HONEYWELL

DPS 6 & LEVEL 6 GCOS 6 MOD 400 APPLICATION DEVELOPER'S GUIDE

SOFTWARE

DPS 6 & LEVEL 6
GCOS 6 MOD 400
APPLICATION DEVELOPER'S GUIDE

SUBJECT

Instruction in MOD 400 System Usage for Application Programmers

SOFTWARE SUPPORTED

See the MOD 400 Guide to Software Documentation for information about Executive Releases supported by this manual.

ORDER NUMBER

CZ15-00

December 1982

Honeywell

PREFACE

This manual has been written for the applications programmer. Its purpose is to provide the information needed to use GCOS 6 MOD 400 system services to write and run application programs.

The reader is assumed to have basic knowledge of application development and processing, and some programming experience in one of the three languages supported. The languages supported are COBOL, BASIC, or FORTRAN.

The major topics presented in this manual are:

- Terminal startup and user access procedures
- File management
- Screen Editor conventions, directives, and user procedures
- Line Editor conventions, directives, and user procedures
- COBOL, BASIC, and FORTRAN compile, link, and execute procedures
- Program debug utility user procedures

USER COMMENTS FORMS are included at the back of this manual. These forms are to be used to record any corrections, changes, or additions that will make this manual more useful.

File No 1R13, 1S13 CZ15-00

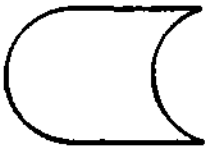
- Networking overview and processing capabilities
- Patch utility user procedures
- File backup and recovery procedures
- Memory dump interpretation procedures.

After reading this manual, the applications programmer should be familiar with MOD 400 system services and be able to write, debug, and run application programs.

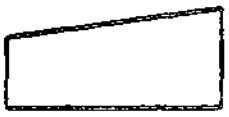
The notation conventions used in this manual follow. The first set of conventions applies to directive syntax as a whole; the second set applies to flow chart symbols; the third set applies to heading hierarchy; and the fourth set applies to user keyins.

Syntax Convention	Meaning
UPPERCASE CHARACTERS	Reserved keyword or symbol. Enter as shown.
lowercase characters	Variable field. Replace by a user-supplied value.
Brackets	Include none or one of the enclosed options.
Braces	Include one of the enclosed options.

Flowchart Symbols

Symbol	Meaning
	Process. Represents performance of a computer operation(s)
	Online storage. Represents information stored on diskette, cartridge disk, or storage module
	Printed card. Represents card input

Document. Represents printed output



Manual input, i.e., terminal input

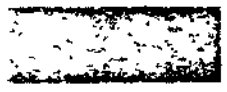
Mandatory. Indicates that designated flow of information, type of processing, or output is required.

Heading Hierarchy

The following conventions are used to indicate the relative levels of topic headings used in this manual.

Level 1 (highest)	ALL CAPITALS, UNDERLINED
Level 2	Initial Capitals, Underlined
Level 3	ALL CAPITALS, NOT UNDERLINED
Level 4	Initial Capitals, Not Underlined

User Keyins



Indicates user input to the system

MANUAL DIRECTORY

The following publications constitute the GCOS 6 MOD 400 manual set. Refer to the "Software/Manual Directory" of the Guide to Software Documentation for the current revision number and addenda (if any) of relevant release-specific publications.

Manuals are obtained by submitting a Honeywell Publications Order Form to the following address:

Honeywell Information Systems Inc.
47 Harvard Street
Westwood, MA 02090

Attn: Publications Services

Honeywell software reference manuals are periodically updated to support enhancements and improvements to the software. Before ordering any manuals, you should refer to the Guide to Software Documentation to obtain information concerning the specific edition of the manual that supports the software currently in use at your installation. If you use the four-character base publication number to order a document, you will receive the latest edition of the manual. The Publications Distribution Center can provide specific editions of a publication only when supplied with the seven- or eight-character order number listed in the Guide to Software Documentation.

Honeywell applications software packages, such as INFO 6, TOTAL 6, and TPS 6, provide specialized services. Contact your Honeywell representative for information concerning the availability of applications software and supporting documentation.

Base Publication Number	Manual Title
CZ01	GCOS 6 MOD 400 Guide to Software Documentation
CZ02	GCOS 6 MOD 400 System Building and Administration
CZ03	GCOS 6 MOD 400 System Concepts
CZ04	GCOS 6 MOD 400 System User's Guide
CZ05	GCOS 6 MOD 400 System Programmer's Guide - Volume I
CZ06	GCOS 6 MOD 400 System Programmer's Guide - Volume II
CZ07	GCOS 6 MOD 400 Programmer's Pocket Guide
CZ09	GCOS 6 MOD 400 System Maintenance Facility Administrator's Guide
CZ10	GCOS 6 MOD 400 Menu Management/Maintenance Guide
CZ15	GCOS 6 MOD 400 Application Developer's Guide
CZ16	GCOS 6 MOD 400 System Messages
CZ17	GCOS 6 MOD 400 Commands
CZ18	GCOS 6 Sort/Merge
CZ19	GCOS 6 Data File Organizations and Formats
CZ20	GCOS 6 MOD 400 Transaction Control Language Facility
CZ21	GCOS 6 MOD 400 Display Formatting and Control
CZ34	GCOS 6 Advanced COBOL Reference
CZ35	GCOS 6 Advanced COBOL Quick Reference Guide
CZ36	GCOS 6 BASIC Reference
CZ37	GCOS 6 BASIC Quick Reference Guide
CZ38	GCOS 6 Assembly Language (MAP) Reference
CZ39	GCOS 6 Advanced FORTRAN Reference
CZ40	GCOS 6 Pascal User's Guide
CZ41	GCOS 6 RPG-II Reference
CZ47	Data Entry Facility-II User's Guide
CZ48	Data Entry Facility-II Operator's Quick Reference Guide
CZ52	DM6 I-D-S/II Programmer's Guide
CZ53	DM6 I-D-S/II Data Base Administrator's Guide
CZ54	DM6 I-D-S/II Reference Card
CZ59	Level 6 to Level 6 File Transmission Facility User's Guide
CZ60	Level 6 to Level 66 File Transmission Facility User's Guide
CZ61	Level 6 to Level 62 File Transmission Facility User's Guide
CZ62	BSC Transport Facility User's Guide
CZ63	2780/3780 Workstation Facility User's Guide
CZ64	HASP Workstation Facility User's Guide

Base Publication Number	Manual Title
CZ65	Programmable Facility/3271 User's Guide
CZ66	Remote Batch Facility/66 User's Guide
CZ71	DM6 TP Development Reference
CZ72	DM6 TP Application User's Guide
CZ73	DM6 TP Forms Processing

In addition, the following publications provide supplementary information:

AS22	Level 6 Models 6/34, 6/36, and 6/43 Minicomputer Handbook
AT97	Level 6 Communications Handbook
CC71	Level 6 Minicomputer Systems Handbook
CD18	Level 6 MOD 400/600 Online Test and Verification Operator's Guide
FQ41	Writable Control Store User's Guide

Users should be aware that a Software Release Bulletin accompanies each software product ordered from Honeywell. You should consult the Software Release Bulletin before using the software. Contact your Honeywell representative if a copy of the Software Release Bulletin is not available.

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

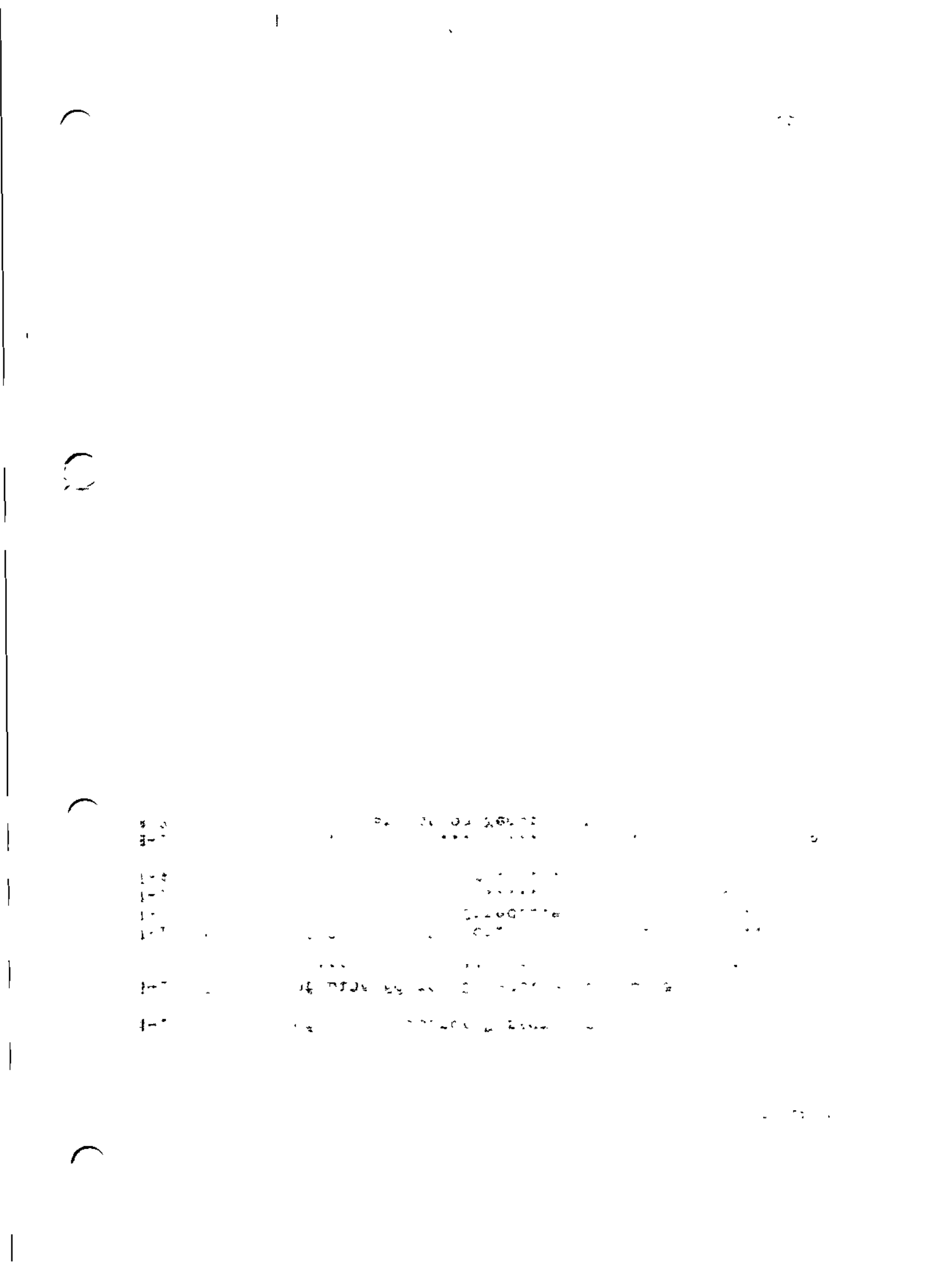
CONTENTS

CONTENTS

ILLUSTRATIONS

ILLUSTRATIONS

TABLES



1. Introduction

Section 1
INTRODUCTION

The Application Developer's Guide describes GCOS 6 system facilities available to the application programmer and provides procedures for using these facilities to write, debug, and run application programs.

SYSTEM FACILITIES

The GCOS 6 MOD 400 Executive supports concurrent execution of one batch stream (such as program development or file maintenance) and one or more online streams.

User-written online applications may be loaded and started at any time after system initialization. The number of applications in operation is determined only by the amount of available memory. When one application is deleted or terminates, its memory is automatically released to another.

MOD 400 allocates memory dynamically from pools and can relocate programs at load time. Once an application is loaded into memory, it is dispatched according to its assigned priority level. When multiple tasks share a priority level, they are serviced in a round-robin fashion. The Memory Management Unit prevents user applications residing in different memory pools from interfering with each other or with the Executive.

APPLICATION DEVELOPMENT COMPONENTS

The components that support applications development are:

Screen Editor -- A full screen, interactive program development, text editing, and documentation preparation system that allows a user to enter an entire screen of data into a work file. The ability to manipulate full screens of data at once makes text editing faster and reduces I/O processing.

Line Editor -- An interactive program development, text editing, and documentation preparation system that works on data a line at a time.

COBOL, BASIC, and FORTRAN Run-time Services -- A total system of language processors including, compile, link, and execute modules that validate and process COBOL, BASIC, and FORTRAN programs.

Forms Processor -- A software component that permits a programmer to define terminal screen layouts, as well as control characteristics of the data transmitted between the terminal and program variable storage.

Multi-User Debugger -- A software diagnostic tool used to debug programs.

Figure 1-1 illustrates the family of application development components.

The Screen Editor, Line Editor, COBOL, BASIC, and FORTRAN run-time services, and the Multi-User Debugger are described in this manual. Forms Processing is described in the Display Formatting and Control manual. The MOD 400 Executive is described in the MOD 400 System Concepts manual.

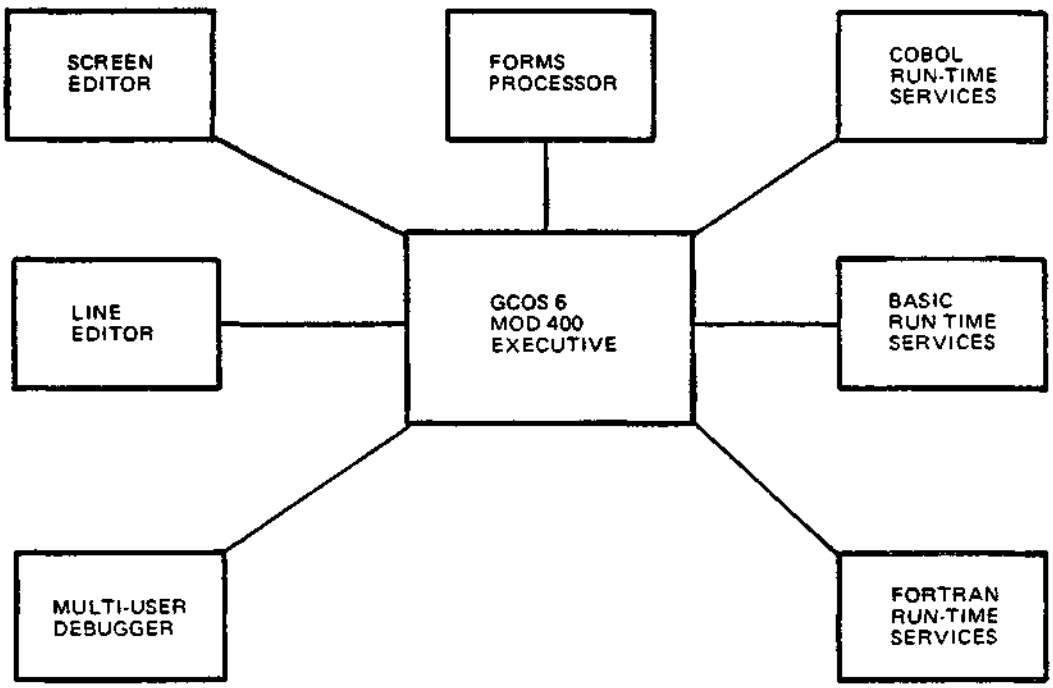
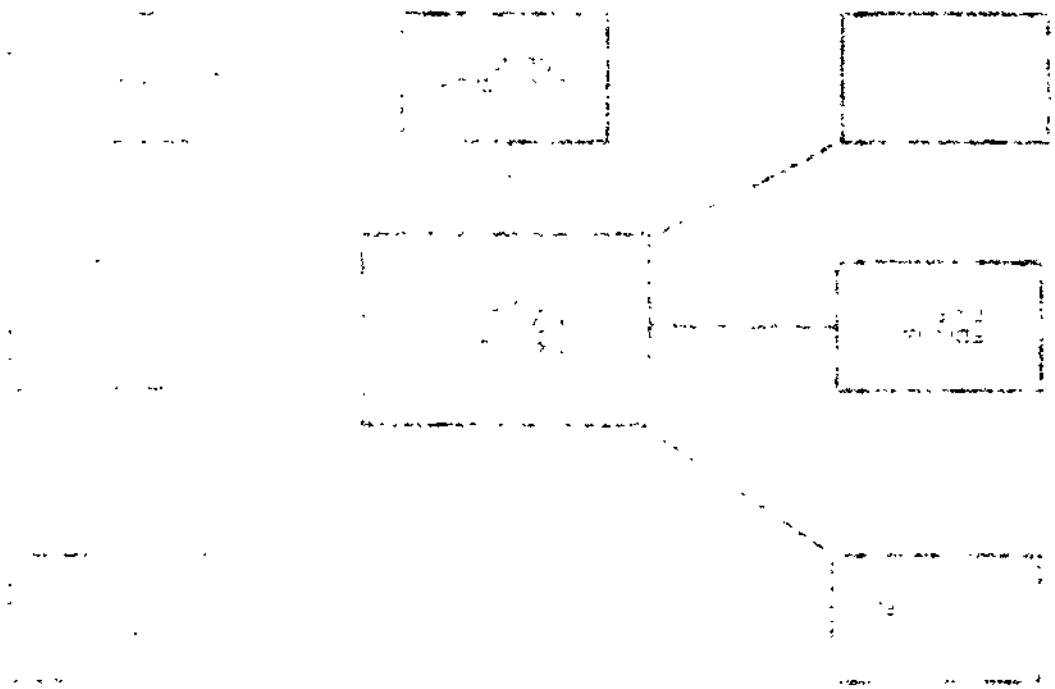


Figure 1-1. Application Development Components



Section 2
SYSTEM ACCESS

This section describes MOD 400 user access procedures.

USER ACCESS PROCEDURES

When you are at a user terminal, access to the system depends on the way your terminal is described to and recognized by the system.

Access to the system requires:

1. Physical connection between your terminal and the central processor
2. Logical connection between you (the user) and the operating system.

In some cases, the Executive performs the second step for you automatically after you have made the physical connection.

CONNECTING THE TERMINAL TO THE CENTRAL PROCESSOR

You can connect your terminal to the central processor by two methods, depending on the type of terminal you have: a direct-connect terminal or a dialup terminal.

Direct-Connect Terminal

For a direct-connect terminal, place the POWER ON/OFF switch in the ON position. This is sufficient to connect the terminal to the central processor.

Dialup Terminal

A telephone line connects a dialup terminal to the central processor. Use the following steps to make the connection:

1. Turn the terminal POWER ON/OFF switch to the ON position.

2. Lift the receiver, press the button marked TALK/CLEAR, and listen for a dial tone.

3. Use the telephone number provided by the system operator for the dialup line to call into the central processor.

4. Press the button marked DATA when you hear a high-pitched tone (this tone lets you know that the connection has been made). Hang up the receiver.

If you are unable to make a connection, hang up the receiver and begin again at Step 2.

CONNECTING A USER TO THE EXECUTIVE

After you have made the physical connection between your terminal and the central processor, you can make the logical connection that identifies and establishes you as a user known to the Executive. The procedure you use depends on whether your terminal is defined as a login terminal or a non-login terminal. If your terminal is a non-login terminal, turn to "Non-Login Terminal" later in this section.

Login Terminal

A login terminal is one that requires you to use the LOGIN command to connect to the Executive. If the system you want to access incorporates user registration, you may be required to be registered as a user in order to log in. The system administrator can register you, or may allow an unregistered user to log in under the user identification USER.

Login terminals can allow a full login or an abbreviated login, or generate an automatic login. Which of these is acceptable to a given terminal is determined by the system startup procedure. See your system operator to determine which type your terminal requires before using it. After you have

logged in, your access to system facilities is governed by control arguments entered in the LOGIN command or, under user registration, in your user profile.

MANUAL LOGIN TERMINAL

When the connection to the system has been made at a manual login terminal, a message-of-the-day and the login prompter message:

```
LOGIN
```

followed by the system identification and the current date and time appears at the terminal. If the terminal (and your user profile) allow it, you may enter a full login line, such as:

```
L JONES
```

to gain access to the system. (Check with the system operator for the correct format of your full login line.) In a user registration system, you may then be required to enter your password in response to the prompter message:

```
PASSWORD?
```

If you enter the login line and password correctly, the system responds with a ready message and you can begin to enter commands. (Typing errors can be corrected when they are made. See Appendix A for instruction on correcting errors.)

ABBREVIATED LOGIN TERMINAL

Some terminals and some user profiles allow login only by abbreviation. Most systems have defined one or more abbreviations that are available to users at any terminal, and may have defined terminal-specific abbreviations in addition. Check with the system administrator for the abbreviations that can be used at your terminal.

If you wish to login by an abbreviation, after the login prompter message has been issued, enter a one-character login abbreviation, such as:

```
S
```

In a user registration system, you may then be required to enter your password in response to the prompter message:

```
PASSWORD?
```

If you have entered the correct abbreviation and password, the system will respond with a ready message and you can begin to enter commands.

AUTOMATIC LOGIN TERMINAL

At a terminal that generates an automatic login, there is no login prompt. The login process occurs automatically after connecting to the central processor. After the message-of-the-day (if there is one) and the ready message are displayed, you can begin to enter commands.

Non-Login Terminal

A terminal can be configured as a non-login terminal. At a non-login terminal, you can start entering data immediately after making a connection with the system. It is recommended that the first command you enter is the Ready On (RDN) command. If the system responds with the message:

```
RDY:
```

you can continue with the session. If the system does not respond to this command (or to any other command), you must request the operator to activate a task group for that terminal. After the task group is activated, your access to system facilities is governed by the control arguments specified in the task group activation command.

Example:

You have made a connection with the system. Your task group is $H. The first command you enter is:

```
RDN
```

to activate a prompter message to signal you that the previous command is complete and the system is ready to accept another command.

The system responds with:

```
RDY:
```

Enter the command to list your working directory:

```
LWD
```

The system responds (in this case) with:

```
^ZSYS51
RDY:
```

To find out what files are under this directory, enter:

```
LS
```

and the system responds with a listing of the files. Figure 2-1 shows a sample listing.

PROCEDURES AND CONVENTIONS AFTER ACCESS

It may be necessary to request operator intervention or interrupt a running task while at your terminal. The procedures and conventions used to perform these actions are described in the following paragraphs.

Sending Messages to the Operator

To send a message from your terminal to the system operator (e.g., when your terminal is remote from the operator terminal), you can enter the Message (MSG) command described under "Working With Files". For example, if you want to abort the current batch request, you could enter:

```
MSG "PLEASE ABORT BATCH REQUEST"
```

Interrupting (Breaking) a Task

You can interrupt or break a running task to reenter commands, temporarily halt the task, or terminate it.

To effect a break from the user terminal, press the BREAK (BRK) key. The system then issues the break prompter message:

```
**BREAK**
```

Your response may be any one of the following:

1. Enter any command (see the Commands manual). This may be followed by another command or by one of the responses described in steps 2 through 4. If the entered command is not Start (SR), Logoff (BYE), New Procedure (NEW_PROC), Unwind (UW), or Program Interrupt (PI) (described later), the lead task again enters break mode and issues another **BREAK** message requesting another response.

```
DIRECTORY:  ^ZSYS51

TIME:  1980/09/25 0724:27

                          PHYSICAL    STARTING    RECORD
 ENTRY NAME    TYPE      SECTORS    SECTOR HEX   LENGTH
**************************************************
ZSYS51          D          13            0         32
Z3EXECUTIVEL    R2        848           18        256
START_UP.EC     S           8          368        256
GROUP$H.EC      S           8          370        256
GROUP$P.EC      S           8          378        256
SUPER.EC        S           8          380        256
GROUP$D.EC      S           8          388        256
DEBUG.WORK      R2         64          390        256
HIS             D           8          430         32
EML             D           8          440         32
MDD             D           8          5B8         32
UDD             D           8          610         32
LDD             D           8          618         32
SID             D          16         15C8         32
SYSLIB1         D           8         1B38         32
SYSLIB2         D          48         1B58         32
FORMS           D           8         3100         32
TV              D           8         3120         32
PROGS           D           0            0         32
TRANS           D           0            0         32
DEBUGDB         R2         56          3D0        256
GROUP$C.EC      S           8          408        256
GROUPB4.EC      S           8          410        256
GROUPB1.EC      S           8          418        256
GROUPB2.EC      S           8          420        256
GROUPB3.EC      S           8          428        256
CONFDIR         D           8         3200         32
DEFMENU         D           8         3228         32
ZDRT            D          16         3348         32
SSR.EC          S           8         3740        256
**************************************************
```

Figure 2-1. Directory Listing

2. Enter one of the following break mode responses to the **BREAK** message:

   a. Start (SR). This resumes execution of the suspended task as though the break had not been made.

   b. Unwind (UW). This releases all tasks and you return to command level.

   c. Logoff (BYE). This aborts and deletes the current task group request.

   d. New Process (NEW_PROC). This aborts all task requests in the task group except for the lead task, then restarts the task group, using the same arguments as specified in the initial task group request.

   Any of these commands terminates the current break; i.e., there will be no other **BREAK** message after they are executed.

3. Enter Unwind (UW). All tasks will be terminated and you return to command level.

   If the terminated task was invoked following a break, the lead task reenters break mode, issues another **BREAK** prompter message, and awaits a response.

4. Enter Program Interrupt (PI). The task interrupted is currently suspended.

   For Linker and Editor, suppress output and return to directive input level. The PI command suppresses output resulting only from the Linker MAP directive.

   The PI command is meaningful only to the Linker and Editor running in a task group whose lead task is the Command Processor. The commands described in steps 1, 2, and 3 may be used with the Linker and Editor.

Example:

You issue a List Names (LS) command and the output begins to appear on the screen at your terminal. You want this output to be printed on the line printer. You should immediately press the Break (BRK) key and take one of the following steps:

1. Enter:

```
FO !LPT00
```

   to change the output destination to the line printer; then enter the SR command to resume execution of the LS command. The output that had already appeared at the terminal will not appear on the hard-copy printout.

2. Enter the UW command to terminate the current LS task; or enter:

```
FO !LPT00
```

   to change the output destination to the line printer; then enter the LS command to restart the LS program from the beginning.

Section 3
FILE CONVENTIONS

This section presents MOD 400 file conventions as well as a procedural scenario titled "Working With Files". This scenario provides a detailed explanation of frequently used file system commands and procedures.

OVERVIEW

A file is a logical unit of data composed of a collection of records. The principal external devices available for storing files are:

- Disk devices (diskettes, cartridge disks, cartridge module disks, and mass storage units)
- Magnetic tape units.

These external devices are referred to as volumes (e.g., diskette volume, tape volume).

Various conventions to identify and locate files have been established for their effective control when stored on disk and magnetic tape. The conventions facilitate the orderly and efficient use of the stored data.

Unit record devices (such as card readers, card punches, and printers) also use the file concepts. However, since unit record devices cannot be used to store files, there is less need to

establish conventions for identification and location. A unit record file is simply the data that is read or written at any one time.

DISK FILE CONVENTIONS

Users must be able to specify an access path to any given file on a disk volume that contains multiple files. Files must, therefore, be organized on the volume in some predictable fashion. MOD 400 provides a set of volume organization conventions by which the system can locate any element that resides on the volume.

The principal elements of this organization, aside from the files themselves, are directories. The access path to any given element on a volume is known as a pathname.

Directories

Files on disk devices reside within a tree-structured hierarchy. The basic elements of this hierarchy are files known as directories. The directories are used to point to the location of data files, which are the endpoints of the tree structure.

A directory on a disk volume functions like a catalog. It contains the names and starting locations (sectors on the volume) of files or other directories (or both). The elements whose names are in the directory are said to be contained in or subordinate to the directory; therefore, the organization of a disk volume is a multilevel structure. The complexity of the access path to any given element in the structure depends on the number of directories between the root and the desired element.

A directory structure is illustrated in Figure 3-1. The base directory on a volume is termed a root directory. In Figure 3-1, the root directory is VOL01. The root directory VOL01 points to two subordinate directories DIR1 and DIR2. The directories DIR1 and DIR2, in turn, point to the data files (FILEA, FILEB, FILEC, and FILED).

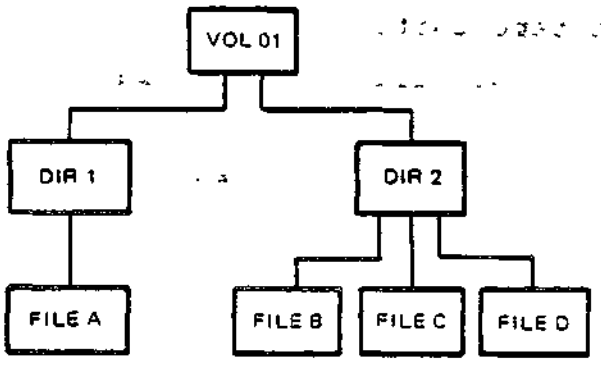


Figure 3-1. Example of Disk File Directory Structure

The following paragraphs describe the root directory and other special types of directories.

ROOT DIRECTORY

There is a tree structure for each disk mounted at any given time. At the base of each tree structure is a directory known as the root directory. This is the directory that ultimately contains every element that resides on the volume either immediately or indirectly subordinate to it.

The root directory name is the same as the volume identifier of the volume on which it resides. The directory VOL01 in Figure 3-1 is a root directory.

SYSTEM ROOT DIRECTORY

One or more disk root directories can be known to the system at any time during its operation. One of these, the system root directory, is required at all times. The volume used by the operator to initialize the system establishes the system root directory. This volume also normally contains system programs, commands, and other routinely used elements. It must contain a number of directories and files that the system needs to perform its functions. These are described in the System Building and Administration manual.

USER ROOT DIRECTORIES

The File System can recognize one or more user root directories. These are root directories of volumes created and used for the installation's own particular needs. They may contain user application programs and their associated data files, application program source and object unit files, listing files, or anything else that you want to store, either temporarily or permanently.

INTERMEDIATE DIRECTORIES

When a volume is first created, it contains only a root directory. You can create, within this directory, any additional directories required to satisfy the needs of your installation. Consider, for example, a volume that is to contain data used by two application projects, each of which has several people associated with it. Each of these people has one or more files of interest to him. The volume has been initialized and contains a root directory name. Two directories can be created subordinate to the root directory, each identified by the project name. Then, subordinate to these directories, a directory can be created for each person associated with each project.

The data files are all contained within the personal directories. This sample intermediate directory structure is illustrated in Figure 3-2.

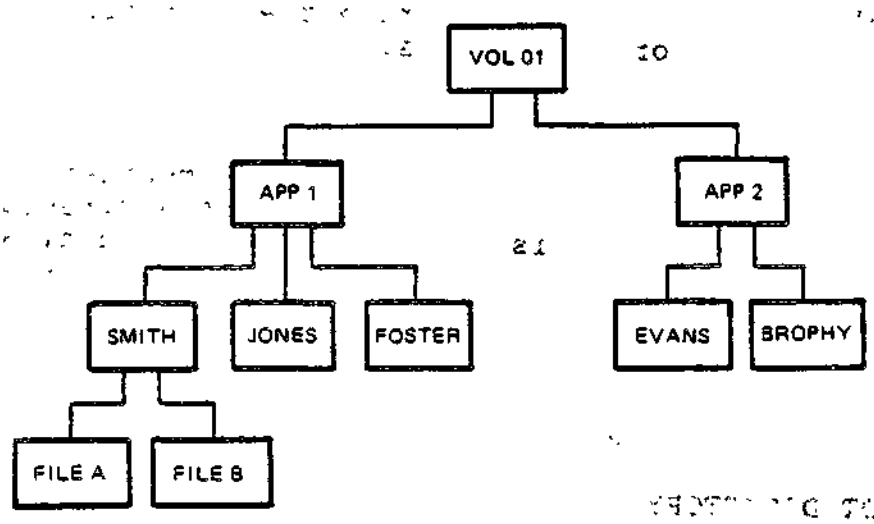


Figure 3-2. Sample Directory Structure

When the need for a user-created directory no longer exists, the directory can be deleted from the File System, making the space it occupied, as well as the space occupied by its attributes in the immediately superior directory, available for reuse. A directory must be empty before it can be deleted; all directories and files subordinate to the one to be deleted must have been previously deleted by explicit commands.

WORKING DIRECTORY

The File System always starts at a root directory when it performs an operation on a disk file or a directory. At times the search for an element residing on a disk volume may traverse a number of intermediate directory levels before locating the desired element; the File System must be supplied with the names of all the branch points it must pass on the way. The files of interest to a user doing work on the system are frequently all contained in a single directory specific to the task being performed; this directory can be three, four, or more levels deep into the structure. It would be convenient to be able to refer to files in relation to a directory at some arbitrary level in the hierarchy rather than in relation to the root directory. The File System allows this to be done by recognizing a special kind of directory known as a working directory.

A working directory establishes a reference point that enables you to specify the name of a file or another directory in terms of its position relative to that directory. If the access path of the working directory is made known to the File System,

and if the desired element is contained in that directory, the element can be specified by just its name. The File System concatenates this name with the names of the elements of the working directory's access path to form the complete access path to the element.

LOCATIONS OF DISK DIRECTORIES AND FILES

The File System has total control over the physical location of space allocated to directories and files; you need never be concerned about where on a volume a directory or file resides. When a volume is first initialized, space is allocated to elements in essentially the order in which they are created. But after the volume has been in use for some time, elements may have been deleted and the space they occupied made reusable. Hence, when a new element is created, it is allocated the first available space ever though that space may eventually be too small to contain the file. If more space is needed for even a single extent of a file, it will be obtained from another free area. Thus, there is not necessarily any relationship between a file's extents and contiguous free disk sectors.

Naming Conventions

Each disk file and directory name in the File System can consist of the following ASCII characters: uppercase alphabetics (A through Z), digits (0 through 9), underscore (_), hyphen (-), and period(.). If lowercase alphabetic characters are used, they are converted to their uppercase counterparts.

The first character of any name must be an alphabetic. The underscore can be used to join two or more words that are to be interpreted as a single name (e.g., DATE_TIME). A period followed by one or more alphabetic or numeric characters after a file name is normally interpreted as a suffix to a file name. This convention is followed, for example, by a compiler when it generates a file that is to be subsequently listed; the compiler identifies this file by creating a name of the form "FILE.L".

The name of a root directory or a volume identifier can consist of from one to six characters. The names of other directories and files can comprise from 1 to 12 characters. The length of a file name must be such that any system-supplied suffix does not result in a name of more than 12 characters.

UNIQUENESS OF NAMES

Within the system at any given time, the access path to every element must be unique. This leads to the following rules:

- Only one volume with a given volume_id can be mounted at any given time. (The system will inform you of an attempt to mount a volume having the same name as one already mounted.)

- Within a given directory, every immediately subordinate directory name must be unique. (The Create Directory command will inform you of an attempt to add a duplicate directory name.)

- Within a given directory, every file name must be unique. (The Create File command will inform you of an attempt to add a duplicate file name.)

PATHNAME

The access path to any File System entity (directory or file) begins with a root directory name and proceeds through zero or more subdirectory levels to the desired entity. The series of directory names (and a file name if a file is the target entity) is known as the entity's pathname. The total length of any pathname, including all hierarchical symbols, cannot exceed 57 characters, except that a working directory pathname cannot exceed 51 characters.

Symbols Used in Pathnames

The following symbols are used to construct pathnames.

- Circumflex (^). Used exclusively to identify the name of a disk volume root directory. The circumflex is used in two forms. In one form it directly precedes the root directory name (e.g., ^VOL011). In the other it directly precedes a greater-than symbol (>) to refer to the root directory of the current working directory (e.g., ^>DIR1>FILEA).

- Greater than (>). Indicates movement in the hierarchy away from the root directory. The symbol is used to connect two directory names or a directory name and a file name. It can also be the first character of a pathname, in which case the element whose name follows it is immediately subordinate to the root directory of the system volume. Each occurrence of the greater-than (>) symbol denotes a change of one hierarchical level; the name to the right of the symbol is immediately subordinate to the name on the left. Reading a pathname from left to right thus indicates movement through the tree structure in a direction away from the root directory. If the root directory ^VOL011 contains a directory name DIR1, the pathname of DIR1 is:

  ```
  ^VOL011>DIR1
  ```

  If the directory named DIR1 in turn contains a file named FILEA, then the pathname of FILEA is:

  ```
  ^VOL011>DIR1>FILEA
  ```

- Less than (<). Used at the beginning of a pathname to indicate movement from the working directory in a direction toward the root directory. Consecutive symbols can be used to indicate changes of more than one level; each occurrence represents a one level change. When followed by elements of a relative pathname, those elements represent changes of direction away from the root directory. One or more of these symbols may precede only a relative pathname.

- ASCII "space" character. Used to indicate the end of a pathname. When represented in memory, a pathname must end with a space character.

The last (or only) element in a pathname is the name of the entity upon which action is to be taken. This element can be a device name, directory name, or file name, depending on the function to be performed. In the Create Directory command, for example, a pathname specifies the name of a directory to be created. The last element of this pathname is interpreted by the command as a directory name; any names preceding the final name are names of superior directories leading to it. An analogous situation occurs in the Create File command, except that in this case the final pathname element is the name of a file to be created.

Absolute and Relative Pathnames

A full pathname contains all necessary elements to describe a unique access path to a File System entity, regardless of the type and location of the device on which it resides. The File System uses this form in referring to a directory or file. However, it is frequently unnecessary to specify all of these elements; the File System can supply some of them when the missing elements are known to it and the abbreviated pathnames are used in the appropriate context. An understanding of these conditions and contexts requires an understanding of absolute and relative pathnames.

Absolute Pathname

An absolute pathname is one that begins with a circumflex (^) or a greater-than symbol (>). (A pathname that begins with a circumflex is a full pathname. This form is used to locate directories and files that reside on a device other than that on which the system volume, the volume from which the system was initialized, is mounted.)

When an absolute pathname begins with a greater-than symbol, the first element named in the pathname is assumed to be immediately subordinate to the system volume root directory. Thus, if the system volume name is SYS01 and the pathname given is >DIR1>FILEA, the full pathname becomes ^SYS01>DIR1>FILEA.

Another volume, USER1, can also contain a >DIR1>FILEA access path and can be known to the File System; the two access paths are made unique by requiring that the root directory be specified when referring to the second volume. The full pathname of this file on the second volume is thus ^USER1>DIR1>FILEA.

Relative Pathname

A relative pathname is one that begins with a file or directory name or less-than symbol (<). When a relative pathname begins with an element name, the first (or only) name in the pathname identifies a directory or file immediately subordinate to the working directory. When the relative pathname begins with one (or more) less-than symbols, the first (or only) name in the pathname identifies a directory or file immediately subordinate to the directory reached by moving from the working directory toward the root the number of levels indicated by the less-than symbol(s).

A relative pathname can consist of one or more elements. If a relative pathname contains more than one element, each element except the last must be a directory name, the first immediately subordinate to the current working directory level, the second immediately subordinate to the first, and so on. The last or only element can be either a directory name or a file name, depending on the function being performed, as described previously.

A simple name is a special case of the relative pathname. It consists of only one element: the name of the desired entry in the working directory.

If a reference is to be made to a file or directory that is on the same volume but not subordinate to the working directory, there are two alternative ways of making this reference: by using an absolute pathname, or by using any of the forms of relative pathname described previously.

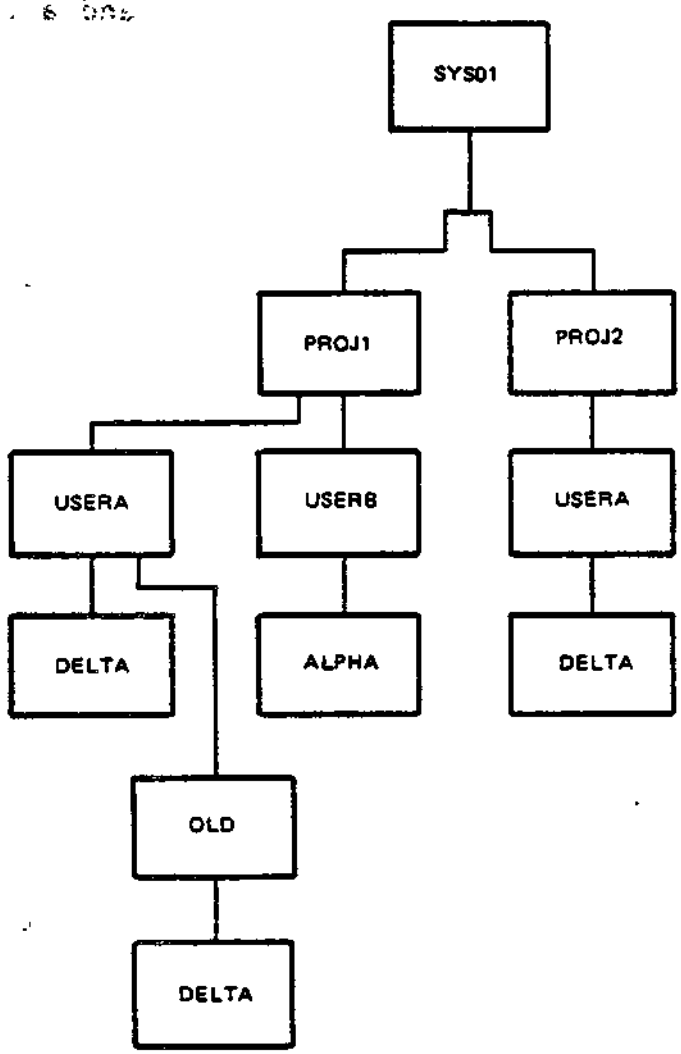
Figure 3-3 shows some relative pathnames and the full pathnames they represent when the working directory pathname is:

>PROJ1>USERA

MAGNETIC TAPE FILE CONVENTIONS

The magnetic tape file conventions include tape file organization, tape file naming conventions, and tape file pathnames.

RELATIVE PATHNAME*	FULL PATHNAME
DELTA	^SYS01>PROJ1>USERA>DELTA
OLD>DELTA	^SYS01>PROJ1>USERA>OLD>DELTA
<USER8>ALPHA	^SYS01>PROJ1>USER8>ALPHA
<<PROJ2>USERA>DELTA	^SYS01>PROJ2>USERA>DELTA
<	^SYS01>PROJ1



*ASSUME CURRENT WORKING DIRECTORY IS SYS01>PROJ1>USERA.

Figure 3-3. Sample Pathnames

Tape File Organization*

Magnetic tape supports only the sequential file organization. Fixed- or variable-length records may be used. Records cannot be inserted, deleted, or modified, but they can be appended to the file. The tape can be positioned forward or backward any number of records.

The unit of transfer between memory and a tape file is a block. Block size varies depending on the number of records and whether the records are fixed or variable in length.

A block can be treated as one logical record called an "undefined" record. An undefined record is read or written without being blocked, unblocked, or otherwise altered by data management. Spanned records (i.e., those that span two or more blocks) are supported. (No record positioning is allowed with spanned records.)

A labeled tape is one that conforms to the current tape standard for volume and file labels issued by the American National Standard Institute. The following types of labeled tapes are supported:

- Single-volume, single-file
- Multivolume, single-file
- Single-volume, multifile
- Multivolume, multifile.

The following types of unlabeled tapes are supported:

- Single-volume, single-file
- Single-volume, multifile.

