ]*) [ ]*([^ ]*)?*' '$1: $2' _
      | kwic -d /extra/spelling/discard _
      | sort -d | unrot -w [width]  >cmd.k
```

First a few words on how Subsystem documentation is stored: The documentation for Subsystem commands resides in a subdirectory named "commands_files". The documentation for each command is in a separate file with the name doc_<command>. The heading line in each file can be identified by the characters ".hd" at the beginning of the line.

The entire command file consists of a single network. The 'files' command produces a list of the full path names (the -f option is passed on to 'lf') of the files in the subdirectory "commands_files" that have path names containing the characters "/doc_" followed by at least one additional character. The next 'change' command generates a 'find' command for each documentation file to find the heading line. These command lines are passed back to the shell ('sh') for execution. The outputs of all of these 'find' commands, namely the heading lines from all the documentation files, is passed back on the first standard output of 'sh'. The second 'change' command uses tagged patterns to isolate the command name and its short description from the header line and to construct a suitable entry for the kwic index generator. Finally, 'kwic', 'sort', and 'unrot' produce the index on the file "cmd.k".

To this point, only serially-executed commands have been discussed, however sophisticated or parameterized. Control structures are necessary for more generally useful applications. The following command file, 'ssr', shows a useful technique for parameter-setting commands. Like many APL system commands, 'ssr' without arguments prints the value it controls (in this case, the user's command search rule), while 'ssr' with an argument sets the search rule to the argument given, then prints the value for verification. 'Ssr' looks like this:

```
# ssr --- set user's search rule and print it

   if [nargs]
      set _search_rule = [arg 1 | quote]
   fi

   _search_rule
```

The 'if' command conditionally executes other commands. It requires one argument, which is interpreted as "true" if it is present, not null, and non-zero. If the argument is true, all the commands from the 'if' to the next unmatched 'else' or 'fi' command are executed. If the argument is false, all the commands from the next unmatched 'else' command (if one is present) to the next unmatched 'fi' command are executed. In 'ssr' above, the argument to 'if' is a function call invoking 'nargs', a command that returns the number of arguments passed to the command file that is currently active. If 'nargs' is zero, then no arguments were specified, and 'ssr' does not set the user's search rule. If 'nargs' is nonzero, then 'ssr' fetches the first argument, quotes it to prevent the command interpreter from evaluating special characters, and assigns it to the user's search rule variable '_search_rule'.

'If' is useful for simple conditional execution, but it is often necessary to select one among several alternative actions instead of just one from two. The 'case' command is available to perform this function. The premier example of 'case' is the command file 'e', which is used to invoke either the screen editor or the line editor depending on which terminal is being used (as well as remembering the name of the file last edited):

```
# e --- invoke the editor best suited to a terminal

   if [nargs]
      set f = [arg 1 | quote]
   fi

   case [line]
      when 10
         se -t consul [se_params] [f]
      when 11
         se -t b200   [se_params] [f]
      when 15
         se -t b150   [se_params] [f]
      when 17
         se -t gt40   [se_params] [f]
      when 18
         se -t b200   [se_params] [f]
      when 25
         se -t b150   [se_params] [f]
      out
         ed [f]
   esac
```

The first 'if' command sets the remembered file name (stored in the shell variable 'f') in the same fashion that 'ssr' was used to set the search rule (above). The 'case' command then selects from the terminals it recognizes and invokes the proper text editor. The argument of 'case' is compared with the arguments of successive 'when' commands until a match occurs, in which case the group of commands after the 'when' is executed; if no match occurs, then the commands after the 'out' command will be executed. (If no 'out' command is present, and no match occurs, then no action is taken as a result of the 'case'.) The 'esac' command marks the end of the control structure. In 'e', the 'case' command selects either 'se' (the screen editor) or 'ed' (the line editor), and invokes each with the proper arguments (in the case of 'se', identifying the terminal type and specifying any user-dependent personal parameters).

The 'goto' command may be used to set up a loop within a command file. For example, the following command file will count from 1 to 10:

```
# bogus command file to show computers can count

   declare i = 1

   :loop
      i
      set i = [eval i + 1]
      if [eval i <= 10]
         goto loop
      fi
```

In actual experience, little need has been found for loops

within command files; thus the need for this contrived example.

4.3.2 Shell Control Variables

Several special shell variables are used to control the operation of the command interpreter. The following table identifies these variables and gives a short explanation of the function of each.

Variable	Function
_ci_name	This variable is used to select a command interpreter to be executed when the user enters the Subsystem. It should be set to the full pathname of the command interpreter desired. The default value is "/bin/sh".
_erase	This variable may be set to a single character or an ASCII mnemonic for a character to be used as the "erase," or character delete, control character for Subsystem terminal input processing. The change in erase character becomes effective only after the Subsystem is re-entered and the initialization routines read the shell variable storage file.
_hello	This variable, if present, is used as the source of a command to be executed whenever the user enters the Subsystem. It is frequently used to implement memo systems, supply system status information, and print pleasing messages-of-the-day.
_kill	This variable may be set to a single character or an ASCII mnemonic for a character to be used as the "kill," or line delete, control character for Subsystem terminal input processing. The change in kill character becomes effective only after the Subsystem is re-entered and the initialization routines read the shell variable storage file.
_prompt	This variable contains the prompt string printed by the command interpreter before any command read from the user's terminal. The default value is a right bracket (]).
_quote_opt	This variable, if set to the value "YES", causes automatic quotation of each line of program output used in a function call. It is mainly provided for compatibility with an older version of the command interpreter, which performed the quoting automatically. The program 'quote' may be used to explicitly

force quotation.

_search_rule This variable contains a sequence of comma-separated elements that control the procedure used by the command interpreter to locate the object code for a command. Each element is either (1) the flag "^int", meaning the command interpreter's table of internal commands, (2) the flag "^var", meaning the user's shell variables, or (3) a template containing the character ampersand (&), meaning a particular directory or file in a directory. In the last case, the command name specified by the user is substituted into the template at the point of the ampersand, hopefully providing a full pathname that locates the object code needed.

4.3.3 Conclusion

This concludes the Application Notes section.

APPENDIX 5

EDITOR
GEORGIA TECH SOFTWARE TOOLS SUBSYSTEM

The Software Tools Subsystem provides two editors, 'ed' and 'se.' The major difference between the two editors is that 'se' allows the user to view a window of text during the editing session. To enhance the window feature, 'se' also provides some additional capabilities, for example special characters for controlling the position of the cursor. The basic editing commands available on both editors, though, are identical.

5.1. ED

'Ed' is an interactive program that can be used for the creation and modification of "text". "Text" may be any collection of character data, such as a report, a program, or data to be used by a program. The nature of the next few sections is that of a tutorial, and as such, a step-by-step journey through an editing session.

5.1.1 Starting an Editing Session

Since you are in the Subsystem, the command interpreter should have just printed the prompt "]". To enter the text editor, type

```
] ed          (followed by a newline)
```

(Throughout this introduction, boldface is used to indicate information typed by the user.) You are now in the editor, ready to go. Note that 'ed' does not print any prompting information; this quiet behavior is preferred by experienced users. (If you would like a prompt, it can be provided; try the command "op/prompt/".)

At this point, 'ed' is waiting for instructions from you. You can instruct 'ed' by using "commands", which are single letters (occasionally accompanied by other information, which you will see shortly).

5.1.2 Entering Text - the Append Command

The first thing that you will need is text to edit. Working with 'ed' is like working with a blank sheet of paper; you write on the paper, alter or add to what you have written, and either file the paper away for further use or throw it away. In 'ed' terminology, the blank sheet of paper you start with is called a "buffer." The buffer is empty when you start editing. All editing operations take place in the

buffer! nothing you do can affect any file unless you make an explicit request to transfer the contents of the buffer to a file.

So the first problem reduces to finding a way to put text into the buffer. The "append" command is used to do this:

```
a
```

This command appends (adds) text lines to the buffer, as they are typed in.

To put text into the buffer, simply type it in, terminating each line with a newline:

```
The quick brown fox
    jumps over
  the lazy dog.
.
```

To stop entering text, you must enter a line containing only a period, immediately followed by a newline, as in the last line above. This tells 'ed' that you are finished writing on the buffer, and are ready to do some editing.

The buffer now contains:

```
The quick brown fox
    jumps over
  the lazy dog.
```

Neither the append command nor the final period are included in the buffer - just the text you typed in between them.

5.1.3 Writing text on a file - the Write command

Now that you have some text in the buffer, you need to know how to save it. The write command "w" is used for this purpose. It is used like this:

```
w file
```

where "file" is the name of the file used to store what you just typed in. The write command copies the contents of the buffer to the named file, destroying whatever was previously in the file. The buffer, however, is untouched; whatever you typed in is still there. To indicate that the transfer of data was successful, 'ed' types out the number of lines written. In this example, 'ed' would type:

```
57
```

It is advisable to write the contents of the buffer out to a file periodically, to insure that you will have an up-to-date version in case of some terrible catastrophe (like a

system crash).

5.1.4 Finishing up - the Quit command

Now that you have saved your text in a file, you may wish to leave the editor. The "quit" command "q" is provided for this:

```
q
```

The next thing you see should be the "]" prompt from the Subsystem command interpreter. If you did not write out the contents of the buffer, the editor will respond:

```
?
(not saved)
```

This is to remind you to write out the buffer, so that the results of your editing session will not be lost. If you intended that the buffer be discarded, just enter "q" again and "ed" will throw away the buffer and terminate.

When you receive the "]" prompt from the Subsystem command interpreter, the buffer has been thrown away; there is absolutely no way to recover it. If you wrote the contents of the buffer to a file, then this is of no concern; if you did not, it may mean disaster.

To check if the text you typed in is really in the file you wrote it to, try the following command:

```
] cat file
```

where "file" is the name of the file given with the "w" command. ("Cat" is a Subsystem command that can be used to print files on the terminal. If, for example, you wished to print your file on the line printer, you could say:

```
] pr file
```

and the contents of "file" would be queued for printing.)

5.1.5 Reading files - the Enter command

Of course, most of the time you will not be entering text into the buffer for the first time. You need a way to fill the buffer with the contents of some file that already exists, so that you can modify it. This is the purpose of the "enter" command "e"; it enters the contents of a file into the buffer. To try out "enter," you must first get back into the editor:

```
] ed
```

"Enter" is used like this:

```
e file
```

"File" is the name of a file to be read into the buffer.

Note that you are not restricted to editing files in the current directory; you may also edit files belonging to other users (provided they have given you permission). Files belonging to other users must be identified by their "full pathname" (discussed fully in The Primos File System -- An Overview). For example, to edit a file named "document" belonging to user "tom," you would enter the following command:

```
e /tom/document
```

After the file's contents are copied into the buffer, 'ed' prints the number of lines it read. In our example, the buffer would now contain:

```
The quick brown fox
    jumps over
  the lazy dog.
```

If anything at all is present in the buffer, the "e" command destroys it before reading in the named file.

As a matter of convenience, 'ed' remembers the file name specified on the last "e" command, so you do not have to specify a file name on the "w" command. With these provisions, a common editing session looks like

```
] ed
e file
[editing]
w
q
```

The "file" command ("f") is available for finding out the remembered file name. To print out the name, just type:

```
f
```

You might also want to check that

```
] ed file
```

is exactly the same as

```
] ed
e file
```

That is, 'ed' will perform an "e" command for you if you

give it a file name on the command line.

5.1.6 Errors - the Query command

Occasionally, an error of some kind will be encountered. Usually, these are caused by misspelled file names, although there are other possibilities. Whenever an error occurs, 'ed' types

```
?
```

Although this is rather cryptic, it is usually clear what caused the problem. If you need further explanation, just enter "?" and 'ed' will respond with a one-line explanation of the error. For example, if the last command you typed was an "e" command, 'ed' is probably saying that it could not find the file you asked for. You can find out for sure by entering "?":

```
e myfile
?
?
I can't open the file to read
```

Except for the messages in reponse to "?", 'ed' rarely gives other, more verbose error messages; if you should see one of these, the best course of action is to report it to someone who maintains the editor.

5.1.7 Printing text - the Print command

You are likely to need to print the text you have typed in, to check it for accuracy. The "print" command "p" is available to do this. "p" is different from the commands seen thus far; "e", "w", and "a" have been seen to work on the whole buffer at once. For a small file, it might be easiest to print the entire buffer just to check on some few lines, but for very large files this is clearly impractical. The "p" command therefore accepts "line numbers" that indicate which lines to print. Try the following experiment:

```
) ed file
1p
The quick brown fox
3p
   the lazy dog.
1,2p
The quick brown fox
    jumps over
1,3p
The quick brown fox
    jumps over
   the lazy dog.
```

"1p" tells 'ed' to print line 1 ("The quick brown fox"). "3p" says to print the third line ("the lazy dog."). "1,2p" tells 'ed' to print the first <u>through</u> the second lines, and "1,3p" says to print the first <u>through</u> the third lines. Suppose we want to print the last line in the buffer, but we don't know what its number is. 'Ed' provides an abbreviation to specify the last line in the buffer:

```
$p
   the lazy dog.
```

The dollar sign can be used just like a number. To print everything in the buffer, we could type:

```
1,$p
The quick brown fox
    jumps over
   the lazy dog.
```

If for some reason you want to stop the printing before it is done, press the BREAK key on your terminal. If you receive no response from BREAK, 'ed' is waiting for you to enter a command. Otherwise, 'ed' will respond with

```
?
```

and wait for your next command.

5.1.8 <u>More Complicated Line Numbers</u>

'Ed' has several ways to specify lines other than just numbers and "$". Try the following command:

```
p
   the lazy dog.
```

'Ed' prints the last line. Does 'ed' always print the last line when it is given an unadorned "p" command? No. The "p" command by itself prints the "current" line. The "current" line is the last line you have edited in any way. (As a matter of fact, the last thing we did was to print all the lines in the buffer, so the last line was edited by being printed.) 'Ed' allows you to use the symbol "." (read "dot") to represent the current line. Thus

```
.p
   the lazy dog.
```

is the same as

```
...p
   the lazy dog.
```

which is the same as just

```
p
   the lazy dog.
```

"." can be used in many ways. For example,

```
1,2p
The quick brown fox
    jumps over
1,.p
The quick brown fox
    jumps over
.,$p
    jumps over
   the lazy dog.
```

This example shows how to print all the lines up to the current line (1,.p) or all the lines from the current line to the end of the buffer (.,$p). If for some reason you would like to know the number of the current line, you can type

```
.=
3
```

and "ed" will display the number. (Note that the last thing we did was to print the last line, so the current line became line 3.)

"." is not particularly useful when used alone. It becomes much more important when used in "line-number expressions." Try this experiment:

```
.-1p
    jumps over
```

".-1" means "the line that is one line before the current line."

```
.+1p
   the lazy dog.
```

".+1" means "the line that is one line after the current line."

```
.-2,.-1p
The quick brown fox
    jumps over
```

".-2,.-1p" means "print the lines from two lines before to one line before the current line."

You can also use "$" in line-number expressions:

```
$-1p
    jumps over
```

"$-1p" means "print the line that is one line before the last line in the buffer (the next to the last line)."

Some abbreviations are available to help reduce the amount of typing you have to do. Typing a newline by itself is equivalent to typing ".+1p"; typing an up arrow, "^", followed by a newline is equivalent to typing ".-1p"; and typing a line-number expression followed by a newline is equivalent to typing that line-number expression followed by "p". Examples:

```
    (type a newline by itself)
   the lazy dog.
^
    jumps over
1
The quick brown fox
```

It might be worthwhile to note here that all commands expect line numbers of one form or another. If none are supplied, "ed" will use default values. Thus,

```
w file
```

is equivalent to

```
1,$w file
```

and

```
a
```

is equivalent to

```
.a
```

(which means, append text <u>after</u> the current line.)

5.1.9 <u>Deleting Lines</u>

As yet, you have seen no way of removing lines that are no longer wanted or needed. To do this, use the "delete" command "d":

```
1,2d
```

Deletes the first through the second lines. "d" expects line numbers that work in the same way as those specified

for "p", deleting one line or any range of lines.

```
d
```

deletes only the current line. It is the same as ".d" or ".,.d".

After a deletion, the current line pointer is left pointing to the first line after the group of deleted lines, unless the last line in the buffer was deleted. In this case, the current line is the last line before the group of deleted lines.

5.1.10 Text Patterns

Frequently it is desirable to be able to find a particular "pattern" in a piece of text. For example, suppose that after proofreading a report you have typed in using 'ed' you find a spelling error. There must be an easy way to find the misspelled word in the file so it can be corrected. One way to do this is to count all the lines up to the line containing the error, so that you can give the line number of the offending line to 'ed'. Obviously, this way is not very fast or efficient. 'Ed' allows you to "search" for patterns of text (like words) by enclosing the pattern in slashes:

```
/jumps/
    jumps over
```

'Ed' looks for the pattern you specified, and moves to the first line which contains the pattern. Note that if we had typed

```
/jumped/
?
```

'ed' informs us that it could not find the pattern we wanted.

'Ed' searches forward from the current line when it attempts to find the pattern you specified. If 'ed' reaches the last line without seeing the pattern, it "wraps around" to the first line in the file and continues searching until it either finds the pattern or gets back to the line where it started (line "."). This procedure ensures that you will get the "next" occurrence of the pattern you were looking for, and that you will not miss any occurrences because of your current position in the file.

Suppose, however, that you do not wish to find the "next" occurence of a word, but the previous occurrence of a word. Very few text editors provide this capability; however, 'ed' makes it simple. Just surround the pattern with back-slashes:

```
\quick\
The quick brown fox
```

Remember: backslashes - search backward. The backward search (or backscan, as it is sometimes called) wraps around the file in a manner similar to the forward search (or scan). The search begins at the line before the current line, proceeds until the first line of the file is seen, then begins at the last line of the file and searches up until the current line is encountered. Once again, this is to ensure that you do not miss any occurrences of a pattern due to your current position in the file.

"Ed" also provides more powerful pattern matching services than simply looking for a given string of characters. (A note to beginning users: this section may seem fairly complicated at first, and indeed you do not really need to understand it completely for effective use of the editor. However, the results you might get from some patterns would be mystifying if you were not provided with some explanation, so look this over once and move on.)

The pattern that may appear within slashes (or backslashes) is called a "regular expression." It contains characters to look for and special characters used to perform other operations. The following characters

```
x  ?  $  [  .  @  (
```

have special meaning to "ed":

x Beginning of line. The "%" character appearing as the first element in a pattern matches the beginning of a line. It is most frequently used to locate lines with some string at the very beginning; for example,

```
/%The/
```

finds the next line that begins with the word "The". The percent sign has its special meaning **only if it is the first element of the pattern**; otherwise, it is treated as a literal percent sign.

? Any character. The question mark "?" in a regular expression matches <u>any</u> character (except a beginning-of-line or a newline). It can be used like this:

```
/a?b/
```

to find strings like

```
a+b
a-b
```

a b
arbitrary

However, "?" is most often used with the "closure" operator "*" (see below).

$ End of line. The dollar sign appearing as the last element of a pattern matches the newline character at the end of a line. Thus,

/today$/

can be used to find a line with the word "today" at the very end. Similar to the percent sign, the dollar sign has no special meaning in positions other than at the end of a pattern.

[] Character classes. The square brackets are used to match "classes" of characters. For example,

/[A-Z]/

will find the next line containing a capital letter,

/%[abcxyz]/

will find the next line beginning with an a, b, c, x, y, or z, and

/[^0-9]/

will find the next line which contains a non-digit. Character classes are also frequently used with the "closure" operator "*".

* Closure. The asterisk is used to mean "any number of repetitions (including zero) of the previous pattern element (one character or a character class in brackets)." Thus,

/a?*b/

will find lines containing an "a" followed by any number of characters and a "b". For example, the following lines will be matched:

ab
abnormal
Recording Media, by Dr. Joseph P. Gunchy

As another example,

/%=.*$/

will match only those lines containing all equal-

signs (or nothing at all). If you wish to ensure that only non-empty lines are matched, use

/%==*$/

Always remember that "*" (closure) will match zero or more repetitions of an element.

@ Escape. The "at" sign has special meaning to both "ed" and the Subsystem I/O routines. It is the "escape" character, which is used to prevent interpretation of a special character which follows. (Note that to enter a single "@" from a terminal, you must type two; the Subsystem I/O routines remove one in the process of interpreting escaped characters.) Suppose you wish to locate a line containing the string "a * b". You may use the following command:

/a @@* b/

(Note that two "at" signs are required to pass one "at" sign to the editor.) The "at" sign "turns off" the special meaning of the asterisk, so it can be used as an ordinary text character. You may have occasion to escape any of the regular expression metacharacters (%, ?, $, [, *, @, or () or the slash itself. For example, suppose you wished to find the next occurrence of the string "1/2". The command you need is:

/1@@/2/

() Pattern tags. As seen in the next section, it is sometimes useful to remember what part of a line was actually matched by a pattern. By default, the string matched by the entire pattern is remembered. It is also possible to remember a string that was matched by only a part of a pattern by enclosing that part of the pattern in braces. Hence to find the next line that contains a quoted string and remember the text between the quotes, we might use

/"(?*)"/

If the line thus located looked like this

This is a line containing a "quoted string".

then the text remembered as matching the tagged part of the pattern would be

quoted string

The last important thing you will need to know about patterns is the use of the "default" pattern. "//" remembers the last regular expression used in any command, to save you the trouble of retyping it. To access the remembered pattern, simply use an "empty" string. For example, the following sequence of commands could be used to step through a file, looking for each occurrence of the string "ICS":

```
/ICS/
//
//
(and so on)
```

One last comment before leaving pattern searching. The constructs

```
/pattern/
\pattern\
```

are not separate commands; they are components of line number expressions. Thus, to print the line after the next line containing "tape", you could say

```
/tape/+1p
```

Or, to print a range of lines from one before to one after a line with a given pattern, you could use

```
/pattern/-1,/pattern/+1p
```

5.1.11 Making Substitutions - the Substitute command

This is one of the most used editor commands. The "substitute" command "s" is used to make small changes within lines, without the need of retyping them. It is used like this:

```
starting-line,ending-line s /pattern/new-stuff/
```

For instance, suppose our buffer looks like this:

```
1,$p
The quick brown fox
    jumps over
   the lazy dog.
```

To change "jumps" to "jumped,"

```
2s/jumps/jumped/p
    jumped over
```

Note the use of the trailing "p" to print the result. If

the "p" had been omitted, the change would have been performed (in the buffer) but the changed line would not have been printed out.

If the last string specified in the substitute command is empty, then the pattern found is deleted:

```
s/jumped//p
     over
s/% */    jumps /p
    jumps over
```

Recalling that a missing pattern means "use the last pattern specified," try to explain what the following commands do:

```
s///p
jumps over
s//    /p
    jumps over
```

(Note that, like many other commands, the substitute command assumes you want to work on the current line if you do not specify any line numbers.)

What if you want to change "over" into "over and over"? We might use

```
s/over/over and over/p
    jumps over and over
```

to accomplish this. There is a shorthand notation for this kind of substitution that was alluded to briefly in the last section. (Recall the discussion of "tagged" patterns.) By default, the part of a line that was matched by the whole pattern is remembered. This string can then be included in the replacement string by typing an ampersand ("&") in the desired position. So, instead of the command in the last example,

```
s/over/& and &/
```

could have been used to get the same result. If portion of the pattern had been tagged, the text matched by the tagged part in the replacement could be reused by typing "&1":

```
s/jump(?*)/vault&1/p
    vaults over and over
```

It is possible to tag up to nine parts of a pattern using braces. The text matched by each tagged part may then be used in a replacement string by typing

```
&n
```

where n corresponds to the nth "(" in the pattern. What does the following command do?

```
s/([^ ]*) (?*)/$$2 $$1/
```

Final words on substitute: the slashes are known as "delimiters" and may be replaced by any other character except a newline, as long as the same character is used consistently throughout the command. Thus,

```
s#vaults#vaulted#p
    vaulted over
```

is legal. Also, note that substitute changes only the first occurrence of the pattern that it finds; if you wish to change all occurrences on a line, you may append a "g" (for "global") to the command, like this:

```
s/ /*/gp
****vaulted*over
```

5.1.12 Line Changes and Insertions

Two "abbreviation" commands are available to shorten common operations applying to changes of entire lines. These are the "change" command "c" and the "insert" command "i".

The change command is a combination of delete and append. Its format is

```
starting-line,ending-line c
```

This command deletes the given range of lines, and then goes into append mode to obtain text to replace them. Append mode works exactly the same way as it does for the "a" command; input is terminated by a period standing alone on a line. Examine the following editing session to see how change might be used:

```
1,$c
Ed is an interactive program used for
the creation and modification of "text.
.
c
the creation and modification of "text."
"Text" may be any collection of character
data.
.
```

As you can see, the current line is set to the last line entered in append mode.

The other abbreviation command is "i". "I" is very closely related to "a"i in fact, the following relation holds:

```
starting-line i
```

is the same as

```
starting-line - 1  a
```

In short, "i" inserts text before the specified line, whereas "a" inserts text after the specified line.

5.1.13 Moving Text

Throughout this introduction, we have concentrated on what may be called "in-place" editing. The other type of editing commonly used is often called "cut-and-paste" editing. The move command "m" is provided to facilitate this kind of editing, and works like this:

```
starting-line,ending-line m after-this-line
```

If you wanted to move the last fifty lines of a file to a point after the third line, the command would be

```
$-49,$m3
```

Any of the line numbers may, of course, be full expressions with search strings, arithmetic, etc.

You may, if you like, append a "p" to the move command and it will print the last line moved. The current line is set to the last line moved.

5.1.14 Global Commands

The "global" command "g" is used to perform an editing command on all lines in the buffer that match a certain pattern. For example, to print all the lines containing the word "editor", you could type

```
g/editor/p
```

If you wanted to correct some common spelling error, you would use

```
g/old-stuff/s//new-stuff/gp
```

which will make the change in all appropriate lines and print the resulting lines. Another example: deleting all lines that begin with an asterisk could be done this way:

```
g/^##*/d
```

"G" has a companion command "x" (for "exclude") that performs an operation on all lines in the buffer that do not match a given pattern. For example, to delete all lines that do not begin with an asterisk, use

```
x/^##*/d
```

"G" and "x" are very powerful commands that are essential for advanced usage, but are usually not necessary for beginners. Concentrate on other aspects of 'ed' before you move on to tackle global commands.

5.1.15 Marking Lines

During some types of editing, especially when moving blocks of text, it is often necessary to refer to a line in the buffer that is far away from the current line. For instance, say you want to move a subroutine near the beginning of a file to somewhere near the end, but you aren't sure that you can specify patterns to properly locate the subroutine. One way to solve this problem is to find the first line of the subroutine, then use the command

```
/subroutine/
   subroutine think
.=
47
```

and write down (or remember) line 47. Then find the end of the subroutine and do the same thing:

```
/end/
   end
.=
71
```

Now you move to where you want to place the subroutine and enter the command