Magnetic Tape File and Volume Names

Each tape file and volume name in the File System can consist of the following ASCII characters:

- Uppercase alphabetics (A through Z)
- Exclamation mark (!)
- Double quotation marks (")
- Percent sign (%)
- Ampersand (&)
- Single quotation mark (')
- Left parenthesis (()
- Right parenthesis ())
- Asterisk (*)
- Plus sign (+)
- Comma (,)
- Hyphen (-)
- Period (.)
- Slash (/)

*This information applies to 9-track magnetic tape only.

- Colon (:)
- Semicolon (;)
- Less-than sign (<)
- Equal sign (=)
- Question mark (?)
- Underscore (_).

The underscore (_) can be used as a substitute for a space. If a lowercase alphabetic character is used, it is converted to its uppercase counterpart.

Any of these characters can be used as the first character of a file or volume name.

The name of a tape volume can be from one through six characters; tape file names can be from 1 through 17 characters.

Magnetic Tape Device Pathname Construction

A magnetic tape volume must be dedicated to a single user. Therefore, the device pathname convention must always be used when referring to magnetic tape volumes or files. The general form of a tape device file pathname is:

```
!dev_name [>vol_id [>filename] ]
```

where dev_name is the symbolic name defined for the tape device at system building, vol_id is the name of the tape volume, and filename is the name of the file on the volume. Tape devices are always reserved for exclusive use (i.e., the reserving task group has read and write access; other users are not allowed to share the files).

Automatic Tape Volume Recognition

Automatic volume recognition dynamically notes the mounting of a tape volume. This feature allows the File System to record the volume identification in a device table, thus making every tape volume accessible to the File System software.

UNIT-RECORD DEVICE FILE CONVENTIONS

Unit-record devices (e.g., card readers, card punches, printers) are used only for reading/writing data; they are not used for data storage and thus do not require conventions for file identification and location.

Refer to a unit-record device by entering a pathname consisting of an exclamation mark (!) followed by the symbolic device name defined during system building. The format is:

```
!dev_name
```

where dev_name is the symbolic device name of the unit record device.

WORKING WITH FILES

The following information addresses selected commands and procedures that you may use frequently, including:

- Using file pathname conventions
- Controlling your files and directories
- Interrupting execution
- Controlling your output
- Controlling printing
- Program execution
- Communicating with other users
- Performing batch processing.

The examples that follow provide full details on performing these functions. Note that some examples do not list all optional control arguments for the commands described. See the Commands manual for a complete description of all commands and their arguments. For information on using Execution Command Files, see Appendix F.

COMMAND PROCESSOR

You communicate with the system through command lines entered at a terminal or read from a command file. Your command lines are read and interpreted by a system software component called the Command Processor.

Standard I/O Files

Four files are always associated with the Command Processor:

- Command-in
- User-in
- User-out
- Error-out.

The command-in file is the file from which the Command Processor takes its input. The command-in file is normally associated with (or assigned to) your terminal. However, it can be reassigned, temporarily, to another device or file, and subsequently reassigned to your terminal.

A command function reads its own input during execution from the user-in file (normally assigned to your terminal). The directives submitted to the Editor following entry of the Editor command, for example, are submitted through user-in. A task group normally writes its output to the user-out file (normally assigned to your terminal). The user-out file can be reassigned to another device or file (see "Controlling Output"). This reassignment remains in effect until another reassignment occurs.

The Command Processor, and any commands it invokes, writes any errors detected to the error-out file. The error-out file is the same as the initial user-out file; it cannot be reassigned by a command or command argument.

You can determine the full pathnames associated with each of these files by issuing a Status Group (STG) command at your terminal.

Command Level

The system indicates that it is at command level by issuing a ready (RDY) message at your terminal. This assumes that you have not disabled the ready message by a previously issued Ready Off (RDF) command; if you have, the system still comes to command level, but you are not informed. You can activate the ready prompt at any time by issuing a Ready On (RDN) command.

When executing a command function, you can return to command level in one of two ways:

- After a command function terminates, the system returns to command level and awaits the entry of another command. This command can be any function you wish to execute or it can be a BYE command, indicating that you have no further work to do and you want to terminate the current session.

- You can interrupt execution of an invoked command by pressing the Break or Interrupt key at your terminal. See "Interrupting Execution" below.

CONTROLLING YOUR OPERATING ENVIRONMENT

The following paragraphs describe the commands and procedures that you may find most useful as an interactive system user. Once at command level, you can perform a wide variety of system operations using these commands and special system procedures. Selected examples are designed to help you become familiar with using the system for applications programming. For full descriptions of all commands and their arguments, refer to the Commands manual.

Volume Control

The following commands illustrate how to create or rename a tape or disk volume.

CREATING VOLUMES

Before you can begin to perform useful work on a previously unused tape or disk volume, you must assign it a unique name (volume identifier or vol_id) that can be recognized by the system. The vol_id designates the volume (or root) directory name of the tape or disk volume.

First you must ask the system operator to mount your diskette or tape on an available drive and notify you of the drive's symbolic peripheral device name. (The symbolic peripheral device name is the name the system uses to recognize the device.)

For example, suppose you want to create a diskette volume and name it WORK. Send the following message to the system operator (see "Communicating With Other Users" later in this section):

```
MSG "MOUNT DISKETTE AND NOTIFY ME OF DEVICE NAME"
```

The operator issues the Status System (STS) operator command to determine which devices are available:

```
STS
```

and the system responds:

	SYMPD NAME	CHANNEL	DEVICE TYPE	VOLUME ID	USAGE	AVAILABLE SECTORS PHYSICAL	LOGICAL	VOL/FILE SET NAME	MEMBER NUMBER
B	RCM00	2800	2380	^DMPVL	0	46504	5813		
B	RCM01	2880	2380	^OPEN	0	51704	6463		
B	FCM01	2880	2385	^SYSTST	0	172792	21599		
B	MSM01	1880	2361	^	D				
B	RCD00	1400	2332	^MINE	0	9465	1182		
B	FCD00	1400	2333	^	0	0	0		
B	RCD01	1480	2332	^RJE	0	1216	152		
B	FCD01	1480	2333	^FCD01	0	19560	2445		
B	FCD02	1500	2333	^FCD02	0	11960	1495		
B	RCD03	1580	2332	^	D				
B	FCD03	1580	2333	^	D				
B	DSK00	0400	2010	^	D				

The operator then mounts a diskette on the available drive, DSK00, and sends you the following message:

```
VOLUME MOUNTED ON DSK00
```

You can now use the Create Volume (CV) command to assign a unique vol_id to your new disk volume, using the following form of the command:

```
CV !DSK00 -FT WORK
```

where WORK is the vol_id you want to assign.

Using the -FT argument initializes all data structures on the volume and establishes WORK as the root directory name; the root directory pathname for this volume is ^WORK.

RENAMING DISK VOLUMES

If disk volumes having the same vol_id are used, one of the volumes must be renamed before the system will accept it. (A tape volume cannot be renamed.) The command:

```
CV !DSK00>OLD -RN NEW
```

renames the volume OLD using the -RN control argument; the new volume name is NEW.

Directory Control

You can create an unlimited number of directories to organize your files. The following commands illustrate how to change your working directory, and create, rename, or delete directories.

CHANGING YOUR WORKING DIRECTORY

The system provides you with tools to keep you aware of your location within the directory and file structure at any moment. You can also request a list of the files and directories under any directory to which you have list access.

To list your working directory, use the List Working Directory (LWD) command:

```
LWD
^SYSVLA>UDD>PROGS>LOWELL
```

The system responds with the absolute pathname of your working directory. If you want to change to some other directory, use the CWD (Change Working Directory) command. For example:

```
CWD ^SYSVLA>UDD>PROGS>JONES
RDY:
LWD
^SYSVLA>UDD>PROGS>JONES
```

The name of uour new working directory is JONES. Any number of users can work in the same directory at one time, as long as each user has list access to move there.

It is usually more convenient to use the relative pathnames of directories. For example, you can change your working directory to LOWELL by typing:

```
CWD <LOWELL
```

When going from a directory to a subdirectory, the system requires you to specify the directory name (there may be more than one directory subordinate to your working directory).

However, when moving up in the file structure, there is no ambiguity. You can move up one or more directory levels by entering one or more "<" signs:

```
CWD <
RDY:
LWD
^SYSVLA>UDD>PROGS
```

If CWD is entered with no arguments, the system returns you to your home directory (your initial working directory):

```
LWD
^SYSVLA>UDD>PROGS>JONES
RDY:
CWD
RDY:
LWD
^SYSVLA>UDD>PROGS>LOWELL
```

CREATING DIRECTORIES

You can create a directory using the Create Directory (CD) command. For example, you may want to put a COBOL program and a BASIC program under separate subdirectories below your home directory (your initial working directory). You first create the directories:

```
CD COBOL_DIR
RDY:
CD BASIC_DIR
```

You can now create your programs in subdirectories subordinate to your home directory (or create them elsewhere and copy them into the directories COBOL_DIR and BASIC_DIR).

As another example, suppose that you have just created, formatted, and named the disk volume WORK, as described under "Creating Volumes". You would like to create two directories, named SHEPARD and COOK, immediately subordinate to the root directory ^WORK.

Before creating your two directories, you enter a CWD command to change your working directory to ^WORK:

```
CWD ^WORK
```

(Note that this step is optional; you need not change your working directory to the volume ^WORK to create subordinate directories or files. You can create directories or files from any location in the File System tree structure by supplying the appropriate absolute or relative pathname of the file or directory you wish to create. However, for the sake of simplicity, only simple pathnames are used here.)

To create the directory SHEPARD, enter the command:

```
CD SHEPARD
```

This directory now resides immediately subordinate to the root directory ^WORK.

To create the directory COOK, enter the command:

```
CD COOK
```

This directory now resides, along with SHEPARD, immediately subordinate to the root directory ^WORK. Figure 3-4 illustrates this directory tree structure.

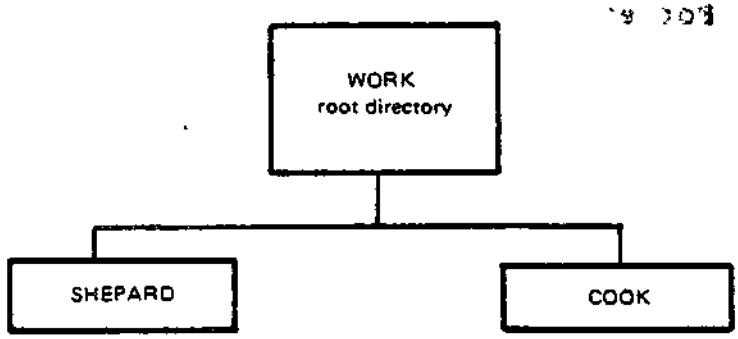


Figure 3-4. Location of Directories SHEPARD and COOK

RENAMING DIRECTORIES

You can change the name of an existing directory using the Rename (RN) command. For example, assume that within your working directory >UDD>PROGS>SMITH, there is a directory TEST. The command:

```
RN TEST WORK
```

changes the pathname of the affected directory from:

>UDD>PROGS>SMITH>TEST to >UDD>PROGS>SMITH>WORK

DELETING DIRECTORIES

You can delete one or more directories using the Delete Directory (DD) command. For example, you may no longer need to use a directory called EXAMPLE. The command:

```
DD EXAMPLE
```

deletes the directory called EXAMPLE from your working directory. Note that you could not delete EXAMPLE if it was your working directory.

As a safety measure, the File System will not allow you to delete a nonempty directory. If you wish to delete a directory, you must first delete any subdirectories or files it contains.

File Control

The following commands show you how to create, rename, delete, copy, and locate files.

CREATING FILES

You create a file in the file structure with the Create File (CF) command. For example:

```
CF DATAFILE
```

produces a sequential file called DATAFILE in your working directory, with a record size of 256 characters (the default) and a length of zero sectors. The following command:

```
CF MIME.D -RSZ 80 -MSZ 800
```

produces a file called MIME.D in the working directory, with a record size of 80 characters and a maximum allowable size of 800 control intervals. This file is meant to be a card file; after reading the cards (see "Copying Files"), a listing reveals the following:

```
LS

DIRECTORY:  ^SYSVL1>UDD>PROGS>DIRA

                          PHYSICAL    STARTING     RECORD
ENTRY NAME     TYPE       SECTORS     SECTOR HEX   LENGTH
  **********************************************************
  START_UP.EC    S            8          580          256
  MIME.D         S           40          8D0           80
  **********************************************************
```

As another example, assume that you wish to create a file under each of the two directories, SHEPARD and COOK, shown in Figure 3-5. Your working directory is the root directory WORK. To create a file named REPORTS under the directory SHEPARD, enter the command:

```
CF SHEPARD>REPORTS
```

where SHEPARD>REPORTS is the relative pathname (relative to your working directory) of the file you wish to create.

The file REPORTS now resides immediately subordinate to the directory SHEPARD, as shown in Figure 3-5.

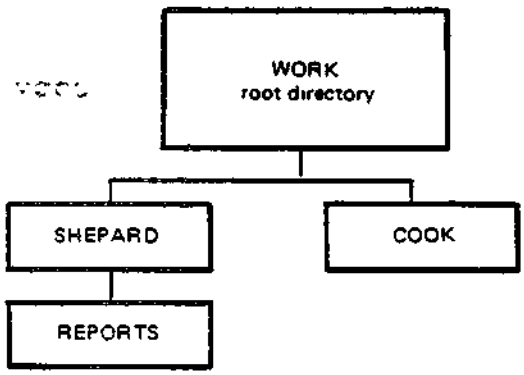


Figure 3-5. Location of Subordinate File REPORTS

Suppose you want to create a file named WORDLIST under the directory COOK. Since your working directory is still the root directory, WORK, enter the command:

```
CF COOK>WORDLIST
```

where COOK>WORDLIST is the relative pathname of the file you want to create. The file WORDLIST now resides immediately subordinate to the directory COOK, as shown in Figure 3-6.

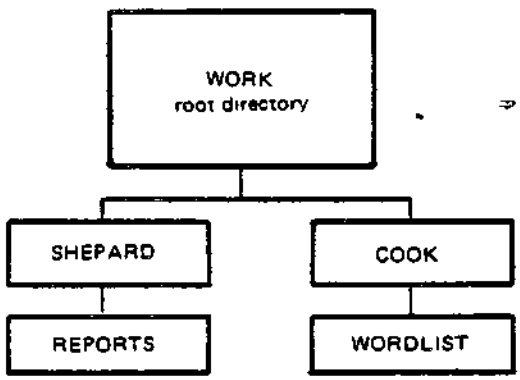


Figure 3-6. Location of Subordinate File WORDLIST

RENAMING FILES

Suppose that Cook wants to name his file more appropriately MATCHTM.D rather than MIME.D. The file can be renamed using the RN command:

```
RN MIME.D MACHTM.D
```

DELETING FILES

You can delete files using the Delete File (DL) command. For example, to delete the file DATAFILE in your working directory, enter:

```
DL DATAFILE
```

COPYING FILES

The Copy (CP) command allows you to copy files between directories, into directories from a card reader, out of directories to a printer, and between tape or disk volumes. For example, suppose Cook wants to read cards into a file MACHTM.D. From the home directory (COOK), enter:

```
CP !CDR00 MACHTM.D
```

All peripheral devices (tapes, card readers, and printers) are referred to by their symbolic peripheral device name; e.g., "CDR" for card reader.

When you read in cards, the card reader must be ready (the READY light must be on). While the command is being processed, your terminal locks; if the card reader is not ready or jams, the operator receives an error message. Until the card reader processes the end-of-file (EOF) card, the copy is not complete, and, if you fail to include an EOF card, the reader and your terminal remain locked. An EOF card is multipunched in column 1, rows 11, 5, 8, and 9. After copying is finished, you will receive the ready message.

Cook wants to copy an Assembly language subroutine, "REC3.A", from his home directory, ^SYSVLA>UDD>PROGS>COOK, currently his working directory, to the directory ^SYSVLA>UDD>PROGS>TOOLS. Note the use of the relative pathname.

```
CP REC3.A <TOOLS>REC3.A
```

The command allows you to omit the second argument if you are copying a file into your working directory. Thus, if Cook were

in the directory ^SYSVLA>UDD>PROGS>TOOLS, and wanted to copy in the file ^SYSVLA>UDD>PROGS>COOK>REC3.A, he needs to type only:

```
CP <COOK>REC3.A
```

The command copies REC3.A into TOOLS and names it REC3.A by default.. You must be in the target directory to use this feature.

For another example, to copy cards onto a tape named BS001, that is already mounted, enter:

```
CP !CDR00 !MT900>BS001>WESTNAMES
```

LOCATING FILES

You can use the Where (WH) command to locate and display a file's full pathname. The system will search your working directory and the two system libraries, SYSLIB1 and SYSLIB2, looking for your file. If the file is found, its full pathname is displayed. If the file is not found, an error message is displayed. You may find this command useful if you know the simple pathname of a file but want to know its absolute pathname, or, to determine if the file you want to locate exists.

LISTING FILES AND DIRECTORIES

You can list the contents of any directory that you have at least list access to by using the List Names (LS) command.

For example, Cook lists the contents of his working directory by entering:

```
LS

 DIRECTORY:  ^SYSVL1>UDD>PROGS>COOK

                          PHYSICAL    STARTING      RECORD
 ENTRY NAME    TYPE       SECTORS    SECTOR HEX     LENGTH
   **************************************************
 START_UP.EC    S            8          580           256
   **************************************************
```

To determine the starting sector of the file for file dumping purposes; the record length is the number of characters per line. Listing Cook's file with the -BF argument would produce this information.

```
LS -BE

DIRECTORY:    ^SYSVL1>UDD>PROGS>COOK
START_UP.EC    S       8

TOTAL SECTORS        8
```

The -DIR argument of the command will list only directories subordinate to your working directory.

With no arguments, the LS command lists all files and directories subordinate to your working directory.

To stop output (scrolling) during execution of a list command, press the space bar on your terminal keyboard. To resume scrolling, enter "@" on the same line, followed by a carriage return.

Interrupting Execution

You can interrupt the execution of any command or program, at any time, by pressing the terminal break (BRK) key. This signals the processor to interrupt execution. You can now enter any system command. To resume execution of the command or program, enter the Start (SR) command for programs or the Program Interrupt (PI) command for system software, such as the Editor or Linker.

If you do not want to resume execution after a break, you can use the Unwind (UW) command to return to command level, or a New Process (NEW_PROC) command to restart your task group; i.e., return it to the state existing immediately after login.

For example, assume you are editing a large file, and you accidentally press the BRK key while listing the file. To resume listing your file as if no break had occurred, you could issue an SR command, or you could save your work in the Editor and resume processing in edit mode by issuing a PI command. You can also enter the UW command if you want to close all your files, return to system command level, and proceed with other work. Your final option would be to issue a NEW_PROC command that would return you to command level in your home directory.

Controlling Output

Normally, all output goes to your terminal. (At login, for example, all four I/O files, user-in, user-out, command-in, and error-out, are assigned to your terminal.) If you are producing a large output, you may want to redirect it elsewhere. The following paragraphs describe how to direct your output to a file or to a printer.

DIRECTING OUTPUT TO A FILE

To direct output to a file (which need not have been previously created) using the FO (File Out) command, enter:

```
FO FILEA
```

All normal system output (such as a response to an LS command) will go to FILEA, which is your new user-out file. Error messages and the ready message that go to the error-out file cannot be redirected and will continue to appear at your terminal. Thus, if you entered an LS command, the system would write the listing to FILEA and respond at your terminal with only the ready message. However, input directed to your terminal is unaffected by the FO command.

DIRECTING OUTPUT TO A PRINTER

If you are performing functions that will lead to many pages of output, you can direct output to a printer. The command:

```
FO !LPT00
```

directs all subsequent output to LPT00 (assuming that you have access to the printer). Note that while you are using the printer, no one else can use it. In a multi-user system you may wish to avoid tying up the printer. (See "Deferred Printing" for information on printing large files.)

REDIRECTING OUTPUT TO YOUR TERMINAL

After you have finished directing output to a printer, you should redirect output to your terminal. Enter the FO command with no arguments:

```
FO
```

(The default is to redirect output to your terminal.)

Printing Control

You can print files at your terminal or you can request deferred printing. If you use the Print (PR) command, output appears on your terminal (i.e., output goes to the user-out file). This is inconvenient, however, if you are printing large files. For large files, you have the option of using deferred printing. The system will store your print request in a first-in, first-out queue.

PRINTING FILES AT YOUR TERMINAL

If you want to print a file at your terminal, issue a PR command

```
PR FILEB.D -SP
```

Remember that not all files are meant to be listed at a terminal. Some files are print files; some are not. Examples of print files are listings from the Linker and compilers, and batch output files. In print files, the first character of each line is a print control character, instructing the printer how many spaces to skip between lines and when to skip to the top of a form. When printing a nonprint file, use the -SP argument of the Print command. This argument instructs the printer to print the first character of each line, and to skip one space between lines.

DEFERRED PRINTING

To print large files, use the Deferred Print (DP) command. The DP command frees you from the need to reserve a printer and allows more efficient use of your system's printer resources. The request is queued on a first-in, first-out basis in one or more print queues.

Arguments of the DP command allow you to address your print output. If you are not at the printer, the person who is can separate printouts and route them to personnel. The Destination (-DS) argument accepts a string of up to 13 characters that appear as the first line of the address page. You can include blanks in the string if you enclose the string in quotes. The Header (HE) argument accepts a string of up to 26 characters that appear as the second line of the address page. For example:

```
DP START_UP.EC -SP -HE "SHEPARD 692" -DS
"WEST COMPUTER" -Q 3
```

produces the following printer output:

An address page, with a header label and destination label; one blank page; one or more pages containing the file; one blank page; and an end page, containing accounting information (e.g., the cost of the print job).

If you do not include a header and destination label, the default is the user name for the header label, and the account name for the address label.

Your request is automatically entered in the queue. In this example, Shepard's print request will not be executed until it reaches the head of print queue three.

NOTE

Deferred print requests are queued on disk and are not lost when the system is restarted.

Program Execution

Most of the programs you write will require some type of input and output. Before you execute a program, you must provide information that tells the program where your input will come from and where your output will go. The GET and REMOVE commands allow you to reserve files and devices for program input and output, and, after program execution, to cancel those reservations.

GET performs two functions. First, it reserves a file or device for use by the executing program. This reservation may set exclusive access or some degree of shared access (see "Reserving Files or Devices"). Secondly, GET establishes a relationship between pathnames and the logical file numbers (LFNs) by which you can gain access to files and devices.*

Once program execution has terminated, you can use the REMOVE command to cancel file/device reservations and the LFNs that your program assigned with the GET command.

For example, if you are compiling the COBOL program CARDIN, CARDIN uses two files, a card reader (from which input will be read) and a disk file (to which output will go). The program refers to these two files by internal file names (IFNs) 0A and 0C, which correspond (map) to logical file numbers (LFNs) 1 and 3.**

After linking your object unit into a bound unit, you must use the GET command to reserve an input file (a card reader) and an output file (a disk file).

To reserve the card reader, specify:

```
GET !CDR00 -LFN 1
```

* Using a GET command will override any internal LFN assignments you have included in your FORTRAN, BASIC, or Assembly language program.

**You assigned these IFNs when you wrote your COBOL program. The system maps these IFNs to corresponding LFNs.

To reserve the disk file, specify:

```
GET COBOL_DIR>MASTER -LFN 3
```

In this example, it is assumed that the file MASTER was previously created under the directory COBOL_DIR. It is also assumed that the directory COBOL_DIR is subordinate to your working directory. (The GET command could have directed program output to any file, not necessarily one named MASTER.)

If you have already loaded the card reader, you can now execute CARDIN by entering the simple pathname (since the bound unit CARDIN is in your working directory):

```
CARDIN
```

The program reads cards into the file MASTER. Once the program terminates, remove the device and file reservations:

```
REMOVE !CDR00 -LFN 1
RDY:
REMOVE COBOL_DIR>MASTER -LFN 3
```

Reserving Files or Devices

You can use the GET command to reserve a file or device for use by your task group. When you reserve a file, you can specify whether other users will be allowed some form of concurrent access.

For example, when you reserve a disk file, you can specify that all users can read the file while you have it reserved, but that only you can alter (write to) it. To do this, enter:

```
GET FILEB -LFN 1 -SHARE R
```

If a directory is reserved exclusively for you (using the -SHARE N argument), then all subdirectories and files are also reserved exclusively. Thus, entering:

```
GET ^VOL04 -SHARE N
```

reserves the entire volume VOL04 for your exclusive use. Note that the system always reserves tapes exclusively for your use when you reserve them.

Communicating With Other Users

You can send messages to the system operator and send (or receive) mail to other system users by using the Message (MSG) or MAIL commands. Messages sent to the operator are displayed immediately on the operator terminal; mail is not displayed until the receiver enters the MAIL command.

To send a message to the operator, you might enter the following request:

```
MSG "PLEASE MOUNT "VOLA"
```

You must enclose your message in quotation marks (or apostrophes) if it contains embedded blanks.

To send mail to another person, you might enter:

```
MAIL LOWELL
```

where LOWELL is the person_id of the receiver. The system will respond:

```
INPUT:
```

You can then enter the text of your message. Terminate the message by entering a period (.) or the letter Q followed by a carriage return. Your message is queued in Lowell's mailbox until Lowell issues a MAIL command to display mail.

To mail a file that might be a program or a long message for many users, use the filename argument of the MAIL command:

```
MAIL LOWELL HEX_AS.A
```

This command mails the file HEX_AS.A to Lowell. Long messages should not be sent to users at a VIP terminal.

NOTE

Before you can receive mail, either you or your system operator must have previously created the mailbox directory and the necessary mailboxes, and have set access controls on these mailboxes. See the System User's Guide for details.

ABSENTEE PROCESSING

MOD 400 offers both interactive and absentee (batch) processing. As an absentee user, you submit requests against the system batch task group ($). Absentee processing allows you to perform multiprocessing on the system, i.e., you can process interactive tasks while the system processes one or more of your absentee requests simultaneously. All system software components are available to you as an absentee user.

The system operator creates the batch task group against which all users place requests on a queued first-in, first-out basis. To enter a request into the batch queue, use the Enter Batch Request (EBR) command. The EBR command requires you to

specify a command-in file containing commands to be executed in the batch task group. Normally, you create this file on disk in your working directory before entering your batch request.

For example, suppose you want to compile, link, and execute an application program called PAYROLL in batch mode. You have previously created the command-in file PAYR_IN in your working directory ^ZSYS01>IW. PAYR_IN is the file from which the Command Processor will read its commands.

The file PAYR_IN contains the following commands:

```
COBOLA PAYROLL -LO
LINKER PAYROLL
LIB ZSYS02>ZCART
LINK PAYROLL
MAP;QT
GET DEPT4 -LFN 2
GET OUTPUT -LFN 3
PAYROLL
BYE
```

Your command-in file can contain any combination of legitimate commands, such as compile/link/execute sequences, including any necessary file control commands (GET, REMOVE); or file print/dump commands. The commands must appear in the file as if they were being executed from your terminal.

Any time after you create the file PAYR_IN, you can enter a batch request against it, by specifying:

```
EBR PAYR_IN -WD ^ZSYS01>IW
```

Output from the COBOL program PAYROLL that would normally appear at your terminal is written to a file named PAYR_IN.AO.

Section 4
SCREEN EDITOR

The Screen Editor (also referred to as SCORPEO in the software) is a full screen, interactive text editing, and documentation preparation tool. You can manipulate full screens of data at once making text editing faster and simpler by using the features of a video display terminal.

This section describes the Screen Editor capabilities as well as the directives used to create, modify, and save files.

In this section, cursor position is designated by a shading.

OVERVIEW

The Screen Editor creates and/or alters character text in a file. If you are creating a source unit file, the statements can be written in FORTRAN, COBOL, BASIC, Pascal, RPG, or Assembly language. See the appropriate language manual for details.

You control editing by using a combination of directives, function keys, and labeled keys.

You can edit files created by the Line Editor with the Screen Editor. The Line Editor is described in Section 5.

All editing is done in a temporary work area called a buffer. This buffer references a pair of temporary workfiles.

When you invoke the Screen Editor, a buffer and associated work files are automatically created for you. To save the Screen Editor output, you must write the contents of the buffer to a file.

NOTE

During a single execution of the Screen Editor, you can read only one file. You must write the contents of the buffer out to another file or to the same file. To edit a second file, you must reinvoke the Screen Editor (see "Calling the Screen Editor" later in this section).

SCREEN EDITOR PROCESSING

You control Screen Editor processing by specifying directives, or using function keys or labeled keys. The subsections that follow describe the following Screen Editor processing functions.

- Terminal and Keyboard Requirements -- Defines the terminals and keyboards supported.
- Suffix Conventions -- Describes file naming conventions.
- Calling the Screen Editor -- Describes the command used to invoke the Screen Editor.
- Screen Description -- Defines and illustrates the three regions of the screen.
- Creating a File -- Describes the procedure for creating a file.
- Changing an Existing File -- Describes the procedure for modifying a file.
- Interrupting Screen Editor Processing -- Describes the procedures used to stop Screen Editor processing.
- Entering Screen Editor Directives -- Defines the rules for entering Screen Editor directives.
- Directive Format -- Defines the rules for specifying directive formats.
- Designating Lines -- Describes the procedure used for locating, adding, and deleting lines from a file.

- Special Characters -- Defines the characters used to specify a processing function.

- Directive Set -- Defines each directive in alphabetical order by directive name. Provides the information necessary for using the directives to create and modify files.

- Function Key Descriptions -- Defines each function key in alphabetical order by function name. Describes function key use in creating and modifying files.

- Labeled Key Descriptions -- Describes each labeled key as it is used by the Screen Editor in creating and modifying files.

TERMINAL AND KEYBOARD REQUIREMENTS

The Screen Editor can be used with the following asynchronous terminal types:

7801	7300 (general purpose)
7802	7300 (word processing)
7803 (word processing)	7300 (data entry)

It is recommended that the baud rate of the terminal is at least 1200 baud.

SCREEN EDITOR SUFFIX CONVENTIONS

When you create a source unit, you should append the appropriate suffix identification character to the name of the file that will contain the source unit. The suffix designates the type of programming language that constitutes the source unit. The suffix must be .C for COBOL programs, .F for FORTRAN programs, .B for BASIC programs, .PS for Pascal programs, and .A or .P for Assembly language programs.

When you specify the file names of Screen Editor input and output files (when calling the Screen Editor, and in selected directives), you must designate the complete file name, including the suffix that denotes the contents of the file (.C for COBOL, .F for FORTRAN, .B for BASIC, .PS for Pascal, and .A or .P for Assembly language programs). The Screen Editor does not append a suffix to its input and output files.

LOADING THE SCREEN EDITOR

To load the Screen Editor when running under the menu subsystem, select the SCORPEO option from one of the selection menus that contains it, and press TRANSMIT.

To load the Screen Editor by a command line, enter the SCORPEO command.

FORMAT:

```
SCORPEO  [path]  [ {-LINES} nnnnn ]
                 [ {-L    }       ]
```

ARGUMENTS:

None or any number of the following control arguments may be entered:

[path]

Pathname of the file you wish to edit. Can be any valid form of pathname. If you are creating a new file without any initial input from another file, do not enter a pathname. If you specify path, the first 17 lines of the file will fill the window (text region).

{-LINES} nnnnn
{-L   }

Approximate number of lines the Screen Editor should hold in main memory during the current editing session. nnnnn is a positive decimal value. If you specify less than 100 lines, a value of 100 lines is used.

Default: 200 lines.

Once you have loaded the Screen Editor, a screen such as that described in Figure 4-1 or Figure 4-2 is displayed. If you are creating a file (you invoked the Screen Editor without a pathname), the screen shown in Figure 4-1 is displayed. If you are modifying (editing) a previously created file, a screen similar to that shown in Figure 4-2 is displayed.

Description of the Screen

The display on the terminal used by the Screen Editor is broken down into three distinct areas called "regions". Each region is described in detail in the following paragraphs. Refer to Figure 4-1 and Figure 4-2 for the location of each region.

```
LEFT MARGIN = 001      CURRENT LINE = 00001
**********TOP OF FILE****************************** TOP OF FILE ***************

                                    (17 blank lines)

---------- ---------- ---------- ---------- ---------- ---------- ----------
DIRECTIVE:
```

Figure 4-1 Sample Screen for Creating a File

```
^VOL1>DIR>INVTNRY
LEFT MARGIN = 001        CURRENT LINE = 00001      M              MODIFIED
**********TOP OF FILE  ***************************** TOP OF FILE  ***************

                        (17 lines of text are displayed here)

---------- ---------- ---------- ---------- ---------- ---------- ----------
DIRECTIVE:
```

Figure 4-2. Sample Screen for Modifying a File

STATUS REGION

The status region of the screen is that area at the top of the screen that shows the status of the file.

When you are creating a file, the information displayed in the status region is: the current position of the left margin and the current line number (as shown in Figure 4-1).

When you are editing a file, the information displayed is: the full pathname of the file you are editing, the flag "MODIFIED" (if modified), and the current position of the left margin (as shown in Figure 4-2).

TEXT REGION

The text region of the screen is the large area in the middle of the display where you actually create and modify the text. The text region is also called the "window." The window is 18 lines long and from 80 to 256 characters wide. (You control the width of the text with the Window Width directive described later in this section.) The maximum record length of a file is 256 characters.

When the first 17 lines of a file appear in the window, you will see at the top of the window a line designated as TOP OF FILE. This line is called the control line. You can use this line to verify that you are at the beginning of the file and to position the cursor when you want to insert text before the first line of your file. This line does not appear within your file.

At the bottom of the window is a line of dashes and vertical bars. This line is called the tab designator line. The position of the vertical bars indicates the current tab stops. To change these tab stops, use the Language Type directive or the TAB SET/TAB CLR key (both are described later in this section).

DIRECTIVE REGION

The directive region is at the bottom of the screen just under the tab stop designator line. The cursor is positioned here when you press the HOME key to enter directives to the Screen Editor. Screen Editor directives are described later in this section.

Immediately below the directive region is the area where you can view system messages.

Creating a Source Unit

To create a source unit, perform the following steps. Directives, function keys, and labeled keys are described later in this section.

1. Change the working directory to a user volume by specifying the Change Working Directory command (see the Commands manual) or by using the CWD form.

2. Call the Screen Editor (see "Calling the Screen Editor" earlier in this section).

3. Enter the source unit text.

4. Make changes, if necessary, by entering the appropriate directives, by using the function keys or the labeled keys, or by typing over existing text.

5. Write the contents of the buffer to a file by using the Write directive.

6. Exit from the Screen Editor by entering the Quit directive.

Changing an Existing Source Unit

To change an existing source unit, perform the following steps. (You can change a source unit that was created using the Line Editor, described in Section 5 and in Appendix A, with the Screen Editor.) Directives, function keys, and labeled keys are described later in this section.

1. Change the working directory to a user volume by specifying the Change Working Directory command (see the Commands manual) or by using the CWD form.

2. Call the Screen Editor (see "Calling the Screen Editor" earlier in this section) optionally specifying the pathname of the source file you wish to modify.

3. If you did not specify a file pathname when you called the Screen Editor, use the Read directive to read into the buffer the source unit you wish to edit.

4. Use the appropriate Screen Editor directives, function keys, and labeled keys or simply type over existing text to modify the source unit.

5. Write the contents of the buffer to the file from which the lines were read or to a different file by using the Write directive.

6. Exit from the Screen Editor using the Quit directive.

Interrupting Screen Editor Processing

You can interrupt Screen Editor processing by either:

- Pressing the QUIT function key.

  Pressing the QUIT function key is the perferred way to terminate processing. If no modified buffer exists (i.e., the user has not changed a line in the file, or has not written a newly created file), then the Screen Editor terminates when the QUIT function key is pressed. If a modified buffer exists, then a question "Modified buffer exists. Do you still wish to quit? (Answer yes of no.)" is displayed and you make a choice (yes = teminate; no = resume). Thus, the system does not process the QUIT function key, but the Screen Editor does in a very specific way, as described here.

- Pressing the INTERRUPT or BREAK key on your terminal.

- Entering Δ CΔBgroup-id on the operator terminal, where group-id is the two character task group name of the group containing the Screen Editor task.

A **BREAK** message is displayed on the 25th line of your terminal when the system interrupts the Screen Editor. At this point, there are four actions you can take:

1. Enter any user command found in the Commands manual.

2. Enter the Start (SR) command. The Screen Editor resumes processing as if it had not been interrupted. All of the changes to the file you were editing at the time of the interrupt are intact.

3. Enter the Unwind (UW) command. The Screen Editor terminates processing and the system returns to command level. None of the changes made to the file since the last write directive are saved.

   Using UW causes the Screen Editor to terminate unconditionally.

4. Enter the New Process (NEW_PROC) command. The current task group is aborted and then restarted using the same arguments specified when you logged in. None of the changes you made to the file since the last write directive are saved.

ENTERING SCREEN EDITOR DIRECTIVES

To enter any of the Screen Editor directives, press the HOME key to position the cursor in the directive region of the screen. See the description of the HOME key later in this section under "Labeled Keys." After entering the directive and its arguments (if any), press the RETURN key to execute the directive.

If you have entered a directive line and you decide not to use the directive before you press the RETURN key, you can cancel the directive by pressing the Cursor Up ( ) labeled key. (This key is described in detail later in this section.) The cursor is placed in the text region at the location on which the cursor was positioned before you pressed the HOME key.

Screen Editor Directive Format Conventions

Most Screen Editor directives consist of only a directive name, a directive name preceded by one or two line numbers, a directive name optionally preceded by one or two line numbers and followed by text, or only a line number. You cannot specify a directive line longer than 70 characters. You can specify only one directive on a line. Directive formats are:

FORMAT 1:

```
line_number
```

FORMAT 2:

```
dirname
```

FORMAT 3:

```
line_number   dirname
```

FORMAT 4:

```
line_number ,line_number   dirname
```

FORMAT 5:

```
dirname  text
```

FORMAT 6:

```
line_number  dirname  text
```

FORMAT 7:

```
line_number ,line_number   dirname  text
```

NOTES

1. If a directive includes text, you must leave at least one space between the directive name and the text.

2. If you specify two line numbers in a directive line, separate them by a comma (,); do not use any spaces.

DESIGNATING LINES

You can locate each line in the buffer by entering a decimal number that indicates the file-relative position of the line within the buffer. The first line in the buffer is line 1; subsequent lines are numbered sequentially in ascending order.

Screen Editor directives may cause lines to be added to or deleted from the buffer. Each time this occurs, all succeeding lines are renumbered. For example, if line 15 is deleted, line 16 becomes 15, and each subsequent line number is decremented by 1.

If you specify a line that is not in the buffer, an error message is displayed.

BLOCK DESCRIPTION

The BLOCK function key defines a "block," or subset of the buffer upon which some action will be taken.

You define a block by positioning the cursor on the desired starting position and pressing the BLOCK function key. Next, you position the cursor on the ending position and again press the BLOCK function key. It is not necessary to set the starting position of the block first. The starting block position is always considered to be the block definition closest to the beginning of the buffer. The ending block position is always considered to be the block definition closer to the end (last line) of the buffer.

A block is defined by its location, i.e., the line and column numbers of its starting and ending points. For example, if you define a block beginning in line 1, column 1, and ending in line 10, column 80, and then you delete lines 5 through 10, the resulting block begins at line 1, column 1, and ends at the "old" line 16 (now the "new" line 10).

Once you define a block, it can be acted upon. You perform actions on a block using the Move Block, Copy Block, Erase Block, and Delete Block function keys, and the Write Block and Change Block directives. These function keys and directives are described later in this section.

When defining a block for a Move Block or Copy Block directive, it is possible to split a line. A line split occurs when you try to insert or delete a block other than at the endpoints of a line. If you do this, the lines are truncated (if you insert) or concatenated (if you delete). Spaces are considered trivial characters and are truncated without a warning message. If concatenation causes an overflow of the maximum line length, the overflowing characters are truncated.

The DELETE BLOCK function key always deletes all the text in the block. If both the block start and block end positions are split lines, the two split lines at the end of each block are concatenated. The result is the display of one line where the block definition had been previously. For example:

The following lines are in the window with the block start and end positions denoted by shaded rectangles.

```
THIS IS AN EXAMPLE TO SHOW WHAT HAPPENS
WHEN THE TEXT IS DELETED BY USING
THE DELETE BLOCK FUNCTION KEY WITH SPLIT LINES.
```

Pressing the DELETE BLOCK function key results in the following:

```
THIS IS AN EXAMPLE WITH SPLIT LINES.
```

Use of the COPY BLOCK function key has a possibility of two different split lines. The left portion of the line to which you are copying (up to the cursor position) is concatenated to the block start position. The right portion of the line on which you are copying is concatenated to the block end position. The result is the same as if you inserted characters. For example:

The following lines are in the window with the block start and end positions designated by shaded rectangles. The position at which you want to copy is designated by an arrow.

```
THIS IS AN EXAMPLE OF ↶ COPY BLOCK FUNCTIONALITY.
THE DEFINED BLOCK WILL BE COPIED AT THE CURSOR POSITION.
THIS LINE IS THE START OF  THE COPY BLOCK.
ALL THE TEXT IN THE BLOCK WILL BE COPIED
TO THE COPY POSITION. THIS SHOWS  THAT THE BLOCK WILL NOT BE
DELETED.
```

Pressing the COPY BLOCK function key produces the following:

```
THIS IS AN EXAMPLE OF THE COPY BLOCK.
ALL THE TEXT IN THE BLOCK WILL BE COPIED
TO THE COPY POSITION. THIS SHOWS COPY BLOCK FUNCTIONALITY.
THE DEFINED BLOCK WILL BE COPIED AT THE CURSOR POSITION.
THIS LINE IS THE START OF THE COPY BLOCK.
ALL THE TEXT IN THE BLOCK WILL BE COPIED
TO THE COPY POSITION. THIS SHOWS THAT THE BLOCK WILL NOT BE
DELETED.
```

The MOVE BLOCK function key operates the same with split lines as does the COPY BLOCK with a DELETE BLOCK. The split lines are the same as shown previously, except that the block is deleted from its previous position and the left portion of the block start line is concatenated to the right portion of the block end line. Using the same example as used in the COPY BLOCK example above, pressing the MOVE BLOCK function key results in the following:

```
    THIS IS AN EXAMPLE OF THE COPY BLOCK.
.   ALL THE TEXT IN THE BLOCK WILL BE COPIED
-   TO THE COPY POSITION. THIS SHOWS COPY BLOCK FUNCTIONALITY.
    THE DEFINED BLOCK WILL BE COPIED AT THE CURSOR POSITION.
    THIS LINE IS THE START OF THAT THE BLOCK WILL NOT BE
    DELETED.
```

The Write Block directive writes a split line as a line by itself. For example assume you defined a block as shown:

```
MOVE PAY TO OUT_PAY.
MOVE CREDIT_UNION TO OUTPUT_CU.
MOVE FED_TAXES TO OUTPUT_FED_TAX.
```

Using the Write Block directive result in the following:

```
TO OUT_PAY.
MOVE CREDIT_UNION TO OUTPUT_CU.
MOVE FED_TAXES
```

The Change Block directive is not affected by split lines.

SPECIAL CHARACTERS

When the following ASCII characters are included in search_expressions or change_expressions, they have special meanings. All special characters can be used only in search_expressions, except the & special character, which can only be used in change_expressions.

Character	Description
*	Request expressions that contain any number (or none) of the preceding character(s). If this character is the first character of a regular expression, it has no special meaning.
^	When designated as the first character of an expression, request lines that begin with the specified expression (excluding the character ^).