```
47,71m.
```

which does exactly what you want.

The problem here is that absolute line numbers are easily forgotten, easily mistyped, and difficult to find in the first place. It is much easier to have 'ed' remember a short "name" along with each line, and allow you to reference a line by its name. In practice, it seems convenient to restrict names to a single character, such as "b" or "e" (for "beginning" or "end"). It is not necessary for a given name to be uniquely associated with one line; many

lines may bear the same name. In fact, at the beginning of the editing session, all lines are marked with the same name, a single space.

To return to our example, using the "k" command, we can mark the beginning and ending lines of the subroutine quite easily:

```
/subroutine/
   subroutine think
kb
/end/
   end
ke
```

We have now marked the first line in the subroutine with "b" and the second line with "e".

To refer to names, we need more line number expression elements: ">" and "<". Both work in line number expressions just like "$" or "/pattern/". The symbol ">" followed by a single character mark name means "the line number of the first line with this name when you search forward". The symbol "<" followed by a single character mark name means "the line number of the first line with this name when you search backward". (Just remember that "<" points backward and ">" points forward.)

Now in our example, once we locate the new destination of the subroutine, we can use "<b" and "<e" to refer to lines 47 and 71, respectively (remember, we marked them). The "move" command would then be

```
<b,<em.
```

Several other features pertaining to mark names are important. First, the "k" command does not change the current line ".". You can say

```
$kx
```

(which marks the last line with "x") and "." will not be changed. If you want to mark a range of lines, the "k" command will take two line numbers. For instance,

```
5,10ka
```

will mark lines 5 through 10 with "a" (i.e., give each of lines 5 through 10 the name "a").

The "n", "!" and apostrophe commands also deal with marks. The "n" command performs two functions. If it is invoked without a mark name following it, like

```
$n
```

it will print the mark name of the line. In this case, it would print the mark name of the last line in the file. If the 'n' command is followed by a mark name, like

```
4nq
```

it marks the line with that mark name, and erases the marks on any other lines with that name. In this case, line 4 is marked with "q" and it is guaranteed that no other line in the file is marked with "q".

The '!' and apostrophe commands are both global commands that deal with mark names. The apostrophe command works very much like the 'g' command: the apostrophe is followed by a mark name and another command; the command is performed on every line marked with that name. For instance,

```
'as/fox/rabbit/
```

will change the first "fox" to "rabbit" on every line that is named "a". The '!' command works in the same manner, except that it performs the command on those lines that <u>are not</u> marked with the specified name. For example, to delete all lines not named "k", you could type

```
!kd
```

5.1.16 <u>Undoing Things -- the Undo Command</u>

Unfortunately, Murphy's Law guarantees that if you make a mistake, it will happen at the worst possible time and cause the greatest possible amount of damage. 'Ed' attempts to prevent mistakes by doing such things as working with a copy of your file (rather than the file itself) and checking commands for their plausibility. However, if you type

```
d
```

when you really meant to type

```
a
```

'ed' must take its input at face value and do what you say. It is at this point that the "undo" command becomes useful. "Undo" allows you to "undelete" the last group of lines that were deleted from the buffer. In the last example, some inconvenience could be avoided by typing

```
^ud
```

which restores the deleted line. (By default "undo" will also delete the current line; "ud" keeps the current line

from being deleted.)

The problem that arises with "undo" is the answer to the question "What was the last group of lines deleted?" This answer is very dependent on the implementation of 'ed' and in some cases is subject to change. After many commands, the last group of lines deleted is well-defined, but unspecified. It is not a good idea to use the "undo" command after anything other than 'c', 'd', or 's'. After a 'c' or 'd' command,

```
ud
```

will place the last group of deleted lines after the current line. After an 's' command (which by the way, deletes the old line and then reinserts the changed line),

```
u
```

will delete the current line and replace it with the last line deleted -- it will exactly undo the effects of the 's' command.

You should be warned that while "undo" works nicely for repairing a single 'c', 'd', or 's' command, it cannot repair the damage done by one of these commands under the control of a global prefix ('g', 'x', 'v' and apostrophe). Since the global prefixes perform their commands many times, only the very last command performed by a global prefix can be repaired.

5.1.17 Summary

This concludes our tour through the world of text editing. For your convenience, a short summary of all available editing commands (many of which were not discussed in this introduction, but which you will undoubtedly find useful) is given in the next few sections.

5.1.17.1 Command Summary

Editor Command Summary

Range	Name	Function
.	a	append text after the specified line
.,.	c	change: delete specified text, then accept text to replace it
.,.	d	delete specified range of lines
	e	edit named file (syntax: e [filename])
	f	print remembered file name (syntax: f[filename])
1,$	g	globally perform command (syntax: g/pattern/command)
.	i	insert text before specified line
^,.	j	join the specified lines into a single line (syntax: j/string/). "string" is inserted between each pair of lines joined.
.,.	k	mark lines with given name (syntax: k<single-character-name>)
.,.	m	move lines from one place to another (syntax: from-here,to-here m there[p])
.,.	n	print mark names of specified lines
none	o	options command. Syntax includes: op/prompt/ to set command prompt
.,.	p	print specified lines
none	q	quit (exit the editor gracefully)
.	r	read a file into the buffer (syntax: r [filename])
.,.	s	substitute (syntax: s/change-this/to-this/[p])
.,.	t	transit (syntax: t/from-range/to-range/[p]). Characters in the "from-range" are converted to their corresponding characters in the "to-range".

...	u	un-substitute (restore lines that were changed by substitution)
...	v	overlay; print all lines in the given range, waiting at the end of each line for text that is to be appended.
1,$	w	write buffer (syntax: w [filename])
1,$	x	exclude; converse of "g"; perform command on all lines that do not match a given pattern (syntax: x/pattern/command)
...	y	copy; reproduce a block of lines in another place (syntax: from-here,to-here y after-here)
.	=	print value; value of the line number expression preceding the equals sign is printed as a decimal integer. (syntax: <expression>=)
1,$	*	global on mark; perform command on all lines having a given mark name (syntax: *<name>command)
1,$	!	exclude on mark; perform command on all lines that do not have a given mark name (syntax: !<name>command)

5.1.17.2 Line Number Expressions

Elements of Line Number Expressions

Form	Value
integer	value of the integer. (Ex: 44)
.	number of the current line in the buffer
$	number of the last line in the buffer
^	number of the previous line in the buffer (.-1)
/pattern/	number of the next line in the buffer that matches the given pattern (Ex: /February/); the search proceeds to the end of the buffer, then wraps around to the beginning and back to the current line
\pattern\	number of the next line in the buffer that matches the given pattern; search proceeds in reverse, from the current line to line 1, then the last line back to the current line. (Ex: \January\)
>name	number of the next line having the given mark name (search wraps around, like //)
<name	number of the next line having the given mark name (search proceeds in reverse, in the same way as \\)
expression	Any of the above operands may be combined with plus or minus signs to produce a line number expression. Plus signs may be omitted if desired. (Ex: /parse/-5, /lexical/+2, /lexical/2, $-5, .+6)

5.1.17.3 Pattern Elements

Summary of Pattern Elements

Element	Meaning
^	Matches the null string at the beginning of a line. However, if not the first element of a pattern, is treated as a literal percent sign.
?	Matches any single character other than newline.
$	Matches the newline character at the end of a line. However, if not the last element of a pattern, is treated as a literal dollar sign.
[<ccl>]	Matches any single character that is a member of the set specified by <ccl>. <ccl> may be composed of single characters or of character ranges of the form <c1>-<c2>. If character ranges are used, <c1> and <c2> must both belong to the digits, the upper case alphabet or the lower case alphabet.
[^<ccl>]	Matches any single character that is not a member of the set specified by <ccl>.
*	In combination with the immediately preceeding pattern element, matches zero or more characters that are matched by that element.
@	Turns off the special meaning of the immediately following character. If that character has no special meaning, this is treated as a literal "@".
{<pattern>}	Tags the text actually matched by the sub-pattern specifed by <pattern> for use in the replacement part of a substitute command.
&	Appearing in the replacement part of a substitute command, represents the text actually matched by the pattern part of the command.
@<digit>	Appearing in the replacement part of a substitute command, represents the text actually matched by the tagged sub-pattern specified by <digit>.

5.2. SE

'Se' works much like the regular editor 'ed', accepting the same commands with a few differences. Rather than displaying only a single line from the file being edited (as 'ed' does), 'se' always displays a "window" onto the file. In order to do this, 'se' must be run from a CRT terminal and must be told what sort of terminal it is. This is accomplished through the specification of a particular parameter when 'se' is invoked.

5.2.1 Terminals Supported

'Se' is capable of being used from a variety of different terminals. New terminal types are easily added by making small additions to the source code. In general, all that is required of a terminal is that it have the ability to home the cursor (position it to the upper left hand corner of the screen) without erasing the screen's contents, although backspacing, a screen clear function, and arbitrary cursor positioning are tremendously helpful.

The terminals currently supported are the following:

adds	ADDS Consul 980. (This is the same as "consul" below.)
adm3a	Lear-Siegler ADM-3A.
b150	Beehive International B150.
b200	Beehive International B200.
cg	Chromatics Color Graphics Terminal.
consul	ADDS Consul 980.
fox	Perkin-Elmer 1100.
haz	Hazeltine 1500 series.
isc	Intelligent Systems Corporation 8001 Color Terminal.
regent	ADDS Regent 100.
sbee	Beehive International Superbee.
sol	Processor Technology Sol computer with software to emulate a Beehive B200.
tvt	Southwest Technical Products TV Typewriter II.

5.2.2 Editing Options

'Se' allows the user to set various options which control the editing envrionment. To set an option, the user must specify the "option" (o) command. This consists of the letter 'o' followed by one of the following strings of characters:

a causes absolute line numbers to be displayed in the left-hand margin of the screen. Default behavior is to display upper-case letters with the letter "A" corresponding to the first line in the window.

c inverts the case of all letters typed by the user (i.e., converts upper-case to lower-case and vice versa). This option will cause commands to be recognized only in upper-case and alphabetic line numbers to be displayed and recognized only in lower-case.

d[<dir>] selects the placement of the current line pointer following a "d" (delete) command. <dir> must be either ">" or "<". If ">" is specified, the default behavior is selected: the line following the deleted lines becomes the new current line. If "<" is specified, the line immediately preceding the delete lines becomes the new current line. If neither is specified, the current value of <dir> is displayed in the status line.

f selects Fortran oriented options. This is equivalent o specifying both the "c" and "t6" (see below) options.

l[<top>] sets the line number display option. Under control of this option, 'se' will continuously display the value of one of three symbolic line numbers. <top> may be any of the following:

. display the current line number

' display the number of the top line on the screen

$ display the number of the last line in the buffer

If <top> is omitted, the line number display is disabled.

t[<tabs>] sets tab stops according to <tabs>. <tabs> consists of a series of numbers indicating column numbers in which tab stops are to be set. If a number is preceded by a plus sign ("+"), it indicates that the number is an increment; stops are

set at regular intervals separated by that many columns, beginning with the most recently specified absolute column number. If no such number precedes the first increment specification, the stops are set relative to column 1. By default, tab stops are set in every third column starting with column 1, corresponding to a <tabs> specification of "+3". If <tabs> is omitted, the current tab spacing is displayed in the status line.

u[<chr>] selects the character that 'se' displays in place of unprintable characters. <chr> may be any printable character; it is initially set to blank. If <chr> is omitted, 'se' displays the current replacement character on the status line at the bottom of the screen.

v[<int>] sets the default "overlay column". This is the column at which the cursor is initially positioned by the "v" command. <int> must be a positive integer, or a dollar sign ($) to indicate the end of the line. If <int> is omitted, the current overlay column is displayed in the status line.

w[<int>] sets the "warning threshold" to <int> which must be a positive integer. Whenever the cursor is positioned at or beyond this column, the column number is displayed in the status line at the bottom of the screen and the terminal's bell is sounded. If <int> is omitted, the current warning threshold is displayed on the status line. The default warning threshold is 74, corresponding to the first column beyond the right edge of the screen on an 80 column crt.

-[<lnr>] splits the screen at the line specified by <lnr> which must be a simple line number within the current window. All lines above <lnr> remain frozen on the screen, the line specified by <lnr> is replaced by a row of dashes, and the space below this row becomes the new window on the file. Further editing commands do not affect the lines displayed in the top part of the screen. If <lnr> is omitted, the screen is restored to its full size.

5.2.3 Control Characters for Editing and Cursor Motion

Since "se" takes its commands directly from the terminal, it cannot be run from a script by using Subsystem I/O redirection, and Subsystem erase, kill, and escape conventions do not exactly apply. In fact, "se" has its own set of control characters for editing and cursor motion; their meaning is as follows:

ctrl-A Toggle insert mode. The status of the insertion indicator is inverted. Insert mode, when enabled, causes characters typed to be inserted at the current cursor position in the line instead of overwriting the characters that were there previously. When insert mode is in effect, "INSERT" appears in the status line.

ctrl-C Insert blank. The characters at and to the right of the current cursor position are moved to the right one column and a blank is inserted to fill the gap.

ctrl-D Cursor up. The effect of this key depends on "se"s current mode. When in command mode, the current line pointer is moved to the previous line without affecting the contents of the command line. If the current line pointer is at line 1, the last line in the file becomes the new current line. In overlay mode (viz. the "v" command), the cursor is moved up one line while remaining in the same column. In append mode, this key is ignored.

ctrl-E Tab left. The cursor is moved to the nearest tab stop to the left of its current position.

ctrl-F "Funny" return. The effect of this key depends on the editor's current mode. In command mode, the current command line is entered as-is, but is not erased upon completion of the command; in append mode, the current line is duplicated; in overlay mode (viz. the "v" command), the current line is restored to its original state and command mode is reentered (except if under control of a global prefix).

ctrl-G Cursor right. The cursor is moved one column to the right.

ctrl-H Cursor left. The cursor is moved one column to the left. Note that this does not erase any characters; it simply moves the cursor.

ctrl-I Tab right. The cursor is moved to the next tab stop to the right of its current position.

ctrl-K Cursor down. As with the ctrl-D key, this key's effect depends on the current editing mode. In command mode, the current line pointer is moved to the next line without changing the contents of the command line. If the current line pointer is at the last line in the file, line 1 becomes the new current line. In overlay mode (viz. the "v" command), the cursor is moved down one line while remaining in the same column. In append mode, ctrl-K has no effect.

ctrl-L Scan left. The cursor is positioned according to the character typed immediately after the ctrl-L. In effect, the current line is scanned, starting from the current cursor position and moving left, for the first occurrence of this character. If none is found before the beginning of the line is reached, the scan resumes with the last character in the line. If the line does not contain the character being looked for, the message "NOT FOUND" is printed in the status line at the bottom of the screen. 'Se' remembers the last character that was scanned for using this key; if the ctrl-L is hit twice in a row, this remembered character is searched for instead of a literal ctrl-L.

ctrl-M Newline. This key is identical to the NEWLINE key described below.

ctrl-N Insert newline. A newline character is inserted before the current cursor position, and the cursor is moved one position to the right. The newline is displayed according to the current nonprintable replacement character (see the "u" option).

ctrl-O Skip right. The cursor is moved to the first position beyond the current end of line.

ctrl-P Interrupt. If executing any command except "a", "c", "i" or "v", 'se' aborts the command and reenters command mode. The command line is not erased.

ctrl-Q Fix screen. The screen is reconstructed from 'se's internal representation of the screen.

ctrl-R Erase right. The character at the current cursor position is erased and all characters to its right are moved left one position.

ctrl-S Scan right. This key is identical to the ctrl-L key described above, except that the scan proceeds to the right from the current cursor position.

ctrl-T Kill right. The character at the current cursor position and all those to its right are erased.

ctrl-U Erase left. The character to the left of the current cursor position is deleted and all characters to its right are moved to the left to fill the gap. The cursor is also moved left one column.

ctrl-V Skip right and terminate. The cursor is moved to the current end of line and the line is terminated.

ctrl-W Skip left. The cursor is positioned at column 1.

ctrl-Y Kill left. All characters to the left of the cursor are erased; those above and to the right of the cursor are moved to the left to fill the void. The cursor is left in column 1.

ctrl-Z Toggle case mapping flag. The status of the case mapping indicator is inverted; if case mapping was on, it is turned off, and vice versa. Case mapping, when in effect, causes all upper case letters to be converted to lower case, and all lower case letters to be converted to upper case. Note, however, that 'se' continues to recognize relative line numbers in upper case only, in contrast to the case mapping option (see the description of options above). When case mapping is on, "CASE" appears in the status line.

NEWLINE Kill right and terminate. The characters at and to the right of the current cursor position are deleted, and the line is terminated.

DEL Kill all. The entire line is erased, along with any error message that appears in the status line.

ESC Escape. The ESC key provides a means for entering 'se's control characters literally as text into the file. In fact, any character that can be generated from the keyboard is taken literally when it immediately follows the ESC key. If the character is unprintable (as are all of 'se's control characters), it will appear on the screen according the current nonprintable replacement character (normally a blank).

The set of control characters defined above can be used for correcting mistakes while typing regular editing commands, for correcting commands that have caused an error message to be displayed, for correcting lines typed in append mode, or for inline editing using the "v" command described below.

5.2.4 'Se' Command Interpretation

There are a few differences in command interpretation between the regular editor and the 'se'. The only effect of the "p" command in 'se' is to position the window so that as many as possible of the "printed" lines are displayed while including the last line in the range. In fact, the window is always positioned so that the current line is displayed. Typing a "p" command with no line numbers positions the window so that the line previously at the top of the window is at the bottom. This can be used to "page" backwards through the file. The ":" command, (which in the regular editor prints about a screenfull of text starting with a specified line), positions the window so it begins at the specified line, and leaves the current line pointer at this line. Thus, a ":" can be used to page forward through the file.

The "overlay" (v) command in the regular editor 'ed' only allows the user to add onto the end of lines, and can be terminated before the stated range of lines has been processed by entering a period by itself, as in the "append" command. But in 'se', this command allows arbitrary changes to be made to the lines, and the period has no special meaning. To abort before all the lines in the range have been covered, use the "funny return" character (ctrl-F). Doing this restores the line containing the cursor to the state it was in before the "v" command was started.

'Se' has extended line number syntax. In general, whatever appears in the left margin on the screen is an acceptable line number and refers to the line displayed in that row on the screen.

APPENDIX 6

FORMATTER
GEORGIA TECH SOFTWARE TOOLS SUBSYSTEM

6.1. BASICS

6.1.1 Introduction

'Fmt' is a program designed to facilitate the preparation of neatly formatted text. It provides many features, such as automatic margin alignment, paragraph indentation, hyphenation and pagination, that are designed to greatly ease an otherwise tedious job.

It is the intent of the next few sections to familiarize the user with the principles of automatic text formatting in general and with the capabilities and usage of 'fmt' in particular.

6.1.2 Usage

'Fmt' takes as input a file containing text with formatting instructions. It is invoked by a command with various optional parameters, discussed below. The resultant output is appropriately formatted text suitable for a printer having backspacing capabilities. The output of 'fmt' is made available on its first standard output port, and so may be placed in a file, sent to a line printer, or changed in any of a number of ways, simply by applying standard Software Tools Subsystem I/O redirection.

When 'fmt' is invoked from the Subsystem, there are several optional parameters that may be specified to control its operation. The full command line syntax is

fmt [-s] [-p<first>[-<last>]] { <file name> }

A brief explanation of the cryptic notation: the items enclosed within square brackets ("[]") are optional -- they may or may not be specified; items enclosed between braces ("{}") may occur any number of times, including zero; items enclosed in angle brackets ("<>") designate character strings whose significance is suggested by the text within the brackets; everything else should be taken literally.

And now for an explanation of what these parameters mean:

-s If this option is selected, 'fmt' will pause at the top of each page, ring the bell or buzzer on your terminal, and wait for a response. This feature is for the benefit of people using hard-copy terminals with paper not having pin-feed margins. The correct response, to be entered after the paper is mounted, is a control-c (hold the 'control' key down and type 'c').

-p ... This option allows selection of which pages of the formatted document will actually be printed. Immediately following the "-p", without any intervening spaces, should be a number indicating the first page to be printed. Following this, a second number may be specified, separated from the first by a single dash, which indicates the last page to be printed. If this second number is omitted, all remaining pages will be produced.

<file> Any number of file names may be specified on the command line. 'Fmt' will open the files in turn, formatting the contents of each one as if they constituted one big file. When the last named file is processed, 'fmt' terminates. If no file names are specified, standard input number one is used. In addition, standard input may be specified explicitly on the command line by using a dash as a file name.

6.1.3 Commands and Text

'Fmt', like almost every other text formatter ever written, operates on an input stream that consists of a mixture of text and formatting commands. Each command starts at the beginning of a line with a 'control character', usually a period, followed by a two character name, in turn followed by some optional 'parameters'. There must not be anything else on the line. For example, in

```
.ta 11 21 31 41
```

the control character is a period, the command name is ta, and there are four parameters: "11", "21", "31" and "41". Notice that the command name and all the parameters must be separated from each other by one or more blanks. Anything not recognizable as a command is treated as text.

6.2. FILLING AND MARGIN ADJUSTMENT

6.2.1 Filled Text

'Fmt' collects as many words as will fit on a single output line before actually writing it out, regardless of line boundaries in its input stream. This is called 'filling' and is standard practice for 'fmt'. It can, however, be turned off with the 'no-fill' command

```
.nf
```

and lines thenceforth will be copied from input to output unaltered. When you want to turn filling back on again, you may do so with the 'fill' command

```
.fi
```

and 'fmt' will resume its normal behavior.

If there is a partially filled line that has not yet been written out when an nf command is encountered, the line is forced out before any other action is taken. This phenomenon of forcing out a partially filled line is known as a 'break' and occurs implicitly with many formatting commands. To cause one explicitly, the 'break' command

```
.br
```

is available.

6.2.2 Hyphenation

If, while filling an output line, it is discovered that the next word will not fit, an attempt is made to hyphenate it. Although 'fmt' is usually quite good in its choice of where to split a word, it occasionally makes a gaffe or two, giving reason to want to turn the feature off. Automatic hyphenation can be disabled with the 'no-hyphenation' command

```
.nh
```

long enough for a troublesome word to be processed, and then reenabled with the 'hyphenate' command

```
.hy
```

Neither command causes a break.

6.2.3 Margin Adjustment

After filling an output line, 'fmt' inserts extra blanks between words so that the last word on the line is flush with the right margin, giving the text a "professional" appearance. This is one of several margin adjustment modes that can be selected with the 'adjust' command

```
.ad <mode>
```

The optional parameter <mode> may be any one of four single characters: "b", "c", "l" or "r". If the parameter is "b" or missing, normal behavior will prevail -- both margins will be made even by inserting extra blanks between words. This is the default margin adjustment mode. If "c" is specified, lines will be shifted to the right so that they are centered between the left and right margins. If the parameter is "l", no adjustment will be performed; the line will start at the left margin and the right margin will be ragged. If "r" is specified, lines will be moved to the right so that the right margin is even, leaving the left margin ragged.

The 'no-adjustment' command

```
.na
```

has exactly the same effect as the following 'adjust' command:

```
.ad l
```

No adjustment will be performed, leaving the left margin even and the right margin ragged. In no case does a change in the adjustment mode cause a break.

6.2.4 Centering

Input lines may be centered, without filling, with the help of the 'center' command

```
.ce N
```

The optional parameter N is the number of subsequent input lines to be centered between the left and right margins. If the parameter is omitted, only the next line of input text is centered. Typically, one would specify a large number, say 1000, to avoid having to count lines; then, immediately following the lines to be centered, give a 'center' command with a parameter of zero. For example:

```
.ce 1000
more lines
than I care
to count
.ce 0
```

It is worth noting the difference between

```
.ce
```

and

```
.ad c
```

When the former is used, an implicit break occurs before each line is printed, preventing filling of the centered lines; when the latter is used, each line is filled with as many words as possible before centering takes place.

6.2.5 Summary - Filling and Margin Adjustment

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.ad c	both	both	no	Set margin adjustment mode.
.br	-	-	yes	Force a break.
.ce N	N=0	N=1	yes	Center N input text lines.
.fi	on	-	no	Turn on fill mode.
.hy	on	-	no	Turn on automatic hyphenation.
.na	-	-	no	Turn off margin adjustment.
.nf	-	-	yes	Turn off fill mode. (Also inhibits adjustment.)
.nh	-	-	no	Turn off automatic hyphenation.
.sb	off	-	no	Single blank after end of sentence.
.xh	on	-	no	Extra blank after end of sentence.

6.3. SPACING AND PAGE CONTROL

6.3.1 Line Spacing

'Fmt' usually produces single-spaced output, but this can be changed, without a break, using the 'line-spacing' command

```
.ls N
```

The parameter N specifies how many lines on the page a single line of text will use; for double spacing, N would be two. If N is omitted, the default (single) spacing is reinstated.

Blank lines may be produced with the 'space' command

```
.sp N
```

The parameter N is the number of blank lines to produce; if omitted, a value of one is assumed. The sp command causes a break; if the current line spacing is more than one, the break will cause blank lines to be output. Then the blank lines generated by sp are output. Thus, if output is being double-spaced and the command

```
.sp 3
```

is given, four blank lines will be generated. If the value of N calls for more blank lines than there are remaining on the current page, any extra ones are discarded. This ensures that, normally, each page begins at the same distance from the top of the paper.

6.3.2 Page Division

'Fmt' automatically divides its output into pages, leaving adequate room at the top and bottom of each page for running headings and footings. There are several commands that facilitate the control of page divisions when the normal behavior is inadequate.

The 'begin-page' command

```
.bp ±N
```

causes a break and a skip to the top of the next page. If a parameter is given, it serves to alter the page number and so it must be numeric with an optional plus or minus sign. If the parameter is omitted, the page number is incremented by one. If the command occurs at the top of a page before any text has been printed on it, the command is ignored, except perhaps to set the page number. This is to prevent the random occurrence of blank pages.

The optionally signed numeric parameter is a form of parameter used by many formatting commands. When the sign is omitted, it indicates an absolute value to be used; when the sign is present, it indicates an amount to be added to or subtracted from the current value.

The page number may be set independently of the 'begin-page' command with the 'page-number' command

.pn ±N

The next page after the current one, when and if it occurs, will be numbered ±N. No break is caused.

The length of each page produced by 'fmt' is normally 66 lines. This is standard for eleven inch paper printed at six lines per inch. However, if non-standard paper is used, the printed length of the page may easily be changed with the 'page-length' command

.pl ±N

which will set the length of the page to ±N lines without causing a break.

Finally, if it is necessary to be sure of having enough room on a page, say for a figure or a graph, use the 'need' command

.ne N

'Fmt' will cause a break, check if there are N lines left on the current page and, if so, will do nothing more. Otherwise, it will skip to the top of the next page where there should be adequate room.

6.3.3 'No-space' Mode

'No-space' mode is a feature that assists in preventing unwanted blank lines from appearing, usually at the top of a page. When in effect, certain commands that cause blank lines to be generated, such as **bp**, **ne** and **sp**, are suppressed. For the most part, 'no-space' mode is managed automatically; it is turned on automatically at the top of each page before the first text has appeared, and turned off again automatically when the first line on the page is written. This accounts for the suppression of **bp** commands at the top of a page and the discarding of excess blank lines in **sp** commands.

'No-space' mode may be turned on explicitly with the 'no-space' command