$	When specified as the last character of an expression, request lines that end with the specified expression (excluding the character $).
.	Can be any character on a line; specify one per character (e.g., .. means any two characters on any line).
&	Can only be used in the change_expression of a change directive to indicate that the strings of characters preceding and following & are to be concatenated to the target string of the search.
!C	Request that the following character be interpreted not as a special character (e.g., !C* means match an asterisk). Specify the "C" in upper case.
[n]x	Request a repeat factor ([n]) of the specified character (x). x can be any character including "." (e.g., [25]. matches any 25 columns characters).

Summary of Screen Editor Directives

Table 4-1 lists each Screen Editor directive mnemonic, summarizes its function, and designates the directive name under which it is more fully described.

SCREEN EDITOR DIRECTIVES

Screen Editor directives are described in detail on the following pages. In the examples, numbers in parentheses are references to line numbers and do not appear in memory or in the text.

Table 4-1. Summary of Screen Editor Directives

Directive Mnemonic	Function	Directive Name
BL	Display last line of buffer.	Bottom Line
C	Change one character string to another character string.	Change
CA	Change all occurrences of a character string in the buffer to another character string.	Change All
CB	Change all occurrences of a character string in a block to another character string.	Change Block
line number	Display a line of text.	Display
LC	Convert all upper case characters in a block to lower case.	Lower Case
LM	Display the left margin of the buffer.	Left Margin
LT	Set tabs stops for the specified programming language.	Language Type
Q	Conditionally terminate execution of the Screen Editor.	Quit
R	Read text from the specified file to the buffer.	Read
RM	Display the right margin of the buffer.	Right Margin
S	Search the buffer for the specified character string.	Search
SB	Search the buffer for the specified character string from the current cursor position backward to line 1.	Search Backward
SC	Change the number of lines to scroll.	Scroll Change

Table 4-1 (cont) Summary of Screen Editor Directives

Directive Mnemonic	Function	Directive Name
SF	Search the buffer for the specified character string from the current cursor position forward to the last line of the buffer.	Search Forward
TB	Do not suppress trailing blanks when text is written to a file.	Trailing Blanks
TL	Display line 1 of the buffer.	Top Line
UC	Convert all lower case characters in a block to upper case.	Upper Case
V	Display the current version of the Screen Editor.	Version
W	Write the contents of the buffer to a file.	Write
WB	Write the specified block of text in the buffer to a file.	Write Block
WW	Set the window width to the specified value.	Window Width

BOTTOM LINE

BOTTOM LINE (BOTTOM_LINE OR BL)

Display the last line of the buffer at the top of the current window.

The cursor is positioned on the last line (the bottom line) of the file in the same column in which it was positioned before it was moved to the directive line.

FORMAT:

```
{BOTTOM_LINE}
{BL         }
```

Example:

```
BL
```

Display the last line of the buffer at the top of the current window.

CHANGE (CHANGE OR C)

Search the buffer for the specified search_expression and replace the first occurrence of the search_expression with the change_expression.

Searching begins at the current cursor position. Searching proceeds in a forward direction until the end of the file is reached, and, if necessary, resumes at the top of the file and proceeds forward to the current cursor position.

The search_expression must be found wholly on a line of the file to be considered a matched string.

When the directive has completed execution and a match was found, the cursor rests on the first character of the changed expression. The changed line is displayed as the first line in the window.

If a match was not found, the message "SEARCH FAILED" is displayed.

It is not necessary to repeat the search_expression for subsequent identical changes. Simply entering the first two delimiters and the change_expression changes the next occurrence of the search_expression to the change_expression.

FORMAT:

```
{CHANGE}   "search_expression"change_expression"
{C     }
```

ARGUMENTS:

"

(Delimiter) Can be any character. You must use the same character in each of the three locations where a delimiter is required. If using a delimiter that is a character within the search_expression or the change_expression, you must use the special character "!C" before the character within the text. It is recommended that you use a delimiter that is not within the search_expression or the change_expression.

search_expression

Character string for which the Screen Editor is searching. The first occurrence of this character string will be replaced with the character string specified in the change_expression.

change_expression

Character string that will replace the first occurrence of the argument "search_expression."

Example 1:

```
C  "ABC"DEF"
```

Change the next occurrence in the file of ABC to DEF.

Example 2:

```
C  ""DEF"
```

Change the next occurrence of the previously defined search_expression ABC to DEF.

Example 3:

```
C  "^.*$"DEF"
```

Change the next line to DEF.

Example 4:

```
C """
```

Delete the next occurrence of the previously designated search_expression.

CHANGE ALL (CHANGE ALL OR CA)

Search the buffer for all occurrences of the specified character string and replace all occurrences of the character string with another specified character string.

Each occurrence of the search_expression must be found wholly on a line of the file to be considered a match string.

After this directive is executed, the cursor is positioned on the first character of the last changed character string. The changed line is displayed as the first line of the window.

FORMAT:

```
[n[,m]] {CHANGE_ALL} "search_expression"change_expression"
        {CA        }
```

ARGUMENTS:

[n[,m]]

Starting line number (n) and ending line number (m) between which the specified search_expression is changed. If you specify both starting and ending line numbers, all occurrences of the specified search_expression found between the line numbers are changed. If you specify only a starting line number, only those occurrences of the search_expression from the specified line number to end of the buffer are changed. If you do not specify line numbers, the search for the search_expression begins at the current cursor position. Searching proceeds in a forward direction until the end of file is reached and resumes at the top of the file until all occurrences of the search_expression are replaced. When the change is completed, the cursor rests on the last changed expression.

(Delimiter) Can be any character. You must use the same character in the three locations where a delimiter is required. If using a delimiter that is a character within the search_expression or change_expression, you must use the special character "!C" before the character within the text. It is recommended that you use a delimiter that is not within the search_expression or the change_expression.

search_expression

Character string for which the Screen Editor is searching. Each occurrence of this string according to the line number values specified by n or m (see above) will be replaced with the character string specified in the "change_expression" argument.

change_expression

Character string that will replace each occurrence of the search_expression.

Example 1:

```
CA  "ABC"DEF"
```

Change all occurrences in the buffer of ABC to DEF.

Example 2:

```
5,10  CA  "ABC"DEF"
```

Between (and including) lines 5 to 10 of ABC to DEF, change all occurrences

Example 3:

```
5  CA  "ABC"DEF"
```

Between (and including) line 5 and the last line of the buffer, change all occurrences of ABC to DEF.

CHANGE BLOCK

CHANGE BLOCK (CHANGE_BLOCK OR CB)

Search the current block for all occurrences of the specified expression and replace each occurrence of the expression with another specified expression.

Before using this directive, you must define the block of text you wish to change (see the description of the Block function key under "Function Keys" later in this section).

See "Block Description" earlier in this section for details on blocks.

FORMAT:

```
{CHANGE_BLOCK}   "search_expression"change_expression"
{CB           }
```

ARGUMENTS:

▪

(Delimiter) Can be any character. You must use the same character in the three locations where a delimiter is required. If using a delimiter that is a character within the search_expression or change_expression, you must use the special character "!C" before the character within the text. It is recommended that you use a delimiter that is not within the search_expression or the change_expression.

search_expression

Character string for which the Screen Editor is searching. Each occurrence of this string within the defined block will be replaced with the character string specified in the "change_expression" argument.

change_expression

Character string that will replace each occurrence of the search_expression.

Example:

You have previously defined a block as

C:=A-B,D:=C-B,E:=D-C,

Enter the directive

CB ",";"

When the directive is executed, the resulting block is:

C:=A-B;D:=C-B;E:=D-C;

DISPLAY

Display a specified line of text.

After executing the Display directive, a new page (window) of text is displayed. The specified line of text appears as the first line of the new window.

The cursor is positioned on the new line in the same column in which it was positioned before you executed the directive.

FORMAT:

line_number

ARGUMENT:

line_number

Line number (decimal) of the text you wish displayed. The line number must be a positive integer whose maximum value is 65535. The specified line will appear at the top of the window.

Example:

Display line 35 of the buffer on the first line of the window.

LANGUAGE TYPE

LANGUAGE TYPE (LANGUAGE_TYPE OR LT)

Set tabs stops for the specified programming language.

FORMAT:

```
{LANGUAGE_TYPE}  [language]
{LT            }
```

ARGUMENT:

language

Programming language in which you are creating or editing your source file. Specify the language as shown below to set the appropriate tab stops:

Language	Tab Stops (Column)
(default)	11, then every 10 columns
Assembly or A	11, then every 10 columns
COBOL or C	8, 12, 21, then every 10 columns through column 61, 73
FORTRAN or F	7, 11, 21, then every 10 columns through column 61, 73
PASCAL or P	11, then every 10 columns
BASIC or B	11, then every 10 columns

If you do not specify "language," the tab stops are set as specified in the default tab stops above.

Example:

```
LT  COBOL
```

Set tab stops at columns 8, 12, 21, and then every 10 columns. The tab stop line at the bottom of the text region is changed accordingly.

LEFT MARGIN (LEFT_MARGIN OR LM)

Display the left margin of the buffer in the current window.

The left margin is always set to the first character of each line.

The cursor is positioned in column 1 on the line on which it was positioned before you executed the directive.

See the Window Left fuction key description later in this section.

FORMAT:

```
{LEFT_MARGIN}
{LM         }
```

Example:

LM

Display the first 80 columns of text in the current window.

LOWER CASE

LOWER CASE (LOWER_CASE OR LC)

Convert all upper case characters in a previously defined block to lower case characters.

If you specify characters within apostrophes ('), those characters are not converted.

You must have previously defined a block before you can use this directive. See the Block function key description later in this section for information on defining blocks.

FORMAT:

```
{LOWER_CASE}
{LC        }
```

Example:

Assume you have defined the following block:

```
THIS PROGRAM CALCULATES THE WEEKLY 'GROSS' AND 'NET' PAY
```

Enter the Lower Case directive:

```
LC
```

The block now reads:

```
this program calculates the weekly 'GROSS' and 'NET' pay
```

QUIT (QUIT OR Q)

Terminate the current screen editing session and close the file associated with it.

You must specify the Quit directive at the end of a screen editing session.

Quit is executed conditionally. If you have modified a file and enter the Quit directive without having saved (written) the file (see the Write directive later in this section), you are warned that a modified file exists. If you want to save the edited text, answer "NO" or "N" to the prompt "Modified buffers exist. Do you wish to quit? (Answer yes or no.)", and enter a Write directive to save the file. Now enter the Quit directive. If you do not wish to save the modified text, answer "YES" or "Y" to the prompt.

The QUIT function key operates exactly as the Quit directive.

FORMAT:

```
{QUIT}
{Q   }
```

Example:

```
Q
```

Terminate the current screen editing session.

READ

READ (READ OR R)

Place the contents of the specified file into the buffer.

The cursor is positioned in column 1 of the first line of the file.

If you have not specified the pathname of the file when you called the Screen Editor (see "Calling the Screen Editor" earlier in this section), use this directive first to read in the file you wish to edit.

You may only use the Read directive once during the current screen editing session (i.e., you may only edit one file at a time).

File concurrency for the specified file is exclusive read and exclusive write.

NOTES

1. During the read, all tab characters are replaced with the appropriate number of blanks according to the currently defined tan stops. If this occurs, the "MODIFIED" status is displayed. When the file is saved (written), it will contain no tab characters.

2. During the read, if any hexadecimal sequence contains a non-ASCII character (i.e., hexadecimal characters 00 to 19 and 80 to FF), each non-ASCII character is replaced by the ASCII period (.) character, and the "MODIFIED" status is displayed.

FORMAT:

```
{READ}
{R   }   path
```

ARGUMENT:

path

Pathname of the file to be read. Can be any valid form of pathname.

Example:

```
R  FILEA
```

Read the contents of the file FILEA into the buffer.

RIGHT MARGIN (RIGHT_MARGIN OR RM)

Display the right margin of the buffer in the current buffer.

The current window is moved to the right so that column 80 of the display coincides with the column that is the current window width.

The cursor is positioned in the last column of the line on which it was positioned before you executed the directive.

Use this directive to view text beyond column 80.

See the Window Right function key described later in this section.

FORMAT:

```
{RIGHT_MARGIN}
{RM          }
```

Example:

```
RM
```

Display the right margin of the buffer in the current window.

SCROLL CHANGE

SCROLL_CHANGE (SCROLL_CHANGE OR SC)

Change the number of lines that move through the text region (window) when you press the Window Up or Window Down function keys. (The Window Up and Window Down function keys are described under "Function Keys" later in this section.)

FORMAT:

```
{SCROLL_CHANGE}  [lines]
{SC           }
```

ARGUMENT:

[lines]

Number of lines to move the current window. Can be any positive decimal integer. If a boundary (top or bottom line) occurs before the specified number of lines are scrolled, the boundary is displayed and scrolling stops. This value remains in effect until explicitly changed by another Scroll Change directive.

Default: 18 lines.

Example:

If the current window displays the lines

```
(1)
(2)
(3)
(4)
(5)
 .
 .
 .
(18)
```

and you enter the directive line SC 4 and press the Window Down function key, the current window will display:

```
(5)
(6)
(7)
 .
 .
 .
(22)
```

SEARCH (SEARCH OR S)

Search the buffer for the specified search_expression (character string).

The cursor is positioned on the first character of the matched search_expression. The line containing the match is displayed on the first line of the window.

If you have previously defined a search_expression, simply entering the directive followed by the two identical delimiters will search for the next occurrence of the search_expression.

Searching begins at the current cursor position, continues to the end of the file, and, if no match is found, begins at the top of the buffer (line 1 of the file) and continues to the current cursor position.

If no match is found, the message "SEARCH FAILED" is displayed at the terminal.

FORMAT:

```
[n[,m]] {SEARCH}   "search_expression"
        {S     }
```

ARGUMENTS:

[n]

Line number at which to begin the search. If you do not specify n, search begins at the current cursor position.

[,m]

Line number at which to end the search. If you do not specify m and have specified n, search ends at the last line of the file.

"

(Delimiter) Can be any character. You must use the same character in the two locations where a delimiter is required. If using a delimiter that is a character within the search_expression, you must use the special character "!C" before the character within the text. It is recommended that you use a delimiter that is not within the search_expression.

search_expression

String of characters that is the object of the search.

Example 1:

```
S  "ABC"
```

Search for the first occurrence after the current cursor position of the string ABC.

Example 2:

```
S  /AB"C/
```

Search for the first occurrence after the current cursor position of the character string AB"C.

Example 3:

```
S  BAB
```

Search for the first occurrence after the current cursor position of A. Since the first non-blank character after the directive (B) is used as the delimiter, the search_expression is that character string found between the first and second occurrence of the character B.

Example 4:

```
S  ""
```

Search for the next occurrence of the previously defined search_expression.

SEARCH_BACKWARD (SEARCH_BACKWARD OR SB)

Search the buffer from the current cursor position back to line 1 for the specified search_expression.

The cursor is positioned on the first character of the matched search_expression. The line containing the match is displayed on the first line of the window.

If you have previously defined a search_expression, simply entering the directive followed by the two identical delimiters will search for the next occurrence of the search_expression.

Searching begins at the current cursor position and continues backwards, from right to left, toward line 1 of the buffer until a match is found. If no match is found, the message "SEARCH FAILED" is displayed.

FORMAT:

```
{SEARCH_BACKWARD}   "search_expression"
{SB              }
```

ARGUMENTS:

"

(Delimiter) Can be any character. You must use the same character in the two locations where a delimiter is required. If using a delimiter that is a character within the search_expression, you must use the special character "!C" before the character within the text. It is recommended that you use a delimiter that is not within the search_expression .

search_expression

String of characters that is the object of the search.

Example 1:

```
SB  "ABC"
```

Search for the first occurrence before the current cursor position of the string ABC.

Example 2:

```
SB  "AB!C"C"
```

Search for the first occurrence before the current cursor position of the string AB"C.

Example 3:

```
SB  BAB
```

Search for the first occurrence before the current cursor position of A. Since the first non-blank character after the directive (B) is used as the delimiter, the search_expression is that character string found between the first and second occurrence of the character B.

SEARCH_FORWARD (SEARCH_FORWARD OR SF)

Search the buffer from the current cursor position to the end of the buffer for the specified search_expression.

The cursor is positioned on the first character of the matched search_expression. The line containing the match is displayed on the first line of the window.

If you have previously defined a search_expression, simply entering the directive followed by the two identical delimiters will search for the next occurrence of the search_expression.

Searching begins at the current cursor position and continues foward, from left to right, toward the last line of the buffer until a match is found. If no match is found, the message "SEARCH FAILED" is displayed.

FORMAT:

```
{SEARCH_FORWARD}   "search_expression"
{SF            }
```

ARGUMENTS:

*

(Delimiter) Can be any character. You must use the same character in the two locations where a delimiter is required. If using a delimiter that is a character within the search_expression, you must use the special character "!C" before the character within the text. It is recommended that you use a delimiter that is not within the search_expression.

search_expression

String of characters that is the object of the search.

Example 1:

```
SF  "ABC"
```

Search for the first occurrence after the current cursor position of the character string ABC.

Example 2:

```
SF  XAB"CX
```

Search for the first occurrence after the current cursor position of the character string AB"C.

Example 3:

```
SF  BAB
```

Search for the first occurrence after the current cursor position of A. Since the first non-blank character after the directive (B) is used as the delimiter, the search_expression is that character string found between the first and second occurrence of the character B.

Example 4:

```
SF  ""
```

Search for the next occurrence of the previously defined search_expression.

TOP LINE (TOP LINE OR TL)

Display the first line (line 1) of the buffer at the top of the current window.

The cursor is positioned on line 1 in the column in which it was positioned before you executed the directive.

FORMAT:

```
{TOP_LINE}
{TL      }
```

Example:

```
TL
```

Display the first line of the buffer at the top of the window.

TRAILING BLANKS

TRAILING_BLANKS (TRAILING_BLANKS OR TB)

Do not suppress trailing blanks on the lines within the buffer when text is written to a file.

A line with trailing blanks is a line in which some number of characters (at least one) at the end of a line are spaces. If you do not specify this directive, these spaces are lost (discarded) when the line is written to a file. If you do specify this directive, the Screen Editor preserves them.

Once entered, this directive remains in effect until the end of the Screen Editor session.

FORMAT:

```
{TRAILING_BLANKS}
{TB             }
```

Example:

```
TB
```

Do not suppress trailing blanks when you write the buffer to a file.

UPPER CASE (UPPER_CASE OR UC)

Convert all lower case characters in a previously defined block to upper case characters.

If you specify characters within apostrophes ('), these characters are not converted.

You must have previously defined a block before you can use this directive. See the Block function key description later in this section for information on defining blocks.

FORMAT:

```
{UPPER_CASE}
{UC        }
```

Example:

Assume you have defined the following block:

```
This program calculates the weekly gross and net pay
```

Enter the Upper Case directive:

```
UC
```

The block now reads:

```
THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY
```

VERSION

VERSION (VERSION OR V)

Display the current Screen Editor version number.

This directive is informational only; the displayed version number appears in the message region of the screen.

FORMAT:

```
{VERSION}
{V      }
```

Example:

```
V
```

Display the current Screen Editor version number. For example, the message may read

```
SCORPEO  09/04/1121
```

where 09 is the month, 04 is the day, and 1121 is the time associated with the date the SCORPEO bound unit was created.

WINDOW_WIDTH (WINDOW_WIDTH OR WW)

Set the window width for the window.

The width of a window is a measure of how far the right margin of the text can be. The Right_Margin directive positions the right margin to the window's width. The Window Right function key moves the window until the window width is column 80 of the display.

The Window Width directive defines the widest part of the records of a file that can be seen by moving the window to the right. It is independent of the record length of the input file. No data is lost when the window width is smaller than the length of the lines read from a file. However, the parts of the lines that are "beyond" the window width cannot be seen until the window width is made wider.

When you call the Screen Editor (see "Loading the Screen Editor" earlier in this section), the window width is initialized to 256 characters.

FORMAT:

```
{WINDOW_WIDTH}  [length]
{WW           }
```

ARGUMENT:

[length]

Maximum length in characters (bytes) of the window. Specify a decimal digit from 1 to 256.

Default: 80 characters.

Example:

```
WW  132
```

Set the window width to 132 characters.

WRITE

WRITE (WRITE OR W)

Save the specified lines of the buffer in a file.

If you specify a file pathname of a file which already contains text, that text is overwritten.

NOTE

If you write text out to a file other than the file which you are editing, the file of reference changes to the pathname specified in this directive.

FORMAT:

[n[,m]] {WRITE} [path]
{W }

ARGUMENTS:

None or any number of the following control arguments may be entered:

[n[,m]]

The starting line number (n) and the ending line number (m) of the text to be placed into a file. If you do not specify starting and ending line numbers, all lines in the buffer are written to the file. If you specify only a starting line number, all lines in the buffer beginning with that line to the end are written to the file.

If you do not specify a pathname, do not specify line numbers.

[path]

Pathname of the file that is to contain the specified lines of text. Can be any valid form of pathname. If you do not specify a file pathname, the text is placed in the current file whose name you specified when you called the Screen Editor or when you read in a file (see the Read directive earlier in this section). This file is called the file of reference. If you specify a file pathname of a file that does not currently exist, the file is created for you. If the file does not exist, the Screen Editor creates a variable sequential file of control interval size of 512 bytes and a maximum record size of 256 bytes. If you do not specify path, the default pathname used is the current file reference.

Example 1:

```
W  ^VOL1>DIR>INVNTRY
```

Write the contents of the buffer to a file whose pathname is ^VOL1>DIR>INVNTRY.

Example 2:

Assume you called the Screen Editor as follows

```
SCORPEO  ^VOL1>DIR>INVNTRY
```

After editing the file, you wish to write all lines back to the same file. Enter the directive

Example 3:

```
10,20  W  ^VOL1>DIR>INV_NEW
```

Write the contents of lines 10 through 20 to the file named ^VOL1>DIR>INV_NEW.

WRITE BLOCK

WRITE BLOCK (WRITE_BLOCK OR WB)

Write the specified block of text into a file.

You must have previously defined a block of text using the Block function key (described later in this section under "Function Keys").

You cannot write a block of text to the file you are currently editing (i.e., the file of reference).

If you specify a file pathname of a file which already contains text, that text is overwritten.

NOTE

If you write text out to a file other than the file which you are editing, the file of reference changes to the pathname specified in this directive.

Use this directive to "save" a piece (block) of the buffer in a file.

See "Block Description" earlier in this section for details on blocks.

FORMAT:

```
{WRITE_BLOCK}   path
{WB          }
```

ARGUMENT:

path

Pathname of the file that is to contain the block of text. Can be any valid form of pathname. You must specify a pathname different from the pathname of the file whose name you specified when you called the Screen Editor or when you read in a file (see the Read directive earlier in this section). If you specify a file pathname of a file that does not currently exist, the file is created for you. If the file currently exists, the new text overwrites the file's current contents. Do not specify the current file of reference.

Example:

Assume you have already defined a block of text such as:

```
CONST
  FEDTAX   = 0.05;
  STATAXLO = 0.04;
  STATAXHI = 0.07;
```

You want to write this block of text to a file named ^VOL1>DIR>PAY whose contents are:

```
PROGRAM PAY (INPUT,OUTPUT);
(*THIS PROGRAM CALCULATES THE WEEKLY GROSS AND
  NET PAY OF AN UNDETERMINED NUMBER OF EMPLOYEES*)
```

By entering the directive line

```
WB  ^VOL1>DIR>PAY
```

the contents of the file ^VOL1>DIR>PAY are now:

```
CONST
  FEDTAX   = 0.05;
  STATAXLO = 0.04;
  STATAXHI = 0.07;
```

FUNCTION KEYS

On the general purpose keyboard, the function keys are on the top row and are numbered F1, F2, etc. On data entry and word processing keyboards the keys are inscribed with other text. When a function code is pressed, the Screen Editor performs the action associated with that key. Each Screen Editor function key may have two definitions: one for normal (unshifted) depression, and one for depression with the SHIFT key.

The keyboard design differs, depending on the kind of keyboard you have. Figures 4-3, 4-4, and 4-5 summarize Screen Editor function keys and associates them with their proper function key by keyboard. Labeled keys are discussed later in this section.

	F1	F2	F3	F4	F5	F6
SHIFT				COPY BLOCK	ERASE BLOCK	APPEND LINE
		BACKWARD WORD	FORWARD WORD	MOVE BLOCK	DEFINE BLOCK	

	F7	F8	F9	F10	F11	F12
SHIFT	DELETE BLOCK	WINDOW LEFT	WINDOW RIGHT			
		WINDOW UP	WINDOW DOWN	QUIT		HELP

Figure 4-3. Screen Editor Template for 780X General Purpose Asynchronous Keyboard

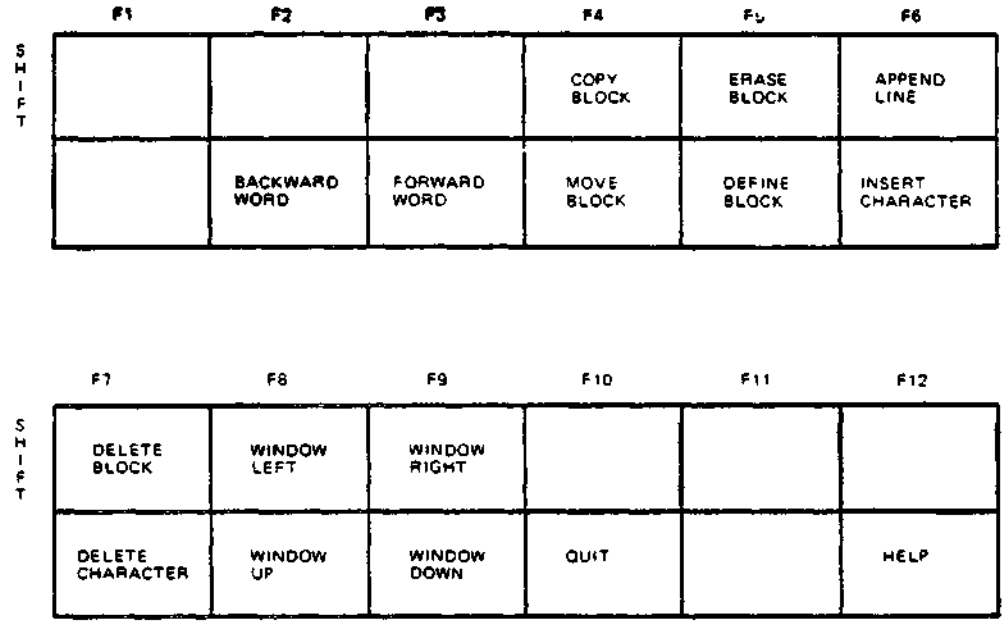


Figure 4-4. Screen Editor Template for 7300 General Purpose Asysnchronous Keyboard

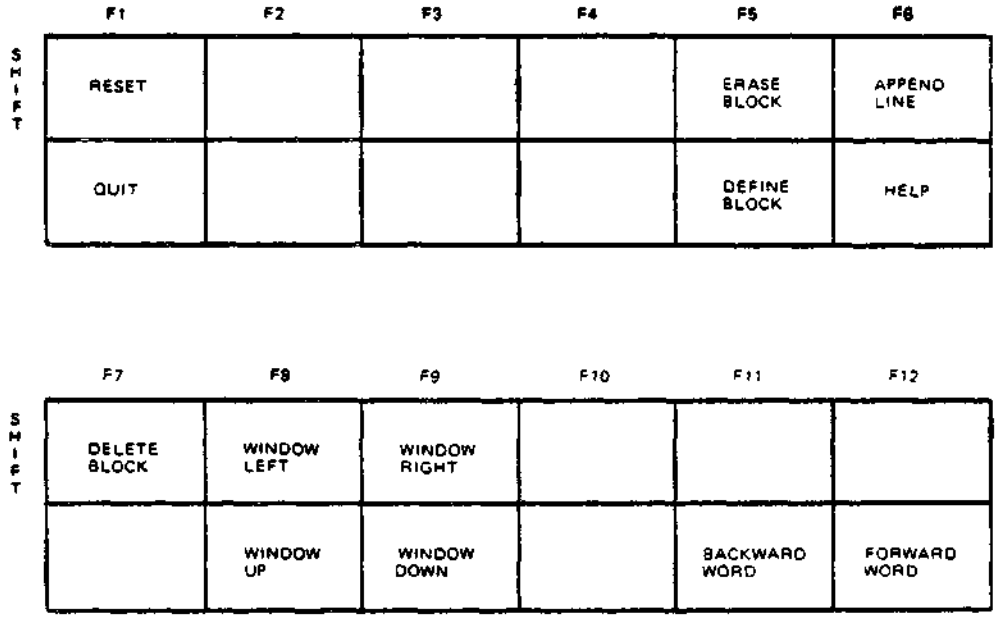
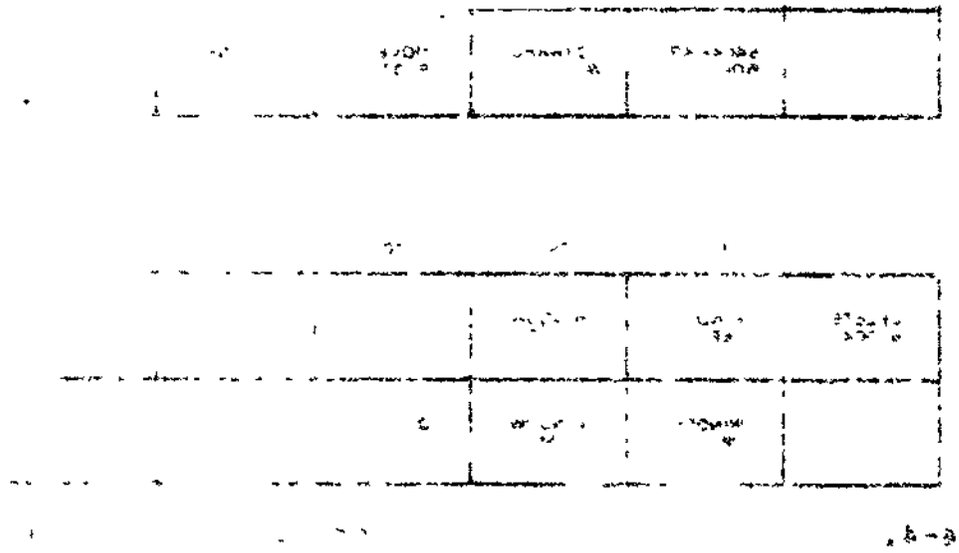
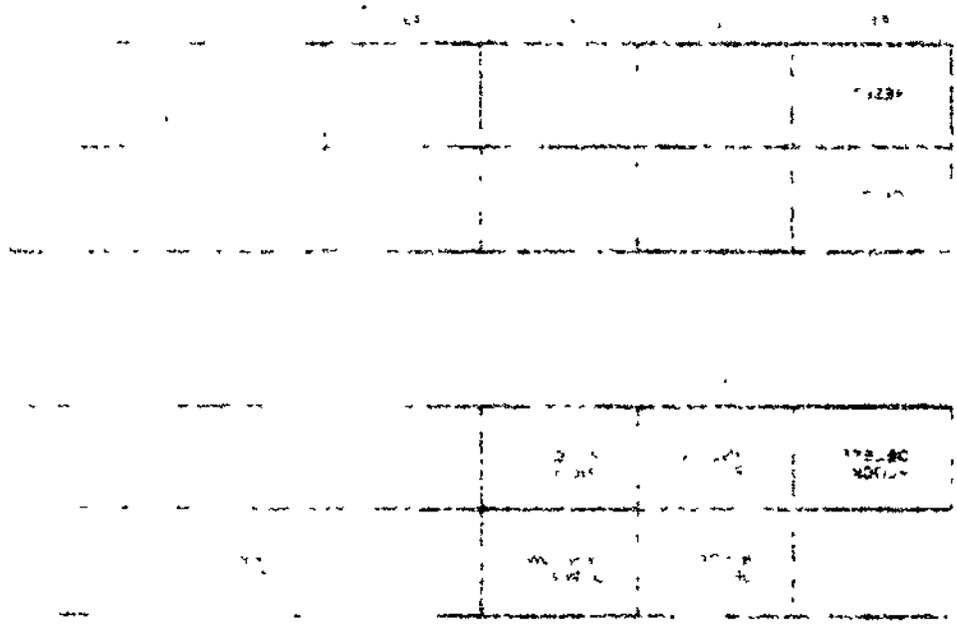


Figure 4-5. Screen Editor Template for 7300 Word Processing Keyboard

Function Key Descriptions

The following function key descriptions are alphabetized by key name for quick reference. Refer to Figures 4-3, 4-4, and 4-5 for locations of function keys by keyboard type.





APPEND LINE

Append a new line after the line on which the cursor is positioned.

The "new" line appears as a blank line on which you may enter text. You must position the cursor in the text region of the screen for the Append Line function key to take effect.

To insert a line before the first line of text (line 1 in the file), position the cursor on the control line, and press the Append Line function key. Enter the new text on the blank line that is displayed.

You cannot insert lines before the Screen Editor control line.

The Append Line function key performs the same actions as the INS LINE labeled key described later in this section.

Example:

```
(*************************************************************
  THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY
```

Press the Append Line function key and the text appears as follows:

```
(*************************************************************

  THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY
```

You may now enter text on the blank line on which the cursor is resting. Be aware that all lines numbers following the new line are incremented by one.

BACKWORD WORD

BACKWARD WORD

Position the cursor from its current position to the first character of the previous word. A word is considered a string of characters delimited by blanks.

If the cursor is positioned in the middle of a word, it is repositioned to the beginning of that word.

If the cursor is positioned on the first word of a line, it is repositioned to the first character of the last word of the previous line.

Example 1:

The current cursor position is:

THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY

Press the Backward Word function key. The cursor is now positioned as follows:

THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY

Example 2:

The current cursor position is:

THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY
OF AN UNDETERMINED NUMBER OF EMPLOYEES.

Press the Backward Word function key. The cursor is now positioned as follows:

THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY
OF AN UNDETERMINED NUMBER OF EMPLOYEES.

BLOCK

Designate the first and last positions of a block of text.

Position the cursor on the character that is the first character of a block of text on which you wish to perform some action. Press the Block function key: this defines the beginning of the block. Position the cursor on the last character of the text you are defining as a block of text. Press the Block function key again: this defines the end of the block. The beginning and end of a block can be defined in any order.

A block is defined by its location, ie., the line and column numbers of its starting and ending points. For example, if you define a block beginning in line 1, column 1, and ending in line 10, column 80, and then you delete lines 5 through 10, the resulting block begins at line 1, column 1, and ends at the "old" line 16 (now the "new" line 10).

The definition of a block remains in effect until you use it, or cancel the block by pressing the Erase Block function key (described later in this section).

You may only define one block at a time.

You must define a block before using any of the block function keys (Erase Block, Delete Block, Copy Block, and Move Block: all are described later in this section), or any of the block directives (Write Block and Copy Block) defined earlier in this section.

Example:

Locate the block of text you wish to define:

THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY

Position the cursor on the character that is the first character of the block you wish to define, in this case "T". Press the Block function key.

Next, position the cursor on the character that is the last character of the block you wish to define, in this case "Y". Press the Block function key.

The block you have just defined is:

THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY

COPY BLOCK

COPY BLOCK

Copy a defined block of text to a specified position.

The text of the defined block is retained in its original position and it is replicated at the current cursor position.

You must have previously defined a block before attempting to copy it. See the Block function key description and "Block Description" for information on defining blocks.

Example:

The previously defined block is:

```
    WEEKLY NET PAY
```

You wish to copy the block into the position specified by the arrow:

```
    **************************
    THIS PROCEDURE CALCULATES
    THE WEEKLY GROSS PAY
    AND <----
    OF AN UNDETERMINED NUMBER OF EMPLOYEES
```

Position the cursor on the second space character on the line beginning with AND. Press the Copy Block key.

The text now reads:

```
    **************************
    THIS PROCEDURE CALCULATES
    THE WEEKLY GROSS PAY
    AND WEEKLY NET PAY
    OF AN UNDETERMINED NUMBER OF EMPLOYEES
```

The block of text remains in its original position and is copied into the new position.

DELETE BLOCK

Delete the previously defined block of text.

You must have previously defined a block before attempting to delete it. See the Block function key description for information on defining blocks.

You do not need to position the cursor on the originally defined block to delete it.

The definition of the block is erased after using Delete Block function key.

See "Block Description" earlier in this section for details on blocks.

Example:

Assume you have previously defined the block designated by the shaded rectangles:

```
THIS IS AN EXAMPLE TO SHOW WHAT HAPPENS
WHEN TEXT IS DELETED BY USING
THE DELETE BLOCK FUNCTION KEY WITH
SPLIT LINES.
```

Press the Delete Block function key. The text now reads:

```
THIS IS AN EXAMPLE WITH
SPLIT LINES.
```

ERASE BLOCK

ERASE BLOCK

Cancel the definition of the previously defined block.

You must have previously defined a block (or have partially defined a block) before attempting to erase it. See the Block function key description for information on defining blocks.

There is no effect on the text within the defined block; only the block definition is cancelled.

You do not need to position the cursor on the originally defined block to erase it.

Example:

Assume you have previously defined the following block:

AND WEEKLY NET PAY

Press the Erase Block function key to cancel the definition of this block.

FORWARD_WORD

Position the cursor on the first character of the next word after the current cursor position.

A word is considered a string of characters delimited by spaces.

If the cursor is on the last word of a line, the cursor is positioned on the first character of the following line.

Example:

The current position of the cursor is:

```
SWT := STATAXLO * GROSSPAY;
```

Press the Forward Word function key. The new cursor position is:

```
SWT := STATAXLO  GROSSPAY;
```

MOVE BLOCK

MOVE BLOCK

Move a previously defined block of text to a specified position.

The block is deleted from its original position.

You must have previously defined a block before attempting to move it. See the Block function key description for information on defining blocks.

The definition of the block is erased when you use this function key.

See "Block Description" earlier in this section for details on blocks.

Example:

The previously defined block is:

```
THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY
```

You wish to move the block into the position specified by the arrow

```
**********************************************************
←
OF AN UNDETERMINED NUMBER OF EMPLOYEES.
```

Position the cursor on the space where you want the first character of the block to appear:

```
**********************************************************
□
OF AN UNDETERMINED NUMBER OF EMPLOYEES.
```

Press the Move Block function key. The text now reads:

```
**********************************************************
THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY
OF AN UNDETERMINED NUMBER OF EMPLOYEES.
```

The block of text is deleted from its original position.

WINDOW DOWN

Move the current window toward the last line of the buffer by the number of lines specified in the scroll amount (see the Scroll Change directive earlier in this section for information on scroll amounts).

Scrolling stops at the last line in the buffer.

If you append lines to the last line in the buffer, the window is automatically moved down to accommodate the appended lines.

Example:

Assume the scroll amount is set at 5 and the text in the window appears as:

```
(10)  IF TOTWRK1 <= STRTIME
(11)  THEN
(12)  SUBGROSS := RATE * TOTWRK1;
(13)  SUBGROSS := SUBGROSS * PENNYRND;
(14)  SUBGROSS := ROUND (SUBGROSS);
(15)  SUBGROSS := SUBGROSS " PENNYRND;
                .
                .
                .
(28)  END
```

Press the Window Down function key. The current window now displays:

```
(15)  SUBGROSS := SUBGROSS " PENNYRND;
                .
                .
                .
(32)  FWT := FWT " PENNYRND;
(33)  WRITE (FWT :9:2);
```

WINDOW LEFT

WINDOW LEFT

Move the current window 40 columns to the left (toward the left margin) of the buffer.

Use this function key when you have entered text beyond column 80 and you wish to view text entered before column 1 of the current window.

If the current window is already at the left margin (displays column 1), no action is taken.

Example:

Assume the following phrase begins in column 81:

ENTERED BEFORE COLUMN 1 OF THE CURRENT WINDOW

Press the Window Left function key. The text that begins 40 characters before column 81 now appears in the window:

BEYOND COLUMN 80 AND YOU WISH TO VIEW TEXT ENTERED BEFORE COL

WINDOW RIGHT

Move the current window 40 columns to the right (toward the right margin) of the buffer.

Use this function key when you have entered text beyond column 80 and you wish to view text entered beyond the last column of the current window.

If you enter text beyond column 80 of the current window, the window automatically moves 40 columns to the right.

If the current window already displays the text at the right margin, no action is taken.

Example:

Assume the following window begins in column 1:

```
USE THIS FUNCTION KEY WHEN YOU HAVE ENTERED TEXT BEYOND
```

Press the Window Right function key. The text that begins 40 characters beyond the first column of the current window is now displayed:

```
RED TEXT BEYOND COLUMN 80 AND YOU WISH TO VIEW TEXT ENTERED B
```

WINDOW UP

WINDOW_UP

Move the current window toward the first line of the buffer by the number of lines specified as the scroll amount (see the Scroll Change directive described earlier in this section for information on scroll amounts).

Scrolling stops at the first line of the buffer.

Example:

Assume the scroll amount is set at 5 and the text in the current window appears as:

```
(15)  SUBGROSS := SUBGROSS " PENNYRND;
                .
                .
                .
(32)  FWT := FWT " PENNYRND;
(33)  WRITE (FWT :9:2);
```

Press the Window Up function key. The current window now displays:

```
(10)  IF TOTWRK1 <= STRTIME
(11)  THEN
(12)  SUBGROSS := RATE * TOTWRK1;
(13)  SUBGROSS := SUBGROSS * PENNYRND;
(14)  SUBGROSS := ROUND (SUBGROSS);
(15)  SUBGROSS := SUBGROSS " PENNYRND;
                .
                .
                .
(28)  END
```

LABELED KEYS

Labeled keys perform the functions described on the key. Depending on the type of terminal you are using you may or may not have these labeled keys. If your terminal does not have the listed labeled key, one of the function keys performs the same action. Function keys are described earlier in this section. Labeled keys are listed alphabetically on the following pages. Those labeled keys that have corresponding function keys are identified in the labeled key description.

NOTE

The labeled keys AUTO LF (Automatic Line Feed) and LOCAL may cause unpredictable results during a screen editing session.

AUTO LF will cause an automatic line feed each time you press carriage return. The Screen Editor then performs another line feed/carriage return. This results in double spacing and loss of the correct cursor position.

LOCAL allows you to move the cursor on the screen without interrupting processing. Using this key causes the loss of the correct cursor position unless the cursor is repositioned to its original location before you exit from local mode.

Use of either of these keys is not recommended during screen editing sessions.

BACKSPACE

BACKSPACE

Move the cursor one position (character) to the left.

If the cursor is positioned on the leftmost column showing on the screen, pressing BACKSPACE has no effect.

If the current window does not display the left margin and the cursor is positioned in column 1 of the current window, pressing the BACKSPACE key automatically moves the window 40 columns to the left.

Example:

Assume the cursor is resting on a line of text as follows:

THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY

Press the BACKSPACE labeled key. The cursor is now positioned as follows:

THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY

CARRIAGE RETURN
CR
ENTER
NEW LINE
RETURN

Move the cursor from its current position to the leftmost column (column 1) of the succeeding line.

Scrolling occurs, bringing the leftmost column into the window, if necesary.

If the cursor is positioned on the last line of the window, the window automatically moves to display the next nine lines in the buffer.