```
.ns
```

and turned off explicitly with the 'restore-spacing' command

```
.rs
```

Neither command causes a break.

6.3.4 Summary - Spacing and Page Control

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.bo ±N	N=1	next	yes	Begin a new page.
.ls N	N=1	N=1	yes	Set line spacing.
.ne N	-	N=1	yes	Express a need for N contiguous lines.
.ns	on	-	no	Turn on 'no-space' mode.
.pl ±N	N=66	N=66	no	Set page length.
.pn ±N	N=1	ignored	no	Set page number.
.rs	-	-	no	Turn off 'no-space' mode.
.sp N	-	N=1	yes	Put out N blank lines.

6.4. MARGINS AND INDENTATION

6.4.1 Margins

All formatting operations are performed within the framework of a page whose size is defined by four margins: top, bottom, left and right. The top and bottom margins determine the number of lines that are left blank at the top and bottom of each page. Likewise, the left and right margins determine the first and last columns across the page into which text may be placed.

6.4.2 Top and Bottom Margins

Both the top and the bottom margins consist of two sub-margins that fix the location of the header and footer lines. For the sake of clarity, the first and second sub-margins of the top margin will be referred to as 'margin 1' and 'margin 2', and the first and second sub-margins of the bottom margin, 'margin 3' and 'margin 4'.

The value of margin 1 is the number of lines to skip at the top of each page before the header line, plus one. Thus, margin 1 includes the header line and all the blank lines preceding it from the top of the paper. By default, its value is three. Margin 2 is the number of blank lines that are to appear between the header line and the first text on the page. Normally, it has a value of two. The two together form a standard top margin of five lines, with the header line right in the middle. It is easy enough to change these defaults if they prove unsatisfactory; just use the 'margin-1' and 'margin-2' commands

```
.m1 ±N
.m2 ±N
```

to set either or both sub-margins to ±N.

The bottom margin is completely analogous to the top margin, with margin 3 being the number of blank lines between the last text on a page and the footer line, and margin 4 being the number of lines from the footer to the bottom of the paper (including the footer). They may be set using the 'margin-3' and 'margin-4' commands

```
.m3 ±N
.m4 ±N
```

which work just like their counterparts in the top margin; none cause a break.

6.4.3 Left and Right Margins

The left and right margins define the first and last columns into which text may be printed. They affect such things as adjustment and centering. The left margin is normally set at column one, though this is easily changed with the 'left-margin' command

```
.lm ±N
```

The right margin, which is normally positioned in column sixty, can be set similarly with the 'right-margin' command

```
.rm ±N
```

To ensure that the new margins apply only to subsequent

text, each command causes a break before changing the margin value.

6.4.4 Indentation

It is often desirable to change the effective value of the left margin for indentation, without actually changing the margin itself. For instance, all of the examples in this guide are indented from the left margin in order to set them apart from the rest of the text. Indentation is easily arranged using the 'indent' command.

.in ±N

whose parameter specifies the number of columns to indent from the left margin. The initial indentation value, and the one assumed if no parameter is given, is zero (i.e. start in the left margin).

For the purpose of margin adjustment, the current indentation value is added to the left margin value to obtain the effective left margin. In this respect, the lm and in commands are quite similar. But, whereas the left margin value affects the placement of centered lines produced by the ce command, indentation is completely ignored when lines are centered.

Paragraph indentation poses a sticky problem in that the indentation must be applied only to the first line of the paragraph, and then normal margins must be resumed. This can't be done conveniently with the 'indent' command, since it causes a break. Therefore, 'fmt' has a 'temporary-indent' command

.ti ±N

whose function is to cause a break, alter the current indentation value by ±N until the next line of text is produced, and then reset the indentation to its previous value. So to begin a new paragraph with a five column indentation, one would say

.ti +5

6.4.5 Page Offset

As if control over the left margin position and indentation were not enough, there is yet a third means for controlling the position of text on the page. The concept of a page offset involves nothing more than prepending a number of blanks to each and every line of output. It is primarily intended to allow output to be easily positioned on the paper without having to adjust margins and indentation (with

all their attendant side effects) and without having to physically move the paper. Although the page offset is initially zero, other arrangements may be made with the 'page-offset' command

.po ±N

which causes a break.

6.4.6 Summary - Margins and Indentation

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.in ±N	N=0	N=0	yes	Indent left margin.
.lm ±N	N=1	N=1	yes	Set left margin.
.m1 ±N	N=3	N=3	no	Set top margin before and including page heading.
.m2 ±N	N=2	N=2	no	Set top margin after page heading.
.m3 ±N	N=2	N=2	no	Set bottom margin before page footing.
.m4 ±N	N=3	N=3	no	Set bottom margin including and after page footing.
.po ±N	N=0	N=0	yes	Set page offset.
.rm ±N	N=60	N=60	yes	Set right margin.
.ti ±N	N=0	N=0	yes	Temporarily indent left margin.

6.5. HEADINGS, FOOTINGS AND TITLES

6.5.1 Three Part Titles

A three part title is a line of output consisting of three segments. The first segment is left-justified, the second is centered between the left and right margins, and the third is right-justified. For example

left part center part right part

is a three part title whose first segment is "left part", whose second segment is "center part", and whose third segment is "right part".

To generate a title at the current position on the page, the "title" command is available:

```
.tl /left part/center part/right part/
```

In fact, this command was used to generate the previous example. The parameter to the title command is made up of the text of the three parts, with each segment enclosed within a pair of delimiter characters. Here, the delimiter is a slash, but any other character can be used as long as it is used consistently within the same command. If one or more segments are to be omitted, indicate this with two adjacent delimiters at the desired position. Thus,

```
.tl ///Page 1/
```

specifies only the third segment and would produce something like this:

```
                                                    Page 1
```

It is not necessary to include the trailing delimiters.

To facilitate page numbering, you may include the sharp character ("#") anywhere in the text of the title; when the command is actually performed, "fmt" will replace all occurrences of the "#" with the current page number. To produce a literal sharp character in the heading, it should be preceded by an "\"

```
.#
```

so that it loses its special meaning.

The first segment of a title always starts at the left margin as specified by the lm command. While the third segment normally ends at the right margin as specified by the rm command, this can be changed with the "length-of-title" command:

```
.lt ±N
```

which changes the length of subsequence titles to ±N, still beginning at the left margin. Note that the title length is automatically set by the lm and rm commands to coincide with the distance between the left and right margins.

6.5.2 Page Headings and Footings

The most common uses for three part titles are page headings and footings. The header and footer lines are initially blank. Either one or both may be set at any time, without a break, by using the 'header' command

```
.he /left/center/right/
```

to set the page heading, and the 'footer' command

```
.fo /left/center/right/
```

to set the page footing. The change will become manifest the next time the top or the bottom of a page is reached. As with the tl command, the "#" may be used to access the current page number. As an illustration, the following commands were used to generate the page headings and footings for this guide:

```
.he /'Fmt' User's Guide///
.fo //- # -//
```

6.5.3 Summary - Headings, Footers and Titles

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.fo 'l'c'r'	blank	blank	no	Set running page footing.
.he 'l'c'r'	blank	blank	no	Set running page heading.
.lt ±#	#=50	#=60	no	Set length of header, footer and titles.
.tl 'l'c'r'	blank	blank	yes	Generate a three part title.

6.6. TABULATION

6.6.1 Tabs

Just like any good typewriter, 'fmt' has facilities for tabulation. When it encounters a special character in its input called the 'tab character' (analogous to the TAB key on a typewriter), it automatically advances to the next output column in which a 'tab stop' has been previously set. Tab stops are always measured from the effective left margin, that is, the left margin plus the current indentation or temporary indentation value. Whatever column the left margin may actually be in, it is always assumed to be column one for the purpose of tabulation.

Originally, a tab stop is set in every eighth column, starting with column nine. This may be changed using the 'ta' command

.ta <col> <col> ...

Each parameter specified must be a number, and causes a tab stop to be set in the corresponding output column. All existing stops are cleared before setting the new ones, and a stop is set in every column beyond the last one specified. This means that if no columns are specified, a stop is set in every column.

By default, 'fmt' recognizes the ASCII TAB, control-i, as the 'tab character', but since this is an invisible character and is guaranteed to be interpreted differently by different terminals, it can be changed to any character with the 'tab-character' command:

.tc <char>

While there is no restriction on what particular character is specified for <char>, it is wise to choose one that doesn't occur elsewhere in the text. If you omit the parameter, the tab character reverts to the default.

When 'fmt' expands a tab character, it normally puts out enough blanks to get to the next tab stop. In other words, the default 'replacement' character is the blank. This too may easily be changed with the 'replacement-character' command:

.rc <char>

As with the tc command, <char> may be any single character. If omitted, the default is used.

A common alternate replacement character is the period, which is frequently used in tables of contents. The following example illustrates how one might be constructed:

```
.ta 52
.tc \
Section Name\Page
.rc .
.sp
.nf
.ta 53
Basics\1
Filling and Margin Adjustment\2
Spacing and Page Control\5
.sp
.fi
```

The result should look about like this:

```
Section Name
Page

Basics.....1
Filling and Margin Adjustment.....2
Spacing and Page Control.....5
```

6.6.2 Summary - Tabulation

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.ta N ...	9 17 ...	all	no	Set tab stops.
.tc c	TAB	TAB	no	Set tab character.
.rc c	BLANK	BLANK	no	Set tab replacement character.

6.7. MISCELLANEOUS COMMANDS

6.7.1 Comments

It is rare that a document survives its writing under the pen of just one author or editor. More frequently, several different people are likely to put in their two cents worth concerning its format or content. So, if the author is particularly attached to something he has written, he is well advised to say so. Comments are an ideal vehicle for this purpose and are easily introduced with the "comment" comman

```
.# <commentary text>
```

Everything after the # up to and including the next newline character is completely ignored by 'fmt'.

6.7.2 Boldfacing and Underlining

'Fmt' makes provisions for **boldfacing** and <u>underlining</u> lines or parts thereof with two commands:

```
.bf N
```

boldfaces the next N lines of input text, while

```
.ul N
```

underlines the next N lines of input text. In both cases, if N is omitted, a value of one is assumed. Neither command causes a break, allowing single words or phrases to be boldfaced or underlined without affecting the rest of the output line.

It is also possible to use the two in combination. For instance, the section heading at the beginning of this section was produced by a sequence of commands and text similar to the following:

```
.ce
.bf
.ul
Miscellaneous Commands
```

As with the 'center' command, these two commands are often conveniently used to bracket the lines to be affected, i.e. specifying a huge parameter value with the first occurrence of the command and a value of zero with the second:

```
.bf 1000
.ul 1000
lots of lines
to be
boldfaced
and
underlined
.bf 0
.ul 0
```

6.7.3 Control Characters

As mentioned in the first section, command lines are distinguished from text by the presence of a 'control character' in column one. In all the examples cited thus far, a period has been used to represent the control character. It is possible to select any character for this purpose. In fact, several occasions arose in the writing of this appendix which called for use of an alternate control character, particularly in the construction of the command summaries at the end of each section. The 'control-character' command may be used anywhere to select a new value:

.cc <char>

The parameter <char>, which may be any single character, becomes the new control character. If the parameter is omitted, the familiar period is reinstated.

It has been shown that many commands automatically cause a break before they perform their function. When this presents a problem, it can be altered. If instead of using the basic control character the 'no-break' control character is used to introduce a command, the automatic break that would normally result is suppressed. The standard no-break control character is the grave accent (`), but may easily be changed with the following command:

.c2 <char>

As with the cc command, the parameter may be any single character, or may be omitted if the default value is desired.

6.7.4 Prompting

Brief, one-line messages may be written to the user's terminal using the 'prompt' command

.er <brief, one-line message>

The text that is actually written to the terminal starts with the first non-blank character following the command name, and continues up to, but not including, the next newline character. If a newline character should be included in the message, the escape sequence

\n

may be used. Leading blanks may also be included in the message by preceding the message with a quote or an apostrophe. 'Fmt' will discard this character, but will then print the rest of the message verbatim. For instance,

```
.er '     this is a message with 5 leading blanks
```

would write the following text on the terminal, leaving the cursor or carriage at the end of the message

```
     this is a message with 5 leading blanks
```

For a multiple-line message, try

```
.er multiple^nline@nmessage&n
```

The output should look like this:

```
multiple
line
message
```

Prompts are particularly useful in form letter applications where there may be several pieces of information that 'fmt' has to ask for in the course of its work. The next section describes how 'fmt' can dynamically obtain information from the user.

6.7.5 Premature Termination

If 'fmt' should ever encounter an 'exit' command

```
.ex
```

in the course of doing its job, it will cause a break and exit immediately to the Subsystem.

6.7.6 Summary - Miscellaneous Commands

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.*	-	-	no	Introduce a comment.
.bf N	N=0	N=1	no	Boldface N input text lines.
.c2 c	'	'	no	Set no-break control character.
.cc c	.	.	no	Set basic control character.
.er text	-	ignored	no	Write a message to the terminal.
.ex	-	-	yes	Exit immediately to the Subsystem.
.ul N	N=0	N=1	no	Underline N input text lines.

6.8. INPUT PROCESSING

6.8.1 Input File Control

Up to this point, it has been assumed that "fmt" reads only from its standard input file or from files specified as parameters on the command line. It is also possible to dynamically include the contents of any file in the midst of formatting another. This aids greatly in the modularization of large, otherwise unwieldy documents, or in the definition of frequently used sequences of commands and text.

The "source" command is available to dynamically include the contents of a file:

.so <file>

The parameter <file> is mandatory; it may be an arbitrary file system pathname, or, as with file names on the command line, a single dash ("-") to specify standard input number one.

The effect of a 'source' command is to temporarily preempt the current input source and begin reading from the named file. When the end of that file is reached, the original source of input is resumed. Files included with 'source' commands may themselves contain other 'source' commands; in fact, this 'nesting' of input files may be carried out to virtually any depth.

'Fmt' provices one additional command for manipulating input files. The 'next file' command