Example:

Assume the cursor is positioned as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

Press CARRIAGE RETURN. The cursor is now positioned as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

CLEAR/RESET

CLEAR/RESET

Cancel the character string just entered on the line.

NOTE

If transmission errors occur or if some other type of data corruption occurs on the screen, pressing the CLEAR/RESET key redisplays the entire screen (the three regions).

Example:

Assume you have just entered the following line and the cursor is positioned as shown:

THISS PROOGRAM CALUCLATE

Rather than using function keys and labeled keys to correct the errors, press the CLEAR/RESET key. The line of text is cancelled, and the cursor is positioned at the beginning of the same line.

CTL TAB
CTRL TAB

CTL TAB
CTRL TAB

Move the cursor back one tab stop from its current position according to the currently defined tab stops.

Example:

With the default tab stops set, the current cursor position is:

THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY

Press the CTL TAB sequence. The cursor is now positioned as follows:

THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY

CURSOR DOWN (↓)

Move the cursor down one line (row) leaving the cursor in the same column.

If the cursor is positioned on the last line of the window, pressing the Cursor Down labeled key positions the cursor on the first line of the window. The column is unchanged.

Example:

Assume the cursor is positioned as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

Press the Cursor Down labeled key. The cursor is now positioned as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

CURSOR LEFT

CURSOR LEFT (←)

Move the cursor one position (character) to the left.

If the cursor is positioned on the leftmost column on the screen, pressing the Cursor Left labeled key moves the cursor to the rightmost character on the screen of the preceding line.

If the cursor is positioned in the leftmost column of line 1 of the current window, pressing the Cursor Left labeled key moves the cursor to the rightmost column of the last line displayed in the window.

Example:

Assume the cursor is positioned as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

Press the Cursor Left labeled key. The cursor is now positioned as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

CURSOR RIGHT(→)

Move the cursor one position (character) to the right of its current position.

If the cursor is positioned on the rightmost column on the screen, pressing the Cursor Right labeled key moves the cursor to the leftmost character on the screen of the succeeding line.

If the cursor is in the rightmost column of the last line in the window, pressing the Cursor Right key positions the cursor in column 1 of the first line in the window.

Example:

Assume the cursor is positioned as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

Press the Cursor Right labeled key. The cursor is now positioned as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

CURSOR UP

CURSOR UP (↑)

Move the cursor up one line leaving the cursor in the same column.

If the cursor is positioned on the first line of the window, pressing the Cursor Up key positions the cursor on the last line of the window. The column is unchanged.

If the cursor is positioned in the Directive Region of the screen, pressing the Cursor Up key returns the cursor to the position in which it was before you pressed the HOME key. (The HOME key is described later in this section.)

Example:

Assume the cursor is positioned as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

Press the Cursor Up labeled key. The cursor is now positioned as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

DEL CHAR

Delete the character on which the cursor is positioned.

To delete a character, position the cursor on the unwanted character and press the DEL CHAR labeled key. The line that contained the deleted character is now one character shorter in length. All characters following the deleted character are moved one position to the left so that the first character following the deleted character is adjacent to the character preceding the deleted character.

The REPEAT key may be used to delete multiple characters quickly.

The DEL CHAR labeled key performs the same action as the Delete Character function key described earlier in this section.

Example:

Assume the text reads as follows:

THIS PROOGRAM CALCULATES THE WEEKLY GROSS AND NET PAY

Position the cursor on the unwanted character:

THIS PROOGRAM CALCULATES THE WEEKLY GROSS AND NET PAY

Press the DEL CHAR labeled key. The text now reads:

THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY

DEL LINE

DEL LINE

Delete the line on which the cursor is resting.

After the line is deleted, the cursor is positioned in the same column but on the line that immediately followed the deleted line.

Example:

Assume the cursor is resting on the line as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SIB
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

Press the DEL LINE labeled key. The text now reads as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

ERASE EOL

Delete any text from (and including) the current cursor position to the end of the line.

To erase characters from the current cursor position to the end of the line, position the cursor on the first character of the text you want to erase, and press the ERASE EOL labeled key.

The cursor is positioned on the same line in the same column in which it was positioned when you pressed the ERASE EOL key.

Example:

Assume the text reads as follows and you have positioned the cursor as shown:

```
THIS PROGRAM CALCULATES THE GROSS AND NET PAY OF AN OF
AN UNDETERMINED NUMBER OF EMPLOYEES.
```

Press the ERASE EOL labeled key. The text now reads:

```
THIS PROGRAM CALCULATES THE GROSS AND NET PAY OF .
AN UNDETERMINED NUMBER OF EMPLOYEES.
```

HOME

HOME

Move the cursor to the directive input line of the screen.

When you press the HOME key, the system "remembers" the cursor position before it is positioned to the directive input line. The "remembered" cursor position is the cursor position used for any cursor position related directives.

Example:

Assume the cursor is positioned as follows:

THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY

Press the HOME labeled key. The cursor position is remembered and is repositioned to the directive input line as follows:

DIRECTIVE:$

INS CHAR

Insert a number of characters to the left of some point in a buffer.

To insert characters, position the cursor to the character that should immediately follow the character(s) that you are inserting. Press the INS CHAR labeled key. The flag "INSERT" appears in the status region to alert you that you are in insert mode. Enter the new characters.

Every character to the right of the current cursor position (including the character on which the cursor is resting) will be moved one space to the right for the insertion of each insert character you enter.

To end character insertion, press the INS CHAR labeled key a second time. The "INSERT" flag is removed from the status region. Any characters you enter now will write over the existing text.

To insert characters at the end of a line, position the cursor to the location you wish to begin the insert and simply enter the characters. It is not necessary to use the INS CHAR labeled key to enter characters at the end of a line.

Pressing the RETURN or LINE FEED key while in insert character mode creates a new line. The characters from the cursor position to the end of the line are placed on that new line. If you press RETURN, the repositioned characters will start in column 1 of the new line. If you press LINE FEED, the new line is blank filled up to the cursor position. The characters begin at the cursor position on the new line. An example of pressing RETURN in insert character mode is:

```
THIS IS A SPLIT LINE.
```

Press RETURN. The result is:

```
THIS IS A
SPLIT LINE.
```

An example of pressing LINE FEED in insert character mode is:

```
THIS IS A SPLIT LINE.
```

Press LINE FEED. The result is::

```
THIS IS A
: SPLIT LINE.
```

The REPEAT key can be used to insert multiple characters of the same value.

If inserting characters in a line causes the maximum line length to be exceeded, those characters in the rightmost columns of the line are lost.

Example:

Position the cursor to the right of the position where you want to insert characters. For example:

```
NETPAY := GROSSPAY - FICA;
```

Press the INS CHAR labeled key. Enter the characters you wish to insert. The new line of text (after insertion) now reads:

```
NETPAY := GROSSPAY - FWT - SWT - FICA;
```

Press the INS CHAR labeled key again to leave insert mode.

LINE FEED

Move the cursor down one line from its current position and leave the cursor in the same column.

If the cursor is positioned on the last line of the window, pressing the Line Feed labeled key moves the window down 9 lines.

Example:

Assume the cursor is positioned as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

Press the LINE FEED labeled key. The cursor is now positioned as follows:

```
IF TOTWRK1 <= STRTIME
THEN
SUBGROSS := RATE * TOTWRK1;
SUBGROSS := SUBGROSS * PENNYRND;
```

TAB
CTL I
CTRL I

TAB
CTL_I
CTRL_I

Move the cursor from its current position to the next tab stop to the right on that line according to the current tab stop definition.

The key sequences CTL I or CTRL I perform the same actions as the TAB key.

Example:

With the default tab stops set, the current cursor position is:

```
THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY
```

Press the TAB labeled key. The cursor is now positioned as follows:

```
THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY
```

TAB CLR

Cancel (clear) the tab stop definition in a specified column.

Position the cursor in the column containing the tab stop and press the TAB CLR labeled key.

Example:

Assume you have set tab stops as shown by the designated cursor positions:

THIS PROGRAM CALCULATES THE WEEKLY GROSS AND NET PAY

Positioning the cursor in the column containing the tab stop you wish to cancel (say, the first cursor position) cancels the definition of that tab stop.

TAB SET

TAB SET

Set a tab stop in a specified column.

Position the cursor in the desired column and press the TAB SET labeled key.

Example:

Assume this line begins in column 1. At each designated cursor position, you have pressed the TAB SET labeled key.

THIS PROGRAM CALCUEATES THE WEEKLY GROSS AND NET PAY

Tab stops are now set at columns 9 and 19.

Section 5
LINE EDITOR

This section describes Line Editor functions and the Line Editor directive set. For procedural information on using Line Editor directives to create and edit files, see Appendix A; sample user dialogs with the Line Editor are provided.

OVERVIEW

The Line Editor creates and/or alters character text that constitutes files; the files usually are source unit files. The statements in a source unit file can be written in FORTRAN, Pascal, COBOL, BASIC, or Assembly language. Throughout this section, it is assumed that source unit files are being edited.

Editing is controlled by directives entered to the Line Editor through the device specified in the in_path argument of the Enter Batch Request (EBR) or Enter Group Request (EGR) command. This device can be reassigned in the command that loads the Line Editor.

All editing is done in a temporary work area called the current buffer. When the Line Editor is invoked, the Line Editor creates a current buffer. To save Line Editor output, you must write the source unit contents of the current buffer to a file.

During a single execution of the Line Editor, the Line Editor can operate in input and/or edit mode. In input mode, you can create a source unit and/or add one or more specified lines to an existing source unit. In edit mode, you can locate and change single characters, words, or a string of characters, read the contents of a file into the current buffer so that the line(s) can be edited, write lines from the current buffer to a file, and terminate execution of the Line Editor.

NOTES

1. During a single execution of the Line Editor, you can create and/or change any number of files. You must delete the contents of the current buffer before you begin to edit another file, unless you want that file to comprise the same information that was in the previous file(s).

2. At any time during execution of the Line Editor you can request a message that will indicate whether input or edit mode is in effect. Each time !? is entered, the following message is issued:

   {INPUT} MODE
   {EDIT }

Line Editor processing can be interrupted by either:

- Pressing the QUIT, INTERRUPT, or BREAK key on your terminal

- Entering Δ CΔBgroup-id on the operator terminal, where group-id is the two-character group identification code associated with the group containing the task to be interrupted.

A **BREAK** message appears on your terminal when the system interrupts the Line Editor. If the Program Interrupt (PI) command is entered, output is suppressed and the task returns to directive input level. See the Commands manual for a detailed description of the break function.

Each Line Editor directive's name and function is listed in Table 5-1. They are described in detail in "Input Mode Description and Directives", "Edit Mode Description and Directives", and "Advanced Usage of the Line Editor." Directives described in the input and edit mode subsections operate within the current buffer.

LINE EDITOR SUFFIX CONVENTIONS

During program preparation, it is convenient to identify output file(s) with the name of the input file.

When you create a source unit, you should append the appropriate suffix identification character to the name of the file that will contain the source unit. The suffix designates the type of text that constitutes the source unit. The suffix must be .C for COBOL programs, .F for FORTRAN programs, .B for BASIC programs, .PS for Pascal programs, and .A for Assembly language programs.

When you specify the file names of Line Editor input and output files (in Line Editor directives), the editor requires that you designate the complete file name, including the suffix that denotes the contents of the file (.C for COBOL, .F for FORTRAN, .B for BASIC, .PS for Pascal programs, and .A for Assembly language programs). The Line Editor does not append a suffix to its input and output files.

LINE EDITOR DIRECTIVE FORMAT CONVENTIONS

Most Line Editor directives consist of only a directive name, a directive name preceded by one or two addresses, or a directive name optionally preceded by one or two addresses and followed by text and termination escape characters (!F) that designate the end of the directive and cause the Line Editor to switch from input mode to edit mode. These formats are illustrated here. Note that if a directive includes text, the text may be specified beginning immediately after the directive name (see Format 4) or beginning on the next line (see Format 5).

FORMAT 1:

```
dirname
```

FORMAT 2:

```
adr  dirname
```

FORMAT 3:

$$adr_1 \left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \ dirname$$

FORMAT 4:

$$\left[adr_1 \left[\left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2 \right] \right] dirname[text]!F$$

FORMAT 5:

$$\left[adr_1\left[\left\{\begin{matrix};\\,\end{matrix}\right\}adr_2\right]\right]dirname$$

[text]

.
.
.

!F

ARGUMENTS:

dirname

Valid Line Editor directive.

adr adr

Valid addresses for the current buffer.

text

Any text.

NOTES

1. Spaces are not permitted, except in the following circumstances:

   a. Spaces are permitted in expressions constituting addresses.

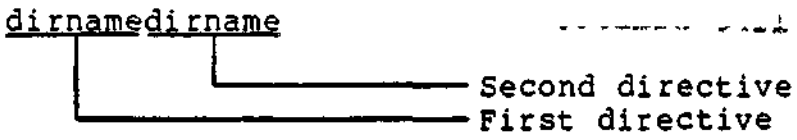
   b. A space is permitted after the Execute, Read, and Write directive names (these directives are described later in this section).

2. One or two addresses may be specified without a directive name; if no directive name is specified, the last (or only) addressed line will be printed (see "Print Directive").

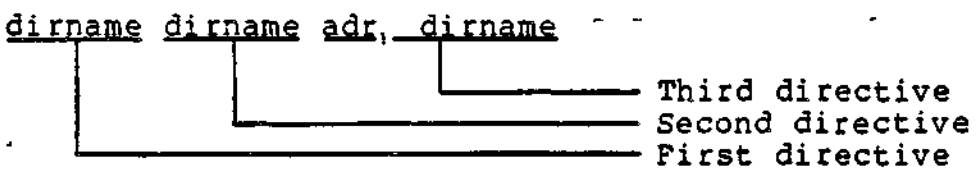
When a single address is specified, the Line Editor locates the specified line in the current buffer. When two addresses are specified within a single directive, the Line Editor locates a specified series of lines in the current buffer; the lines that are located depend on whether the addresses are separated by a comma or a semicolon (see "Referencing a Series of Lines"). If a Line Editor directive format designates that either a single address or a pair of addresses may be entered, you can enter that directive and omit one or both addresses; their default value(s) will be used. Address default values are described later in this section under each directive's argument descriptions.

Multiple Line Editor directives can be entered on a single line; it is not necessary to separate each directive with a delimiter, but one or more spaces can be specified, as illustrated below:

Directives not separated by delimiters:



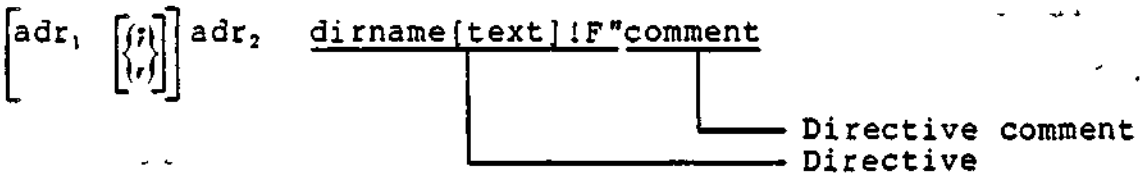
Directives separated by delimiters:



A comment can be included at the end of a directive line (i.e., at the end of the last or only directive); the comment must be preceded by a quotation mark ("), as illustrated:

```
adr  dirname dirname"comment
```

To include a comment after an input mode directive, specify the comment after the terminator !F; otherwise, the comment is included as text.



If a terminal is the directive input device, press RETURN at the end of each line.

Methods of Specifying Addresses

Each address can be specified by one of the following methods or by a combination of these methods:

- Number of line
- Position of line relative to the "current" line
- Contents of line.

DESIGNATING A LINE NUMBER AS AN ADDRESS

Each line in the current buffer can be located by a decimal number that indicates the current position of the line within the buffer.* The first line in the buffer is line 1; subsequent lines are numbered sequentially in ascending order. Multiple decimal numbers separated by plus or minus signs can be specified to represent a line number.

Example:

```
10
5+5
```

Each of the expressions above request line number 10. The last line can be referenced by its line number or by the character $.

If an address designates a line that is not in the current buffer, an error message is issued.

Line Editor directives may cause lines to be added to or deleted from the current buffer. Each time this occurs, all succeeding lines are renumbered. For example, if line 15 is deleted, line 16 becomes 15, and each subsequent line number is decremented by 1.

DESIGNATING THE POSITION OF A LINE RELATIVE TO THE "CURRENT" LINE AS AN ADDRESS

Most Line Editor directives affect either the current line or a line a designated number of positions from the current line. If the last Line Editor directive entered was an Input directive (i.e., input mode was in effect), the current line is the last line added or read by the Line Editor (regardless of whether the condition specified in the directive was met). If the last Line Editor directive entered was an Edit directive (edit mode was in effect), the current line is the last line of text edited. The current line can be located by specifying a period (.).

NOTE

If you do not know which line is the current line, you can obtain a display of the line number of the current line by specifying the Print Line Number directive, which is described under "Advanced Usage of the Line Editor" later in this section.

*To determine the line number of a specified line in the current buffer, enter the Print Line Number directive; to determine the line number and contents of specified line(s) in the buffer, enter the Print With Line Number directive. (These directives are described under "Advanced Usage of the Line Editor", later in this section.)

You can locate lines relative to the current line by specifying an address that consists of a period followed by one or more signed decimal numbers. For example, the address .+1 specifies the line immediately following the current line, the address .-1 specifies the line immediately preceding the current line, and .+5+5-3 specifies the seventh line after the current line.

When specifying an increment to the current line number, you can omit the plus (+) sign; e.g., .5 is interpreted as .+5. When specifying a decrement to the current line number, you can omit the period; e.g., -3 is interpreted as .-3, and .5+5-3 is interpreted as .+7.

DESIGNATING CONTENTS OF LINE AS AN ADDRESS

You can designate that the Line Editor locate the first line that contains a specified character or a specified sequence of characters by designating those characters in an expression as an address. An expression comprises one or more ASCII characters, which must be delimited by slashes (e.g., /ASCII characters/).

The Line Editor will search the lines in the current buffer until it finds the *first* occurrence of the specified expression; unless specified otherwise,* the expression can be in any position within the line. The Line Editor searches from the line immediately following the current line (i.e., .+1) through the last line in the buffer; if a line containing the specified expression is not found, the Line Editor then searches line 1 to the current line. In the directive format:

```
/BBB/dirname
```

the address is the expression BBB. The Line Editor searches as many lines as necessary for the first occurrence of BBB. The contents of the source unit being searched are listed below. (The numbers within parentheses represent line numbers.)

```
(1) AAA
(2) BBB
(3) CCC (current line)
(4) BBB
```

The specified directive causes the Line Editor to locate line number 4, since this is the first line after the current line that contains the expression BBB.

*If a circumflex (^) is designated as the first character of the expression, the expression must be the first expression on the line; if $ is designated as the last character of the expression, the expression must be the last expression on the line. Use of these special characters is described in the following paragraphs.

When the following ASCII characters are included in expressions, they have special meanings:

Character	Description
*	Requests expressions that contain any number (or none) of the immediately preceding character(s).
^	When designated as the first character of an expression, requests lines that begin with the specified expression (excluding the character ^).
$	When specified as the last character of an expression, requests lines that end with the specified expression (excluding the character $).
.	Can be any character on any line; specify one period per character (e.g., .. means any two characters on any line).
&	Can be used in the string expression of a Substitute directive to indicate that the strings of characters preceding and following & are to be concatenated to the target string of the search. See the description of the substitute directive later in this section.
line feed (hex 0A) (see Note 3)	The occurrence of a line feed in the string expression determines the point in the resulting line at which the line is to be split into two lines. See the Substitute directive for further details.

NOTES

1. The special meanings of the above characters, / (which delimits an expression) and !? (which causes display of the mode currently in effect), can be removed by preceding the special character with !C. For example, !C!? causes !? to be interpreted as text rather than as a request for display of the mode that is in effect.

2. The characters . and $ can be specified as line numbers or as special characters in expressions; the Line Editor can interpret their meaning from the way they are used.

3. For the Line Editor, two hexadecimal characters can be interpreted as one ASCII byte by using the escape sequence !Hxx, where xx are the two hex characters. However, this feature must be used with care since some of the hexadecimal characters may be confused with control or special characters in ASCII strings. The following is a list of the hexadecimal characters whose use is restricted:

   0A is the line feed character; in a string expression, it is interpreted as a request for advancement to a new line.

   2E and AE in a regular expression are treated as ".".

   26 and A6 in a string expression are treated as "&".

   2A and AA in a regular expression are treated as "*".

   24 at the end of a regular expression is interpreted as "end-of-line ($)".

   5E at the beginning of a regular expression becomes "beginning-of-line (^)".

   Rather than attempting to substitute in an expression using the characters above, execute a Change directive, reentering the line using hexadecimal and ASCII characters for the entire line.

Following are some examples of expressions specified as addresses in Line Editor directives. Following each expression is a description of the line/character(s) in the current buffer for which the Line Editor will search. In each case, the Line Editor searches the lines sequentially, starting with the line immediately following the current line to the end of the file, and then from line one through the current line.

Expression	Description
/A/	Locates the first line that contains the expression A in any position in that line.
/ABC/	Locates the first line that contains the expression ABC in any position on that line.
/AB*C/	Locates the first line that contains the expression AC or A followed by any number of B's and a C.

Expression	Description
/IN..TO/	Locates a line that contains IN and TO separated by any two characters.
/IN.*TO/	Locates a line that contains IN and TO, in that order, with any or no characters between those two words.
/^ABC/	Locates a line that begins with the expression ABC.
/ABC$/	Locates a line that ends with the expression ABC.
/ABC!C$/	Locates a line that contains the expression ABC$. ABC$ can be in any character positions, since the character $ was preceded by !C.
/^ABC.*DEF$/	Locates a line that begins with ABC and ends with DEF; there may be any number of characters between ABC and DEF.
/.*/	Locates any line.

The Line Editor remembers the last specified expression. That expression can be reinvoked in a subsequent Line Editor directive by specifying a null expression (e.g., //).

Examples:

/ABC/dirname

Expression ABC specified as address

2dirname

Second line in buffer specified as address

//dirname

Specifies ABC as an address, since ABC was last specified expression

An address can be specified as an expression followed by one or more signed decimal integers. Each of the following three expressions requests the second line after the line that contains ABC.

```
/ABC/2
/ABC/+2
/ABC/+5-3
```

An address can be formed by combining any of these methods. If a compound address contains a line number, the line number must be the first element of the address.

The first element of the compound address determines the starting location from which the Line Editor will search for the designated expression. If the first element is a line number, the Line Editor searches for the expression starting with the line that immediately follows the specified line number. (Ordinarily, the Line Editor searches starting with the line that immediately follows the current line.)

Example 1:

```
10/ABC/
```

The Line Editor searches the lines in the current buffer for the characters ABC, starting with line 11.

Example 2:

```
.-8/ABC/
```

The Line Editor searches the lines in the current buffer for the characters ABC, starting eight lines before the current line.

Example 3:

```
/ABC//DEF/
```

The Line Editor searches for the first line containing DEF that occurs <u>after</u> the first line containing ABC.

Each expression in a compound address can be followed by a signed decimal integer.

Example 4:

```
/ABC/-10/DEF/5
```

The Line Editor searches for the first occurrence of the character string DEF that is within 10 lines before the first line that contains ABC. After DEF is found, the current line is the fifth line after the line containing the match for DEF.

Referencing a Series of Lines

A Line Editor directive that permits two addresses to be specified causes the Line Editor to locate a series of lines in the buffer. The addresses can be separated by a comma or a semicolon. If the second address is relative to the current line (plus or minus), both the addresses and the plus or minus sign determine which lines will be located by the Line Editor; otherwise, only the addresses are relevant.

If the addresses are separated by a comma, the Line Editor locates the line at the first address through the line at the second address, inclusive. The current line remains unchanged until the directive is executed; the current line then becomes the line specified by the second address.

If the addresses are separated by a semicolon, the line located by the first address becomes the current line and the value of the second address is calculated.

Example 1:

```
1,5dirname
```

These addresses specify lines 1 through 5, inclusive. After the directive is executed, line 5 becomes the current line.

Example 2:

```
1,$dirname
```

These addresses specify line 1 through the last line in the buffer, inclusive. After the directive is executed, the last line becomes the current line.

Example 3:

```
.1,/ABC/
```

These addresses specify the line immediately following the current line through the first line that contains ABC. The first line that contains ABC then becomes the current line.

Example 4:

```
.1,.2dirname
```

The contents of a sample source unit are following. The numbers within parentheses represent line numbers.

```
(1) ABC
(2) DEF (current line)
(3) GHI
(4) ABC
(5) XYZ
(6) ABC
```

These addresses specify the line immediately following the current line through the second line after the current line. The Line Editor locates lines 3 and 4. Line 4 becomes the current line.

Example 5:

```
.1;2dirname
```

These addresses are the same as those in Example 4, but they are separated by a semicolon. If the contents of the sample source unit are the same as in Example 4, this directive causes the Line Editor to locate lines 3, 4, and 5. This first address specifies the line immediately after the current line, i.e., line 3. Line 3 then becomes the current line. The second address specifies that the Line Editor locate through the second line after the (new) current line, i.e., lines 4 and 5.

The same series of lines can be requested by specifying their addresses in more than one way, using different delimiters.

Example 6:

```
/ABC/,/ABC/+3dirname
/ABC/;.+3dirname
```

The contents of a sample source unit follows. The numbers within parentheses represent line numbers.

```
(1) ABC
(2) DDD (current line)
(3) EEE
(4) FFF
(5) GGG
(6) HHH
```

The first series of addresses specifies that the Line Editor locate the first line that contains ABC (line 1) through the third line after that line (lines 2, 3, and 4). Line 4 becomes the current line.

The second series of addresses specifies that the Line Editor locate the first line that contains ABC (line 1), make that line the current line, and then reference three lines from the "new" current line (lines 2, 3, and 4). Line 4 becomes the current line.

Loading the Line Editor

The Line Editor command loads the Line Editor. Upon loading, a message indicating the current Line Editor release number is sent to the error-out file.

To load the Line Editor, enter the ED command.

FORMAT:

ED[?SILENT] [ctl_arg]

ARGUMENTS:

[?SILENT]

Optional entry point that suppresses the welcome message.

[ctl_arg]

None or any number of the following control arguments may be entered:

-IN path

File from which Line Editor directives are to be read. -IN path in the Line Editor command line results in the user-in file being changed to "path" or the contents of "path" being copied to buffer (EXEC). Execution starts with the first line of (EXEC). Default: Directives are obtained from the current user-in file.

{-LINE_LEN nn}
{-LL nn }

Alter the line length to be acted upon by the Line Editor and can be any value from 20 to 256. Default: nn equals 80.

{-PROMPT}
{-PT }

Print the prompt characters E? (in edit mode) or I? (in input mode) on the user-in file upon completion of the previous Line Editor directive; no carriage return follows. If the user-in is other than a terminal-like device, this argument is ignored.

{-NO_BLANK_SUPPRESS}
{-NBS }

No blank suppression; i.e., the Line Editor does not suppress trailing blanks on the input line (for one invocation only). Subsequent invocations without -NBS will suppress trailing blanks.

{-FILE_SIZE nn}
{-FS nn }

Alter the initial size of the work file to the size in the user-supplied value of nn, where nn is a decimal integer comprising up to four characters and designates the number of 256-byte control intervals. If an output file is created, it is initialized to the same size.

Default: 4.

{-ARGS strings}
{-ARG strings }

Up to nine character strings that are numbered sequentially and may be passed to the Line Editor in the "Change Origin of Text During Edit Mode" (!B) Line Editor directive. Each argument following the -ARG keyword is copied to buffer (ARGn). n denotes the position of the argument following the -ARG and can be any value from one through nine. If specified, this argument and its strings must be entered last.

{-SAFE name}
{-SF name }

Permanent work files called name.EDWK1 and name.EDWK2 contain the latest copy of the current buffer. Name can be from one to six characters. Abnormal termination causes the work files to be closed in their current state and saved for later use, and normal termination releases them. To reuse the work files, invoke the Line Editor without -SAFE or with -SAFE and a different name. Default: Work files are temporary files and are released under all conditions.

{-SIZE nn}
{-SZ nn }

Define the number of 1024-byte words to be used for dynamic storage in memory. nn can be any value from 1 to 64. The formula for calculating the number of lines possible is (2*nnK/LL+6)-3, where K is 1024 and LL is the line length value (or 80 by default).

Default: 1.

SUMMARY OF LINE EDITOR DIRECTIVES AND ESCAPE SEQUENCES

Table 5-1 lists each Line Editor directive name and escape sequence, summarizes its function, and designates the topic in this section under which the directive/escape sequence is described. The topics refer to the following level headings:

- "Input Mode Description and Directives" (input mode)
- "Edit Mode Description and Directives" (edit mode)
- "Advanced Usage of the Line Editor"

  -"General Advanced Line Editor Directives" (advanced usage -- general)

  -"Auxiliary Buffer Directives and Escape Sequences" (advanced usage -- auxiliary buffers)

  -"Line Editor Debugging Directives" (advanced usage -- debugging)

  -"Line Editor Programming Directives" (advanced usage -- programming).

Table 5-1. Summary of Line Editor Directives and Escape Sequences

Directive Name/Escape Sequence	Function	Topic Under Which Described
A	Add line(s) after specified address.	Append directive (input mode)
B	Make specified auxiliary buffer the current buffer.	Change Buffer directive (advanced usage -- auxiliary buffers)
C	Delete specified line(s) and insert other line(s).	Change directive (input mode)
D	Delete specified line(s) from current buffer.	Delete directive (edit mode)
E	Execute command other than Line Editor without exiting from the Line Editor.	Execute directive (advanced usage -- general)
G	Search for specified line(s) that contain specified character string.	Global directive (advanced usage -- general)

Table 5-1 (cont). Summary of Line Editor Directives and Escape Sequences

Directive Name/Escape Sequence	Function	Topic Under Which Described
I	Add line(s) before a specified address.	Insert directive (input mode).
K	Copy line(s) in current buffer to specified auxiliary buffer. Do not delete lines from current buffer. Overlay existing line(s) in auxiliary buffer.	Copy directive (advanced usage -- auxiliary buffers)
L	Send line feed to the user-out file.	Line Feed directive (advanced usage -- general)
M	Move line(s) from current buffer to specified auxiliary buffer; delete the lines from current buffer and overlay existing line(s) in auxiliary buffer.	Move directive (advanced usage -- auxiliary buffers)
N	Designate different line as the current line.	New Current Line directive (advanced usage -- general)
P	Print specified line(s) in current buffer.	Print directive (edit mode)
Q	Conditionally terminate execution of Line Editor.	Quit directive (edit mode)
R	Read text from file to current buffer.	Read directive (edit mode)
S	Substitute character string with another character string.	Substitute directive (edit mode)
T	Display line of text on user-out file. Subsequent input/output will be on the next line.	Type directive (advanced usage -- programming)
U	Convert specified uppercase expression to lowercase.	Lowercase directive (advanced usage -- general)

Table 5-1 (cont). Summary of Line Editor Directives and Escape Sequences

Directive Name/Escape Sequence	Function	Topic Under Which Described
V	Search for specified line(s) that do not contain specified character string.	Exclude directive (advanced usage -- general)
W	Write specified line(s) from current buffer to specified file.	Write directive (edit mode)
X	Request status of auxiliary buffers.	Buffer status directive (advanced usage -- auxiliary buffers)
ZDUMP	Print contents of specified line(s).	Hexadecimal dump directive (advanced usage -- debugging)
ZREGEXP	Display last specified expression.	ZREGEXP directive (advanced usage -- debugging)
ZTRACE	Display each directive line before it is executed.	ZTRACE directive (advanced usage -- debugging)
!B	Change origin of text to specified auxiliary buffer or execute specified auxiliary buffer.	Change origin of text during input/edit mode (advanced usage -- auxiliary buffers)
!C	Remove meaning of following special character.	
!F	Terminate an input mode directive.	(Input mode)
!Hxx	Interpret two following hexadecimal characters as one ASCII byte.	
!K	Copy line(s) in current buffer to specified auxiliary buffer; do not delete existing line(s) in auxiliary buffer.	Copy-append directive (advanced usage -- auxiliary buffers)

Table 5-1 (cont). Summary of Line Editor Directives and Escape Sequences

Directive Name/Escape Sequence	Function	Topic Under Which Described
!L	Send line feed to the error-out file.	Line feed directive (advanced usage -- general)
!M	Move line(s) from current buffer to specified auxiliary buffer; delete the line(s) from current buffer and append them to existing line(s) in auxiliary buffer.	Move-append directive (advanced usage -- auxiliary buffers)
!P	Type line number and contents of specified line(s) in current buffer.	Print With Line Number directive (advanced usage -- general)
!Q	Unconditionally terminate execution of Line Editor.	Quit directive (edit mode)
!R	Accept single line from terminal.	Accept Single Line from Terminal directive (advanced usage -- auxiliary buffers)
!T	Display line of text on user-out file; subsequent input/output will be on the same line.	Type directive (advanced usage -- programming)
!U	Convert specified lowercase expression to uppercase.	Uppercase directive (advanced usage -- general)
!?	Cause message indicating whether input or edit mode is in effect.	
#	If current buffer contains data, execute specified directive(s).	If Data directive (advanced usage -- programming)
address #	If current line is specified line, execute specified directive(s).	If Line directive (advanced usage -- programming)

Table 5-1 (cont). Summary of Line Editor Directives and Escape Sequences

Directive Name/Escape Sequence	Function	Topic Under Which Described
^S	Replace each occurrence of specified character string with another character string.	Substitute directive (edit mode)
addresses #	If current line is within specified lines, execute specified directive(s).	If Range directive (advanced usage -- programming)
^B	Release a specified auxiliary buffer.	Destroy directive (advanced usage -- auxiliary buffers)
^ #	If current buffer does not contain data, execute specified directive(s).	If Empty directive (advanced usage -- programming)
address ^ #	If current line is not specified line, execute specified directive(s).	If Not Line directive (advanced usage -- programming)
addresses^#	If current line is not within specified lines, execute specified directive(s).	If Not Range directive (advanced usage -- programming)
*	If specified expression is within specified lines, execute specified directive(s).	Search directive (advanced usage -- programming)
^*	If specified expression is not within specified lines, execute specified directive(s).	Search Not directive (advanced usage -- programming)
:	Define location to which Line Editor can be directed for subsequent directive(s).	Label directive (advanced usage -- programming)
=	Type line number of specified line in current buffer.	Print Line Number directive (advanced usage -- general)

Table 5-1 (cont). Summary of Line Editor Directives and Escape Sequences

Directive Name/Escape Sequence	Function	Topic Under Which Described
>	Accept subsequent directive(s) from specified location in current buffer or interactively.	Go To directive (advanced usage -- programming)
?	If specified line is in current buffer, execute specified directive(s).	Address Prefix directive (advanced usage -- programming)
*	Annotate Line Editor files.	Comment directive (advanced usage -- programming)

CREATING A SOURCE UNIT

To create a source unit, perform the following steps listed. Input mode directives are described under "Input Mode Description and Directives". Each of the directives referenced is described under "Edit Mode Description and Directives".

1. Change the working directory to a user volume by specifying the Change Working Directory (CWD) command (see the Commands manual).

2. Load the Line Editor. (See "Loading the Line Editor" earlier in this section.)

3. If there already are lines in the current buffer, clear the buffer by specifying: 1,$D.

4. Enter the appropriate Input directive and text to be included.

5. Make changes, if necessary, by entering the appropriate Input and/or Edit directive(s).

6. Write the contents of the current buffer to a file by using the Write directive.

7. Exit from the Line Editor by entering the Quit directive (Optional).

CHANGING AN EXISTING SOURCE UNIT

To change an existing source unit, perform the following steps. Input mode directives are described under "Input Mode Description and Directives". Each of the directives referenced is described under "Edit Mode Description and Directives" later in this section.

1. Change the working directory to a user volume by specifying the Change Working Directory (CWD) command (see the Commands manual).
2. Load the Line Editor, if it is not already loaded. (See "Loading the Line Editor" earlier in this section.)
3. If there already are lines in the current buffer, delete unwanted lines by specifying the Delete directive.
4. Use the Read directive to read into the current buffer the source unit to be edited.
5. Enter the appropriate Edit and/or Input directive(s).
6. Write the contents of the current buffer to the file from which the lines were read or to a different file by using the Write directive.
7. Exit from the Line Editor by entering the Quit directive (Optional).

INPUT MODE DESCRIPTION AND DIRECTIVES

During input mode, you can create a source unit or add lines to an existing source unit by entering through the directive input device one or more input directives.

Input directives have the following capabilities:

- Add lines after a specified address (Append directive).
- Delete specified lines and insert other specified lines (Change directive).
- Add lines before a specified address (Insert directive).

You can create a source unit by using the Append or Insert directive. You can add lines to an existing source unit by using any or all of the above directives.

Each input directive must have one of the following formats:

FORMAT 1:

$$\left[\begin{matrix} adr_1 \\ [text] \end{matrix}\right] \left[\left\{\begin{matrix} ; \\ , \end{matrix}\right\}\right] adr_2 \quad dirname$$

.
.
.

!F*["comment]

FORMAT 2:

$$\left[adr_1 \quad \left[\left\{\begin{matrix} ; \\ , \end{matrix}\right\} \; adr_2\right]\right] \quad dirname[text]!F*["comment]$$

If directives are being entered through a terminal, the directive name can either be immediately followed by a carriage return, and then text (i.e., the lines to be included in the source unit) or directive name can be immediately followed by text, with additional lines of text (if any) added on subsequent lines. The text can be any number of lines of ASCII characters. The maximum number of characters per line is determined by the value specified in the -LINE_LEN n argument of the ED command. The last line of text must be followed by the escape sequence !F* to terminate input mode; otherwise, the next Line Editor directive is interpreted as additional text. The escape sequence !F can be entered at the end of the last line of text or in the first character position of the next line. The next directive can begin in the next character position or on the next line.

NOTES

1. To enter a blank from the operator terminal, as the first character on a line, precede it with an !C sequence.

2. The characters !F can be included as text by preceding them with !C; in this case, !F does not designate the end of the text.

Input directives are described in detail on the following pages. In the examples, numbers in parentheses are references to line numbers and do not appear in memory or in text.

*When entering directives from a card reader, the punch for an exclamation point is 12-8-7.

APPEND

APPEND (A)

Move one or more specified lines into the current buffer after a specified address. If multiple lines are specified, they are put into the buffer in the order in which they were entered. The Append directive can be used to create a source unit or to add lines to an existing source unit.

After the Append directive is executed, the current line is the last line appended. The appended line(s) are given line numbers and subsequent lines, if any, are renumbered.

FORMAT 1:

```
[adr]A
text
.
.
.
!F
```

FORMAT 2:

```
[adr]Atext!F
```

ARGUMENT:

adr

Address of the line immediately after which the specified line(s) will be inserted.

Default: Current line. If the buffer is empty, the current line is line number 0.

NOTE

If you are creating a new source unit, there is no need to specify an address.

Example 1, Creating a New Source Unit:

In this example, the buffer is empty.

```
A
WWW
XXX
YYY
ZZZ
!F
```

This Append directive puts lines WWW, XXX, YYY, and ZZZ into the current buffer. Since the buffer is empty, it is not necessary to specify an address. The lines will be inserted, in the order in which they were entered, starting at line 1. The lines put into the buffer constitute a new source unit which can then be edited and/or written to a file.

Example 2, Adding Lines to an Existing Source Unit:

```
/TTT/A
UUU
!F
3A
WWW
XXX
!F
```

These Append directives put line UUU into the buffer immediately after the first line that contains TTT, and lines WWW and XXX into the buffer immediately after the third line.

The contents of the buffer are:

```
(1) TTT
(2) VVV
```

After the first Append directive is executed, the buffer will contain:

```
(1) TTT
(2) UUU (current line)
(3) VVV
```

After the second Append directive is executed, the buffer will contain:

(1) TTT
(2) UUU
(3) VVV
(4) WWW
(5) XXX (current line)

CHANGE (C)

Delete a single line or a series of lines in the current buffer and then insert the text specified between the directive name and the insert terminator !F.

After the Change directive is executed, the current line is the last line of inserted text. The inserted line(s) are given line numbers and subsequent lines, if any, are renumbered.

FORMAT 1:

$$\left[\begin{matrix} adr_1 \left[\begin{matrix} ; \\ , \end{matrix}\right. adr_2 \left.\right] \\ text \end{matrix}\right] C$$

•
•
•
!F

FORMAT 2:

$$\left[adr_1 \left[\begin{matrix} ; \\ , \end{matrix}\right. adr_2 \left.\right]\right] C text\,!F$$

ARGUMENTS:

adr

Address of the first or only line to be deleted and replaced. Default: Current line.

adr

Address of the last line to be deleted and replaced. Default: Only the line identified by adr is deleted and changed.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is deleted and replaced.

In the following examples, the contents of the current buffer are:

```
(1) AAA
(2) BBB
(3) CCC (current line)
(4) DDD
(5) EEE
```

Example 1:

```
2C
XXX
YYY
!F
```

This Change directive deletes the second line and replaces it with lines XXX and YYY. Subsequent lines are renumbered.

After the Change directive is executed, the buffer will contain:

```
(1) AAA
(2) XXX
(3) YYY (current line)
(4) CCC
(5) DDD
(6) EEE
```

Example 2:

```
/BBB/,.1C
XXX
YYY
ZZZ!F
```

This Change directive deletes the first line that contains BBB (line 2) through the line immediately after the current line (line 4) and replaces them with lines XXX, YYY, and ZZZ, respectively.

After the Change directive is executed, the buffer will contain:

```
(1) AAA
(2) XXX
(3) YYY
(4) ZZZ (current line)
(5) EEE
```

Example 3:

```
.,5C        .,$C
XXX    or   XXX
!F          !F
```

Each of the Change directives above deletes the current line through line 5 and replaces them with a single line containing XXX.

After the change directive is executed, the buffer will contain:

(1) AAA
(2) BBB
(3) XXX (current line)

INSERT

INSERT (I)

Insert one or more specified lines into the current buffer before a specified address. If multiple lines are specified, they are inserted in the order in which they were entered.

The Insert directive can be used to create a source unit or to add lines to an existing source unit.

After the Insert directive is executed, the current line is the last line inserted. The inserted line(s) are given line numbers, and subsequent lines, if any, are renumbered.

FORMAT 1:

```
[adr]I
text
.
.
.
!F
```

FORMAT 2:

```
[adr]Itext!F
```

ARGUMENT:

adr

Address of the line immediately before which the specified line(s) will be inserted. Default: Current line.

NOTE

If you are creating a new source unit, there is no need to specify an address.

Example 1:

In this example, the current buffer is empty.

```
I
AAA
BBB
CCC
DDD
!F
```

This Insert directive creates in the current buffer a new source unit comprising lines AAA, BBB, CCC, and DDD, respectively. The lines can then be edited and/or written to a file.

In Examples 2, 3, and 4, the contents of the current buffer are:

```
(1) AAA
(2) BBB
(3) CCC
(4) DDD (current line)
```

Example 2:

```
-2I
XXX
!F
```

This Insert directive designates that a line containing XXX be inserted two lines before the current line.

After the Insert directive is executed, the current buffer will contain:

```
(1) AAA
(2) XXX (current line)
(3) BBB
(4) CCC
(5) DDD
```

Example 3:

```
/AAA/I
H!C!FH
KKK
!F
```

This Insert directive designates that lines H!FH and KKK be inserted into the current buffer immediately before the first line that contains AAA. Note that when !F is part of the text, it is preceded by !C; when !F delimits the last line of text, it is not preceded by !C.

After the Insert directive is executed, the buffer will contain:

```
(1) H!FH
(2) KKK (current line)
(3) AAA
(4) BBB
(5) CCC
(6) DDD
```

Example 4:

```
I
XXX
!F
```

This Insert directive designates that a line containing XXX be inserted immediately before the current line.

After the Insert directive is executed, the current buffer will contain:

```
(1) AAA
(2) BBB
(3) CCC
(4) XXX (current line)
(5) DDD
```

EDIT MODE DESCRIPTION AND DIRECTIVES

During edit mode you can create a source unit or edit an existing source unit.

Edit mode directives have the following capabilities:

- Delete specified line(s) from the current buffer (Delete directive)
- Print on the user-out file specified line(s) in the current buffer (Print directive)
- Terminate execution of the Line Editor (Quit directive)
- Read text from specified file into the current buffer (Read directive)
- Substitute a designated string of characters in specified line(s) with another specified string of characters (Substitute directive)
- Write specified line(s) from the current buffer to specified file (Write directive).

NOTES

1. To edit an existing source unit, the Read directive must be previously specified.
2. Until you are familiar with the Line Editor, enter Print directives frequently so you can determine the status of the lines being edited.
3. To save the results of an edited or newly created source unit, you must specify the Write directive before you terminate execution of the Line Editor.

Most edit mode directives have one of the following formats:

FORMAT 1:

```
dirname["comment]
```

FORMAT 2:

```
adr dirname["comment]
```

FORMAT 3:

$$\left[adr_1 \quad \left\{\begin{matrix};\\,\end{matrix}\right\}\right] adr_2\ dirname["comment]$$

Edit mode directives are described alphabetically on the following pages. In the examples, numbers in parentheses are references to line numbers and do not appear in memory or in text.

DELETE (D)

Delete a single line or consecutive lines from the current buffer.

After the Delete directive is executed, each subsequent line in the buffer is renumbered, and the current line is the line that immediately follows the last line deleted or the last line in the buffer if the previous "last line" was deleted.

FORMAT:

$$\left[\text{adr}_1 \left[\left\{\begin{matrix};\\,\end{matrix}\right\} \text{adr}_2\right]\right]\text{D}$$

ARGUMENTS:

adr_1

Address of the first or only line to be deleted. Default: Current line.

adr_2

Address of the last line to be deleted. Default: Only the line identified by adr is deleted.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is deleted.

In the following examples, the contents of the current buffer are:

(1) AAA
(2) BBB (current line)
(3) CCC
(4) DDD
(5) EEE

Example 1:

```
1,3D
```

This Delete directive deletes lines 1 through 3. After this Delete directive is executed, the current buffer will contain:

(1) DDD (current line)
(2) EEE

Example 2:

```
/CCC/D
```

In this Delete directive, adr is CCC and adr is not specified, so the only line that will be deleted is the first line that contains CCC. After this Delete directive is executed, the current buffer will contain:

(1) AAA
(2) BBB
(3) DDD (current line)
(4) EEE

Example 3:

```
.,3D
```

This Delete directive deletes the current line through line 3. After this Delete directive is executed, the current buffer will contain:

(1) AAA
(2) DDD (current line)
(3) EEE

Example 4:

```
D
```

This Delete directive does not include any addresses so only the current line, line number 2, is deleted. After this directive is executed, the current buffer will contain:

(1) AAA
(2) CCC (current line)
(3) DDD
(4) EEE

PRINT (P)

Print a single line or consecutive lines in the current buffer. You can specify the address(es) of the line(s) to be printed, or you can request a printout of the first line that contains a specified expression. The printout is issued to the user-out file; i.e., the file designated in the -OUT out_path argument of the Enter Batch Request (EBR) or Enter Group Request (EGR) command, unless the file was reassigned in the File Out (FO) command. If the printout occurs on the operator terminal, each line of text is preceded by the group identification characters.

After the Print directive is executed, the current line is the last (or only) line printed.

FORMAT 1:

Format including directive name P:

$$\left[adr_1 \left[\left\{\begin{matrix};\\,\end{matrix}\right\} adr_2\right]\right]P$$

ARGUMENTS:

adr

Address of the first or only line to be printed. The Line Editor begins its search at the second line in the current buffer.
Default: Current line.

adr

Address of the last line to be printed. Default: Only the line identified by adr_1 is printed.

NOTE

If both adr and adr are omitted and P is specified, only the current line is printed.

FORMAT 2:

Format excluding directive name P:

$$adr_1 \left[\left\{\begin{matrix};\\,\end{matrix}\right\} adr_2\right]$$

ARGUMENTS:

adr_1

If adr_2 is not specified, adr_1 designates the address of the only line to be printed.

adr_2

Address of last line to be printed.

In the following examples, the contents of the current buffer are:

```
(1) AAABBB
(2) CCCDDD (current line)
(3) EEEFFF
(4) GGGHHH
```

Example 1:

```
1,$P
```

This Print directive causes a printout of each line in the current buffer.

```
AAABBB
CCCDDD
EEEFFF
GGGHHH
```

After this directive is executed, the current line is line number 4.

Example 2:

```
P
```

This Print directive causes a printout of only the current line.

```
CCCDDD
```

After this directive is executed, the current line still is line number 2.

Example 3:

```
4P
```

This Print directive causes a printout of line number 4.

```
GGGHHH
```

After this directive is executed, the current line is line number 4.

Example 4:

```
.,4P
```

This Print directive causes a printout of the current line (line number 2) through line number 4:

```
CCCDDD
EEEFFF
GGGHHH
```

After this directive is executed, the current line is line number 4.

Example 5:

```
/AAA/
```

This Print directive causes a printout of the first line that contains AAA.

```
AAABBB
```

After this directive is executed, the current line is line number 1.

Example 6:

```
3D/AAA/
```

This example illustrates a directive line that contains both a Delete directive and a Print directive in which only an expression is designated.

This directive line deletes line number 3 and causes a printout of the first line that contains AAA. After the directives are executed, the current buffer will contain:

```
(1) AAABBB
(2) CCCDDD
(3) GGGHHH
```

There will be a printout of line number 1, and that line will be the current line.

QUIT (Q OR !Q)

Exit from the Line Editor. Quit must be specified at the end of the editing session. This directive must be the last or only directive on a line. If the directive input device is a terminal, the Quit directive must be immediately followed by a carriage return.

Quit is executed conditionally or unconditionally, depending on which Quit format is specified. In a conditional Quit request (Format 1), if a buffer has a pathname associated with it via a Read or Write directive and the contents of the buffer have been modified but not written to a file before the Quit directive is entered, a warning message is issued and Quit is not executed. After the message, any Line Editor directive(s), including Write, may be entered. If Write is not specified and Quit is reentered, the Quit directive is executed and changes specified in previous Line Editor directives are not saved. In an unconditional Quit request (Format 2), modified buffers are not checked before Quit is executed.

FORMAT 1:

```
Q
```

FORMAT 2:

```
!Q
```

Example:

A	Append directive, which puts specified lines into current buffer.
AAABBB CCCDDD EEEFFF	Lines that will be put into current buffer.
!F	Designate the end of the insertion.
2D	Delete the second line of text (e.g., CCCDDD).
W FIRST	Write all lines in buffer to file named FIRST.
Q	Return control from the Line Editor to the Command Processor.

READ

READ (R)

Read text from a specified file into the current buffer. The Read directive must be the only or last directive on a line. After the Read directive is executed, the current line is the last line read from the file.

FORMAT:

[adr]R[path]

ARGUMENTS:

adr

Address of a line in the current buffer; the contents of the specified file will be appended after this line. Default: Last line in the buffer; if the buffer is empty, the file is appended starting at the first line in the buffer.

path

Pathname of the ASCII file to be read into the current buffer. (Methods of specifying pathnames are described in Section 2.) The pathname may be preceded by any number of blanks. Default: Pathname specified in the latest Read or Write directive associated with the current buffer. To determine which pathname was specified last, specify the Buffer Status directive, which is described under "Advanced Usage of the Line Editor" later in this section. If the path argument is not specified and a pathname was not previously specified, an error message is issued.

NOTE

!CDR or any other device name beginning with an exclamation point (!) may cause errors. The exclamation point is a Line Editor escape character. A read of !CDRxx (R !CDRxx) will try to read file name DRxx because !C is a conceal flag. Use >SPD> in place of the exclamation point (e.g., R >SPD>CDRxx), or conceal a C (e.g., R !!CCDRxx).

Example 1:

```
R START
```

This Read directive reads into the current buffer the contents of a file whose simple pathname is START. Since an address is not specified, the lines are read into the buffer after the last line that currently is in the buffer.

The contents of START are:

```
(1) AAA
(2) BBB
(3) CCC
```

If the buffer is empty, after the Read directive is executed, the current buffer will contain:

```
(1) AAA
(2) BBB
(3) CCC (current line)
```

If the buffer already contains:

```
(1) XXX
(2) YYY
(3) ZZZ
```

After the Read directive is executed, the current buffer will contain:

```
(1) XXX
(2) YYY
(3) ZZZ
(4) AAA
(5) BBB
(6) CCC (current line)
```

Example 2:

```
/CCC/R NEW
```

This Read directive designates that the contents of the file whose simple pathname is NEW be read into the current buffer after the first line in the current buffer that contains CCC.

The contents of the current buffer are:

```
(1) AAA
(2) BBB (current line)
(3) CCC
(4) CCC
```

The contents of NEW are:

```
(1) XXX
(2) ZZZ
```

After the Read directive is executed, the current buffer will contain:

```
(1) AAA
(2) BBB
(3) CCC
(4) XXX
(5) ZZZ (current line)
(6) CCC
```

Example 3:

This example illustrates the Read directive used in conjunction with Append and Write directives. The current buffer is empty.

```
A       Puts subsequent lines into the current buffer.
AAA
BBB
CCC
!F      Designates the end of the insert.
W  NOW  Writes the contents of the current buffer to the
        file whose simple pathname is NOW.
R       Reads into the current buffer, after the last line
        in the buffer, the contents of NOW; NOW is the
        pathname specified in the last Write directive.
```

After the Read directive is executed, the current buffer will contain:

```
(1) AAA
(2) BBB
(3) CCC
(4) AAA
(5) BBB
(6) CCC (current line)
```

SUBSTITUTE (S OR !S)

Replace each occurrence of a specified string of characters in a single line or in a sequence of lines with another specified string of characters.

After this directive is executed, the current line is the last line located by the Line Editor.

FORMAT:

$$\left[adr_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} adr_2\right]\right] S/regexp/string/$$

or

$$\left[adr_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} adr_2\right]\right] !S/regexp/string/ \quad \text{(See Note 3)}$$

ARGUMENTS:

adr

Address of the first line to be searched for the specified string of characters. The search begins at the second line in the current buffer. Default: Current line.

adr

Address of the last line to be searched for the specified string of characters. Default: adr_1.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is searched.

/

(Delimiter) Can be any character that is not in regexp or string. However, the same delimiter must be used in each of the three locations where a delimiter is required.

regexp

String of characters for which the Line Editor is searching; each occurrence of this character string within the specified addresses will be replaced with the character(s) specified in the argument "string".

Default: The last regexp specified. This can be determined by entering the ZREGEXP directive, which is described under "Line Editor Debugging Directives".

string

String of characters that will replace each occurrence of regexp.

NOTES

1. If string contains the character "&" in any position, each occurrence of regexp to be replaced will be replaced with regexp included in string, in place of "&". For example, if regexp is "in" and string is "&to", each occurrence of "in" becomes "into". To ignore the special meaning of "&", precede it with !C.

2. The occurrence of a line feed in the string expression determines the new line characters, i.e., point in the resulting line at which the line is to be split into two lines.

3. If the directive name !S is used (as illustrated in the second directive format) and the specified substitution fails, no error message is issued and execution of the command file (if any) continues.

Example 1:

```
S/ABGDEF/ABC linefeed DEF/
```

This Substitute directive searches the current line and (1) replaces each occurrence of ABGDEF with ABCDEF and (2) causes the character string to be split between two lines. ABC will be on the first line, and DEF will be on the second line.

Example 2:

The contents of the current buffer are:

(1) E
(2) NTE
(3) R
(4) YOUR

1,3S/linefeed key//

After this Substitute directive is entered, the current buffer will contain:

(1) ENTERYOUR

In the following examples, the contents of the current buffer are:

(1) AAACCC
(2) BBBAAA (current line)
(3) CCCBBB
(4) DDDAAA

Example 3:

2,4S/AAA/XXX/

This Substitute directive searches lines 2 through 4 and replaces each occurrence of AAA with XXX.

After this directive is executed, the current buffer will contain:

(1) AAACCC
(2) BBBXXX
(3) CCCBBB
(4) DDDXXX (current line)

Example 4:

.,4S-CCC-UUU-

This Subsitute directive searches the current line (line 2) through line number 4 and replaces each occurrence of CCC with UUU.

After this directive is executed, the current buffer will contain:

(1) AAACCC
(2) BBBAAA
(3) UUUBBB
(4) DDDAAA (current line)

Example 5:

```
-1,/DDD/S//&JJJ/
```

This Substitute directive searches one line before the current line (line 1) through the first line that contains DDD (line 4) and replaces each occurrence of DDD with DDDJJJ.

After this directive is executed, the current buffer will contain:

(1) AAACCC
(2) BBBAAA
(3) CCCBBB
(4) DDDJJJAAA (current line)

Example 6:

```
/BBB/S//XXX/
```

This Substitute directive searches the first line after the current line through the current line (line 2) and changes the first occurrence of BBB to XXX.

After this directive is executed, the current buffer will contain:

(1) AAACCC
(2) BBBAAA
(3) CCCXXX (current line)
(4) DDDAAA

WRITE (W)

W

Write a specified line or a series of lines in the current buffer to a specified file. If the file does not already exist, a new file is created with the specified file name. If the named file does exist and currently contains other data, the line(s) written to the file via the Write directive replace the existing contents.

To save the results of previously specified Line Editor directives, you must specify the Write directive before you terminate execution of the Line Editor (i.e., Write must be specified before Quit).

The Write directive must be the last directive on a line. After the Write directive is executed, the specified line(s) remain in the current buffer; a copy of them is written to the specified file.

FORMAT:

$$\left[adr_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} adr_2\right]\right] W [path]$$

ARGUMENTS:

adr_1

Address of the first line to be written to a specified file. Default: First line in the current buffer.

adr_2

Address of the last line to be written to a specified file. Default: Last line in the current buffer.

NOTE

If both adr_1 and adr_2 are omitted, all lines in the current buffer are written to the specified file.

path

Pathname of the file to which the specified line(s) will be written. (Methods of specifying pathnames are described in Section 2.) The pathname may be preceded by any number of spaces. Default: Pathname specified in the latest Read or Write directive associated with the current buffer. If a pathname was not previously specified, an error message is issued.

Example 1:

```
W IDENT
```

This Write directive writes all lines in the current buffer to a file whose simple pathname is IDENT.

Example 2:

This example illustrates use of a Write directive in a sample Line Editor session. In this example, there is a file named EXIST that contains the following lines:

```
(1) AAA
(2) BBB
(3) CCC
(4) DDD
```

```
R EXIST
```

Read into the current buffer the contents of the file named EXIST. The current buffer will contain:

```
(1) AAA
(2) BBB
(3) CCC
(4) DDD (current line)
```

```
1,$S/AAA/XXX/
```

Search each line in the current buffer and change each occurrence of AAA to XXX. The buffer will contain:

```
(1) XXX
(2) BBB
(3) CCC
(4) DDD (current line)
```

```
1,3W
```

Write lines 1 through 3 to the file specified in the last Read or Write directive; i.e., EXIST. EXIST will contain:

```
(1) XXX
(2) BBB
(3) CCC
```

```
Q
```

Terminate execution of the Line Editor.

ADVANCED FUNCTIONS OF THE LINE EDITOR

The directives described on the previous pages permit you to create a source unit and perform basic editing. The following subsections describe Line Editor directives that perform general advanced functions, permit usage of auxiliary buffers, perform debugging, and perform programming functions. Within each subsection the directives are summarized and then described in detail alphabetically by full directive name.

GENERAL ADVANCED LINE EDITOR DIRECTIVES

The general advanced Line Editor directives have the following capabilities:

- Cause another specified directive to act on only those lines that do not contain a specified character string (Exclude directive)
- Permit execution of a command instead of Line Editor directives without exiting from the Line Editor (Execute directive)
- Cause another specified directive to act on only those lines that contain a specified character string (Global directive)
- Send line feed to user-out file and error-out file (Line Feed directive)
- Convert the specified expression to lowercase (Lowercase directive)
- Make a different line the current line (New Current Line directive)
- Print the line number of a specified line in the current buffer (Print Line Number directive)
- Print the line number and contents of specified line(s) in the current buffer (Print With Line Number directive)
- Convert the specified expression to uppercase (Uppercase directive).

EXCLUDE

EXCLUDE (V)

Exclude specified elements. The Exclude directive can be used in conjunction with Delete, Print, Print Line Number, and Print With Line Number directives so that the specified directive acts on only those lines that do not contain a specified character string.

After the Exclude directive is executed, the current line is the last line searched by the Line Editor; i.e., the line specified in adr_1 (see below).

FORMAT:

$$\left[adr_1 \left[\left\{\begin{matrix};\\,\end{matrix}\right\} adr_2\right]\right] Vx/regexp/$$

ARGUMENTS:

adr_1

Address of the first line to be searched. Default: First line in the current buffer.

adr_2

Address of the last line to be searched. Default: Last line in the current buffer.

NOTE

If both adr_1 and adr_2 are omitted, all lines in the buffer are searched.

x

Directive name with which the Exclude directive is being isued; must be one of the following:

D - VD deletes line(s) that do not contain regexp.

P - VP prints the contents of line(s) that do not contain regexp.

!P - V!P prints the line number(s) and contents of line(s) that do not contain regexp.

= - V= prints the line number(s) of line(s) that do not contain regexp.

(Delimiter) Can be any character that does not occur in regexp. The same delimiter must be used before and after regexp.

regexp

String of characters for which the Line Editor will search; only lines that do not contain regexp will be acted upon by the Line Editor during execution of the directive name specified in argument x.

In the following examples, the contents of the current buffer are:

```
(1) JJJKKK (current line)
(2) LLLMMM
(3) NNNPPP
(4) RRRJJJ
```

Example 1:

```
1,3V!P/JJJ/
```

This Exclude Print with line number directive causes the Line Editor to search lines 1 through 3 and to print the line number and contents of each line that does not contain JJJ.

Typeout:

```
2 LLLMMM
3 NNNPPP
```

Current line: 3

Example 2:

```
VD*JJJ*
```

This Exclude Delete directive deletes each line that does <u>not</u> contain JJJ; since no addresses are specified, each line in the current buffer is searched.

After this directive is executed, the current buffer will contain:

```
(1) JJJKKK
(2) RRRJJJ (current line)
```

EXECUTE

EXECUTE (E)

Cause execution processing. The Execute directive permits you to execute a command instead of Line Editor directives without exiting from the Line Editor; i.e., you can enter any command and then continue to use the Line Editor. For example, the Execute directive can be used to designate a printer as the Line Editor output file. Otherwise, if you want a printout of Line Editor output, the printout is issued to the terminal, which is the original user-out file. If the user-out file is a line printer and a Quit directive is entered to exit from the Line Editor, the user-out file remains set to the printer.

The Execute directive must be the last directive on a line.

The current line is not affected by Execute directives.

FORMAT:

E command

ARGUMENT:

command

Any command (see the Commands manual).

Example:

```
E FO  >SPD>LPT00
```

This Execute directive includes a File Out (FO) command, which sets the user-out file to the line printer whose pathname is >SPD>LPT00.

GLOBAL (G)

Act on only those lines that contain a specified character string and can be used in conjunction with Delete, Print, Print Line Number, and Print With Line Number directives.

After the Global directive is executed, the current line is the last line searched by the Line Editor.

FORMAT:

$$\left[adr_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} adr_2\right]\right] Gx/regexp/$$

ARGUMENTS:

adr_1

Address of the first line to be searched. Default: First line in the current buffer.

adr_2

Address of the last line to be searched. Default: Last line in the current buffer.

NOTE

If both adr_1 and adr_2 are omitted, all lines in the current buffer are searched.

x

Directive name with which the Global is being used; must be one of the following:

D - Delete all line(s) in the specified range containing regexp.

P - Print the contents of line(s) containing regexp.

!P - Print the line number(s) and contents of line(s) containing regexp (see "Print With Line Number Directive" later in this section).

= - Print the line number(s) of line(s) containing regexp (see "Print Line Number Directive" later in this section).

/

(Delimiter) Can be any character that does not occur in regexp. The same delimiter must be used before and after regexp.

regexp

String of characters for which the Line Editor will search; only lines that contain regexp will be acted upon by the directive name specified in argument x.

In the following examples, the contents of the current buffer are:

(1) JJJKKK
(2) LLLMMM
(3) NNNPPP
(4) RRRJJJ

Example 1:

```
1,3G!P/JJJ/
```

This Global Print With Line Number directive causes the Line Editor to search lines 1 through 3 and print the line number and con- tents of each line that contains JJJ.

Typeout:

```
1 JJJKKK

Current line:  3
```

Example 2:

```
GD*JJJ*
```

This Global Delete directive deletes each line that contains JJJ; since no addresses are specified, all lines in the buffer are searched.

After this directive is executed, the current buffer will contain:

(1) LLLMMM
(2) NNNPPP (current line)

LINE FEED (L OR !L)

Send line feeds to the user-out file and the error-out file, respectively. After the Line Feed directive is executed, the current line is unchanged. Default: none (addresses are ignored).

FORMAT:

L or !L

LOWERCASE

LOWERCASE (U)

Convert all occurrences of a specified expression within specified addresses from uppercase to lowercase. After the Lowercase directive is executed, the current line is the last line read.

FORMAT:

$$\left[adr_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} adr_2\right]\right] U/regexp/$$

ARGUMENTS:

adr_1

Address of the first line to be searched. Default: Current line.

adr_2

Address of the last line to be searched. Default: adr_1.

regexp

String of characters for which the Line Editor searches. Only uppercase letters (A through Z) are converted; others are not changed.

Example:

```
U/ADR/
```

This Lowercase directive searches the current line and changes each occurrence of ADR to adr. If the current line is:

```
ADR FIRST
```

after the Lowercase directive is executed, the line contains:

```
adr FIRST
```

NEW CURRENT LINE (N)

Cause the specified line to become the new current line. The contents of the new current line are not printed after the directive is executed.

FORMAT:

```
adrN
```

ARGUMENT:

adr

Address of the line that is to be the new current line.

Example:

```
/CCC/N
```

If the following condition exists prior to execution of the N directive:

```
AAA (current line)
BBB
CCC
DDD
```

The situation will be as follows after the N directive is executed.

```
AAA
BBB
CCC (current line)
DDD
```

PRINT LINE NUMBER

PRINT LINE NUMBER (=/!P)

Print out the line number of a specified line in the current buffer.

The printout is issued to the user-out file, i.e., the file designated in the -OUT out_path argument of the Enter Batch Request (EBR) or Enter Group Request (EGR) command, unless that file was reassigned.

After this directive is executed, the current line is the line whose line number was typed.

FORMAT:

```
[adr]=
```

ARGUMENT:

adr

Address of the line whose line number is to be typed.

Default: Current line.

In the following examples the contents of the current buffer are:

```
(1) AAABBB (current line)
(2) CCCDDD
(3) CCCEEE
```

Example 1:

```
/CCC/=
```

This Print Line Number directive causes a printout of the line number of the first line that contains CCC.

Printout:

```
2
```

Current line: 2

Example 2:

```
=
```

This Print Line Number directive causes a printout of the line number of the current line.

Printout:

```
1
```

Current line: 1

PRINT WITH LINE NUMBER

PRINT WITH LINE NUMBER (!P)

Print out the line number and contents of a single line or consecutive lines in the current buffer. The printout is issued to the user-out file, i.e., the file designated in the -OUT out path argument of the Enter Batch Request or Enter Group Request command, unless the file was reassigned. If the printout occurs on a terminal, each line of text is preceded by the group identification characters.

After this directive is executed, the current line is the last line whose line number and contents were typed.

FORMAT:

$$\left[adr_1 \left[\left\{\begin{matrix};\\,\end{matrix}\right\} adr_2\right]\right] !P$$

ARGUMENTS:

adr_1

Address of the first line whose line number and contents are to be typed. Default: Current line.

adr_2

Address of the last line whose line number and contents are to be typed. Default: Address specified for adr_1.

NOTE

If both adr_1 and adr_2 are omitted, there is a printout of the line number and contents of the current line.

In the following examples, the contents of the current buffer are:

(1) AAA
(2) BBB (current line)
(3) CCC
(4) DDD

Example 1:

```
1,$!P
```

This Print With Line Number directive causes a printout of the line number and contents of each line in the current buffer.

Printout:

```
1 AAA
2 BBB
3 CCC
4 DDD
```

Current line: 4

Example 2:

```
!P
```

This Print With Line Number directive causes a printout of the line number and contents of only the current line.

Printout:

```
2 BBB
```

Current line: 2

UPPERCASE

UPPERCASE (!U)

Convert all occurrences of a specified expression within specified addresses from lowercase to uppercase.

After the Uppercase directive is executed, the current line is the last line read.

FORMAT:

$$\left[adr_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} adr_2\right]\right] !U/regexp/$$

ARGUMENTS:

adr_1

Address of the first line to be searched. Default: Current line.

adr_2

Address of the last line to be searched. Default: adr_1.

regexp

String of characters for which the Line Editor searches. Only lowercase letters (a through z) are converted; others are not changed.

Example:

```
!U/adr/
```

This Uppercase directive searches the current line and change each occurrence of adr to ADR. If the current line is:

```
adr first
```

after the Uppercase directive is executed, the line contains:

```
ADR first
```

COMMENT (")

Annotate Line Editor command files. The text after the Comment directive appears as program output but is ignored by the Line Editor.

FORMAT:

"comment

AUXILIARY BUFFER DIRECTIVES AND ESCAPE SEQUENCES

In the previous pages of this section, it was assumed that there is only a single buffer, the current buffer. The current buffer must be used, but one or more additional buffers, called auxiliary buffers, also can be used. There are 64 auxiliary buffers available for use.

The most common use of auxiliary buffers is for moving or copying text from one part of a file to another.

To make an auxiliary buffer available and to put lines into it, specify the Move, Move-Append, Copy, and/or Copy-Append directives, which are described in the following paragraphs.

Lines cannot be written directly from an auxiliary buffer to a file; the auxiliary buffer must be designated in the Change Buffer directive as the current buffer or the lines must be read back to the current buffer via the escape sequence !B, which is described under "Change Origin of Text During Input Mode", later in this section. Lines can be written from the current buffer to a file via the Write directive (see "Write Directive" earlier in this section).

You can determine the status of each buffer currently in use by specifying the Buffer Status directive.

Auxiliary buffer directives have the following functions:

- Cause Line Editor to accept a line from terminal (Accept Single Line From a Terminal directive)
- Determine status of each buffer in use (Buffer Status directive)
- Make specified auxiliary buffer the current buffer (Change Buffer directive)
- Cause Line Editor to accept subsequent text from a specified auxiliary buffer
  - During edit mode (Change Origin of Text During Edit Mode directive)
  - During input mode (Change Origin of Text During Input Mode directive)
- Copy line(s) in current buffer to specified auxiliary buffer; lines in current buffer are not deleted
  - Delete existing lines in auxiliary buffer (Copy directive)
  - Do not delete lines in auxiliary buffer (Copy-Append directive)

- Destroy a buffer (i.e., release its file space) (Destroy directive)

- Move line(s) from current buffer to specified auxiliary buffer; lines in current buffer are deleted

  - Lines overlay existing lines, if any, in auxiliary buffer (Move directive)

  - Lines appended to existing lines, if any, in auxiliary buffer (Move-Append directive).

ACCEPT SINGLE LINE FROM A TERMINAL

ACCEPT SINGLE LINE FROM A TERMINAL (!R)

Permit a single line of directives or text to be entered through a terminal. !R normally is used when Line Editor directives are being executed from a buffer. When the Line Editor encounters !R, the entire escape sequence is removed from the input stream and replaced with the line read from the user-in file.

FORMAT:

```
!R
```

Example:

```
T/ENTER YOUR NAME/
A!R!F
```

These directives are in the buffer that is being executed.

There will be the following message on the terminal:

```
ENTER YOUR NAME
```

You will respond with your name, i.e., Jane Jones.

Following the current line in the current buffer will be:

```
Jane Jones
```

BUFFER STATUS (X)

Cause a message of the status of each buffer currently in use. The current line is not changed.

FORMAT:

X

DESCRIPTION:

The following information is designated:

- Name of each buffer. The original current buffer is always named 0.
- Number of lines in each buffer.
- Indicator as to which buffer is the current buffer. The name of the current buffer is preceded by ->.

If a buffer has been read into and/or written from, the message includes the pathname specified in the last read or write.

If the contents of the current buffer have been modified (i.e., in the message, MOD is designated before its name), all of the following conditions must exist:

- Lines from an existing file have been read into the current buffer via a Read directive or the contents of the current buffer have been written to a file.
- The contents of the buffer were modified via one or more Line Editor directives.

Each message has the following format:

```
number of lines  ->[MOD]  (buffer-name)  [pathname]
[number of lines   [MOD]  (buffer-name)  [pathname]]
        .           .          .            .
        .           .          .            .
        .           .          .            .
```

Example:

This example illustrates usage of the buffer status directive. The file USE, which is in the working directory, comprises the following lines:

```
(1) AAA (current line)
(2) BBB
(3) CCC
(4) DDD
```

```
R USE
```

Read the contents of USE into the current buffer, which is named 0.

```
1,$S*BBB*XXX*
```

Search the first line through the last line in the current buffer and changes each occurrence of BBB to XXX. After this directive is executed, the current buffer will contain:

```
(1) AAA
(2) XXX
(3) CCC
(4) DDD
```

```
3,4M2
```

Move lines 3 and 4 of the current buffer into auxiliary buffer 2. After this directive is executed, the current buffer will contain:

```
(1) AAA
(2) XXX
```

Auxiliary buffer 2 will contain:

```
(1) CCC
(2) DDD
```

```
X
```

Request the status of each buffer currently in use. The following message will be issued:

```
2  ->MOD  (0)  USE
2         (2)
```

CHANGE BUFFER (Bx)

Designate that a specified auxiliary buffer is to become the current buffer. The previously designated current buffer becomes an auxiliary buffer.

After this directive is executed, lines can be written from the new current buffer to a file.

FORMAT:

Bx

ARGUMENT:

x

Buffer name. The name must be 1 to 6 ASCII characters. If the name comprises more than a single character, the name must be enclosed within parentheses; otherwise, the parentheses are optional. The original current buffer name is 0. This name can never be altered. An auxiliary buffer name, once specified, cannot be altered during the current Line Editor session.

Example:

B3

This directive designates that auxiliary buffer 3 is the current buffer. If desired, lines can now be written from this buffer to a file.

CHANGE ORIGIN OF TEXT DURING EDIT MODE

CHANGE ORIGIN OF TEXT DURING EDIT MODE (!B)

Cause the Line Editor to read subsequent directives from a specified auxiliary buffer. !B can be specified within an expression, pathname, text to be typed (i.e., in the Type directive), or as a directive. When the Line Editor encounters this sequence in an expression, pathname, or text, the entire escape sequence is removed from the input stream and replaced with the literal contents of the first line of the specified buffer; if !B is a directive, the input stream is replaced with the entire literal contents of the specified buffer. If another !B escape sequence is encountered while accepting input from buffer x, the newly encountered escape sequence will also be replaced by the contents of its named buffer.

The buffer to which the input stream is redirected may contain Line Editor requests, literal text, or both. If the Line Editor is executing a request obtained from an auxiliary buffer and an error occurs, the usual error comment is suppressed and the remaining contents of that buffer are skipped. Control returns to the statement immediately following the !B escape sequence that called the auxiliary buffer. For example, if one thinks of the escape sequence !B(x) as a subroutine call statement, the failure to match a regular expression specified by some request in buffer x may be thought of as a return statement. Once the last commands in the auxiliary buffer have been processed, control returns to the statement immediately following the !B escape sequence that called the auxiliary buffer.

The buffer name may be in the format (ARGn), where n is a number from 1 to 9 that refers to the nth argument that followed the -ARG argument of the ED command. The escape sequence is replaced with the first (or only) line of the buffer (ARGn) created during initialization of the Line Editor.

FORMAT:

!Bx

ARGUMENT:

x

Name of the buffer that contains subsequent Line Editor text. The buffer name must be 1 through 6 ASCII characters. if the buffer name comprises more than a single character, the name must be enclosed within parentheses; otherwise, the parentheses are optional.

Example 1: !B as a directive

```
!B(TEST)
```

In this example, the contents of the current buffer and the auxiliary buffer named TEST are:

Current buffer:

```
(1) A
(2) B
(3) A
(4) D
(5) E
```

Auxiliary buffer:

```
1,$S/A/X/
```

This Substitute directive designates that in the current buffer all occurrences of A be replaced with X. After the Substitute directive is executed, the current buffer will contain:

```
(1) X
(2) B
(3) X
(4) D
(5) E
```

The auxiliary buffer named TEST will contain:

```
1,$S/A/X/
```

Example 2: !B Within an Expression

```
2S/AAA/!B2/
```

This Substitute directive designates that in the second line of the current buffer, each occurrence of AAA should be replaced with the first line of auxiliary buffer 2.

The contents of the current buffer and auxiliary buffer 2 are:

Current buffer:

```
(1) AAABBB
(2) CCCAAA
(3) XXXYYY
```

Auxiliary buffer 2:

```
DDD
EEE
```

After the Substitute directive is executed, the current buffer contains:

```
(1) AAABBB
(2) CCCDDD
(3) XXXYYY
```

Example 3: !B Within Text to be Typed

```
T/!B2/
```

This Type directive (which is described later in this section) requests that the first line of auxiliary buffer B2 be displayed on the user-out file.

Example 4: Buffer Name (ARGn)

The ED command includes the argument -ARG ABC "MY NAME" XYZ

```
S/DEF/!B(ARG3)/
```

This Substitute directive searches the current line and replaces each occurrence of DEF with XYZ (i.e., the third argument following -ARG in the ED command).

CHANGE ORIGIN OF TEXT DURING INPUT MODE

CHANGE ORIGIN OF TEXT DURING INPUT MODE (!B)

Cause the Line Editor to accept subsequent text from a specified auxiliary buffer. The escape sequence !B can appear within text of an Input directive,

When the Line Editor encounters !B, the entire escape sequence is removed from the input stream and replaced with the literal contents of the specified buffer. If another !B escape sequence is encountered after accepting text from the specified buffer, the newly encountered escape sequence will also be replaced with the contents of the named buffer.

FORMAT:

[text]!Bx [[text]!B] ...

ARGUMENT:

x

Name of the buffer that contains subsequent Line Editor text. The buffer name must be 1 through 6 ASCII characters. If the buffer name comprises more than a single character, the name must be enclosed within parentheses; otherwise, the parentheses are optional.

Example:

```
/D/I
!B(TEST)!F
```

In this example, the contents of the current buffer and the auxiliary buffer named TEST are:

Auxiliary buffer:

```
(1) X
(2) Y
(3) Z
```

Current buffer:

```
(1) A
(2) B
(3) C
(4) D
(5) E
```

This Insert directive designates that the contents of the auxiliary buffer named TEST be inserted into the current buffer before the line that contains D.

After the Insert directive is executed, the current buffer will contain:

```
(1) A
(2) B
(3) C
(4) X
(5) Y
(6) Z
(7) D
(8) E
```

The auxiliary buffer named TEST will contain:

```
(1) X
(2) Y
(3) Z
```

COPY (K)

Write into a specified auxiliary buffer a single line or consecutive lines contained in the current buffer. The lines in the current buffer are <u>not</u> deleted; i.e., the lines are in both the current and the auxiliary buffers. Any lines previously in the auxiliary buffer are destroyed during execution of the Copy directive.

After the Copy directive is executed, the current line in the current buffer is the line immediately after the last line moved to the auxiliary buffer. There is no current line in the auxiliary buffer until that auxiliary buffer is changed to the current buffer via a change buffer directive.

FORMAT:

$$\left[adr_1 \left[\left\{\begin{matrix};\\,\end{matrix}\right\} adr_2\right]\right] Kx$$

ARGUMENTS:

adr_1

Address of the first line to be written into the specified auxiliary buffer. Default: Current line.

adr_2

Address of the last line to be written into the specified auxiliary buffer. Default: adr_1.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is written into the specified auxiliary buffer.

x

Name of the auxiliary buffer into which the specified line(s) will be written. The name must be 1 through 16 ASCII characters. If the name comprises more than a single character, the name must be enclosed within parentheses; otherwise, the parentheses are optional.

Example:

```
1,3K(52)
```

This Copy directive copies into auxiliary buffer 52 lines 1 through 3 in the current buffer. The contents of the current buffer are:

```
(1) FIRST (current line)
(2) SECOND
(3) THIRD
(4) FOURTH
```

After the Copy directive is executed, the contents of the current buffer are unchanged, but the current line is line number 4. Auxiliary buffer 52 will contain:

```
(1) FIRST
(2) SECOND
(3) THIRD
```

There will be no current line in the auxiliary buffer.

COPY-APPEND

COPY-APPEND (!K)

Write a line or lines from the current buffer to an auxiliary buffer without destroying the contents of the auxiliary buffer. The lines copied from the current buffer are appended to the contents of the auxiliary buffer. The lines written are also retained in the current buffer.

After the Copy-Append directive is executed, the current line in the current buffer is the line immediately after the last line written to the auxiliary buffer or the last line in the buffer. There is no current line in the auxiliary buffer.

FORMAT:

$$\left[adr_1\left[\begin{Bmatrix};\\,\end{Bmatrix} adr_2\right]\right]!Kx$$

ARGUMENTS:

adr_1

Address of the first line to be written to the specified auxiliary buffer. Default: Current line.

adr_2

Address of the last line to be written to the specified auxiliary buffer. Default: adr_1.

NOTE

If both addresses are omitted, only the current line is written to the auxiliary buffer.

x

Name of the auxiliary buffer into which the specified line(s) will be written. The name must be from 1 to 16 ASCII characters. If the name is more than one character, it must be enclosed within parentheses; otherwise, parentheses are optional.

Example:

```
1,3!K(ABUF)
```

This directive appends lines 1 through 3 of the current buffer to the contents of auxiliary buffer ABUF. Thus, if the current buffer and ABUF contain the following lines prior to execution:

Current	ABUF
(1) AAA (current line)	(1) MMM
(2) BBB	(2) NNN
(3) CCC	
(4) DDD	

They will contain the following after execution:

Current	ABUF
(1) AAA	(1) MMM
(2) BBB	(2) NNN
(3) CCC	(3) AAA
(4) DDD (current line)	(4) BBB
	(5) CCC

DESTROY (^B)

Release a specified auxiliary buffer's file space. Any buffer other than buffer 0 and the current buffer may be removed; if the current buffer name is specified, the directive is ignored and an error message is issued.

FORMAT:

^Bx

ARGUMENT:

x

Name of the auxiliary buffer to be destroyed. The name must be from 1 to 6 ASCII characters. If the name comprises more than one character, it must be enclosed within parentheses; otherwise, parentheses are optional.

Example:

^B(AX)

This Destroy directive removes buffer AX.

MOVE

MOVE (M)

Move a single line or consecutive lines from the current buffer to a specified auxiliary buffer; the lines no longer exist in the current buffer. If the auxiliary buffer already contains lines, those lines are destroyed.

After the Move directive is executed, the current line in the current buffer is the line after the last line moved to the auxiliary buffer or the last line in the buffer. There is no current line in the auxiliary buffer.

FORMAT:

$$\left[adr_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} adr_2\right]\right] Mx$$

ARGUMENTS:

adr_1

Address of the first line to be moved from current buffer to auxiliary buffer.

Default: Current line.

adr_2

Address of the last line to be moved from current buffer to auxiliary buffer.

Default: adr_1.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is moved from the current buffer to the auxiliary buffer.

x

Name of the auxiliary buffer to which the specified line(s) will be moved. The name must be 1 through 6 ASCII characters. If the name comprises more than a single character, the name must be enclosed within parentheses; otherwise, the parentheses are optional.

Example:

```
1,3M5
```

This Move directive moves lines 1 through 3 from the current buffer to the auxiliary buffer named 5. In this example, the contents of the current buffer are:

```
(1) FIRST (current line)
(2) SECOND
(3) THIRD
(4) FOURTH
```

After the Move directive is executed, the current buffer will contain:

```
(1) FOURTH (current line)
```

Auxiliary buffer 5 will contain:

```
(1) FIRST
(2) SECOND
(3) THIRD
```

MOVE-APPEND

MOVE-APPEND (!M)

Move one or more lines of text from the current buffer to the specified auxiliary buffer. The lines are appended to the existing contents of the auxiliary buffer; the existing contents of the auxiliary buffer are not overlaid. If the auxiliary buffer contains no text, the lines are placed in the auxiliary buffer starting at line 1. The lines moved are deleted from the current buffer.

FORMAT:

$$\left[adr_1\left[\left\{\begin{matrix},\\;\end{matrix}\right\} adr_2\right]\right]!Mx$$

ARGUMENTS:

adr_1

Address of the first line to be moved from the current buffer to the auxiliary buffer. Default: Current line.

adr_2

Address of the last line to be moved from the current buffer to the auxiliary buffer. Default: adr_1.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is moved from the current buffer to the auxiliary buffer.

x

Name of the auxiliary buffer to which the specified line(s) will be moved. The name must be 1 through 6 ASCII characters. A name of more than one character must be enclosed in parentheses; otherwise, parentheses are optional.

Example:

```
1,3!M(SOOZ)
```

This directive appends lines 1 through 3 to the contents of auxiliary buffer SOOZ. If the contents of the buffers are as follows prior to the move:

Current	SOOZ
(1) FIRST (current line)	(1) AAAAA
(2) SECOND	(2) BBBBB
(3) THIRD	
(4) FOURTH	

The buffers will contain the following after the move:

Current	SOOZ
(1) FOURTH (current line)	(1) AAAAA
	(2) BBBBB
	(3) FIRST
	(4) SECOND
	(5) THIRD

LINE EDITOR DEBUGGING DIRECTIVES

The functions of Line Editor debugging directives are:

- Print contents of specified line(s) on the terminal (Hexadecimal Dump directive)
- Display, on the user-out file, the last specified regular expression (ZREGEXP directive)
- Display each directive line before it is executed (ZTRACE directive).

HEXADECIMAL DUMP (ZDUMP)

Print the contents of specified line(s) on the terminal in both hexadecimal and ASCII formats. The output format consists of the line number, the length (number of characters) expressed in hexadecimal, eight words in hexadecimal format, and eight words in ASCII format.

The display of each buffer line is separated from following displays by a blank line. If a buffer line is too long to be displayed on a single line, it is continued on the next line, with no blank line separation.

After this directive is executed, the current line is the last (or only) line printed.