```
.nx <file>
```

may be used for either one of two purposes. If you specify a <file> parameter, all current input files are closed (including those opened with so commands), and the named file becomes the new input source. You can use this for repeatedly processing the same file, as, for example, with a form letter. If you omit the <file> parameter, 'fmt' still closes all of its current input files. But instead of using a file name you supply with the nx command, it uses the next file named on the command line that invoked 'fmt'. If there is no next file, then formatting terminates normally.

Neither the so command nor the nx command causes a break.

6.8.2 Functions and Variables

Whenever 'fmt' reads a line of input, no matter what the source may be, there is a certain amount of 'pre-processing' done before any other formatting operations take place. This pre-processing consists of the interpretation of 'functions' and 'variables'. A 'function' is a predefined set of actions that produces a textual result, possibly based on some user supplied textual input. For example: one hypothetical function might be named 'time', and its result might be a textual representation of the current time of day:

```
09:31:19
```

A 'variable', on the other hand, is simply one of 'fmt's internal parameters, such as the current page length or the current line-spacing value. The result of a variable is just a textual representation of the value of that parameter.

From the standpoint of a user, functions and variables are very similar. In fact, they are invoked identically by enclosing the appropriate name in square brackets:

```
[time]
[ls]
```

When 'fmt' sees such a construct in an input line, it exises everything in between the brackets, including the brackets themselves, and inserts the result of the function or variable in its place. Naturally, anything not recognizable as a variable or a function is left alone. To place the string "[time]" in some text, just type "\[time]"; the "\" preceding the left bracket makes it lose its special meaning.

The 'number registers' are ten accumulators on which simple arithmetic operations can be performed. They exist in the formatter specifically for the implementation of automatically numbered sections and paragraphs.

At this writing, functions and variables are still evolving. But the features just described are rather stable and will probably remain so in future expansions. At this point, the available functions and variables are:

Functions:

date	Current date; e.g. 09/28/79
day	Current day of the week; e.g. friday
time	Current time of day; e.g. 09:31:32

Variables:

cc	Current basic control character
c2	Current no-break control character
in	Current indentation value
lm	Current left margin value
ln	Current line number on the page
ls	Current line-spacing value
ml	Current macro invocation level
n1	Current margin 1 value
n2	Current margin 2 value
n3	Current margin 3 value
n4	Current margin 4 value
pl	Current page length value
pn	Current page number
po	Current page offset value
rm	Current right margin value
tc	Current tab character
ti	Current temporary indentation value

6.8.3 Summary - Input Processing

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.nx file	-	next arg	no	Move on to the next input file.
.so file	-	ignored	no	Temporarily alter the input source.

6.9. MACROS

6.9.1 Macro Definition

A macro is nothing more than a frequently used sequence of commands and/or text that have been grouped together under a single name. This name may then be used just like an ordinary command to invoke the whole group in one fell swoop.

The definition (or redefinition) of a macro starts with a ".de" command

```
.de xx
```

whose parameter is a one or two character string that becomes the name of the macro. The macro name may consist of any characters other than blanks, tabs or newlines; upper and lower case are significant. The definition of the macro continues until a matching ".en" command

```
.en xx
```

is encountered. Anything may appear within a macro definition, including other macro definitions. The only processing that is done during definition is the interpretation of variables and functions (i.e. things surrounded by square brackets). Other than this, lines are stored exactly as they are read from the input source. To include a function call in the definition of a macro so that its interpretation will be delayed until the macro is invoked, the opening bracket should be preceded by the escape character "@". For example,

```
.# tm --- time of day
.de tm
@[time]
.en tm
```

would produce the current time of day when invoked, whereas

```
.# tm --- time of day
.de tm
[time]
.en tm
```

would produce the time at which the macro definition was processed.

6.9.2 Macro Invocation

Again, a macro is invoked like an ordinary command: a control character at the beginning of the line immediately followed by the name of the macro. So to invoke the above "time-of-day" macro, one might say

```
.tm
```

As with ordinary commands, macros may have parameters. In fact, anything typed on the line after the macro name is available to the contents of the macro. As usual, blanks and tabs serve to separate parameters from each other and from the macro name. If it is necessary to include a blank or a tab within a parameter, it may be enclosed in quotes. Thus,

```
"parameter one"
```

would constitute a single parameter and would be passed to the macro as

```
parameter one
```

To include an actual quotation mark within the parameter, type two quotes immediately adjacent to each other. For instance,

```
"""quoted string"""
```

would be passed to the macro as the single parameter

```
"quoted string"
```

Within the macro, parameters are accessed in a way similar to functions and variables: the number of the desired parameter is enclosed in square brackets. Thus,

```
[1]
```

would retrieve the first parameter,

```
[2]
```

would fetch the second, and so on. As a special case, the name of the macro itself may be accessed with

```
[0]
```

Assume there is a macro named "mx" defined as follows:

```
.# mx --- macro example
.de mx
Macro named '[0]', invoked with two arguments:
'[1]' and '[2]'.
.en mx
```

Then, typing

```
.mx "param 1" "param 2"
```

would produce the same result as typing

```
Macro named 'mx', invoked with two arguments:
'param 1' and 'param 2'.
```

Macros are quite handy for such common operations as starting a new paragraph, or for such tedious tasks as the construction of tables like the ones appearing at the end of each section in this guide. For some examples of frequently used macros, see the applications notes in the following pages.

6.9.3 Summary - Macros

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.de xx	-	ignored	no	Begin definition or redefinition of a macro.
.en xx	-	ignored	no	End macro definition.

6.10. APPLICATIONS NOTES

The next few sections will illustrate the capabilities of 'fmt' with some actual applications. Most of the examples are macros that assist in common formatting operations, but attention has also been given to table construction. All of the macros presented here are available for general use in the file '/extra/fmacro/report', which may be named on the command line invoking 'fmt' or may be included with a 'source' command as follows:

```
.so /extra/fmacro/report
```

6.10.1 Paragraphs

One standard way of beginning a new paragraph is to skip a line and indent by a few spaces. This can be done by giving an **sp** command followed by a **ti** command. A better way is to define a macro. This allows procrastination on deciding the format of paragraphs and facilitates change at some later date without a major editing effort.

Here is an example of a paragraph macro:

```
.# pp --- begin paragraph
.de pp
.sp
.ne 2
.ti 0[in]
.ti +5
.ns
.en pp
```

First a line is skipped via the "space" command. Then, after checking that there is room on the current page for the first two lines of the new paragraph, a temporary indentation is set up that is five columns to the right of the running indentation with the two **ti** commands. Finally, no-space mode is turned on to suppress unwanted blank lines.

6.10.2 Sub-headings

Sub-headings may be easily produced with the following macro:

```
.# sh --- sub-heading
.de sh
.sp 2
.ne 4
.ti 0[in]
.bf
[1]
.pp
.en sh
```

First, two blank lines are put out. Then it is determined if there are four contiguous lines on the current page: one for the heading itself, one for the blank line after the heading, and two for the first two lines of the next paragraph. The temporary indentation value is then set to coincide with the current indentation value. Next, the first parameter passed to the macro (the text of the sub-heading) is boldfaced and a new paragraph is begun. The "pp" macro will insert the blank line after the heading.

6.10.3 Major Headings

An example of a macro to produce major headings is the following:

```
.# mh --- major heading
.de mh
.sp 3
.ne 5
.ce
.ul
.bf
[1]
.sp
.pp
.en mh
```

This is similar to the sub-heading macro: three blank lines are put out; a check for enough room is made; the parameter is centered, underlined and boldfaced; another blank line is put out; and a new paragraph is begun.

6.10.4 Quotations

Lengthy quotations are often set apart from other text by altering the left and right margins to narrow the width of the quoted text. Here is a pair of macros that may be used to delimit the beginning and end of a direct quotation:

```
.# bq --- begin direct quote
.de bq
.sp
.ne 2
.in +5
.rm -5
.lt +5
.en bq


.# eq --- end direct quote
.de eq
.sp
.in -5
.rm +5
.en eq
```

Notice the lt command in the first macro. To avoid affecting page headings and footings, the left margin is not adjusted; rather, an additional indentation is applied. But to increase the right margin width, there is no other alternative but to use the rm command. The 'title-length' command is thus necessary to allow headings and footings to remain unaffected by the interim right margin.

6.10.5 Italics

Since most printers can't easily produce italics, they are frequently simulated by underlining. The following macro "italicizes" its parameter by underlining it.

```
.# it --- italicize (by underlining)
.de it
.ul
[1]
.en it
```

6.10.6 Boldfacing

While "fmt" has built-in facilities for boldfacing, their use may be somewhat cumbersome if there are many short phrases or single words that need boldfacing; each phrase or word requires two input lines: one for the bf command and one for the actual text. The following macro cuts the overhead in half by allowing the command and the text to appear on the same line.

```
.# bo --- boldface parameter
.de bo
.bf
[1]
.en bo
```

6.10.7 Examples

This appendix is peppered with examples, each one set apart from other text by surrounding blank lines and additional indentation. The next two macros, used like the "bq" and "eq" macros, facilitate the production of examples.

```
.# bx --- begin example text
.de bx
.sp
.ne 2
.nf
.in +10
.en bx


.# ex --- end example text
.de ex
.sp
.fi
.in -10
.en ex
```

6.10.8 Table Construction

One example of table construction (for a table of contents) has already been mentioned in the section dealing with tabs. Another type of table that occurs frequently is that used in the command summaries in this appendix. Each entry of such a table consists of a number of "fields", followed on the right by a body of explanatory text that needs to be filled and adjusted.

The easiest way to construct a table like this involves using a combination of tabs and indentation, as the following series of commands illustrates:

```
.in +40
.ta 14 24 34 41
.tc \
```

The idea is to set a tab stop in each column that begins a field, and one last one in the column that is to be the left margin for the explanatory text. The extra indentation moves the effective left margin to this column. To begin a new entry, temporarily undo the extra indentation with a ti command, and then type the text of the entry, separating the fields from one another with a tab character:

```
.ti -40
field 1\tfield 2\tfield 3\tfield 4\tExplanatory text
```

The first line of the entry will start at the left margin. Then all subsequent lines will be filled and adjusted between column forty and the right margin.

6.11. SUMMARY OF COMMANDS SORTED ALPHABETICALLY

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.a	-	-	no	Introduce a comment.
.ad c	both	both	no	Set margin adjustment mode.
.bf N	N=0	N=1	no	Boldface N input text lines.
.bp ±N	N=1	next	yes	Begin a new page.
.br	-	-	yes	Force a break.
.c2 c	'	'	no	Set no-break control character.
.cc c	.	.	no	Set basic control character.
.ce N	N=0	N=1	yes	Center N input text lines.
.de xx	-	ignored	no	Begin definition or redefinition of a macro.
.en xx	-	ignored	no	End macro definition.
.er text	-	ignored	no	Write a message to the terminal.
.ex	-	-	yes	Exit immediately to the Subsystem.
.fi	on	-	no	Turn on fill mode.
.fo 'l'c'r'	blank	blank	no	Set running page footing.
.he 'l'c'r'	blank	blank	no	Set running page heading.
.hy	on	-	no	Turn on automatic hyphenation.
.in ±N	N=0	N=0	yes	Indent left margin.
.lm ±N	N=1	N=1	yes	Set left margin.

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.ls N	N=1	N=1	no	Set line spacing.
.lt ±N	N=60	N=60	no	Set length of header, footer and titles.
.m1 ±N	N=3	N=3	no	Set top margin before and including page heading.
.m2 ±N	N=2	N=2	no	Set top margin after page heading.
.m3 ±N	N=2	N=2	no	Set bottom margin before page footing.
.m4 ±N	N=3	N=3	no	Set bottom margin including and after page footing.
.na	-	-	no	Turn off margin adjustment.
.ne N	-	N=1	yes	Express a need for N contiguous lines.
.nf	-	-	yes	Turn off fill mode. (Also inhibits adjustment.)
.nh	-	-	no	Turn off automatic hyphenation.
.ns	on	-	no	Turn on "no-space" mode.
.nx file	-	next arg	no	Move on to the next input file.
.pl ±N	N=66	N=66	no	Set page length.
.pn ±N	N=1	ignored	no	Set page number.
.po ±N	N=0	N=0	yes	Set page offset.
.rc c	BLANK	BLANK	no	Set tab replacement character.
.rm ±N	N=60	N=60	yes	Set right margin.
.rs	-	-	no	Turn off "no-space" mode.

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.sb	off	-	no	Single blank after end of sentence.
.so file	-	ignored	no	Temporarily alter the input source.
.sp N	-	N=1	yes	Put out N blank lines.
.ta N ...	9 17 ...	all	no	Set tab stops.
.tc c	TAB	TAB	no	Set tab character.
.ti ±N	N=0	N=0	yes	Temporarily indent left margin.
.tl 'l'c'r'	blank	blank	yes	Generate a three part title.
.ul N	N=0	N=1	no	Underline N input text lines.
.xb	on	-	no	Extra blank after end of sentence.

6.12. SUMMARY OF COMMANDS GROUPED BY FUNCTION

6.12.1 Filling and Margin Adjustment

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.ad c	both	both	no	Set margin adjustment mode.
.br	-	-	yes	Force a break.
.ce N	N=0	N=1	yes	Center N input text lines.
.fi	on	-	no	Turn on fill mode.
.hy	on	-	no	Turn on automatic hyphenation.
.na	-	-	no	Turn off margin adjustment.
.nf	-	-	yes	Turn off fill mode. (Also inhibits adjustment.)
.nh	-	-	no	Turn off automatic hyphenation.
.sb	off	-	no	Single blank after end of sentence.
.xb	on	-	no	Extra blank after end of sentence.

6.12.2 Spacing and Page Control

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.bp ±N	N=1	next	yes	Begin a new page.
.ls N	N=1	N=1	no	Set line spacing.
.ne N	-	N=1	yes	Express a need for N contiguous lines.
.ns	on	-	no	Turn on 'no-space' mode.
.pl ±N	N=66	N=66	no	Set page length.
.pn ±N	N=1	ignored	no	Set page number.
.rs	-	-	no	Turn off 'no-space' mode.
.sp N	-	N=1	yes	Put out N blank lines.

6.12.3 Margins and Indentation

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.in ±N	N=0	N=0	yes	Indent left margin.
.lm ±N	N=1	N=1	yes	Set left margin.
.m1 ±N	N=3	N=3	no	Set top margin before and including page heading.
.m2 ±N	N=2	N=2	no	Set top margin after page heading.
.m3 ±N	N=2	N=2	no	Set bottom margin before page footing.
.m4 ±N	N=3	N=3	no	Set bottom margin including and after page footing.
.po ±N	N=0	N=0	yes	Set page offset.
.rm ±N	N=60	N=60	yes	Set right margin.
.ti ±N	N=0	N=0	yes	Temporarily indent left margin.

6.12.4 Headings, Footings and Titles

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.fo 'l'c'r'	blank	blank	no	Set running page footing.
.hc 'l'c'r'	blank	blank	no	Set running page heading.
.lt ±N	N=60	N=60	no	Set length of header, footer and titles.
.tl 'l'c'r'	blank	blank	yes	Generate a three part title.

6.12.5 Tabulation

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.ta N ...	9 17 ...	all	no	Set tab stops.
.tc c	TAB	TAB	no	Set tab character.
.rc c	BLANK	BLANK	no	Set tab replacement character.

6.12.6 Miscellaneous Commands

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.#	-	-	no	Introduce a comment.
.bf N	N=0	N=1	no	Boldface N input text lines.
.c2 c	'	'	no	Set no-break control character.
.cc c	.	.	no	Set basic control character.
.er text	-	ignored	no	Write a message to the terminal.
.ex	-	-	yes	Exit immediately to the Subsystem.
.ul N	N=0	N=1	no	Underline N input text lines.

6.12.7 Input Processing

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.nx file	-	next arg	no	Move on to the next input file.
.so file	-	ignored	no	Temporarily alter the input source.

6.12.8 Macros

Command Syntax	Initial Value	If no Parameter	Cause Break	Explanation
.de xx	-	ignored	no	Begin definition or redefinition of a macro.
.en xx	-	ignored	no	End macro definition.

APPENDIX 7

MACRO PROCESSOR
GEORGIA TECH SOFTWARE TOOLS SUBSYSTEM

"Macro" is an enhanced version of Kernighan and Plauger's macro preprocessor from Chapter 8 of Software Tools. "Macro" is an exceedingly complex and powerful program; it is possible to use it as a general programming language. A complete description of its capability is beyond the scope of this document, but a few samples are presented here to help the user become proficient in its usage.

7.1. THE FORMAT OF A MACRO DEFINITION

The basic format of a macro definition is:

```
define(macro-name, replacement-text)
```

"Macro-name" is an identifier, i.e. a sequence of letters or digits beginning with a letter. "Replacement-text" is a (possibly empty) sequence of characters, which may be specially interpreted by "macro".

Macro arguments are referred to by a construct of the form "$<integer>" in the replacement text. The <integer> must be a digit from 0 to 9, inclusive. (Digits 1-9 represent the first through the ninth arguments; digit 0 represents the name of the macro itself). For example, the following macro could be used to skip blanks and tabs in a string, starting at a given position:

```
define(skipbl,
   while ($1 ($2) == BLANK | $1 ($2) == TAB)
      $2 = $2 + 1
   )
```

Here are a few examples of the use of this macro:

```
skipbl(line, i)
skipbl(str, j)
```

In order to prevent premature evaluation of a string, the string may be surrounded by square brackets. For example, suppose we wished to redefine an identifier. The following sequence will not work:

```
define(x,y)
define(x,z)
```

This is because "x" in the second definition will be replaced by "y", with the net result of defining "y" to be "z". The correct method is

```
define(x,y)
define([x],z)
```

The square brackets prevent the premature evaluation of "x".

7.2. BUILT-IN FUNCTIONS

'Macro' provides several "built-in" functions. These are given below:

divert(filename) or divert(filename,append) or divert
"Filename" is opened for output and its file descriptor is stacked. Whenever 'macro' produces output, it is directed to the named file. If the second argument is present, output is appended to the named file, rather than overwriting it. If both arguments are missing, the current output file is closed and output reverts to the last active file (the one in use when the 'divert' command was recognized).

dnl or dnl(commentary information)
As suggested by Kernighan and Plauger, 'dnl' may be used to delete all blanks and tabs up to the next newline, and the newline itself, from the input stream. There is no other way to prevent the newline after each 'define' from being passed to the output. Any arguments present are ignored, thus allowing 'dnl' to be used to introduce comments.

ifelse(a,b,c,d)
If a and b are the same string, then c is the value of the expression; otherwise, d is the value of the expression. Example: this macro returns "OK" if the value of i is "1", "ERR" otherwise:

```
define(status,ifelse(i,1,OK,ERR))
```

include(filename)
"Filename" is opened and its file descriptor is stacked. The next time 'macro' requests input, it receives input from the named file. When end-of-file is seen, 'macro' reverts to the last active input file (the one containing the include) and picks up where it left off.

incr(n)
increment the value of the integer represented by n, and return the incremented value. For instance, the following pair of defines set MAXCARD to 80 and MAXLINE to 81:

```
define(MAXCARD,80)
define(MAXLINE,incr(MAXCARD))
```

substr(s,m,n)
return a substring of string s starting at position m with length n. substr(abc,1,2) is ab; substr(abc,2,1) is b; substr(abc,4,1) is empty. If n is omitted, the rest of the string is used: substr(abc,2) is bc.

undefine(name)
"Undefine" is used to remove the definition associated with a name. Note that the name should be surrounded by brackets, if it is supplied as a literal, otherwise it will be evaluated before it can be undefined. Example:

```
undefine([substr])
undefine ([x])
```

APPENDIX 8

THE PRIMOS FILE SYSTEM

The PRIMOS operating system for the Prime 400 computer supports as one of its services a flexible, hierarchical file system that provides users with the facility to maintain large quantities of data in an orderly, logical manner. These next few sections are indended to provide a brief overview of the file system's capabilities and features. It is somewhat tutorial in nature and does not attempt to cover all of the available features, nor to present the details of implementation.

8.1. ORDINARY FILES

A file is a named collection of information, preserved upon some storage medium, such as magnetic disk. Some files may contain text as in an article or a book, others may contain binary data produced by or to be used as input to some program, and still others may contain the actual executable instructions of the program itself. In other words, no particular structure is forced upon a file by the system. While some programs may operate on files with a definite structure, it is their responsibility to maintain this structure and of no real concern to the system.

We mentioned above that a file has a name; this raises the question of what are the acceptable names for files. A file name consists of 32 or fewer characters chosen from the Roman alphabet, the Arabic digits and the following special characters:

```
$  %  &  *  -  .  /  _
```

The first character, however, must not be a digit. Use of the slash ("/") is strongly discouraged for reasons which will soon become apparent. The case of the alphabetic characters is insignificant since the system forces them all to upper case. Thus

```
GEORGE  Harry  my_file  Your.File  file1
```

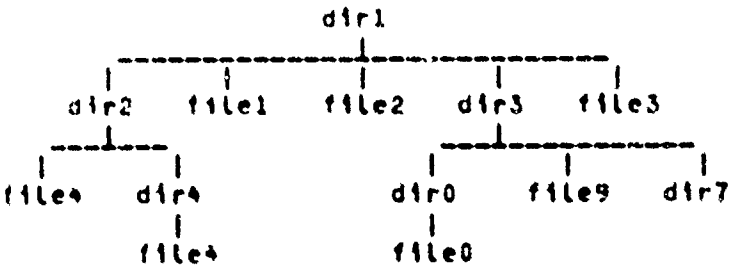
are all legal file names, while

```
(bad_file_name)  naughty_file!  666
```

are not.

B.2. DIRECTORIES

PRIMOS associates the name of a file with its contents through the use of "directories", which themselves are nothing more than ordinary files that PRIMOS treats specially. A directory contains a number of "entries", each of which holds the name of a file plus other information (which we will get to later), and the location on the storage volume of the actual contents of the file. Each file with a corresponding entry in a given directory is said to "reside within" that directory, and that directory is said to "contain" all files for which it holds an entry. Now there is nothing that prevents us from having within a directory a file that is itself a directory. This phenomenon is known as "nesting" of directories and may be carried out to any depth, giving rise to a hierarchical structure:



It should be noted that while the names of all files within a given directory must be unique among themselves, it is perfectly legal to have, in separate directories, two different files with the same name. So, in the example above, file4 that resides within dir4 is distinctly separate from file4 that resides within dir2.

At the topmost level of the hierarchy is the master file directory (MFD), which always begins at a fixed location on a given storage volume. In addition to several entries required by the system, the MFD contains any other directories and/or ordinary files that an installation may see fit to maintain. (Normally, a directory is established within the MFD for each individual user of the system. Each such directory is known as a user file directory or UFD.)

Now that we have a structure containing multiple nested directories and potentially duplicate file names, the problem arises as to how a specific file, say the one named "file0" in the example, is referenced; or worse, how one file named "file4" is referenced distinctly from another by the same name but in a different directory. Pathnames, which we take up in the next section, are the solution.

8.3. PATHNAMES

A pathname is a syntax for uniquely specifying any file contained within the file system. But before we can go any further, a couple of details should be brought to light. When a user logs in to PRIMOS, he is automatically "attached" to a specific directory whose name is usually the same as his login name. This directory is said to be his "current" or "default" directory and has a special significance in the interpretation of pathnames. There is a way to walk around within the file system, changing the current directory, but we won't go into that here.

As we said, a pathname allows you to uniquely specify any file anywhere in the hierarchy by describing a path to the file from some known point. Two such known points are the current directory and the MFD. A pathname, then, consists of a number of directory names, separated by slashes ("/", recall our previous admonition) and ending with the name of the desired file. If the pathname starts with a slash, the path starts in the MFD; otherwise, it starts in the current directory. A simple file name that contains no slashes refers to a file within the current directory; a pathname consisting only of a single slash refers to the MFD itself; and the empty pathname refers to the current directory. Thus, the pathnames