FORMAT:

$$\left[adr_1\left[\left\{\begin{matrix};\\,\end{matrix}\right\}\ adr_2\right]\right]\text{ZDUMP}$$

ARGUMENTS:

adr_1

Address of the first buffer line to be dumped.

Default: Current line.

adr_2

Address of the last buffer line to be dumped.

Default: adr_1.

NOTE

If both addresses are omitted, only the current line will be dumped.

Example:

The contents of lines 1 and 2 of the current buffer are:

```
(1) START EDIT
(2) VDEF ZFVER,X'3031'

1,2ZDUMP
```

This Hexadecimal Dump directive produces the following output at the terminal:

```
0001 000A 5354 4152 5420 4544 4954                  START EDIT

0002 0012 5644 4546 205A 4656 4552 2C58 2733 3033 VDEF ZFVER,X'303
          3127                                      1'
```

Thus, 0001 indicates line number 1; 000A indicates a length of 10 characters (A); followed by the hexadecimal equivalent of START EDIT. A blank line is followed by the dump of line 2, with a length of 18 characters (12). Because nine words are required to fully dump the line, the output continues on the next line of the terminal, with no blank line intervening.

ZREGEXP

Display the last specified expression on the user-out file. The current line is not changed.

FORMAT:

ZREGEXP

Example:

```
S/ABC/DEF/
ZREGEXP
```

This ZREGEXP directive displays the last specified expression, i.e., /ABC/.

ZTRACE

ZTRACE

Display each directive line on the user-out file before it is executed.

FORMAT:

```
ZTRACE {ON }
       {OFF}
```

ARGUMENTS:

ON Each directive line is displayed before it is executed.

OFF Subsequent lines are not displayed before they are executed.

Example:

This example illustrates a program that includes an ED command to load the Line Editor and a ZTRACE ON directive. Following is a printout of the Line Editor output.

Program including ED command and ZTRACE ON directive:

```
 1  RL DIRECTORY
 2  FO DIRECTORY
 3  WS &1 "LS -BF"
 4  FO
 5  &A
 6  ED
 7  ZTRACE ON
 8  B1
 9  I
10  R DIRECTORY
11  GD/^ &/
12  GD/^.   ENTRY NAME  TYPE$/
13  GD/ D$/
14  1,$S/^.   //
15  1,$S/^DIRECTORY:  . //
16  $N
17  :C ?/^^/;M(2)
18  :D ^*/^^/S/^.*$/&   !C!B2>&/?+1;>D
19  ?+1,-1N>C
20  */^^/D!F
21  B0
22  !B1
23  W DIRECTORY
24  Q
```

```
25  ED -NBS -LL  160
26  R DIRECTORY
27  1,$S/..$/
28  1,$S/^.....
29  1,$S/!.*$//
30  1,$S/^/!H00/
31  W DIRECTORY
32  Q
33  SORT -IN SORT_CMD_SD -FF
34  F0 >SPD>LPT00
35  PR SORTED_DIR -LL 132
36  F0
```

Line Editor output:

```
 EDIT-0200-09/11/0948
**EDIT**  B1
**EDIT**  I
**INPUT** R DIRECTORY
**INPUT** GD/^ $/
**INPUT** GD/^.   ENTRY NAME  TYPE$/
**INPUT** GD/ D$/
**INPUT** 1,$S/^.   //
**INPUT** 1,$S/^DIRECTORY:  . //
**INPUT** $N
**INPUT** :C ?/^^/;M(2)
**INPUT** :D ^*/^^/S/^.*$/&!C!B2>&/?+1;>D
**INPUT** ?+1,-1N>C
**INPUT** */^^/D!F
**EDIT**  B0
**EDIT**  !B1
**EDIT**  R DIRECTORY
**EDIT**  GD/^ $/
**EDIT**  GD/^.   ENTRY NAME  TYPE$/
**EDIT**  GD/ D$/
**EDIT**  1,$S/^.   //
**EDIT**  1,$S/^DIRECTORY:  . //
**EDIT**  $N
**EDIT**  :C ?/^^/;M(2)
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1:>D
**EDIT**  ?+1,-1N>C
**EDIT**  :C ?/^^/;M(2)
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  ?+1,-1N>C
**EDIT**  :C ?/^^/;M(2)
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
```

```
**EDIT**  :D ^*/^^/S/^.*$/&!B2>&/?+1;>D
**EDIT**  ?+1,-1N>C
**EDIT**  */^^/D
**EDIT**  W DIRECTORY
**EDIT**  Q
```

LINE EDITOR PROGRAMMING DIRECTIVES

Line Editor programming directives cause conditional execution of subsequent directives, change the location of subsequent Line Editor input, and display a line of text on the user-out file. Programming directives can be in the directive input file (specified in the -IN path argument of the ED command) or an auxiliary buffer, or they can be entered through a terminal.

Each conditional directive includes one or more other Line Editor directives. The directives must be on a single line. If the specified condition exists, the subsequent embedded directive(s) are executed. The following conditions can be tested:

- Does specified line exist (Address Prefix directive)
- Does current buffer contain data (If Empty and If Data directives)
- Is current line a specified line (If Line and If Not Line directives)
- Is current line within specified lines (If Range and If Not Range directives)
- Is specified expression within specified lines (Search and Search Not directives).

Programming directives also have the following capabilities:

- Change location from which Line Editor accepts subsequent directives (Go To directive)
- Define location that can be the endpoint of a Go To directive (Label directive)
- Display a line of text on the user-out file (Type directive).

NOTE

If a directive format comprises multiple directives, the directives may be separated by spaces for readability.

ADDRESS PREFIX

ADDRESS PREFIX (?)

Execute the directives contained in the Address Prefix Line if the specified line exists in the current buffer; otherwise, they are not.

FORMAT:

```
?adr{;}directive [directive] ...
    {,}
```

ARGUMENTS:

adr

Address of the line for which the Line Editor will search.

NOTE

If adr is immediately followed by a semicolon, adr becomes the current line. If adr is immediately followed by a comma, the current line is not changed.

directive

Any Line Editor directive(s); they are executed only if the specified line is found.

Example 1:

```
?8;P
```

This Address Prefix directive specifies that if there is a line 8 in the current buffer, print the contents of that line; that line will become the current line.

Example 2:

In this example, the contents of the current buffer are:

```
(1) DEFGHI
(2) ABCXYZ
(3) ABCGGG (current line)

?/ABC/;S/ABC/DEF/
```

This Address Prefix directive designates that if there is a line that contains ABC, make that line the current line, and in that line replace each occurrence of ABC with DEF.

After this directive is executed, the current buffer will contain:

(1) DEFGHI
(2) DEFXYZ (current line)
(3) ABCGGG

GO TO (>)

Change the location from which the Line Editor accepts subsequent directives.

If the Go To directive is encountered in the buffer that is currently being executed, the Line Editor accepts subsequent directives from a specified location in that buffer. The location must have been previously defined in that buffer by a label directive.

If the Go To directive is entered interactively, only directives in the current directive line are used.

FORMAT:

```
>label
```

ARGUMENT:

label

Location to which control is transferred; the Line Editor accepts subsequent directives from this location.

If the label comprises multiple characters, they must be enclosed within parentheses; otherwise, the parentheses are optional.

Example 1:

In this example, the contents of the current buffer are:

```
(1) EAST ROCKAWAY, NY
(2) LONG BEACH, NY
(3) BRIGHTON, MASS
(4) ANDOVER, MASS
(5) HEWLETT, NY
```

Buffer 2 contains the following directives:

```
:(REPEAT)1,$P
```

Assign label REPEAT to Print directive line.

```
1,$S/MASS$/MASSACHUSETTS/P
```

Substitute each occurrence of MASS at the end of a line with MASSACHUSETTS and prints the contents of the last line in the buffer (i.e., line number 5).

NOTE

When the Line Editor searches the buffer the second time and does not find MASS at the end of a line, control returns to the previous buffer or to the terminal.

1,$S/NY/NEW YORK/>(REPEAT)

Substitute each occurrence of NY with NEW YORK and prints the contents of all lines (i.e., lines 1 through 5).

Example 2:

:A?/ABC/;S/ABC/DEF/P>A

If this directive is entered interactively, the following actions take place. The information to the right of each action indicates how the action is requested in the directive line.

Assign label A to directive line. :A

If ABC exists, take the subsequent actions. ?/ABC/

Change the current line to the location of ABC. ; preceding the substitute directive

Replace each occurrence of ABC with DEF. S/ABC/DEF/

Print the current line. P

Go to line A (i.e., reexecute the same directive line) >A

After all lines containing ABC have been acted upon (i.e., each occurrence of ABC has been replaced with DEF and the resulting lines printed), control returns to the next directive entered interactively.

IF DATA

IF DATA (#)

Execute the directives contained on the If Data directive line if the current buffer contains data; otherwise, they are not.

FORMAT:

```
#directive [directive] ...
```

ARGUMENT:

directive

Any Line Editor directive(s); they are executed only if the current buffer contains data.

IF EMPTY (^#)

Execute the directives contained in the If Empty directive line if the current buffer is empty; otherwise, they are not executed.

FORMAT:

^#directive [directive] ...

ARGUMENT:

directive

Any Line Editor directive(s); they are executed only if the current buffer does not contain data.

IF LINE

IF LINE (adr#)

Execute the directives contained on the If Line Directive line if the current line is the specified line; otherwise, they are not executed.

FORMAT:

```
adr#directive [directive] ...
```

ARGUMENTS:

adr

Address of the line being checked to see if it is the current line.

directive

Any Line Editor directive(s); they are executed only if the specified line is the current line.

IF NOT LINE (adr ^#)

Execute the directives on the If Not Line directive line if the current line is not the specified line; otherwise, they are not executed.

FORMAT:

adr^#directive [directive] ...

ARGUMENTS:

adr

Address of the line being checked to see if it is the current line.

directive

Any Line Editor directive(s); they are executed only if the specified line is not the current line.

IF RANGE

IF RANGE (adr(s) #)

Execute the directives on the If Range directive line if the current line is within specified lines; otherwise, they are not executed.

FORMAT:

$adr_1 \left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2$ #directive [directive] ...

ARGUMENTS:

adr_1

Address of the first line to be searched.

adr_2

Address of the last line to be searched.

directive

Any Line Editor directive(s); they are executed only if the current line is within addresses adr_1 through adr_2. The current line is unchanged.

IF NOT RANGE (adrs ^#)

Execute the directives on the If Not Range directive line if the current line is not within specified lines; otherwise, they are not executed.

FORMAT:

$adr_1 \left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2$ ^#directive [directive] ...

ARGUMENTS:

adr_1

Address of the first line to be searched.

adr_2

Address of the last line to be searched.

directive

Any Line Editor directive(s); they are executed only if the current line is not within addresses adr_1 through adr_2. The current line is unchanged.

Example:

```
1,10^#S/yes/no/
```

This If Not Range directive specifies that if the current line is not within lines 1 through 10, in the current line substitute each occurrence of "yes" with "no".

SEARCH

SEARCH (*)

Execute the directives on the Search directive line if a specified expression is within specified lines; otherwise, they are not executed.

FORMAT:

$adr_1 \left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2$ */regexp/directive [directive] ...

ARGUMENTS:

adr_1

Address of the first line to be searched for the regular expression. Default: Current line.

adr_2

Address of the last line to be searched for the regular expression. Default: adr_1.

NOTE

If both adr_1 and adr_2 are omitted, only the current line is searched.

regexp

String of characters for which the Line Editor is searching.

directive

Any Line Editor directive(s); they are executed only if the specified expression is within the specified addresses.

SEARCH NOT (^*)

Execute the directives on the Search Not directive line if a specified expression is not within specified lines; otherwise, they are not executed. The current line is unchanged.

FORMAT:

$adr_1 \left\{ \begin{matrix} ; \\ , \end{matrix} \right\} adr_2$ ^*/regexp/directive [directive] ...

ARGUMENTS:

adr_1

Address of the first line to be searched for the regular expression. Default: Current line.

adr_2

Address of the last line to be searched for the regular expression. Default: adr_1.

NOTE

If both adr_1 and adr_2 are omitted, the directives are executed only if the regular expression is not in the current line.

regexp

String of characters for which the Line Editor is searching.

directive

Any Line Editor directive(s); they are executed only if the specified expression is not within the specified addresses. The current line is unchanged.

LABEL

LABEL (:)

Define a location to which the Line Editor can be directed (via a Go To directive) for subsequent directives. If a Go To directive is entered interactively, only the current directive line is searched for the label. The Label directive must be specified at the beginning of a line.

FORMAT:

```
:labeldirective [directive] ...
```

ARGUMENTS:

label

Location that can be the argument value of a Go To statement; i.e., a location to which control can be transferred. If multiple characters constitute the label, they must be enclosed within parentheses; otherwise, parentheses are optional.

directive

Any Line Editor directive(s); they are executed when control passes to the specified label.

TYPE (T)

Display a line of text on the user-out file. If the optional exclamation point (!) is specified in the directive format, the next input or output will appear immediately after the printout, on the same line; otherwise, the next printouts are on subsequent lines.

FORMAT:

[!]T/text/

ARGUMENTS:

/

(Delimiter) Can be any nonblank character, but the same character must be used in each place where a delimiter is required.

text

Text to be displayed. Default: One blank line.

Example 1:

T/IDENTIFICATION NUMBER/

This Type directive prints IDENTIFICATION NUMBER. Since the optional exclamation point was not specified, subsequent input or output will appear on subsequent lines.

Example 2:

!T/IDENTIFICATION NUMBER !B2/

This Type directive prints IDENTIFICATION NUMBER and the contents of auxiliary buffer B2. If B2 contains FOR THIS YEAR, the printout will be: IDENTIFICATION NUMBER FOR THIS YEAR. Since the directive name T was immediately preceded by an exclamation point, the next input or output will appear immediately after the printout, on the same line.

PROGRAMMING CONSIDERATIONS

1. Tabbing causes embedded tab characters to be replaced with the appropriate number of spaces so that printed output on a printer or terminal has "tab stops" at character position 11 and at every subsequent 10 character positions. Tab characters can be entered into Assembly language source lines by pressing CTRL I on the terminal device while entering insert and/or substitute directive(s). CTRL I is a nonprinting tab character that has a hexadecimal value of 09. Tabbing is not apparent until a printout occurs.

2. The Line Editor uses a minimum of two temporary work files in the working directory. These files are created by the Line Editor when the Line Editor is invoked; they exist only during the current execution of the Line Editor. A minimum of 16 diskette or 8 cartridge sectors must be available in the working directory for temporary work files. Additional temporary files are created for each auxiliary buffer used; the number of temporary files is limited by the space available in the working directory.

3. If you specify a buffer name comprising more than a single character and omit the parentheses, only the first character is considered the buffer name; subsequent characters are treated as directives.

4. If a file manager error (190223, lack of space) or a physical error (190107) is encountered, use the Quit directive to exit from the Line Editor, and restart after the problem has been corrected. Attempting to recover by other means (such as the escape sequences) may cause unspecified results. If an error occurs while processing a work file (this situation is indicated by an error message that is not followed by a file name), the Line Editor may terminate processing and a fatal error message is issued.

5. An error occurs if the maximum number of lines that the Line Editor will accept in a program has been reached. Control is returned to command level.

6. Linker

Section 6
LINKER

OVERVIEW

The Linker combines object units created by the language processors (compilers and the Assembler) into a bound unit that you can then execute. During a single execution of the Linker, a single bound unit is created. A bound unit contains a root or a root with one or more overlays. The root and overlays cannot exceed the physical memory available in your system's configuration.

LINKER FUNCTIONS

The Linker functions are:

- CREATE A BOUND UNIT -- A bound unit is the output file that results from Linker execution. The bound unit is an executable program.
- BUILD A SYMBOL TABLE -- During the linking process, the Linker builds an internal symbol table used for resolving external references. You can define a symbol within an object unit or by using Linker directives defined later in this section.

- PRODUCE A LISTING -- The linker listing has two parts, a dynamic part and a static part.

  - The dynamic part is generated continuously and contains information about each object unit linked, the directives used, and a summary.

  - The static part is produced in response to the MAP or MAPU directive and is a picture of the state of the link when the MAP(U) directive is processed. It lists the external definitions currently in the symbol table and the undefined external references, if any exist.

  During the link process, summary information about the bound unit is automatically output to a list file. The format of this information is:

```
**********
ROOT  TESTP2
* HIGHEST OVERLAY NUMBER:  2
 LAF
**********
** CMMN  DATA              BASE:  000000  START:  000000  .F..  HIGH:  000011
   ROOT  TESTP2            BASE:  000000  START:  000000  ..U.  HIGH:  00003F
 † OVLY  OVLNO      # 0001 BASE:  00003F  START:  00003F  ....  HIGH:  000060

KEY:  S=SHAREABLE; F=FLOATING; I=CONTAINS AN IMA; U=CONTAINED AN UNDEFINED
      REFERENCE; ->=IN-LINE DIRECTIVE; [...]=EMBEDDED DIRECTIVE

**********
SIZE OF ROOT AND FIXED OVERLAYS:  000060
LAST BU RECORD NUMBER:  4
**********
LINK DONE
**********
```

- RESOLVE EXTERNAL REFERENCES -- The Linker resolves addresses or values of external symbol references in object units being linked. To do this, the Linker uses external definitions found in the object units or declared by the LDEF or VDEF directives. (LDEF and VDEF are described fully later.) When a bound unit is linked, the unresolved external references are listed at the end of the link map. If unresolved external references exist at the end of the list, an error message is displayed on the error-out file, usually the terminal.

---

* Each control interval (logical record on the bound unit file) has a size of 256 bytes (128 words).

**This line only appears if common has been gathered into one contiguous area. The -R ECL parameter was specified.

†This line repeated for each overlay.

LINKER DIRECTIVE CATEGORIES

The Linker directive set may be grouped into eight functional categories described in the following paragraphs.

Specifying Object Unit(s) to be Linked

LINK, LINKN, LINKnn, and LINKO designate that one or more specified object units are to be linked. Object units specified in LINK directives are not linked immediately; their names are put into a link request list. Once a directive has been entered which requires that all preceding link requests are honored, linking begins. Specified object units in the primary input directory are linked before specified object units in the secondary input directory; within each directory, the object units are linked in the order in which they were requested.

LINKN causes the Linker to link object units already named in the link request list, and then to link object units specified in the LINKN directive in the order in which they were requested.

LINKO performs in the same manner as LINKN, except that all embedded directives in the named object unit(s) are ignored by the Linker. LINKnn is a special form of LINKN used to perform selective linking.

Specifying Location(s) of Object Unit(s) to be Linked

Object units to be linked must be in at least one directory. The Linker searches the primary directory first, proceeding to the secondary, and tertiary directories if they exist. When the Linker is loaded into memory, the primary directory is the working directory, and there are no other directories. The directives used to specify location(s) of object unit(s) to be linked are listed below.

IN is used to designate a different directory as the primary directory.

LIB is used to designate a directory as the secondary directory.

LIB2 is used to designate the third directory to be searched.

LIB3 is used to designate the fourth directory to be searched.

LIB4 is used to designate the fifth directory to be searched.

LSR is used to request a list of the directories in the order in which they are to be searched.

RETURN may be used in an INCLUDE file. It acts like an EOF; it returns the Linker to USER-IN. Return provides a way to signal EOF from a user-terminal.

INCLUDE is used to cause the Linker to accept Linker directives from the indicated pathname rather than USER-IN. When the Linker encounters EOF or RETURN in the INCLUDE file, it returns to seek directives from USER-IN.

Creating a Root and Optional Overlay(s)

START is used to specify the relative address at which the root or overlay will begin executing when it is loaded into memory by the Loader.

BASE is used to define relative addresses (within the bound unit) for subsequent object units to be linked. Note that when the lowest address of a root or overlay has been established (i.e., an object unit has been linked), it is invalid to define a lower BASE address within the root or overlay.

OVLY is used to name the nonfloatable overlay that follows, and designates the end of the preceding root or overlay.

FLOVLY is used to name the floatable overlay that follows, and designates the end of the preceding root or overlay.

CC permits a COBOL program that used CALL and CANCEL statements to call overlays by their names.

IST is used to identify the beginning of initialization code in the root.

SHARE is used to designate that the bound unit is sharable within the task group.

QUIT is used to designate that the last Linker directive has been entered. Execution of the Linker terminates after the bound unit has been created.

FLOATB6 is used to suppress certain error checking on local common references when the -R Linker argument has not been specified. Local common references are relocated as if B6 pointed to the base of the containing overlay.

STACK is used to specify the size of the stack area.

GSHARE is used to specify that the bound unit is globally sharable.

SEG is used to specify that the subsequent object unit is to be linked into one or two physical segments in memory.

SYS is used to designate that the bound unit can be loaded into the system area as part of the system.

LINK, LINKN, and LINKO are used to specify those object units to be linked. The order in which specified object units are linked, and when they are linked, is determined by the link directive used.

Producing Link Map(s)

LDEF is used to assign a relative location to an external symbol. When a symbol is defined, its definition is put into the Linker symbol table so that it can be used to resolve references to the symbol during linking.

VDEF is used to assign a value to an external symbol. When a symbol is defined, its definition is put into the linker symbol table so that it can be used during linking to resolve external references.

MAP is used to create a map that lists both defined and undefined symbols.

MAPU is used to create a map that lists the undefined symbols only.

-V ECL option will automatically list symbols as they are defined.

Defining External Symbols

EDEF permits definitions in the Linker symbol table to be made part of the bound unit so that they are available to the Loader at execution time.

OVERLAYTABLE is used to put a value definition containing the name of each overlay and its overlay number in the bound unit symbol table.

COMM is used to define a labeled common block. A symbol can be defined as a relative location or value by specifying the LDEF or VDEF directive, respectively. The symbol's definition is then put into the symbol table by the Linker.

VAL is used to specify a value definition at LINK time. This value is equivalent to the difference between two external location definitions.

Protecting or Purging Symbol(s)

CPROT and CPURGE are used to protect and remove symbols associated with labeled common blocks.

PROT and PURGE are used to protect and remove symbols and object unit names from the symbol table. PROT prevents certain symbols and/or object unit names from being removed from the symbol table. Symbols are protected if they identify a specified address or an address within a specified range; object unit names are protected if they are equated to a specified address or an address within a specified range.

PURGE is used to remove from the symbol table unprotected symbols that define a specified address or an address within a specified range, and/or object unit names equated to a specified address or an address within a specified range.

VPURGE is used to remove a specified value definition from the symbol table.

Reloading After System Failure

RR indicates a sharable bound unit can be reloaded after a system failure into locations other than those it occupied at checkpoint.

Terminating the Linker

QUIT is used to terminate the Linker. If a bound unit is being created, execution of the Linker terminates after the bound unit has been created. If no bound unit is being created, QUIT terminates execution of the Linker.

Subsections that follow include full information on:

Loading the Linker -- Describes the Linker command used to call the Linker and initiate Linker processing.

Entering Linker Directives -- Describes the format line used to enter directives.

Linker Directive Set -- Provides an alphabetic listing of the Linker directives. Detail descriptions of each directive and examples of use are provided.

Linker Procedures -- Describes frequently used Linker procedures.

LOADING THE LINKER

The command LINKER is used to load the Linker.

After the Linker is loaded, a message is sent to the error-out file indicating the version. The message format is:

LINKER-nnnn-mm/dd/hhmm

where nnnn is a release identification, mm/dd is the month and day the Linker component was linked, and hhmm the time (hour, minutes) at which that link took place.

FORMAT:

LINKER bound-unit-path [ctl_arg]

ARGUMENTS:

bound-unit-path

Pathname of the bound unit file. The pathname can be simple, relative, or absolute and must be preceded by a space. If the specified file already exists, the existing information in the file is deleted and replaced with the new bound unit. The bound unit pathname must be specified. It may be up to 57 characters in length. The format of the bound unit file is relative.

ctl_arg

Control arguments; none or any number of the following control arguments can be entered, in any order:

{-IN} path
{-I }

Pathname of the device disk, card reader, operator's terminal, or another terminal that will read Linker directives.

Default: Device specified in the in_path argument of the Enter Group Request command.

When this argument is specified, the prompt character will not appear.

-PT

If the -IN argument is not specified, -PT can be specified to produce a prompt character on the user terminal. A prompt character is issued only if -PT is specified.

{-COUT }list-path-name
{-COUTA}

Designate the list file. The list file can be sent to a disk, another terminal, or a printer. The list-path-name is associated with this list file. If -COUT is not specified, the list-path-name has a default value of bound-unit-name.M in the working directory. If -COUTA is specified, the listing is appended to the specified file.

Error messages are written to the error-out file and the list file. Linker error mesages are described in the System Messages manual.

{-SIZE}nn
{-SZ }

nn designates the maximum number of 1024-word (1K) blocks of memory available for the Linker symbol table; nn must be from 1 to 64. At least 1024 words must be available.

Default: 2

-W

Specify that the Linker work files are to be saved.

Default: Linker work files are automatically released by the Linker upon Linker termination.

-R

Designate that a bound unit is to be created, where all data areas defined as common are separated from all other code. Required for sharable bound units containing common data areas.

{-VERBOSE}
{-V   }

Cause externally defined symbols to be written on the list file as they are defined. Eliminates the need for the MAP directive.

-NOMAP

Suppress the list file.

{-SYMBOL}
{-SYM }

Specify that a debugger information file is to be created. This file is used for symbolic debugging. The name of the file is buname.v. This option should only be used for FORTRANA or COBOLA programs.

Example:

```
LINKER MYPROG -IN MYDISK>CNL -COUT !LPT00 -SIZE 6
```

This LINKER command loads the Linker and designates the following:

1. Bound unit will be a relative file named MYPROG in the working directory.

2. Linker directives will be entered through disk file MYDISK>CNL.

3. List file goes to a line printer (configured as LPT00), rather than to a variable sequential file named MYPROG.M in the working directory.

4. The symbol table will use a maximum of 6K words of memory.

NOTE

LPT00 must have been previously defined in the DEVICE configuration directive at system generation time.

ENTERING LINKER DIRECTIVES

Linker directives are entered through the directive input device. Several directives can also be embedded in Assembly language CTRL statements. They are: LINK, LINKN, LINKO, SHARE, EDEF, SYS, COMM, LSR, and VAL.

Linker directives consist of a directive name or a directive name followed by one or more arguments. Each directive name may be preceded by zero or more blank spaces. If one or more arguments are to be specified in a Linker directive, the directive name must be immediately followed by one or more spaces.

Multiple directives can be entered on a line by specifying a semicolon (;) after each directive, except for the last directive on the line.

The last directive on a line can be followed by a comment; to include a comment, specify a space and a slash (/) after the last directive and then enter the comment.

FORMAT:

directive [Δargument$_1$] [argument$_2$] [Δ/comment]

If the directive input device is the operator's terminal or another terminal, press RETURN at the end of each line (i.e., at the end of the comment, or at the end of the last directive if there is no comment). There is no continuation between lines; the values associated with a single directive cannot be continued on a second line.

If an error occurs when entering a directive, an error message is written to the error-out file. Linker error messages are described in the System Messages manual. Determine what caused the error, and reenter the directive correctly. If multiple directives are entered on a line and an error occurs, the error does not affect the execution of previously designated directives. The directive that caused the error and subsequent directives on that line are not executed.

LINKER DIRECTIVES SET

Linker directives are described in alphabetic order on the following pages. Examples are provided to illustrate directive usage.

BASE

BASE

Define the relative link address within the bound unit for subsequent object units to be linked. At load time, all addresses are relative to the beginning of available memory (relative 0) in the memory pool of the task group. When a task group is created, you specify the memory pool into which its bound units are to be loaded.

Unless BASE directives specify otherwise, the root will be linked, by default, at relative 0, and subsequent object units are linked at successive relative addresses. A BASE directive can be used at any point during linking to change the relative locations of the root, overlays, or individual object units. A floatable overlay always begins at relative 0; therefore, in a floatable overlay, BASE can be specified only <u>after</u> the first LINK, LINKN, or LINKO directive. A BASE directive can specify a previously used or defined location, or an address relative to the beginning of the available memory.

If unprotected symbols define locations that are equal to or greater than the location designated in the BASE directive, those symbols are removed from the symbol table.

The BASE directive cannot be embedded in Assembly language control statements.

FORMAT:

```
        ( $                                  )
        ( %                                  )
        ( X'address'                         )
        ( =object-unit-name                  )
BASE    ( xdef   {±}  X'offset'              )
        ( #                                  )
        ( *ODD                               )
        ( *EVEN                              )
        ( *X'offset'                         )
```

ARGUMENTS:

$

Next location after the highest address of the linked root or previously linked nonfloatable overlay.

$

Highest address+1 ever used in the linked root or any previously linked nonfloatable overlay.

X'address'

A one- to five-character hexadecimal address enclosed in single quotation marks and preceded by X. The specified address is relative to the beginning of the root (relative 0).

=object-unit-name

Specified object unit's base address; the subsequent root, overlay, or object unit will be linked at the same relative address as the specified object unit, which must have already been linked. Furthermore, the object unit name must still exist in the symbol table (i.e., it has not been purged).

xdef [{±} X'offset']

Address of any previously defined (non-common) external symbol. If an offset is specified, it must be a hexadecimal integer with an absolute value less than 8000 (32768 decimal).

#

The current address.

***ODD**

Establishes base address at current address, if it is odd; if it is even, base address is converted to current address+1.

***EVEN**

Establishes base address at current address, if it is even; if it is odd, base address is converted to current address+1.

*X'offset'

Establish base address at the next location whose rightmost hexadecimal characters equal the offset. (Where the offset is a hexadecimal integer of four or fewer characters.)

Default: $ with the following exceptions:

Root - 0
Floatable overlay - 0

Example:

```
LINKER TEXT -COUT !LPT00

START TEXTEN -PT          Designate address where execution
                          begins when the root is loaded.

LINKER-2100-08/27/1042    Linker identification message.
L?                        Linker prompt.

IST INIT                  Define INIT as the beginning of
                          initialization code.

L?
LINK OBJ1,OBJ2            Request that OBJ1.O and OBJ2.O be
                          linked.

L?
MAP                       Cause OBJ1.O and OBJ2.O to be
                          linked, and produce a link map.

L?
OVLY ABLE                 Designate end of the root, and that
                          a nonfloatable overlay named ABLE
                          immediately follows.  The Linker
                          assigns the number 00 to this
                          overlay.

L?
BASE =OBJ2                Subsequent object unit(s) consti-
                          tuting overlay ABLE will be linked
                          starting at the base address of the
                          object unit OBJ2.O; this address
                          can be determined from the map.
                          Unprotected symbols that define
                          locations equal to or greater than
                          the address of OBJ2 are removed
                          from the symbol table.
```

```
L?
LINK OBJ5
```
Request that OBJ5.O be linked.

```
L?
MAP
```

```
L?
LINK OBJ6
```
Request that OBJ6.O be linked.

```
L?
OVLY FOX
```
Designate the end of the above overlay, and specifies that a nonfloatable overlay named FOX immediately follows. The Linker assigns the number 01 to this overlay.

```
L?
BASE $
```
Subsequent object unit(s) constituting the overlay named FOX will be linked starting at one location higher than the ending address of OBJ6.O. This is the default BASE address, so BASE $ need not be specified.

```
L?
LINK OBJA,OBJB
```
Request that OBJA.O and OBJB be linked.

```
L?
MAP
```
Request the status of the symbol table and causes OBJA.O and OBJB.O link requests to be honored, i.e., linked.

```
L?
OVLY ZEBRA
```
Designate end of above overlay 01 and names subsequent nonflotable overlay. The Linker assigns the number 02 to this overlay.

```
L?
BASE X'1105'
```
Designate that subsequent object units constituting overlay ZEBRA will be linked starting at relative location 1105.

```
L?
LINK OBJC                 Object unit OBJC.O will be linked
                          starting at relative location 1105.
L?
LINK OBJD                 Request that OBJD.O be linked.

L?
MAP

L?
FLOVLY FLOAT              Designate end of above overlay, and
                          that a floatable overlay named
                          FLOAT immediately follows.  The
                          Linker assigns the number 03 to
                          this overlay.  This overlay will be
                          linked starting at the default base
                          address of 0.

L?
LINK OBJE                 Request that OBJE.O be linked.

L?
MAP

L?
QUIT

ROOT TEXT
LINK DONE
RDY:
```

Figure 6-1 illustrates use of BASE directives in a bound unit that consists of a root and overlays. This example assumes that the bound unit being created will be executed as part of task group A1, and memory pool AA will be used by this task group. Figure 6-1 also shows memory pool AA's location in memory relative to the system pool and another pool. The object units specified by the following directives are loaded into memory pool AA during execution of the bound unit.

Figure 6-2 shows the configuration of memory pool AA at different times during execution. Note that OBJ.C of the root is overlayed by overlay ABLE and that overlay FOX is partially overlayed by overlay ZEBRA. Also note that overlay FLOAT is positioned by the Loader and is not necessarily at the location shown in the diagram.

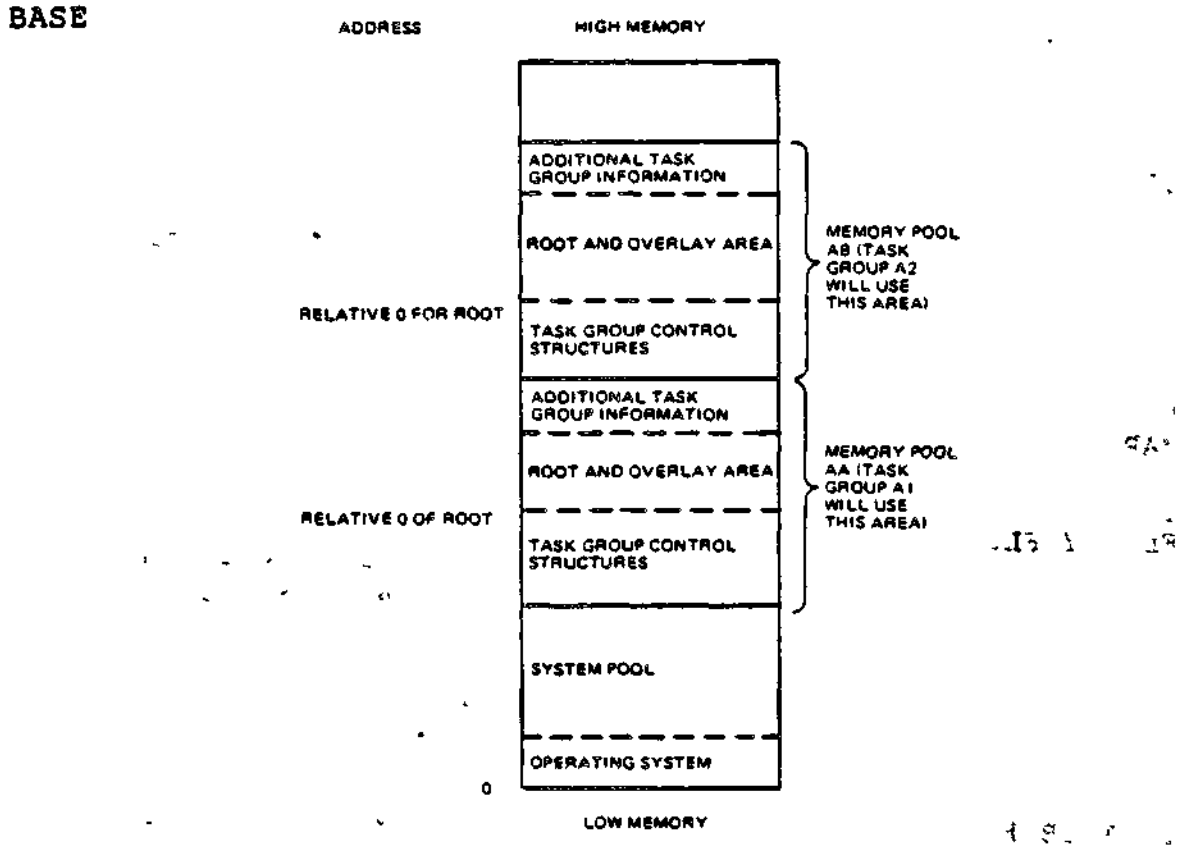


Figure 6-1. Relative Location in Memory of Memory Pool AA

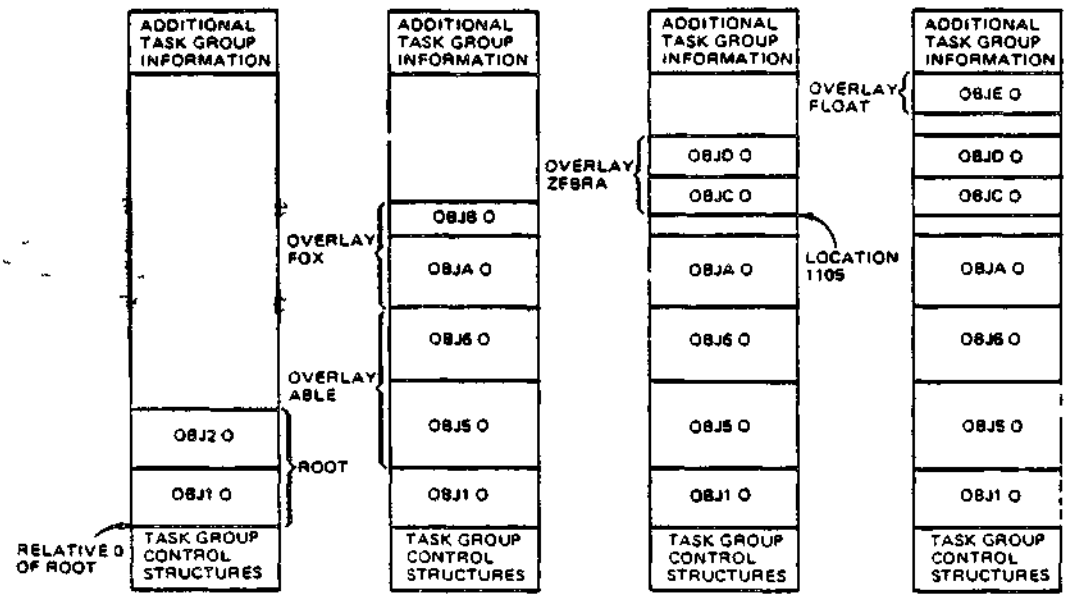


Figure 6-2. Overlays in Memory Pool AA

CC (CALL_CANCEL)

Place each overlay name and its associated Linker-generated overlay number into the bound unit attribute table so that the COBOL program can call/cancel overlays by name. This directive is used when linking COBOL programs that contain CALL/CANCEL statements that invoke overlays.

To support the CALL/CANCEL facility, two object units are required: ZCCEC.O and ZCCECO.O. ZCCEC will be automatically linked into the root, with ZCCECO linked as an overlay. These object units require only the CC link directive.

The CC directive must be specified before the first LINK, LINKN or LINKO directive in the root, and cannot be embedded in Assembly language control statements.

FORMAT:

CC

COMMON

COMMON

Define a labeled "common" area of a specified size. It may not be embedded in source code.

FORMAT:

```
{COMMON}
{COMM  }  symbol,X'size'
```

ARGUMENTS:

symbol

Identifies the external symbol to be treated as common.

X'size'

Size is specified as a one- to four-character hexadecimal number bound by single quotes and preceded by the letter X.

CPROT

Prevent specified common symbols from being removed from the symbol table.

This directive cannot be embedded in Assembly language control statements.

FORMAT:

CPROT symbol

ARGUMENT:

symbol

Name of the external symbol that is to be protected. The symbol must be specified in the COMM directive or defined as common during Assembly or compilation.

CPURGE

CPURGE

Remove an unprotected common symbol from the symbol table.

FORMAT:

```
CPURGE symbol
```

ARGUMENT:

symbol

Identify the external symbol to be removed from the symbol table. The symbol must have been defined as common.

EDEF

Make a symbolic definition available to the Loader at load time.

When EDEF is specified, the symbol's definition must already be in the symbol table.

Secondary entry points of bound units, whose code is to execute under control of a task, must be defined in an EDEF directive. This includes secondary entry points of overlays and the root entry point when it will be explicitly used in a Create Group command. The start address of the root and of each overlay is placed by the Linker in the bound unit attribute table and does not need an EDEF definition. The bound unit attribute table is part of the bound unit.

If a bound unit is memory-resident, symbols (entry points and references) can be defined by EDEF so that they can be invoked by any bound unit loaded by the system. At system configuration time, when the resident bound units are loaded using the LDBU system configuration directive, these symbols are placed in the system symbol table. When the Loader loads other bound units that contain unresolved references, it tries to resolve them with the list of symbols defined for resident bound units.

If the bound unit is not memory-resident, the symbols in the attribute table of the bound unit are meaningful only as definitions of secondary entry points. Although shared bound units can be in the address space of more than one task group, the bound unit attribute table is available to the Loader only when the bound unit is being loaded. Unresolved references in any bound unit will be resolved only to symbols defined in attribute tables of resident bound units.

The EDEF directive can be embedded in Assembly language control statements.

FORMAT:

$$\left\{\begin{matrix}\text{EDEF}\\ \text{EF}\end{matrix}\right\} \quad \text{symbol}_1\ [,\text{symbol}_2\ ]$$

ARGUMENTS:

$symbol_1$

Any external definition comprising one to six characters. The symbol must have been previously defined; it can name a root or overlay once the root or overlay has been linked. If the symbol was multiply defined, the first definition will be used.

$symbol_2$

Name of the symbol incorporated in the bound unit comprising 1 to 12 characters. If symbol is not specified, the name of the symbol placed in the bound unit is that specified by symbol .

Example:

LINKER MYPROG -PT	Load the Linker. The bound unit named MYPROG will be created on the working directory. The list file MYPROG.M is also created on the working directory.
LINKER-2100-08/27/1042 L? >	
LINK A	
L? LINKN B	
L? MAP	
L? EDEF B	B is a symbol defined as an external location or value in B.O.
L? LDEF SYM,X'1234'	Assign relative location 1234 to external symbol named SYM.
L? OVLY FIRST	Designate end of root, and names non-floatable overlay that immediately follows.
L? LINK X,Y	

```
L?
EDEF SYM

L?
QUIT                    Designate that the last Linker direc-
                        tive has been entered.  Execution of
                        the Linker terminates after the bound
                        unit has been created.

ROOT MYPROG
LINK DONE
RDY:

LINKER PROG 2-COUT !LPT00 -SIZE 2 -PT

                        Load the Linker; the bound unit to be
                        created is named PROG2.  The list file
                        is the printer.  The symbol table is a
                        maximum of 2K words of memory.

LINKER-2100-08/27/1042

L?
BASE X'2222'            Subsequent object units will be loaded
                        into memory starting at the relative
                        address 2222.

L?
LINKN W                 Request that object unit W.O be
                        linked.

L?
MAP                     Produce a link map; in this map, it is
                        determined that object unit W.O
                        contains an unresolved reference to
                        the symbol SYM, which was defined in
                        the root of the bound unit MYPROG.

L?
QUIT
ROOT PROG 2
***BU CONTAINS UNRESOLVED REFERENCES
LINK DONE
```

This example illustrates use of EDEF directives in bound units.

If MYPROG is loaded into memory via an LDBU configuration directive, when the Loader loads PROG2 the Loader will resolve the unresolved reference in PROG2 to the symbol SYM, which was defined in the root of MYPROG.

NOTE

An EDEF directive cannot be entered on the directive line in which the object unit is specified. For example, if the symbol TAG is defined in object unit A, the following directive line is <u>not</u> allowed: LINK A;EDEF TAG.