```
/bin/cd
mydir/file
/
(empty string)
```

refer to a file named "cd" within a directory named "bin" which is contained in the MFD; a file named "file" within a directory named "mydir" which sprouts from the current directory, wherever that may be; the MFD; and the current directory, respectively.

8.4. PROTECTION

Users have the option, if they so desire, to protect their files from unwanted perusal or alteration by other users. Two mechanisms are involved in providing this feature. First, with each directory is associated an "owner" and "password". The owner of a directory is, by default, the user who created it, though this may be changed. When a directory becomes a user's current directory, either of two conditions may prevail: the user may be declared "owner" of the directory if his login name matches that of the directory's creator, or failing this, he is declared a "non-owner". The password comes into play in this latter case. The owner of a directory may require that any non-owner who wishes to make it his current directory must first specify a password. If he is able to do this, then he is attached to the directory as a non-owner; otherwise, he is not allowed

to attach to it.

Once attached to a directory, either as owner or non-owner, the second protection mechanism comes into play. As part of the "other information" that we mentioned in the directory entry for a file, the system keeps two sets of "protection keys", one that applies to the owner of the directory, and the other, to non-owners. These keys, which may be changed on a per-file basis by the directory's owner, control the kinds of things that can be done to a file. There are three operations that may be individually allowed or denied to both owner and non-owners: reading, writing and truncating (deleting). If a user has read permission for a file, he may read the contents of it but may in no way alter them. If write permission is granted, then the file may be written upon (possibly overwriting existing information or extending the file), but nothing may be read from it. If truncate permission is granted, the file may be shortened to any length or even removed completely, but its contents may neither be read nor written.

B.5. SUMMARY

There are many more features and subtleties in the PRIMOS file system that both enhance its power and usefulness, as well as add to its awkwardness. Those things we have talked about here, though, seem to be the most important for a general understanding. For further details on these other features and on the implementation of the file system, you might want to consult the Reference Guide, File Management System, published as PDR3110 by Prime Computer, Inc., Framingham, Mass.

APPENDIX 9

COBOL.wbc

This appendix contains some of the preliminary work towards defining COBOL.wbc that has been accomplished. The first section provides some insight into the frequency of the use of various features of COBOL, while the second section shows the actual discrepancies discovered between some of these features.

9.1. USAGE OF COBOL FEATURES

A Study of the Usage of COBOL Features

The listing below provides the frequency (static) of use of reserved words discovered in a file containing about 20,000 lines of code from Army programs.

ACCEPT	0	CF	0
ACCESS	12	CH	0
ACTUAL	0	CHARACTER	0
ADD	136	CHARACTERS	27
ADDRESS	0	CLOCK-UNITS	0
ADVANCING	0	CLOSE	20
AFTER	0	COBOL	0
ALL	2	CODE	0
ALPHABETIC	1	CODE-SET	0
ALSO	0	COLLATING	0
ALTER	0	COLUMN	0
ALTERNATE	0	COMMA	0
AN	2	COMMON	0
AND	36	COMMUNICATION	0
APPLY	0	COMP	94
ARE	22	COMPUTATIONAL	0
AREA	0	COMPUTE	47
AREAS	4	CONFIGURATION	6
ASCENDING	0	CONTAINS	47
ASSIGN	41	CONTROL	0
AT	59	CONTROLS	0
AUTHOR	4	COPY	0
BEFORE	0	CORR	0
BEGINNING	0	CORRESPONDING	0
BLANK	0	COUNT	0
BLOCK	20	CURRENCY	0
BOTTOM	0	DATA	16
BY	269	DATE	0
CALL	59	DATE-COMPILED	1
CANCEL	0	DATE-WRITTEN	12
CD	0	DAY	0

DE	0	GO	1243
DEBUG-CONTENTS	0	GREATER	148
DEBUG-ITEM	0	GROUP	0
DEBUG-LINE	0	HEADING	0
DEBUG-NAME	0	HIGH-VALUE	0
DEBUG-SUB-1	0	HIGH-VALUES	27
DEBUG-SUB-2	0	I-O	3
DEBUG-SUB-3	0	I-O-CONTROL	0
DEBUGGING	0	IDENTIFICATION	13
DECIMAL-POINT	0	IF	886
DECLARATIVES	0	IN	0
DELETE	0	INDEX	3
DELIMITED	0	INDEXED	82
DELIMITER	0	INDICATE	0
DEPENDING	18	INITIAL	0
DESCENDING	0	INITIALIZE	0
DESTINATION	0	INPUT	14
DETAIL	0	INPUT-OUTPUT	13
DISABLE	0	INSPECT	2
DISPLAY	377	INSTALLATION	11
DIVIDE	0	INTO	2
DIVISION	56	INVALID	39
DOWN	16	IS	337
DUPLICATES	0	JUST	0
DYNAMIC	0	JUSTIFIED	0
EGI	0	KEY	61
ELSE	728	KEYS	0
EMI	0	LABEL	41
ENABLE	0	LAST	1
END	54	LEADING	1
END-OF-PAGE	0	LEFT	0
ENDING	0	LENGTH	0
ENTER	0	LESS	37
ENVIRONMENT	14	LIMIT	0
EOP	0	LIMITS	0
EQUAL	357	LINAGE	0
ERROR	1	LINAGE-COUNTER	0
ESI	0	LINE	0
EVERY	0	LINE-COUNTER	0
EXAMINE	0	LINES	0
EXCEPTION	0	LINKAGE	8
EXIT	193	LOCK	0
EXTEND	0	LOW-VALUE	7
FD	41	LOW-VALUES	9
FILE	22	MEMORY	0
FILE-CONTROL	13	MERGE	0
FILE-LIMIT	0	MESSAGE	0
FILE-LIMITS	0	MODE	22
FILLER	1012	MODULES	2
FINAL	0	MOVE	3364
FIRST	0	MULTIPLE	0
FOOTING	0	MULTIPLY	5
FOR	6	NATIVE	0
FROM	113	NEGATIVE	0
GENERATE	0	NEXT	583
GIVING	42	NO	4

NOT	153	REPORT	0
NOTE	0	REPORTING	0
NUMBER	0	REPORTS	0
NUMERIC	2	RERUN	0
OBJECT-COMPUTER	0	RESERVE	0
OBJECT-PROGRAM	0	RESET	0
OCCURS	104	RETURN	0
OF	19	REVERSED	0
OFF	0	REWIND	0
OMITTED	1	REWRITE	10
ON	23	RF	0
OPEN	21	RH	0
OPTIONAL	0	RIGHT	0
OR	162	ROUNDED	7
ORGANIZATION	3	RUN	26
OUTPUT	8	SA	0
OV	0	SAME	0
OVERFLOW	0	SD	0
PAGE	0	SEARCH	13
PAGE-COUNTER	0	SECTION	84
PERFORM	668	SECURITY	0
PF	0	SEEK	0
PH	0	SEGMENT	0
PIC	4833	SEGMENT-LIMIT	0
PICTURE	9	SELECT	41
PLUS	0	SEND	0
POINTER	0	SENTENCE	583
POSITION	0	SEPARATE	0
POSITIVE	0	SEQUENCE	0
PRINTING	0	SEQUENCED	0
PROCEDURE	14	SEQUENTIAL	4
PROCEDURES	0	SET	262
PROCEED	0	SIGN	0
PROCESSING	0	SIZE	0
PROGRAM	28	SORT	0
PROGRAM-ID	14	SORT-MERGE	0
QUEUE	0	SOURCE	0
QUOTE	0	SOURCE-COMPUTER	5
QUOTES	0	SPACE	61
RANDOM	7	SPACES	222
RD	6	SPECIAL-NAMES	0
READ	56	STANDARD	41
RECEIVE	0	STANDARD-1	0
RECORD	61	START	8
RECORDS	43	STATUS	0
REDEFINES	147	STOP	26
REEL	0	STRING	0
REFERENCES	0	SUB-QUEUE-1	0
RELATIVE	0	SUB-QUEUE-2	0
RELEASE	0	SUB-QUEUE-3	0
REMAINDER	0	SUBTRACT	40
REMARKS	3	SUM	0
REMOVAL	0	SUPPRESS	0
RENAMES	0	SYMBOLIC	0
RENAMING	0	SYNC	23
REPLACING	0	SYNCHRONIZED	0

TABLE	2	UNSTRING	0
TALLY	7	UNTIL	86
TALLYING	2	UP	95
TAPE	0	UPON	0
TERMINAL	0	USAGE	0
TERMINATE	0	USE	0
TEXT	0	USING	65
THAN	177	VALUE	1724
THEN	0	VALUES	0
THROUGH	2	VARYING	68
THRU	456	WHEN	11
TIME	0	WITH	0
TIMES	80	WORDS	0
TO	5091	WORKING-STORAGE	14
TOP	0	WRITE	21
TRAILING	1	ZERO	486
TYPE	0	ZEROES	26
UNIT	0	ZEROS	58

9.2. COMPARISON OF COBOL FEATURES

Some Examples of the Comparison of COBOL Features Offered by the 74 COBOL Standard, PDP-11 COBOL, and PRIME COBOL

The following are excerpts from a more complete study of the differences found in various features of COBOL on several machines.

ACCEPT

ACCEPT identifier-1 [FROM mnemonic-name]

ACCEPT identifier-1 FROM (DATE | DAY | TIME)

1) 74 COBOL

   a) Data is transferred according to the rules of MOVE.

   b) Mnemonic-name must be specified in the SPECIAL-NAMES paragraph.

   c) If the FROM phrase is not given, the device that the implementor specifies as standard is used.

   d) If the size of the data being transferred is identical to the size of identifier-1, the data is transferred and stored in identifier-1.

e) If the size of identifier-1 is less than the size of the data being transferred, the left-most characters of the data being transferred are stored in identifier-1, left-justified; characters to the right are ignored.

f) If the size of identifier-1 is greater than the size of the data being transferred, the data is stored left-justified and additional data is requested. For subsequent data transfers, the size of identifier-1 is taken to be equal to the as yet unfilled portion of it.

g) DATE is implicitly described as an elementary data item with PICTURE of 9(6)V; it contains year of the century, month of the year, and day of the year in that order.

h) DAY is implicitly described as an elementary data item with PICTURE of 9(5)V; it contains year of the century and the day of the year numbered from 1 to 366.

i) TIME is implicitly described as an elementary data item with PICTURE of 9(8)V; the value represents, in a 24-hour system, the number of elapsed hours, minutes, seconds, and hundredths of seconds after midnight in that order from left to right.

2) PDP-11

a) As in 74 COBOL.

b) As in 74 COBOL.

c) If the FROM phrase is not given, the data is transferred from the user's terminal.

d) As in 74 COBOL.

e) As in 74 COBOL.

f) If the size of identifier-1 is greater than the size of the data being transferred, this data is stored left-justified and the remaining space padded with blanks.

g) As in 74 COBOL.

h) This is identical to DATE.

i) As in 74 COBOL, but the positions for the hundredths of a second are filled with zeroes.

3) PRIME

a) Characters are moved without change.

b) As in 74 COBOL.

c) If the FROM phrase is not given, the data is transferred from the user's terminal.

d) As in 74 COBOL.

e) As in 74 COBOL.

f) If the size of identifier-1 is less than the size of the data being transferred, the left-most characters of the data being transfered are stored in identifier-1, left-justified, and the remaining space is filled with blanks.

g) As in 74 COBOL.

h) As in 74 COBOL.

i) TIME is implicitly described as an elementary data item with PICTURE of 9(6)V; it contains hours, minutes, and seconds in that order.

ADD

ADD (identifier-1 | literal-1) [, (identifier-2 | literal-2)] ... TO identifier-m [ROUNDED] [, identifier-n [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]

ADD (identifier-1 | literal-1) , (identifier-2 | literal-2) [, (identifier-3 | literal-3)] ... GIVING identifier-m [ROUNDED] [, identifier-n [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]

ADD (CORRESPONDING | CORR) identifier-1 TO identifier-2 [ROUNDED] [; ON SIZE ERROR imperative-statement]

1) 74 COBOL

a) In formats 1 and 2, each identifier must refer to an elementary numeric item, except that in format 2 each identifier following the word GIVING must refer to either an elementary numeric item or an elementary numeric edited item; in format 3, each identifier must refer to a group item.

b) Each literal must be a numeric literal.

c) The composite of operands must not exceed 18 digits.

d) In format 1, values of operands preceding the word TO are added together; then the sum is added to the current value of identifier-m storing the result immediately into identifier-m, and repeating this process respectively for each operand following the word TO.

e) In format 2, the values of the operands preceding the word GIVING are added together; then the sum is stored as the new value of each identifier-m, identifier-n, ... the resultant-identifiers.

f) In format 3, the data items in identifier-1 are added to and stored in corresponding data items in identifier-2. A pair of data items, one from identifier-1 and one from identifier-2 correspond if (1) they are not designated by the word FILLER and have the same data-name and the same quantifiers up to, but not including, identifier-1 and identifier-2, (2) both data items are elementary numeric data items.

g) The compiler insures that enough places are carried so as not to lose any significant digits during execution.

h) Size error condition:
   i) A size error exists when, after execution of the addition and subsequent decimal point alignment, the integer portion of the absolute value of the result is too large to fit into the receiving item. This does not apply to intermediate results. If rounding is specified, it occurs prior to the check for the size error.
   ii) If a size error occurs and the SIZE ERROR phrase is specified, the values of the receiving items affected by size errors are not altered. After execution of the statement, the imperative statement of the SIZE ERROR clause is executed (it is executed only once per statement execution).
   iii) If a size error occurs and the SIZE ERROR phrase is not specified, the values of the items affected by size errors are unpredictable.

i) If truncation has occurred and the ROUNDED option is specified, then if the most significant digit of the truncated part is five or more, a one is added to the absolute value of the result.

2) PDP-11

a) As in 74 COBOL.

b) As in 74 COBOL.

c) If the size of the intermediate result field is greater than 18, the excess high- 'er digits are truncated.

d) As in 74 COBOL.

e) As in 74 COBOL.

f) As in 74 COBOL, but a pair of data items correspond if they are elementary numeric data items, and they have the same name.

g) A maximum of 18 digits are carried, thus anything over this is lost.

h) Size error condition:
   i) As in 74 COBOL.
   ii) As in 74 COBOL.
   iii) If a size error occurs and the SIZE ERROR phrase is not specified, the high-order digits are truncated.

i) As in 74 COBOL.

3) PRIME

ADD (identifier-1 | literal-1) [, (identifier-2 | literal-2)] ... TO identifier-m [ROUNDED] [; ON SIZE ERROR imperative-statement]

ADD (identifier-1 | literal-1) (, (identifier-2 | literal-2)) [, (identifier-3 | literal-3)] ... GIVING identifier-m [ROUNDED] [ON SIZE ERROR imperative-statement]

a) As in 74 COBOL, but cannot have multiple receiving identifiers or ADD CORRESPONDING.

b) As in 74 COBOL.

c) As in 74 COBOL.

d) As in 74 COBOL, but can only have one identifier after the word TO.

c) As in 74 COBOL, but can only have on identifier after the word GIVING.

f) The CORRESPONDING feature is not available.

g)

h) As in 74 COBOL.

i) As in 74 COBOL.

COMPUTE

COMPUTE identifier-1 [ROUNDED] [, identifier-2 [ROUNDED]] ... = arithmetic-expression [; ON SIZE ERROR imperative-statement]

1) 74 COBOL

a) Identifier-1, identifier-2, ... must be elementary numeric items or elementary numeric edited items.

b) This statement allows the user to combine arithmetic operations without the restrictions on composite of operands and/or receiving data items imposed by the statements ADD, SUBTRACT, MULTIPLY, and DIVIDE.

2) POP-11

a) As in 74 COBOL.

b) The lengths of the integer and decimal components of an intermediate result are determined by an algorithm which uses the lengths of the operands in determining these values. The sum of the lengths of the integer and decimal components must be less than 18.

3) PRIME

COMPUTE identifier-1 [ROUNDED] = arithmetic-expression [; ON SIZE ERROR imperative-statement]

a) As in 74 COBOL, but can only have one identifier before the equal sign.

b) The composite of operands must not contain more than 18 decimal digits.

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.