FLOATB6

Suppress certain error checking on local common references when the -R argument has not been used. The directive tells the Linker that the user will manage $B6 himself and causes each local common reference to be relocated as if the $B6 pointed to the base of the floatable or fixed overlay containing the reference. Normally, $B6 is set by the system to the base of the fixed (root and fixed overlay) area, and local common references within floating overlays would be invalid.

Before using this directive, consult with the person responsible for system building and determine available system memory.

This directive must be specified before the first object unit containing a local common reference is linked.

FORMAT:

FLOATB6

FLOVLY

Assign the specified name and a number to the floatable overlay that immediately follows, and designate the end of the preceding root or overlay. The characteristics of floatable overlays are described at the end of this directive.

FLOVLY must be specified as the first directive of each floatable overlay.

The Linker assigns a two-digit number to each overlay. Overlays are numbered sequentially in ascending order; the first overlay is 00.

FORMAT:

FLOVLY name

ARGUMENTS:

name

Name of the floatable overlay that immediately follows. The overlay name must consist of one to six alphanumeric characters; the first character must be alphabetic.

Example:

LINKER BU -PT	Load the Linker and designate BU as the bound unit name.
LINKER-2100-08/27/1042	
L? LINK A,B	
L? MAP	Produce a link map.
L? FLOVLY GR	Designate the end of the root that consists of object units A.O and B.O, and specify that the next overlay is a floatable overlay named GR. The Linker assigns the number 00 to this overlay.

```
L?
LINK X,Y; MAP
L?
FLOVLY BR
```

Designate the end of floatable overlay GR and designate that the floatable overlay that immediately follows is named BR. The Linker assigns the number 01 to this overlay.

```
L?
LINK R6

L?
MAP

L?
QUIT
ROOT BU
LINK DONE
```

NOTE

External location definitions defined within a floating overlay will automatically be purged at the end of the overlay, because they cannot be referenced from outside the overlay.

A floatable overlay must have the following characteristics:

1. External location definitions in the overlay are not referenced by the root or any other overlay.

2. There cannot be external references between floatable overlays.

3. The overlay does not contain external references that are not resolved by the Linker.

4. The overlay must be linked after all desired nonfloatable overlays have been linked.

5. The overlay cannot contain P+DSP references to any other overlay in the root.

6. The overlay cannot contain IMA (immediate memory address) referenes within itself.

7. There can be IMA references (with or without offsets) to locations in the root or any nonfloatable overlay.

GSHARE

GSHARE

Indicate that the bound unit is globally sharable, which means that the program is sharable between groups and the root is always loaded into the system memory pool. This directive should not be used if a SHARE directive would suffice. System performance may be affected if this directive is misused. Floatable overlays are loaded into user space and are not sharable unless overlay area tables (OATs) are used.

Before using this directive, consult with the person responsible for system building and determine available system memory.

NOTE

Nonsharable bound units (linked without SHARE or GSHARE) are always loaded into the user's memory pool.

FORMAT:

GSHARE

Designate a different directory as the primary directory. (The primary directory is the first directory that the Linker searches for the specified object unit(s) to be linked.) This directive permits the linking of object units that are in directories other than the default primary directory or secondary directory (if any). If the IN directive is not specified, the working directory is the primary directory. (The secondary directory is designated in the LIB directive.)

NOTE

The IN directive must be specified before the first LINK, LINKN, or LINKO directive that requests the linking of an object unit that is in the specified directory.

The specified directory remains the primary directory until another IN directive is entered. If the primary directory is changed via an IN directive and at a later time you want the task group's working directory to be the primary directory, enter the IN directive and omit the pathname.

FORMAT:

IN [path]

ARGUMENTS:

[path]

Pathname of the directory being designated as the primary directory. The pathname can contain a maximum of 57 characters. A simple, relative, or absolute pathname can be specified (methods of designating pathnames are described in Section 3 of this manual). If path is omitted, the working directory becomes the primary directory.

NOTE

The IN directive can not be embedded in Assembly language control (CTRL) statements.

Example 1:

```
IN ^DIR>PRIM
```

This directive designates that ^DIR>PRIM is the primary directory.

Example 2:

`LINKER OUTPUT -PT`	Load the Linker; a bound unit named OUTPUT will be created on the working directory.
`LINKER-2100-08/27/1042`	
`L?`	
`LINKN X`	Request the linking of object unit X.O; X.O is in the working directory.
`L?`	
`IN ^NEW>PRIM`	Designate that ^NEW>PRIM is now the primary directory.
`L?`	
`LINKN A,C`	Request the linking of object unit A.O and C.O in the primary directory. ^NEW>PRIM>A.O is the pathname of A.O and ^NEW>PRIM>C.O is the pathname of C.O, as expanded by the Linker.
`L?`	
`IN`	Designate that the primary directory is now the working directory.
`L?`	
`LINKN Y`	Request the linking of object unit Y.O, in the working directory. WORK>CURR>Y.O is the pathname of Y.O, as expanded by the Linker.
`L?`	
`MAP, QUIT`	

```
ROOT OUTPUT
LINK DONE
```

This example illustrates use of the IN directive in conjunction with directives that request the linking of object units. Assume the primary directory is the working directory, whose relative pathname is WORK>CURR; object units X.O and Y.O, are in the working directory. A.O and C.O are not in the working directory.

INCLUDE

INCLUDE

Accept directives from a file other than user-in or the file specified in the -IN ECL argument. When the Linker encounters an end of file or a RETURN directive in the file specified by the INCLUDE directive, it again seeks directives from the previously active file. If used, the INCLUDE directive must be the last directive entered on a line.

FORMAT:

INCLUDE [path]

ARGUMENT:

[path]

Pathname of the file from which the Linker directives are to be read. A simple pathname can be up to 12 characters in length; an absolute pathname can be up to 57 characters in length.

Example:

INCLUDE !READER

This directive causes the Linker to accept directives from the card reader.

NOTES

1. The directive file specified by the INCLUDE directive cannot contain an INCLUDE directive.
2. The INCLUDE directive cannot be embedded in Assembly language control statements.

IST

Identify the beginning of the initialization start address in the root. Initialization code is to be executed once, immediately after the root is loaded at system boot time. After the initialization code is executed, its space can be made available for overlays. The IST directive must be associated with an LDBU directive that specifies an Initialization Subroutine Table (IST). LDBU, a CLM directive, is explained in the MOD 400 System Concepts manual. IST does not execute unless the bound unit is specified in an LDBU directive.

FORMAT:

$$\left\{\begin{matrix}\text{IST} \\ \text{IT}\end{matrix}\right\} \text{external symbol}$$

ARGUMENTS:

external symbol

Symbol specified by label in IST section of LDBU.

NOTE

The IST directive cannot be embedded in Assembly language control statements.

LDEF

LDEF

Assign a relative location to an external symbol. A symbol should be defined only once, either as a location or as a value. When a symbol is defined, its definition is put into the Linker symbol table so that it can be used to resolve references to the symbol during linking. When a symbol defined as a location is no longer used, its symbol table entry can be cleared by specifying the PURGE directive. PURGE has no effect if a PROTECT (PROT) directive was previously specified.

FORMAT:

```
{LDEF}
{LF  }   symbol,  {$                                          }
                  {%                                          }
                  {X'address'                                 }
                  {=object-unit-name                          }
                  {                                           }
                  {xdef   [{±} X'offset']                     }
                  {#                                          }
```

ARGUMENTS:

symbol

One to six alphanumeric characters.

$

Next location after the highest address of the linked root or previously linked nonfloatable overlay.

%

Highest address+1 ever used in the linked root or any previously linked nonfloatable overlay.

X'address'

Hexadecimal address comprising one to five integers enclosed in single quotation marks and preceded by X. The specified address is relative to the beginning of available memory (relative 0) in the memory pool.

=object-unit-name

Specified object unit's base address.

xdef[± X'offset']

Address of any previously defined external symbol. If an offset is specified, it must be a hexadecimal integer with an absolute value less than 8000 (32768 decimal).

*

The current address.

NOTE

The LDEF directive cannot be embedded in Assembly language control (CTRL) statements.

Example:

```
LINKER BOUND -PT          Load the Linker and designate
                          BOUND as the bound unit name.

LINKER-2100-08/27/1042

L?
LINK A, B, C

L?
MAP

L?
LDEF SYM, X'1234'         SYM assigned relative location 1234.

L?
OVLY FIRST                Designate end of root and name first
                          nonfloatable overlay.

L?
LINK R; MAP

L?
LDEF QUIZ,=C              QUIZ assigned base location of the
                          previously linked object unit named
                          C.O.

L?
OVLY SECOND
```

```
L?
LINKN D; LINK F; MAP

L?
LDEF NEW,SYM
```

NEW assigned same location as the symbol SYM, which was defined in the root; i.e., NEW is assigned relative location 1234.

```
L?
OVLY NEXT

L?
BASE X'1300'

L?
LINK W,X; MAP

L?
LDEF ANY,$
```

ANY assigned next location after highest address of the previously linked nonfloatable overlay, SECOND.

```
L?
OVLY THIRD

L?
LINK Z

L?
LINK Q; MAP

L?
LDEF FIND,%
```

FIND assigned next location after highest address of the root or <u>any</u> previously linked nonfloatable overlay. (A previous nonfloatable overlay was named SECOND; if it ended at location 1566 and this is the highest ever reached during the linking of object units constituting this bound unit, FIND would be assigned location 1567.)

```
L?
QUIT
ROOT BOUND
LINK DONE
RDY:
```

This example illustrates the use of each format of the LDEF directive.

LIB OR LIB1

LIB or LIB1

Designate a directory as the secondary directory. This directory permits the linking of object units that are in directories other than the primary directory. If an object unit specified in the LINK, LINKN, or LINKO directive cannot be found in the primary directory, the Linker searches the secondary directory.

If LIB is not specified, there is no secondary directory; the Linker searches only the primary directory.

The specified secondary directory remains in effect until the LIB directive is respecified with a different directory name, or without any directory name.

NOTES

1. The LIB directive must be specified before the first LINK, LINKN, or LINKO directive that requests the linking of an object unit in the secondary directory.

2. This directive cannot be embedded in Assembly language control (CTRL) statements.

FORMAT:

```
LIB [path]
```

ARGUMENT:

[path]

Pathname of the directory being designated as the secondary directory. A relative or absolute pathname can be specified. (Methods of specifying pathnames are described in Section 3.) If path is omitted, no search of that secondary directory is made.

Example 1:

```
LIB DIR>SECND
```

This directive designates that DIR>SECND is the relative pathname of the secondary directory.

Example 2:

LIB DIR>SECND	Designate that DIR>SECND is the relative pathname of the secondary directory.
LINK B	Request the linking of object unit B.O; B.O resides in the primary directory.
LINK A	Request the linking of object unit A.O; A.O resides in the primary directory.
LINK W	Request the linking of object unit W.O; W.O resides in the secondary directory. DIR>SECND>W.O is the full pathname of W.O, as expanded by the Linker.

This example illustrates usage of a secondary directory that contains unit W.O, Y.O, and Z.O.

All specified object units in the primary directory are linked first; then all specified object units in the secondary directory are linked, and so on. To cause object units to be linked in a specific order, the LINKN or LINKO directive must be used.

LIB

LIB {2 / 3 / 4}

Designate directories as the third, fourth, or fifth directory. If an object unit specified in the Linker directive cannot be found in the primary or secondary directory, then the third directory is searched and so on.

The specified directories remain in effect until another LIB2, LIB3, LIB4 statement is given.

NOTES

1. The LIB2, LIB3, LIB4 directive must be specified before the first LINK, LINKN, or LINKO directive that requests the linking of an object unit that is in one of these directories.

2. The LIB2, LIB3, LIB4 directive cannot be embedded in assembly language control statements.

FORMAT:

{LIB2 / LIB3 / LIB4} [path]

ARGUMENT:

[path]

Pathname of the third, fourth, or fifth directory to be searched (if LIB is specified) if the object unit specified in a Linker directive is not found in the preceding directories. A simple, relative, or absolute pathname can be specified. If path is omitted, the specified directory (2, 3, or 4) is removed from the list of directories to be searched by the Linker.

LINK

Link one or more specified object units. Each specified object unit name is put into the link request list. The object units are linked when the first subsequent directive other than LINK or START is encountered. When this occurs, the Linker searches the primary directory and links the specified object units in the primary directory in the order that they were requested. If all of the object units are not found and there is a secondary directory, the Linker searches the secondary directory and links specified object units found there, in the order that they were requested. If there is a copy of an object unit in both the primary and secondary directory, the copy in the primary directory is linked.

The order that object units are linked is important for the following reasons: (1) it determines which object units will be in memory when parts of the root or overlay are overlaid (2) within the root and each overlay, the first start address encountered by the Linker (either in an END statement or a START directive) is used as the start address for that root or overlay.

During each execution of the Linker, at least one LINK, LINKN, or LINKO directive must be entered for each root or overlay. Multiple LINK directives can be specified within a single root or overlay. If LINK and/or LINKN and/or LINKO directives request that the same object unit be linked more than once within a single bound unit, only the first request is honored, unless the object unit name has been purged.

LINK directives can be embedded in Assembly language control statements; the specified object unit(s) are added to the end of the current link request list. See "LINKN Directive" and "LINKO Directive" for the order that object units are linked if there are embedded LINK directives and/or LINKN and/or LINKO directives.

FORMAT:

```
LINK  obj-unit1[,obj-unit2]...
LK
```

ARGUMENTS:

obj-unit

Name of an object unit to be linked. An object unit name consists of one to six characters, each of which must be an alphanumeric character or a dollar sign ($), a period (.), or an underscore (_). If multiple object units are specified, they are linked in the most efficient order. The first character must be a letter or a dollar sign ($).

NOTE

For a COBOL segmented program, an object-unit name can be up to eight characters. See the Advanced COBOL Reference manual for further details.

LINKN

Link object units in the exact order specified.

If directives designate that an object unit be linked more than once within a single bound unit, only the first request is honored, unless the object unit name has been purged.

During each execution of the Linker, at least one LINKN, LINK, or LINKO directive must be specified for each root or overlay.

Multiple LINKN directives can be specified within a single root or overlay.

LINKN directives can be embedded in Assembly language control (CTRL) statements; the specified object unit(s) are added to the end of the link request list and the library search restarts at the primary directory.

FORMAT:

$$\begin{Bmatrix} \text{LINKN} \\ \text{LN} \end{Bmatrix} \text{obj-unit}_1[,\text{obj-unit}_2]\ldots$$

ARGUMENT:

obj-unit

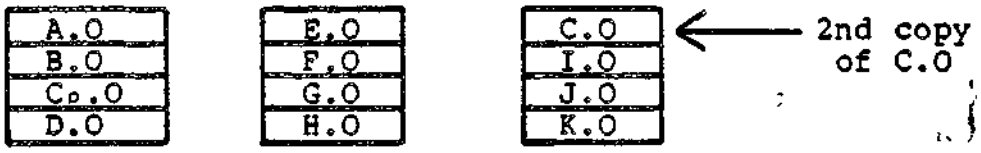
Name of an object unit to be linked. An object unit name must be one to six alphanumeric characters and must not include a suffix; the first character must be a letter or dollar sign ($). The Linker appends the suffix .O to each object unit name and searches for the specified object unit name, including the suffix.

NOTE

For a COBOL segmented program, an object unit name may be up to eight characters. See the Advanced COBOL Reference manual for further details.

Examples of LINK and LINKN

In the following examples, assume that the working directory is the primary directory and LIB and LIB2 directives have been specified.



Example 1:

```
LINK A,G,K,C,F
```

The modules will be linked in the following order:

```
A,C,G,F,K
```

Example 2:

```
LINKN A,G,K,C,F
```

The modules will be linked in the following order:

```
A,G,K,C,F
```

Example 3:

```
LINK A,G,K,Cp,F
```

Assume that module G.O contains "CTRL LINK B,J". The modules will be linked as follows:

```
A,Cp,G,F,K,B,J
```

Once Linker has started to search LIB, it does not return to the primary directory unless a new link request list is found. The two embedded requests were added to the current link request list, forcing a rescan of all libraries.

Example 4:

```
LINKN A,G,K,C,F
```

Assume that module G.O contains "CTRL LINKN B,J". The modules will be linked as follows:

```
A,G,K,C,F,B,J
```

Example 5:

```
LINKN G,B
```

Assume that module G.O contains "CTRL LINK C". The modules will be linked as follows:

```
G,B,Cp
```

Example 6:

```
LINK G,D,F
```

Assume that module G.O contains "CTRL LINK C,B". The modules will be linked as follows:

```
D,G,C,B,F
```

Example 7:

```
LINK G,D,F
```

Assume that module G.O contains "CTRL LINKN C,B". The modules will be linked as follows:

```
D,G,F,C,B
```

In this example, C and B are not added to the current link request list because LINKN was specified instead of LINK.

LINKnn

Check if the LINK directive is to be processed; i.e., allows selective linking.

The LINKnn directive must be used in conjunction with the VDEF directive (or a VALDEF directive in a compilation unit). The VDEF directive is used to modify the bit setting in a 32-bit array. The leftmost 16 bits in the array are set by the symbol Z_MSKR; the rightmost 16 bits in the array are set by the symbol Z_MSKU. Through the VDEF directive, you assign a value to Z_MSKR or Z_MSKU that sets the appropriate bit "on" (a value of 1) or "off" (a value of 0).

Each occurrence of LINKnn causes the array to be indexed by nn. If the referenced bit is on (1), the link request is processed. If the referenced bit is off (0), the link request is ignored.

The bits in the array are initially set on; i.e., all LINKnn directives are processed. The array is modified by the VDEF directive (as described above). The VPURGE directive must be used to remove Z_MSKR and Z_MSKU from the symbol table before these symbols can be redefined.

FORMAT:

LINKnn obj-unit$_1$ [,obj-unit$_2$...]

ARGUMENTS:

nn

Two-digit hexadecimal value between 00 and 1F used as an index in a 32-bit array.

obj-unit$_n$

Name of the object unit to be verified for linking.

LINKO

LINKO

Operate in the same manner as the LINKN directive, except that all embedded link directives in the named object units are ignored.

Only the object units named are linked.

The LINKO directive cannot be embedded in Assembly language control (CTRL) statements.

FORMAT:

$$\begin{Bmatrix}\text{LINKO}\\\text{LO}\end{Bmatrix}\text{obj-unit}_1[,\text{obj-unit}_2]\ldots$$

ARGUMENT:

obj-unit

Name of an object unit to be linked. An object unit name must be one to six alphanumeric characters and must not include a suffix; the first character must be a letter or dollar sign ($). The Linker appends the suffix .O to each object unit name and searches for the specified object unit name, including the suffix.

NOTE

For a COBOL segmented program an object unit name may be up to eight characters. See the Advanced COBOL Reference manual for further details.

LSR

List the Linker search rules. The directories to be searched by the Linker for the object unit(s) are listed in the order in which they will be searched.

The LSR directive can be embedded in Assembly language control (CTRL) statements.

FORMAT:

```
LSR
```

MAP AND MAPU

MAP and MAPU

Create a link map containing: (1) defined symbols that were not purged and (2) undefined symbols to be written to the list-file (see -COUT in the Linker command).

The MAPU directive functions in the same manner, but only applies to undefined symbols. Both the MAP and MAPU directives can be embedded in Assembly language control statements.

If MAP is specified, each defined and undefined symbol generated by the linking of object units is listed in the map and preceded by the name of the object unit in which it is located. A map also includes the names of object units that were linked because of embedded Linker directives, and the symbols contained in those object units. If the MAP directive immediately precedes a QUIT directive, the link map will contain all the defined symbols and undefined symbols of the completed bound unit that have not been removed (i.e., purged).

If MAPU is specified, the map contains each undefined symbol and the object unit in which it is located.

MAP and MAPU directives can be interspersed among other Linker directives. When these directives are encountered all object units named in the link request list are linked before a map is produced. Maps are useful for determining whether all required object units have been linked, and whether all symbols referenced in those object units have been defined.

If there are any undefined references remaining after the last object unit is linked, a MAPU directive is automatically generated by the linker.

FORMAT:

```
{MAP }
{MP  }
{MAPU}
{MU  }
```

Default: No map produced.

A full link map (a map generated by the MAP directive) comprises the following sections:

START — Address at which execution of the root or overlay will begin; specified in the START directive or in a linked object unit.

LOW	Lowest memory address at which the current root or overlay was based.
HIGH	Next location after the highest address of the current root or overlay.
$COMM	Address assigned to COMMON for the bound unit. If no common defined, this does not appear on the MAP.
CURRENT	Next location after the current address of the root or overlay (when the map was created).
EXTERNAL DEFINITIONS	All external symbols currently defined in the symbol table. Unprotected symbols defined in the root or a previously linked overlay will appear in the map unless the symbols are purged via a PURGE or BASE directive. Symbols erroneously defined as both a value and a location will appear twice under EXT DEFS.
UNDEFINED REFERENCES	All references to undefined symbols contained in the object unit root and overlay(s) are listed in the map.

For the root and each overlay containing undefined symbols, the following information is presented:

- Root and overlay(s) containing references to undefined symbol(s)
- Relative address of the last reference to the symbol

If an undefined symbol is referenced in multiple overlays, the symbol will be listed in the map more than once.

If there are external references in both P-relative and Immediate Memory Address forms to an undefined symbol, the symbol is listed twice under UNDEF.

Figure 6-3 illustrates the formats of maps generated by the MAP and MAPU directives.

NOTE

The date and time that the bound unit was created is automatically put in the bound unit's attribute section.

```
LINKER-0303-06/18/0912                          GCOS6 MOD400-L3.0-06/18/1621
BU# EXMPL        LINKED ON: 1982/07/07 1048:04.4  -SLIC -R  -SYM

-> LIB ^CYLON>LOD>ZF1RT

-> IN ^M4LNKR

-> LIB2 ^M4LNKR>TESTPROGS>ASSEMBLER

-> VDEF Z_M3KR,X'4000'

-> BASE X'10'

-> START START1

-> LINK RTPROG

-> MAP

^M4LNKR>TESTPROGS>ASSEMBLER>RTPROG.O
RTPROG                (000010)
1981/08/31 0821:21.1 ASSEMBLER- 2.1-01/09/0913  GCOS6/MINICS 17.26.0
COMMON: COMM1    SIZE: 000032 ADDRESS: 000000
COMMON: COMM2    SIZE: 000064 ADDRESS: 000032
LOCAL COMMON: LCOMM1   SIZE: 000032 ADDRESS: 000096
LOCAL COMMON: $LCOMM   SIZE: 000064 ADDRESS: 0000C8
    C  COMM1  000000    C  COMM2  000032    C  LCOMM1 000096    C  $LCOMM 0000C8
       START1 00001E       ST2    000011       ST3    000013       ST4    000015
       ERP    000017       OLI1   000010       OLI2   000025       OLI3   000029

( EDEF START1)

( EDEF OLI1)

( EDEF OLI2)

( EDEF OLI3)

    OLI4   000035
```

LINKER ID, LINK DATE, AND LINK

LINKER DIRECTIVES

COMMON BLOCKS DEFINED IN RTPROG

EMBEDDED LINKER DIRECTIVES

Figure 6-3. Link Map Formats

```
* * * * * * * * * * * * * * * * * * *
* * * * *       M A P       * * * * *
* * * * * * * * * * * * * * * * * * *

EXMPL              1982/07/07 1048:04.4

**START:     00001E
**LOW:       000010
**HIGH:      00003F
**CURRENT:   00003F

***** COMMON BLOCK DEFINITIONS *****

** EXMPL  000010
 * RTPROG 000010
    C  COMM1  000000    C  COMM2  000032    CX LCOMM1 000096    CX SLCOMM 0000C8


***** EXTERNAL DEFINITIONS *****

     P ZHCOMM 000000    P ZHREL  000000      Z_MSKR   4000

** EXMPL  000010

 * RTPROG 000010
    C  COMM1  000000    C  COMM2  000032    CX LCOMM1 000096    CX SLCOMM 0000C8
       START1 00001E       ST2    000011       ST3    000013       ST4    000015
 ->    ERP    000017       OLI1   000010       OLI2   000025       OLI3   000029
       OLI4   000035
 ->

***** UNDEFINED REFERENCES *****

** EXMPL  000010
 * RTPROG 000010
     V OVLAY1 00001F    V OVLAY2 000029    V OVLAY3 000031    V OVLAY4 000039
```

MAP #1

Figure 6-3 (cont). Link Map Formats

```
KEY: *=OBJECT FILE NAME; **=ROOT OR OVERLAY NAME, OR HEADING;  C=COMMON
     D=DISPLACEMENT REFERENCE; V=VALUE REFERENCE; P=PROTECTED; X=PURGED

* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *

-> OVLY OVLAY1                      LINKER DIRECTIVES

-> LINKN RTPROG,00

^M4LNKR>TEST PROGS>ASSEMBLER>RTPROG.00.O
OVLAY1                (00003F)
1979/06/08 1408:00.6 ASSEMBLER-0200-07/16/0800  GCOS6 MOD400-L200-07/20/2251
COMMON: COMM1    SIZE: 000032 ADDRESS: 000000
COMMON: COMM2    SIZE: 000064 ADDRESS: 000032
LOCAL COMMON: LCOMM1  SIZE: 000032 ADDRESS: 00012C
LOCAL COMMON: $LCOMN  SIZE: 000096 ADDRESS: 00015E
LOCAL COMMON: LCOMM2  SIZE: 000032 ADDRESS: 0001F4
    C  COMM1  000000   C  COMM2  000032   C  LCOMM1 00012C   C  $LCOMN 00015E
    C  LCOMM2 0001F4                                                          COMMON BLOCKS DEFINED IN OVLAY1

-> LINK01 TIME

-> LINK02 WAIT                      LINK DIRECTIVES

-> LDEF ST1A,ST4+X'10'

^CYLON>LDD>ZF1RT>TIME.O
TIME    82020100   (000053)
1982/03/01 1331:23.8 ASSEMBLER- 2.1-01/09/0808  GCOS6 MOD400-L3.0-01/18/0944

{ LINK Z1TIME}

^CYLON>LDD>ZF1RT>Z1TIME.O
Z1TIME  82020100   (000053)
1982/02/19 0903:38.5 MAP-1.0 -10/25/81  GCOS6 MOD400-L3.0-01/18/0944   PAGE
       TIME   000053
```

Figure 6-3 (cont). Link Map Formats

```
-> EDEF ST3

-> MAP


* * * * * * * * * * * * * * * * * *
* * * * *      M A P      * * * * *
* * * * * * * * * * * * * * * * * *

EXMPL                1982/07/07 1048:04.4

**START:    000045      ADDRESS INFORMATION FOR OVLAY1

**LOW:      00003F
**HIGH:     000078      ADDRESS INFORMATION CONTINUED
**CURRENT:  000078

***** COMMON BLOCK DEFINITIONS *****

** EXMPL  000010
 * RTPROG 000010
    C  COMM1  000000   C  COMM2  000032   CX LCOMM1 000096   CX $LCOMM 0000C8

** OVLAY1 00003F
 * .00    00003F
    CX LCOMM1 00012C   CX $LCOMM 00015E   CX LCOMM2 0001F4


***** EXTERNAL DEFINITIONS *****

   P ZHCOMM 000000    P ZHREL  000000      Z_MSKR   0000

** EXMPL  000010
 * RTPROG 000010
    C  COMM1  000000   C  COMM2  000032   CX LCOMM1 000096   CX $LCOMM 0000C8
       START1 00001E      ST2    000011      ST3    000013      ST4    000015
```

Figure 6-3 (cont). Link Map Formats

```
       ERP    000017      OLI1   000010      OLI2   000025      OLI3   000020
       OLI4   00003S      OVLAY1   0001

** OVLAY1 000035
 * .00    000035
    CX LCOMM1 00012C   CX $LCOMM 00015E   CX LCOMM2 0001F4
 * TIME   000053
 * ZITIME 000053
       TIME   000053      ST4A   00002S


***** UNDEFINED REFERENCES *****

** EXMPL  000010
 * RTPROG 000010
     V OVLAY2 000029    V OVLAY3 000031    V OVLAY4 000039

KEY: *=OBJECT FILE NAME; **=ROOT OR OVERLAY NAME, OR HEADING;  C=COMMON
     D=DISPLACEMENT REFERENCE; V=VALUE REFERENCE; P=PROTECTED; X=PURGED

* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *

=> MAPU                          MAPU


* * * * * * * * * * * * * * * *
* * * * *     M A P     * * * * *
* * * * * * * * * * * * * * * *

EXMPL              1982/07/07 1048:04.4

**START:    000045
**LOW:      00003F
**HIGH:     000078
```

Figure 6-4 (cont). Link Map Formats

```
**CURRENT:  000078

***** UNDEFINED REFERENCES *****

** EXMPL   000010
 * RTPROG 000010
     V OVLAY2 000029     V OVLAY3 000031     V OVLAY4 000039

KEY: *=OBJECT FILE NAME; **=ROOT OR OVERLAY NAME, OR HEADING;
     D=DISPLACEMENT REFERENCE; V=VALUE REFERENCE; P=PROTECTED; X=PURGED

* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * *

-> PROTECT OLI4

-> PURGE ST2

-> BASE OLI3

-> OVLY OV_AY2

-> LINK0 RTPROG,01

^M4LNKR>TE3TPROG3>ASSEMBLER>RTPROG.01.0
OVLAY2              (00002D)
1981/08/31 0921:29.4 ASSEMBLER- 2.1-01/09/091
COMMON: COMM1    SIZE: 000032 ADDRESS: 000000
COMMON: COMM2    SIZE: 000064 ADDRESS: 000032
LOCAL COMMON: LCOMM1   SIZE: 000032 ADDRESS: 000226
LOCAL COMMON: %LCOMM   SIZE: 0000C8 ADDRESS: 000258
LOCAL COMMON: LCOMM2   SIZE: 000032 ADDRESS: 000320
    C  COMM1  000000   C  COMM2  000032   C  LCOMM1 000226   C
    C  LCOMM2 000320
```

Figure 6-3 (cont). Link Map Formats

```
-> LINK01 WAIT
                                    EMBEDDED LINK
-> MAP

^CYLON>LDD>ZF1RT>WAIT.O
WAIT    82320100   (000041)
1982/03/01 1331:37.0 ASSEMBLER- 2.1-01/09/0808  GC086 MOD400-L3.0-01/18/0944

[ LINK Z1WAIT]

^CYLON>LDD>ZF1RT>Z1WAIT.O
Z1WAIT  82320100   (000041)
1982/02/19 0909:24.5 MAP-1.0 -10/25/81    GC086 MOD400-L3.0-01/18/0944   PAGE
       WAIT   000041


* * * * * * * * * * * * * * * * * *
* * * * *     M A P     * * * * * *    MAP BS
* * * * * * * * * * * * * * * * * *

EXMPL              1982/07/07 1048:04.4

**START:    000033

**LOW:      00002D
**HIGH:     00005A
**CURRENT:  00005A

***** COMMON BLOCK DEFINITIONS *****

** EXMPL  000010
 * RTPROG 000010
    C  COM#1  000000   C  COMM2  000032
```

Figure 6-3 (cont). **Link Map Formats**

```
** OVLAY2  00002D
 * .01     00002D
    CX LCOMM1 000226    CX SLCOMM 000258    CX LCOMM2 000320

***** EXTERNAL DEFINITIONS *****

     P ZHCOMM 000000     P ZHREL  000000      Z_MSKR   4000

** EXMPL   000010
 * RTPROG  000010
    C  COMM1  000000    C  COMM2  000032      START1 00001E      ST3    000013
       ST4    000015       ERP    000017      OLI1   000010      OLI2   000025
     P OLI3   000035       OVLAY1   0001

** OVLAY1  00003F
 * .00     00003F
 * TIME    000053
 * Z1TIME  000053
       ST4A   000025      OVLAY2   0002

** OVLAY2  00002D
 * .01     00002D
    CX LCOMM1 000226    CX SLCOMM 000258    CX LCOMM2 000320
 * WAIT    000041
 * Z1WAIT  000041
       WAIT   000041

***** UNDEFINED REFERENCES *****

** EXMPL   000010
 * RTPROG  000010
     V OVLAY3 000031     V OVLAY4 000039

KEY: *=OBJECT FILE NAME; **=ROOT OR OVERLAY NAME, OR HEADING; C=COMMON
     D=DISPLACEMENT REFERENCE; V=VALUE REFERENCE; P=PROTECTED; X=PURGED
```

Figure 6-3 (cont). Link Map Formats

```
* * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * *

-> L$R

**********
PRIM   ^M4L¥KR
LIB    ^CYL]N>LOD>ZF!RT
LIB2   ^M4L¥KR>TESTPROG$>ASSEM$LER
**********

-> Q


* * * * * * * * * * * * * * * * * * *

* * * * *      M A P      * * * * *
* * * * * * * * * * * * * * * * * *

EXMPL                1982/07/07 1048:04.4

**START:     000033
**LOW:       00002D
**HIGH:      000000
**CURRENT:   00005A


***** UNDEFINED REFERENCES *****

** EXMPL   000010
 * RTPROG  000010
     V OVLAY3 000031     V OVLAY4 000039

KEY: *=OBJECT FILE NAME; **=ROOT OR OVERLAY NAME, OR HEADING; C=COMMON
     D=DISPLACEMENT REFERENCE; V=VALUE REFERENCE; P=PROTECTED; X=PURGED

* * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * *
```

ACTIVE OBJECT UNIT DIRECTORIES

Figure 6-3 (cont). Link Map Formats

```
**********
ROOT  EXMP_
HIGHEST OVERLAY NUMBER: 3
NUMBER OF EDEFS:  5
 SLIC
**********
CMMN  DATA                BASE:  000000  START:  000000  .F..  HIGH: 000352
ROOT  EXMP_               BASE:  000010  START:  00001E  ..UI  HIGH: 00003F
OVLY  OVLAY1      # 0001  BASE:  00003F  START:  000045  ..U.  HIGH: 000078
OVLY  OVLAY2      # 0002  BASE:  00002D  START:  000033  ..U.  HIGH: 00005A

KEY: S=SHAREABLE; F=FLOATING; I=CONTAINS AN IMA; U=CONTAINED AN UNDEFINED
     REFERENCE; ->=IN-LINE DIRECTIVE; {...}=EMBEDDED DIRECTIVE

**********
SIZE OF ROOT AND FIXED OVERLAYS:  000078
LAST RU RECORD NUMBER:  12

*** BU CONTAINS UNRESOLVED REFERENCES.

**********
LINK DONE
**********

*** THERE WERE 1 ERRORS DURING THE LINK.


EOF
```

Figure 6-3 (cont). Link Map Format

OVERLAYTABLE

OVERLAYTABLE

Include the name of each overlay and its associated Linker-generated overlay number in the set of symbols passed to the Loader at load time.

FORMAT:

```
{OVERLAYTABLE}
{OE          }
{OT          }
```

OVLY

OVLY

Assign the specified name to the non-floatable overlay that immediately follows, and designate the end of the preceding root or overlay.

OVLY must be specified as the first directive of each non-floatable overlay.

The Linker assigns a two-digit number to each overlay. Overlays are numbered sequentially, in ascending order; the first overlay is 00.

FORMAT:

OVLY name

ARGUMENT:

name

Name of the nonfloatable overlay that immediately follows. The overlay name must consist of one to six alphanumeric characters; the first character must be alphabetic.

Example:

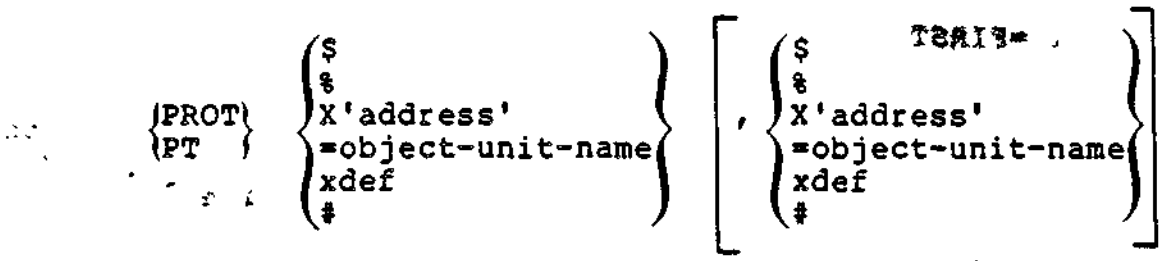
```
LINKER BU -PT                 Load the Linker and designate BU
                              as the bound unit name.

LINKER-2100-081/27/1042

L?
LINK A, B; MAP

L?
OVLY A2                       Designate the end of the root (which
                              comprises object units A.O and B.O)
                              and specify that the next overlay is
                              a nonfloatable overlay named A2.
                              The Linker assigns the number 00 to
                              this overlay.

L?
LINK X

L?
LINK Y
```

```
L
MAP

L?
QUIT
ROOT BU
LINK DONE
RDY:
```

PROTECT

Prevent certain symbols and/or object unit names from being removed from the symbol table. Symbols that identify addresses within the range of addresses specified by the first operand through the second operand are protected, and object unit names equated to addresses within that range are protected. If a second operand is not specified, the symbol at the address of the first operand and any other symbols or object unit names equated to that address are protected. Once a symbol or object unit name is protected, it cannot be purged later. The protect directive cannot be embedded in Assembly language control (CTRL) statements.

FORMAT:



ARGUMENTS:

$

Next location after the highest address of the linked root or previously linked nonfloatable overlay.

%

Highest address+1 ever used in the linked root or any previously linked nonfloatable overlay.

X'address'

Hexadecimal address comprising one to five integers enclosed in apostrophes and preceded by X. The specified address is relative to the beginning of the root (relative 0).

=object-unit-name

Specified object unit's base address.

xdef

Address of any previously defined external symbol.

The current address.

Example 1:

```
PROT X'1234',X'4565'
```

This directive protects symbols and object unit names that identify addresses from 1234 through 4565.

Example 2:

```
PT =FIRST
```

This directive protects symbols that identify the base address of the object unit FIRST and all symbols equated to that address. The base address of FIRST is determined by producing a link map (see "MAP and MAPU Directives").

Example 3:

```
PROT SYM,X'5555'
```

This directive protects symbols that identify addresses from the address of the previously defined external symbol named SYM through 5555; object unit names equated to those addresses also are protected.

PURGE

Remove the following items from the symbol table: unprotected symbols that define addresses greater than or equal to the first address and less than or equal to the second address. If a second operand is not specified, the symbol at the address of the first operand and any other symbols or object unit names equated to that address are purged.

An object unit currently being linked can contain definitions used for previously linked object units that will not be used for subsequent object units to be linked. By removing symbols that are no longer required, there is more room for symbols that will be required by subsequently linked object units.

NOTES

1. Undefined symbols cannot be purged.
2. Symbols and object unit names that are protected by a PROTECT directive cannot be purged.
3. Only symbol addresses (not values) can be purged by this directive.
4. The PURGE directive cannot be embedded in Assembly language control (CTRL) statements.

FORMAT:

```
{PURGE}   {$                    }   [ , {$                    } ]
{PE   }   {%                    }       {%                    }
          {X'address'           }       {X'address'           }
          {=object-unit-name    }       {=object-unit-name    }
          {xdef                 }       {xdef                 }
          {#                    }       {#                    }
```

ARGUMENTS:

$

Next location after the highest address of the linked root or previously linked nonfloatable overlay.

%

Highest address+1 ever used in the linked root or any previously linked nonfloatable overlay.

X'address'

Hexadecimal address comprising one to five integers enclosed in apostrophes and preceded by X. The specified address is relative to the beginning of the root (relative 0).

=object-unit-name

Specified object unit's base address.

xdef

Address of any previously defined external symbol.

#

The current address.

Example 1:

```
PURGE X'1234',X'4565'
```

This directive purges unprotected symbols that identify addresses from 1234 through 4565, and unprotected object unit names equated to addresses within that range.

Example 2:

```
PE =FIRST
```

This directive purges unprotected symbols that identify the base address of the load unit FIRST and any other unprotected symbol names equated to that address.

Example 3:

```
PURGE SYM,X'5555'
```

This directive purges unprotected symbols that identify addresses from the address of the previously defined external symbol SYM through 5555; unprotected object unit names equated to addresses within that range also are purged.

QUIT

Designate that the last Linker directive has been entered. Specify QUIT after the last overlay, or at the end of the root if there are no overlays.

If object units were successfully linked, the bound unit is completed and the Linker terminates; otherwise, the Linker terminates execution immediately.

The QUIT directive is required; it cannot be embedded in Assembly language control statements.

FORMAT:

```
{QUIT}
{QT  }
{Q   }
```

RERUN RELOCATABLE

RERUN RELOCATABLE (RR)

Specify that the sharable bound unit can be reloaded at restart into locations other than those it occupied when the checkpoint was taken (see the Commands manual for details on checkpoint-restart). If this directive is not specified, the bound unit is reloaded at the same system memory pool locations it occupied when the checkpoint was taken.

The RR directive can be embedded in Assembly language control statements.

FORMAT:

RR

NOTE

If the RR directive is used, it is important to remember that after reloading, the current values of the IMAs' referencing locations in the bound unit are no longer valid; therefore, if the bound unit contains IMAs (see the link map or compiler list file to determine this), RR should not be used.

RETURN

Signal the Linker to expect subsequent directives from the user-in file. This directive should only be specified in an INCLUDE file. A RETURN directive in a file specified in an INCLUDE directive is logically equivalent to an EOF mark; it returns the Linker to the user-in file.

FORMAT:

```
RETURN
```

SEG

Cause the bound unit to occupy one or two physical segments in memory. Before using this directive, consult with the person responsible for system building and determine the segment numbers available to task groups. The SEG directive can be entered at any point. You can specify the physical segment number(s) to be assigned as well as the access (read, write, and execute) to the segment(s). This directive is only meaningful when the bound unit is executed in a swap pool.

FORMAT:

```
{SEG}  [{X} 'code_segment_no']  [{,code_access[,data_access]}]
{SG }  [{Z}                   ]  [{,,data_access            }]
```

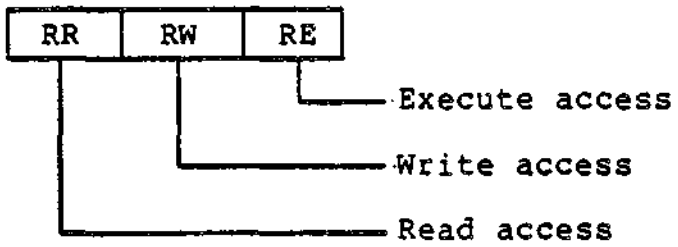
ARGUMENTS:

code_segment_no

Hexadecimal number from 1 to F that specifies the number of the physical segment containing the whole bound unit if the -R ECL parameter in the Linker command is not specified. When -R ECL argument is specified, code segment_no is a hexadecimal number from 2 to F and data segment_no is equal to code_segment_no -1. There are 15 big segments in memory. Each big segment is at most 64K. The user can use big segment numbers from 1 to F.

code_access and data_access

Bit strings of exactly 6 bits that specify the access right for the readable and writable segments, respectively. Each bit string represents the corresponding access fields in the segment descriptor. Representation of the format of the access argument is:



The two bit positions that designate each access represent ring number:

Bit	Ring Number	
11	0	Executive
10	1	Privileged real time
01	2	Unprivileged real time
00	3	Batch

A program making a reference (read, write, and execute) to memory is given access, if the value of the ring number of the program's privilege is less than or equal to ring value of the desired memory location.

The defaults for the access fields are:

Data access - -R - always		000000
Code access - sharable		001100
	- globally sharable	
	- otherwise	000000

Example 1:

```
SEG X'06',,000100
```

In this example if -R was not specified, segment 6 with default access is assigned to the bound unit and the specified data access is ignored. If -R was specified, segment 6 with default access is assigned to the code and segment 5 with the specified access is assigned to the data.

Example 2:

```
SEG ,001000
```

In this example, if -R was not specified, the loader assigns a segment number with the specified access to the bound unit. If -R was specified, the loader assigns a segment number with the specified access to the code and segment number -1 with default access to the data.

SHARE

SHARE

Designate that the bound unit is sharable within the task group. If another task requests that the bound unit be loaded, instead of another copy of the bound unit being loaded, the existing copy in memory is used. The bound unit must have reentrant code, but the system does not check to see that it does.

SHARE must be specified in the definition of the root before the first overlay is defined.

SHARE directives can be embedded in Assembly language control statements.

FORMAT:

```
{SHARE}
{SE   }
```

STACK

Specify the size of the stack in a decimal number of words. If no STACK directive is specified, the Linker will use the largest stack size specified in a object unit linked into the bound unit.

FORMAT:

STACK value-n

ARGUMENT:

value-n

Indicate the size of the stack in a decimal number of words.

START

START

Designate the relative location within a root or overlay at which execution of the root or overlay will begin once it is loaded into memory by the Loader.

If a linked object unit contains a start address (an Assembler or compiler END statement was specified) and the START directive is specified, the first start address encountered (in either a START directive or an END statement) is used by the Linker for that root or overlay.

FORMAT:

```
{START}
{ST   }   symbol
```

ARGUMENT:

symbol

Name of the external symbol whose address designates the relative address that the root or overlay will begin executing.

Default: Start address specified in the first linked object unit that has a start address. If the symbol is never defined or a start address is not found, the start address is the first non-common location in the root or overlay.

NOTE

For very large programs, the start address must be within 64K of the beginning of the root or overlay.

SYS

Indicate that this overlay or root can be run as a system task. This directive does not control where the bound unit is loaded, rather it allows a bound unit to be executed either as a system or user task. Before using this directive, consult with the person responsible for system building and determine available system memory. The SYS directive can be embedded in Assembly language (CTRL) statements.

FORMAT:

```
{SYS}
{SS }
```

Example:

```
SYS
```

VAL

VAL

Create a value definition at link time that is equivalent to the difference between two external location definitions.

VAL directives can be embedded in Assembly language control statements.

FORMAT:

```
VAL  symbol, external location1 - external location2
VL
```

ARGUMENTS:

symbol

Assign a name to the value of the distance between two locations.

external location

Location defined externally.

VDEF

Assign a value to an external symbol. The VDEF directive cannot be embedded in Assembly language control statements. A symbol should be defined only once, as a value or as a location. When a symbol is defined, its definition is put into the Linker symbol table so that it can be used during linking to resolve external references.

FORMAT:

```
{VDEF}
{VF  }    symbol,X'value'
```

ARGUMENTS:

symbol

One to six alphanumeric characters.

X'value'

Value of the designated symbol; must be a one-word hexadecimal integer enclosed in single quotes and preceded by X.

language control statements.

FORMAT:

VPURGE value-definition-symbol

ARGUMENT:

value-definition-symbol

External symbol name associated with a particular value.

LINKER PROCEDURES

This subsection describes the frequently used Linker procedures. The examples provided show different methods for linking COBOL programs, including one example that uses overlays.

Overview

The Linker is a system software program that functions as the final stage of program development before program execution is possible. Before linking, a program must be compiled (or assembled) to produce one or more object units or compile units that the Linker identifies for linking. The Linker recognizes object units by the .O suffix (appended to each file name by the compiler). The Linker combines one or more object units to produce a bound unit. A bound unit is an executable program consisting of a root segment and zero or more overlay segments that can be loaded into memory.

Using Overlays

In situations where memory is limited, it may be necessary for you to divide your program into one or more overlay segments so that individual portions of your program may be called into a single memory area only when they are needed. Unlike the root segment, which cannot be reloaded once it is read into memory, an overlay segment can be read in as often as it is needed. See Example 4 for a link session that uses overlays.

Interrupting Linker Execution

If at any time during Linker execution you want to interrupt processing, you can perform one of the following actions:

- Press the QUIT, INTERRUPT, or BREAK key at your terminal.
- Enter ΔCΔBID at the operator terminal, where id is your two-character task group identification.

After performing one of the above actions, a **BREAK** message will appear on your terminal. You can now:

- Enter any valid ECL command.
- Resume Linker execution as if no break had occurred by entering the Start (SR) command.

- Terminate Linker processing and return to command level by entering the Unwind (UW) command.

- Restart your task group by issuing a New Process (NEW_PROC) command.

NOTE

If you interrupt a MAP, and you want to terminate the MAP operation and jump to the next Linker directive, issue a Program Interrupt (PI) command.

Sample Link Sessions

The sample link sessions that follow will help you become familiar with Linker procedures.

The first three examples illustrate different methods for linking a COBOL program. The fourth example describes a method for linking a COBOL program that contains two overlays.

Example 1:

This example illustrates a link session requiring a minimum of Linker directives.

The COBOL program CARDIN has just been compiled. A List (LS) command is issued to examine the contents of the programmer's (Cook) working directory:

```
LS
```

Entry Name	Type	Physical Sectors	Starting Sector Hex	Record Length
CARDIN.C	S	8	3250	256
CARDIN.O	S	8	3258	256
CARDIN.L	S	16	3260	256

The file CARDIN.C contains COOK's source program. The files CARDIN.O and CARDIN.L were produced by the COBOL compiler. CARDIN.O contains the object unit that the Linker will use to produce a bound unit named CARDIN.

Cook now wants to link his program into a bound unit. He enters the command:

```
LINKER CARDIN -PT -COUT !LPT0@
```

Cook has specified that he wants to create a bound unit named CARDIN. Cook also specifies that his link map will be directed to printer !LPT00 rather than to a file named CARDIN.M in his working directory. (The contents of CARDIN.M are described later in this example.) The -PT argument causes the Linker prompt L? to appear when the Linker is ready to receive input. It is recommended that new users include this argument in the Linker command format.

The Linker responds:

```
LINKER-1982/06/18 0912:50.5
L?
```

Cook now enters Linker directives. Each directive has been keyed to the explanatory notes that follow. (The Linker prompts have been omitted from the text.)

```
(1)     LSR
            **********
            PRIM WORKING DIRECTORY
            **********

(2)     LIB ^SYSRES>LDD>ZCART
(3)     LSR
            **********
            PRIM WORKING DIRECTORY
            LIB  ^SYSRES>LDD>ZCART
            **********
(4)     LINK CARDIN
(5)     MAP
(6)     QUIT
            ROOT CARDIN
            LINK DONE
```

NOTES

1. Cook asks the Linker to list the search rules it currently uses to locate object units to be linked. The Linker's response indicates that the primary directory searched is Cook's working directory.

   This directive is optional. Omitting it will not affect the linking process. It is included here to illustrate how the Linker's search rules are modified by the LIB directive.

2. Because Cook's COBOL program requires the run-time routines located in the directory ZCART, Cook must designate ZCART as the secondary directory to be searched by the Linker. If the required object units cannot be found in Cook's primary directory, the Linker will automatically search the secondary directory ^SYSRES>LDD>ZCART.

3. Cook lists the Linker's modified search rules. (See Note 1.)

4. This LINK directive queues the object unit CARDIN.O for linking.

5. The MAP directive produces a link map that is written out to the printer LPT00 (as specified in the -COUT argument of the Linker command). This link map is shown in Figure 6-4. It also causes CARDIN.O to be linked before the map is produced.

6. Cook enters the QUIT directive to indicate that there are no more directives. The Linker builds the bound unit and terminates.

The linking process has been successfully completed; Cook now enters an LS command to examine the contents of his working directory:

```
LS

                          Physical      Starting      Record
Entry Name      Type      Sectors       Sector Hex    Length

CARDIN.C        S           8             3250        256
CARDIN.O        S           8             3258        256
CARDIN.L        S          16             3260        256
CARDIN          R2          8             3240        256
                            8             3270
```

The bound unit CARDIN now resides in Cook's working directory, ready for execution. Note that CARDIN.M, the link map, is not listed in the working directory. This file was written out to the printer LPT00 when Cook issued the MAP directive.

```
LINKER-1982106118 0912:50.5                    GCOS6 MOD400-S3.0-12/12/1635
BU= CARDIN          LLINKED ON:  1982/05/05 0809:00.6 LAF
→LIB ^SYSRES>LDD>ZCIRT/
→LINK CARDIN/                    LINK DIRECTIVES FOR LINKING CARDIN
→MAP/

CARD
CARDIN  82/04/29    (000000)
 COBOL REV. 300 DATE 82/04/29 TIME 0811

LOCAL COMMON:  $LCOMW:  SIZE:  000133  ADDRESS:  000000
   [LINK ZCIPER   /]
   [LINK ZCSTOP   /]   LOCAL COMMON-AREAS OF MEMORY
   [EDEF ZCMAIN   /]   DEFINED FOR SEPARATE CODE AND
   [LINK ZCIPIO   /]   DATA PORTIONS FOR THE COBOL
                       PROGRAM

^SYSRES>ZCIRT>ZCIPER.0
ZCIPER  79010900    (000307)
HRS ASSEMBLER 5.05   01/10/81   1647.5 est Wed
(C) COPYRIGHT 1977 BY HONEYWELL INFORMATION SYSTEMS INC

^SYSRES>ZCIRT>ZCSTOP.0
ZCSTOP   79010900   (00036C)
HRS ASSEMBLER 5.05 01/10/81   1641.6 est Wed
(C) COPYRIGHT 1977 BY HONEYWELL INFORMATION SYSTEMS INC

^SYSRES>ZCIRTZZCIPI0.0
ZCIPIO   79050200   (000387)
HRS ASSEMBLER 6.00 05/03/81   0906.8 edt Thu   RUN-TIME MODULES
   [LINK ZCDEAD   /]                           REQUIRED BY THE
                                               COBOL PROGRAM
^SYSRES>ZCIRT>ZCDEAD.0
ZCDEAD   79010900   (0003A2)

HRS ASSEMBLER 5.05 01/10/81   1647.7 est Wed
(C) COPYRIGHT 1977 BY HONEYWELL INFORMATION SYSTEMS INC

* * * * * * * * * * * * * * * * * *
* * * *          MAP            * * * *
* * * * * * * * * * * * * * * * * *

** CARDIN          LINK MAP   1982/05/05 0808:00.6
**START   000173
**LOW     000000   MEMORY ADDRESS
**HIGH    0003B5   LIMITS FOR THE
**CURRENT 0003B5   ROOT CARDIN

***** COMMON BLOCK DEFINITIONS ****

    **CARDIN   000000
     *CARDIN   000000
       C $LCOMMW 000000
```

Figure 6-4. Sample Link Map (CARDIN.M)

```
***** EXTERNAL DEFINITIONS ****

P    ZHCOMM  000000
P    ZHREL   000000

**   ROOT    000000
 *   CARDIN  000000
 C   $LCOMW  000000
     ZCZERO  000133  CARDIN  0773  ZCMAIN   0170
 *   ZCIPER  0001DE
     ZCRTER  0001F0
 *   ZCSTOP  000210
 *   ZCIOIO  000213
     ZCOP1   000213   ZCOP2   000217  ZCOP3   00021B  ZCOP4   00021E
     ZCOP5   000221   ZCOP6   000224  ZCRD1   000273  ZCRD2   000275
     ZCRD3   00028A   ZCWR1   00031A  ZCWR2   00031E  ZCWR3   000338
     CZWR4   00033F   ZCWR5   00031C  ZCST1   000393  ZCST2   00039E
     ZCST3   0003A9   ZCCL1   00036A  ZCRW1   000374  ZCRW2   00037C
     ZCDL1   000383   ZCDL2   000388  ZCS     00038D  ZCA     00038F
 *   ZCDEAD  0003B4

KEY:  *=OBJECT FILE NAME; **=ROOT OR OVERLAY NAME, OR HEADING; C=COMMON
      D=DISPLACEMENT REFERENCE; V=VALUE REFERENCE; P=PROTECTED; X=PURGED

* * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * *

  QUIT/
**********
ROOT CARDIN

HIGHEST OVERLAY NUMBER: 1
NUMBER OF EDEFS: 1
LAF
**********
ROOT CARDIN         BASE 000000   ST000173        HIGH=0003B5
**********
*SIZE OF ROOT AND FIXED OVLYS= 03B5
LAST BU RECORD NUMBER:9
**********
LINK DONE
**********
```

MEMORY ADDRESSES OF EXTERNAL NAMES IN THE BOUND UNIT

Figure 6-4 (cont). Sample Link Map (CARDIN.M)

Example 2:

This example shows you how to specify a directive input device (such as a file, another terminal, or card reader) from which the Linker will read its directives. This procedure is useful if you have many directives to enter, or if you wish to create a Linker directive file for future use.

Programmer Cook wants to have the Linker read its directives from a file named LKDIR. He invokes the Editor, types in his Linker directives, and writes the file to the pathname ^SYSRES>WORK>LKDIR. The contents of LKDIR are listed below. (The object unit to be linked, CARDIN.O, resides in Cook's working directory.)

```
LIB ^SYSRES>LDD>ZCART
LINK CARDIN
MAP
QUIT
```

Figure 6-5. Contents of LKDIR

To activate Linker processing, Cook need only enter the following command:

```
LINKER CARDIN -IN ^SYSRES>WORK>LKDIR
```

Cook has specified that he wants to create a bound unit named CARDIN. The -IN argument specifies the pathname of the file from which Linker directives will be read. Cook could also have designated another terminal or a card reader as the directive input device.

The complete dialog as it appears at Cook's terminal is shown below:

```
LINKER CARDIN -IN ^SYSRES>WORK>LKDIR
    LINKER-1982?06?18 912:50.5
    ROOT CARDIN
    LINK DONE
```

Example 3:

In this example, Cook wants to link the object unit CARDIN.O, which resides on a diskette volume named ^DSK, and whose full pathname is ^DSK>MYDIR>CARDIN.O. Since Cook wants to create the bound unit CARDIN in his working directory ^SYSRES>WORK, he must designate the directory MYDIR as the primary directory the Linker will search.

A second object unit NEXT.O is also to be linked into the bound unit. It resides on the current working directory. Cook wants to link NEXT.O to CARDIN.O which has been linked.

Cook's dialog with the system is shown below. The dialog has been keyed to the explanatory notes that follow.

```
(1)     LINKER CARDIN
            LINKER-1982/06/18 0912:50.5
(2)     LIB ^SYSRES>LDD>ZCART
(3)     IN ^DSK>MYDIR
(4)     LSR
            **********
            PRIM ^DSK>MYDIR
            LIB  ^SYSRES>LDD>ZCART
            **********

(5)     LINK CARDIN
(6)     IN
(7)     LSR
            **********
            PRIM WORKING DIRECTORY
            LIB  ^SYSRES>LDD>ZCART
            **********

(8)     LINK NEXT
(9)     MAP
(10)    QUIT
            ROOT CARDIN
            LINK DONE
(11)    LS
            DIRECTORY:  ^SYSRES>WORK
```

Entry Name	Type	Physical Sectors	Starting Sector Hex	Record Length [1]
CARDIN.M	S	8	3240	256
		8	3278	
CARDIN	RZ	8	3248	256
		8	3280	
NEXT.O		8	32A8	256

NOTES

1. Invoke the Linker and specify CARDIN as the name of the bound unit to be created.

2. Request that the Linker search the secondary directory ZCART for the required COBOL run-time routines.

3. Specify the IN directive, designating ^DSK>MYDIR as the primary directory in which the Linker should search for the required object unit.

4. List the Linker's search rules. The Linker's response indicates that the primary directory to be searched is ^DSK>MYDIR.

5. Request that CARDIN.O be queued for linking.

6. Enter the IN directive again, this time omitting a pathname. This action modifies the Linker's search rules; the primary directory to be searched is Cook's working directory.

7. List the Linker's search rules again, and notes that the Linker's primary directory has been redirected to his working directory.

8. Request that NEXT.O be queued for linking.

9. Issue a MAP directive. A link map is written to a file named CARDIN.M in Cook's working directory.

10. Indicate that there are no more Linker directives.

11. Specify an LS command to verify that the bound unit CARDIN has been created in his working directory. CARDIN.M contains link map information.

Example 4:

This example describes how to link a program containing two overlays.

Programmer Shepard has written a COBOL program called COBPRG which calls two overlays, PART2 and PART3. Figure 6-6 shows the relationship between the root COBPRG and the two overlays. Source listings of the root program and the two overlays are shown in Figures 6-7, 6-8, and 6-9. (Source listings are included to show you the relationships that exist between a root program and its overlays. In Figure 6-4, for example, note how the source program calls in its overlays. If Shepard's link is successful, each greeting message will join with the others in the specific order Shepard intends.)

Following COBOL compilation of all three source files, Shepard issues an LS command to display the contents of her working directory:

```
LS
  DIRECTORY:  ^SYSVOL>SHEPARD

                        Physical     Starting     Record
Entry Name      Type    Sectors      Sector Hex   Length

                          8          3278
COBPRG.C        S         8          3280         256
COBPRG.O        S         8          3288         256
COBPRG.L        S        16          3298         256
                          8          3248
PART2.C         S         8          32A8         256
PART2.O         S         8          32B0         256
PART2.L         S        16          32B8         256
PART3.C         S         8          32C8         256
PART3.O         S         8          32D0         256
PART3.L         S        16          32D8         256
```

Note that all three object units to be linked are located in Shepard's working directory. COBPRG.O is the object unit that will form the root segment of the bound unit COBPRG. PART2.O and PART3.O are object units that will form the two overlay segments of the bound unit. Figure 6-6 shows the bound unit Shepard will create when she links the root segment COBPRG and the two overlay segments, PART2 and PART3.

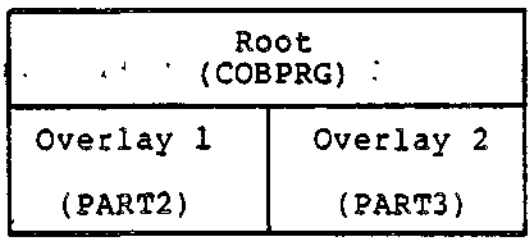


Figure 6-6. Structure of the Bound Unit COBPRG

Shepard is ready to link her programs. Her dialog with the system is described below. The dialog has been keyed to the explanatory notes that follow.

```
(1)     LINKER COBPRG
            LINKER-1982/06/18 0912:50.5
(2)     LIB ^SYSVOL>LDD>ZCART
(3)     CC
(4)     LINK COBPRG
(5)     MAP
(6)     OVLY PART2
```

```
(7)     BASE $
(8)     LINK PART2
(9)     MAP
(10)    OVLY PART3
(11)    BASE =PART2
(12)    LINK PART3
(13)    MAP
(14)    QUIT
        ROOT COBPRG
        LINK DONE
```

NOTES

1. Invoke the Linker, specifying COBPRG as the name of the bound unit to be created.

2. Request the Linker search the secondary directory ZCART for the required COBOL run-time routines.

3. Specify the CC (Call-Cancel) directive since Shepard's main COBOL program contains CALL/CANCEL statements that invoke overlays. By specifying this directive, Shepard tells the Linker to use the call/cancel logic for the programs. (This directive applies only to COBOL programs and must be specified before the first LINK directive in the root.

4. Queue the object unit COBPRG.O for linking.

5. Specify a MAP directive. A link map is written to a file named COBPRG.M in Shepard's working directory.

6. Designate the end of the root segment and the beginning of the first overlay PART2.

7. Identify the relative load address for PART2 within the bound unit. The BASE $ directive specifies that PART2 will be linked beginning with the next location after the highest address of the root segment COBPRG. This is the default base address for PART2. This BASE directive could be omitted.

8. Queue the object unit named PART2 for linking.

9. Specify a MAP directive.

10. Designate the end of the overlay PART2 and the beginning of the second overlay PART3.

11. Request that the overlay named PART3 be loaded starting at the same relative address as the object unit PART2. Overlay PART2 and PART3 can never be in memory at the same time.

12. Queue PART3 for linking.

13. Request a link map.

14. Terminate Linker processing.

Shepard is now ready to execute the bound unit COBPRG. (No data files are required.) She enters the bound unit name:

```
COBPRG
```

The program responds:

```
HELLO FROM PROGRAM SAMPLE.
HELLO FROM PART2--THE FIRST OVERLAY
*****BACK TO SAMPLE *****
HELLO FROM PART3--THE SECOND OVERLAY
GOODBYE FROM PROGRAM SAMPLE.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.
AUTHOR. FOSTER.
INSTALLATION. PHOENIX,ARIZ.
DATE-WRITTEN. 042980.
SECURITY. NONE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. HIS-SERIES-60 LEVEL-6.
OBJECT-COMPUTER. HIS-SERIES-60 LEVEL-6.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  PNAME PIC X(5).
01  QNAME PIC X(5).
77 REC-NUM USAGE COMP-1 VALUE ZERO.
PROCEDURE DIVISION.
GREETING.
    DISPLAY "HELLO FROM PROGRAM SAMPLE.".
    ADD 10 TO REC-NUM.
OVERPROC.
    MOVE "PART2" TO PNAME.
    CALL PNAME.
    MOVE "PART3" TO QNAME.
    CANCEL PNAME.
    DISPLAY "****** BACK TO SAMPLE ******".
    CALL QNAME.
    CANCEL QNAME.
    DISPLAY "GOODBYE FROM PROGRAM SAMPLE".
    STOP RUN.
END COBOL.
```

Figure 6-7. Source Listing of Root Segment COBPRG

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PART2.
AUTHOR. FOSTER.
INSTALLATION. PHOENIX,ARIZ.
DATE-WRITTEN. 042880.
SECURITY. NONE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. HIS-SERIES-60 LEVEL-6.
OBJECT-COMPUTER, HIS-SERIES-60 LEVEL-6.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  PNAME PIC X(5).
01  QNAME PIC X(5).
77  REC-NUM USAGE COMP-1 VALUE ZERO.
PROCEDURE DIVISION.
GREETING.
    DISPLAY "HELLO FROM PART2--THE FIRST OVERLAY".
    EXIT PROGRAM.
END COBOL.
```

Figure 6-8. Source Listing of First Overlay Segment PART2

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PART3.
AUTHOR. FOSTER.
INSTALLATION. PHOENIX,ARIZ.
DATE-WRITTEN. 042980.
SECURITY. NONE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. HIS-SERIES-60 LEVEL-6.
OBJECT-COMPUTER. HIS-SERIES-60 LEVEL-6.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  QNAME PIC X(5).
01  QNAME PIC X(5).
77  REC-NUM USAGE COMP-1 VALUE ZERO.
PROCEDURE DIVISION.
GREETING.
    DISPLAY "HELLO FROM PART3--THE SECOND OVERLAY".
    EXIT PROGRAM.
END COBOL.
```

Figure 6-9. Source Listing of Second Overlay Segment PART3

Section 7
MULTI-USER DEBUGGER (SYMBOLIC MODE)

The following two debuggers are available in the MOD 400 environment:

- $DEBUG - A special debugger used for system maintenance. This debugger is described in the System Programmer's Guide, Volume II.

- The Multi-User Debugger - A general purpose tool used for normal user applications. This debugger operates in two modes:

  - Numeric Mode - Used for applications written in Assembly language.

  - Symbolic Mode - Used for applications written in Advanced COBOL or Advanced FORTRAN.

Numeric debugging is described in the System Programmer's Guide, Volume II. Symbolic debugging is described in this section; full information on symbolic debug functions and directives is provided. See Appendix E for a sample symbolic debug session.

DEBUGGER OVERVIEW

The debugger can be used with object units that have been compiled with the debug option (-SYMBOL). The debug option causes the compiler to generate a file for later use by the debugger.

Additionally, the object unit must be linked with the Linker's debug option (-SYMBOL). In any bound unit there may be a mixture of programs compiled with and without the debug option.

DEBUGGER CAPABILITIES

The debugger uses the object unit tables and a Linker symbol table to manipulate breakpoints, process action lines, and alter and display (dump) data variables. The debugger uses the same referencing format for variables as in the source program. This referencing format can be variable name, label, or line number.

The various debug directives can be used to halt a program at selected breakpoints during execution, restart the program from the same point, or change sequence and start from a different point. While the program is halted, you can examine and alter program data and set further breakpoints.

The debugger can be used to debug Advanced COBOL and Advanced FORTRAN programs. Programs must be compiled with the debug option (-SYMBOL). This option generates an object unit file.

The debugger uses the object unit symbol tables and link symbol table (produced by the Linker debug option, -SYMBOL) to manipulate breakpoints, process actions lines, and alter data variables. The symbol table is called object_unit_name.Z. The special link map is called bound_unit_name. object_unit_name is the name of the Advanced COBOL or Advanced FORTRAN source program. Do not try to edit these symbol table files because you may destroy necessary information.

INVOKING THE DEBUGGER

To use the debugger effectively, you should become familiar with the terms, symbols, and reserved keywords listed in Tables 7-1 through 7-4.

After the program to be debugged has been compiled and linked with the debug option, you can invoke the debugger with the following command:

FORMAT:

```
DEBUG program_name
```

ARGUMENTS:

program_name

Name of the bound unit to be debugged.

Table 7-1 lists the debugger directives by function, indicating the directive name and its meaning. Table 7-2 is a list of terms used in debugger directives. Table 7-3 is a list of debugger special symbols and their meanings. Table 7-4 is a list of reserved keywords. The reserved keywords and special symbols should not be used as variable names or labels in programs to be run with the debugger.

Table 7-1. Summary of Debugger Directives

Function	Directive Name	Meaning
Breakpoint control	AT	Set breakpoints
	CLEAR	Clear specified or all breakpoints
	LIST	List current breakpoints
Trace trap control	TRACE	Trace flow of program
Display and modification of data	CHANGE	Change specified variable's control contents
	DUMP	Display specified variable (dump)
	SET	Set values represented by special symbols
General execution	GO	Resume execution
	IF	Conditional requirement for breakpoint and request lists
	MODE	Switch between symbolic and numeric modes
	ACTIVATE	Change reference to a different object unit
	PAUSE	Enter interactive mode
	QT	Terminate debugger (quit)
	SP	Temporarily suspend the Multi-User Debugger; return control to the Command Processor (sleep)
	STEP	Execute one program statement

Table 7-2. Terms Used in Debugger Directives

Term	Definition
Character string	A string of characters enclosed in apostrophes (') or quotes. The string may include all printable characters except those used for terminal editing. Use two apostrophes to include an apostrophe in the string.
Directive	A statement to the debugger containing keywords, which cause the debugger to perform a specified action (e.g., AT, CLEAR, TRACE).
Hex value	A maximum of eight hexadecimal digits prefixed by a percent (%) sign.
Identifier	A name containing a maximum of 30 characters. Valid characters include all alphanumerics, the hyphen, and the underscore. The first character must be alphabetic. NOTE A minus sign in an expression must be preceded and followed by a blank to distinguish it from a hyphen.
Input unit	A line consisting of one or more debugger directives separated by semicolons. The maximum length is the input device's maximum line length.
Integer	A value less than 65535 and greater than or equal to -65536 entered as a string of decimal digits.
Statement	A single source statement that generates executable code.
Type	Identification of the internal storage and external display formats of variables. All source-defined variables have a basic type such as integer, real, character, string, or alphabetic.
Variable	A single field, array element, entire array, record, or single component of a record.

Table 7-3. Debugger Special Symbols

Symbol	Definition
$R1-$R7	Index registers (containing data)
$B1-$B7	Base registers (containing addresses)
$P	Program counter
$	Current breakpoint
$T	True
$F	False
$L	Prefix for numeric statement labels

Table 7-4. Debugger Reserved Keywords

```
AT          P
CH          QT
CHANGE      SP
C           -NUM
CLEAR       -NUMERIC
DUMP        -SYMBOLIC
IF          $
L           $P
LIST        $R1-$R7
MODE        $B1-$B7
ACTIVATE    $L
PAUSE       $F
SET         TR
TRACE
```

After invoking the debugger, you are prompted with the greater than character (>). To initialize the debugger, set, list, and clear breakpoints using the AT, LIST, and CLEAR directives.

After you have arranged breakpoints to your satisfaction, use the SP directive to return to the Command Processor. Execute your program normally. When a breakpoint is reached, program execution is suspended and the debugger enters interactive mode. Now you can enter any valid debugger directive to debug your program.

To leave the debugger, either continue execution of the program to the normal end (perhaps clearing unwanted breakpoints with the CLEAR directive) and then enter the GO directive, or enter the Quit directive which terminates debugger control and resumes normal execution of your program.

While the debugger is in operation, it maintains two internal variables which identify the current bound unit and the current debugging mode:

- current_object_unit identifies the object unit to which symbolic debugging directives refer. The initial default is the first linked object module with a symbol table. During execution of a program, current_object_unit is automatically changed to identify the object unit in which the most recent breakpoint occurred. You can override the automatic setting with the SET directive.

- current_mode is automatically set to symbolic if the debugger is invoked with a bound unit name.

DEBUGGER AND BREAK KEY FUNCTIONALITY

Typing DEBUG after pressing the Break key and getting the **BREAK** message transfers you to the debugger. To return to the previous level, enter the SP directive or terminate the debugger completely with the QUIT directive.

If DEBUG was the task that was broken, the responses allowed are:

- Any command
- UW, PI, SR or NEW_PROC

NOTE

The PI response will return the user group to the debugger input mode and allow the entry of debugger directives.

If the Debugger task was broken and DEBUG was entered as the response, the user would be placed in the debugger input mode.

The UW response will cause the debugger to be exited as if a GO or SP directive had been input depending on which was appropriate for the current state of the debugger. That is, if debug had been invoked as the result of encountering a breakpoint, a GO is appropriate to exit debug.

PLANNING CONSIDERATIONS

Controlling Execution of the User's Program

The following directives can be used to control execution of the user program.

IF is used to specify if statement processing.

GO causes execution to resume.

STEP steps through a program one executable statement at a time.

PAUSE enters input mode and allows you to request display of variables and registers.

PAUSE used with IF causes execution to stop and entry into interactive mode.

Setting Breakpoints

You can set, list, and clear breakpoints during initialization as well as during execution. All other functionality can only be done during execution. The AT directive sets breakpoints; CLEAR deletes them. The LIST directive lists breakpoints.

Monitoring the Value of Variables

The IF directive can be used in a number of ways to perform other directives when a certain condition is met. The primary use of the IF directive is to monitor the value of a variable during execution of the user's program. If the value meets a given condition, program execution stops and the user is notified.

Controlling Output

The DUMP directive displays the contents of variables and constants.

Maintaining a Trace History

The TRACE directive controls tracing program execution. It can be used in conjunction with IF for conditional tracing. You can issue the TRACE directive only while stopped at a breakpoint.

Altering Values

The CHANGE directive alters the values of variables. SET sets or alters the values of special symbols.

DEBUGGER DIRECTIVES

The remainder of this section provides an alphabetic listing of the debugger directives with detailed descriptions for each directive.

AT

Set breakpoints in the program.

FORMAT:

AT location_list [(request_list)]

ARGUMENTS:

location_list

One or more places in the program where you want to set a breakpoint. Location specifiers are separated by commas. Individual statements in the program are identified either by statement label or source line number. Set breakpoints only on executable statements. For FORTRAN programs, do not set breakpoints on FORMAT statements.

request_list

Optional list of one or more directives to be executed when a breakpoint is reached. A request list can be a single directive or a series of directives delimited by parentheses. Directives in a request list are separated by either semicolons or newline characters. GO is understood to be the last directive in a breakpoint request list. If no request list is given, PAUSE is assumed.

DESCRIPTION:

A breakpoint is set in the program for every statement in the location list. A breakpoint identifier is assigned to each breakpoint, and a brief message is printed showing the line number, label (if any), and assigned identifier number. Breakpoint identifiers are numbers assigned in descending order beginning with 31. You can set a maximum of 32 breakpoints (from 31 through 0, inclusive). Request_list directives are saved in the DEBUG.SM file for each breakpoint. To distinguish FORTRAN statement labels from line numbers, prefix statement labels with "$L".

Examples:

```
AT 1020,LOOP (PAUSE)
AT 1020,LOOP
```

These two directives are equivalent. Both cause program breakpoints to occur before execution of statement number 1020 and the statement labeled LOOP. When a breakpoint occurs, the debugger enters interactive mode and prompts you (with the "greater than" (>) sign) to enter directives from the terminal.

```
AT LOOP1 (DUMP INVENT_PART_NO,A(J_INDEX),J_INDEX;CH BUFNO=1)
```

Each time execution reaches LOOP1, the DUMP and CHANGE directives are performed, and execution of the target program resumes without user intervention.

CHANGE

Alter the contents of variables.

FORMAT:

```
{CHANGE}
{CH    }  change_list
```

ARGUMENT:

change_list

List of change statements of the form:

```
                                 {decimal integer    }
variable name                    {hexadecimal string }
record component  =              {character string   }
                                 {$T                 }
                                 {$F                 }
```

DESCRIPTION:

For each change statement, the item on the left is assigned the value of the element on the right. Change statements are separated by commas.

Example:

```
CHANGE TAGA = %3031, TAGC = 5
```

The hexadecimal value 3031 is placed in TAGA and 5 is loaded into TAGC.

The data types of the left side and right side must match. For example, a variable name defined as a hexadecimal string may not be changed to a decimal integer. The one exception is that any type variable may change to a hexadecimal string that represents exactly the internal format of the data. There is no other conversion of data types.

CLEAR

CLEAR

Delete breakpoints.

FORMAT:

```
{CLEAR}  breakpoint_list
{C    }
```

ARGUMENT:

breakpoint_list

List which can include:

- Individual breakpoint numbers
- A range of breakpoints (e.g., N1 to N2)
- $ (indicating the current breakpoint)
- * (indicating all currently defined breakpoints)

DESCRIPTION:

All breakpoints specified in the breakpoint list are deleted.

Examples:

```
CLEAR *
```

Clear all currently defined breakpoints.

```
C $, 14, 15, 0 TO 5
```

Clear multiple breakpoints.

DUMP

Display program variables and other information about the program execution.

FORMAT:

```
{DUMP}
{DP  }  dump_list
```

ARGUMENT:

dump_list

List of items separated by commas, which can include:

- Variable names representing individual values, or record structures:

```
DUMP HEAD_OF_LIST, TABLE, MASS_REC
```

- Range of variables as declared textually in the program:

```
DUMP ABC TO J_MODE
```

- Record component:

```
DUMP C OF B OF A
DUMP C IN B IN A (these two are equivalent).
```

If "#" precedes a variable name, the variable is displayed in hexadecimal.

DESCRIPTION:

Each requested item is displayed on a new line. Record components are indented according to level. The item name is printed on the left followed by the item value. The value is printed in a format conforming to its data type unless the hexadecimal override character (#) precedes the name.

Example:

```
DUMP ABC, 'RESULT'
```

Display the contents of variable ABC and print the character string RESULT.

GO

GO

Resume execution of the program.

FORMAT:

```
GO [program_location]
```

ARGUMENT:

program_location

Either a statement label or a statement line number.

DESCRIPTION:

If no argument is given, the debugger resumes execution of the program where it left off. If a program location is given, the debugger resumes execution at the new location.

NOTE

The program_location option should be used with caution because registers used by the bound unit may be scrambled by such a jump.

IF

Provide a simple conditional for use in breakpoint and trace request lists.

FORMAT:

```
                  =          character string
                  ^=         hexadecimal string
IF  variable      >     {    decimal literal     }   request_list
                  >          $T
                  <          $F
                  <
```

ARGUMENTS:

= Equals

^= Not equal

> Greater than

>= Greater than or equal to

< Less than

<= Less than or equal to

request_list

Required list of one or more directives to be performed when the IF expression evaluates to True. A request_list consists of a single directive or a series of directives delimited by parentheses. Directives in a request_list are separated by either semicolons or new line characters.

DESCRIPTION:

When an IF directive is performed, the comparative expression is evaluated. If the result is True, the request_list is performed. When execution of the request is completed (without encountering a GO or PAUSE), processing continues with the next directive following the list.

If the result of the comparison is False, the request_list is ignored.

The data types on the left and right of the relation must match. For example, a variable name defined as a hexadecimal string may not be compared to a decimal literal. The one exception is that any type variable may be compared to a hexadecimal literal which represents exactly the internal format of the data. There is no other conversion of data types.

The length of a comparison corresponds to the language rules. If an expression being evaluated is determined to be invalid, an error message is issued and the debugger pauses.

Examples:

```
AT 1020 (IF VAR1 = 'ABCD'(PAUSE))
AT 1020 (IF VAR1 = %41424344 (P))
```

These two directives are both valid assuming the base type of VAR1 is a character string. A True condition causes a Pause; False causes a Go.

```
AT 1020 (IF BOOL = $F (PAUSE))
```

The debugger pauses only if BOOL is False.

```
TR (IF J_INDEX < 0 (PAUSE))
```

Establish tracing for all statements in the compiled program. At each statement, the IF expression is evaluated. If J_INDEX is ever less than zero, the PAUSE occurs. Otherwise, the program continues.

LIST

List current breakpoints.

FORMAT:

```
{LIST}  [breakpoint_list]  [-LG]
{L   }
```

ARGUMENT:

breakpoint_list

List, which can include:

- Individual breakpoint numbers
- A range of breakpoints (e.g., N1 to N2)
- * (indicating all currently defined breakpoints)
- $ (indicating the current breakpoint).

DESCRIPTION:

The breakpoint identifier (an integer between 0 and 31), the line number, and label (if any) are printed for all specified breakpoints. In addition, if -LG is given, the request list (if any) associated with each breakpoint is printed.

Examples:

```
LIST 3,4,10 TO 15 -LG
```

Print the basic information and request lists for breakpoints 3, 4, and 10 through 15.

```
L $
```

Print the basic information for the breakpoint at which the program is currently stopped.

MODE

MODE

Define the debugger mode of operation desired. Specify Symbolic mode for Advanced COBOL and Advanced FORTRAN program debugging. Specify NUMERIC mode for Assembly language program debugging. See the System Programmer's Guide, Volume I for information on numeric debugging operations.

FORMAT:

```
MODE { {NUMERIC} }
     { {NUM     } }
     {           }
     { {SYMBOLIC} }
     { {SYM     } }
```

DESCRIPTION:

The debugging mode is set as specified.

If the debugger is currently in symbolic mode, type MODE NUMERIC to put the debugger in numeric mode. See the System Programmer's Guide, Volume I for a description of numeric directives.

If the debugger is currently in numeric mode, type MODE SYMBOLIC to put the debugger in symbolic mode.

Symbolic mode can only be used with bound units which have been compiled with the debug option on. Also a bound unit must have been initialized for symbolic debugging prior to a switch from numeric to symbolic mode.

ACTIVATE

Change reference to a different object unit.

FORMAT:

```
{ACTIVATE}    object_unit_name[/overlay_name]
{AC      }
```

ARGUMENTS:

object_unit_name

Name of the object unit other than the current one.

overlay_name

Number of the overlay in which the bound unit is linked.

DESCRIPTION:

The object unit named becomes the current object unit.

Example:

```
AC PROC1
DUMP ABC
AC PROC2
DUMP DEF
AC PROC3/4
DUMP XYZ
```

ABC, DEF, and XYZ are variables declared in three separately compiled object units. This sequence of directives displays ABC, activates the PROC2 symbol block, and displays DEF. Then the symbol table for PROC3, linked in overlay 4, is activated, and variable XYZ is displayed.

PAUSE

PAUSE

Enter interactive mode.

FORMAT:

```
{PAUSE}
{P    }
```

DESCRIPTION:

When a PAUSE directive is performed, the debugger enters interactive mode, sends a prompt message (the "greater than" sign (>)) to the user-in file, and reads the user-out file (generally a terminal) to obtain its next directive.

When a PAUSE is encountered within a request_list, it takes effect immediately, and any directives remaining in the list are ignored.

Examples:

```
AT 1020 (DUMP I,NEXT;IF NEXT=%40 (PAUSE))
```

Whenever line 1020 is reached, the variables I and Next are dumped. Then, the IF expression is evaluated. If NEXT is a hexadecimal 40, the debugger pauses. Otherwise, execution of the program resumes at statement 1020.

QUIT

Clear all breakpoints, close all debugger work files, and disable the debugger trap handler before terminating the debugger task.

FORMAT:

QT

SET

Set values represented by special symbols.

FORMAT:

$$\text{SET} \begin{Bmatrix} \$Bn \\ \$Rn \\ \$P \end{Bmatrix} = \begin{Bmatrix} \$Bn \\ \$Rn \\ \$P \\ \text{hex value} \\ \text{dec value} \end{Bmatrix} \begin{Bmatrix} + \\ - \\ * \\ / \end{Bmatrix} \left[\begin{Bmatrix} \$Bn \\ \$Rn \\ \$P \\ \text{hex value} \\ \text{dec value} \end{Bmatrix} \right]$$

ARGUMENTS:

$Bn	base register
$Rn	index register
$P	p-counter
hex value	hexadecimal value
dec value	decimal value

DESCRIPTION:

For each set_list item, the item on the left is assigned the value of the item on the right. The expression must evaluate to a hexadecimal value when setting base or P-registers, or to a decimal or hexadecimal value when setting index registers.

Example:

```
SET $R1 = %F3E2
```

Register R1 is set to hexadecimal value F3E2.

NOTE

The SET directive is generally not useful to users running only COBOL or FORTRAN programs.

SP (SLEEP)

Return processing to command level after initial breakpoints have been set.

FORMAT:

SP

DESCRIPTION:

The SP directive temporarily suspends the execution of the debugger and returns control to the Command Processor. You may now start execution of the bound unit in the standard manner. The debugger becomes active again if:

1. A breakpoint is reached in the user's program.
2. The user types DEBUG from the command level.

TRACE

TRACE

Trace the flow of a program. It can also be used in conjunction with the IF directive to monitor the contents of a variable. You must be at a breakpoint to issue the TRACE directive.

FORMAT:

```
{TRACE}  -OFF    [(request_list)]
{TR   }
```

ARGUMENTS:

-OFF

Turn tracing off for the current bound unit.

request_list

Optional list of one or more directives to be performed when a tracepoint is reached. A request list consists of a single directive or a series of directives delimited by parentheses. Directives in a request list are separated by either semicolons or new line characters.

DESCRIPTION:

A trace is defined and becomes active when the next Go directive is entered. As each statement is executed, a trace message consisting of the line number, bound unit name, and statement label (if any) is listed, the request list (if any) is performed, and execution of the program continues.

Multiple Trace directives may be entered, but only one request_list (the last one entered) will be in effect.

Examples:

```
TRACE (DUMP TAG)
```

Print the variable TAG at every statement.

```
TR (IF J_INDEX = 0 (PAUSE))
```

Monitor the contents of J_INDEX during execution of the current object unit. If J_INDEX goes to zero, execution pauses.

Section 8
NETWORK PROCESSING FUNCTIONS

DPS 6 DSA is Honeywell's minicomputer-based networking system. DPS 6 DSA combines processing power and high-speed information dissemination over communications lines. This is also called data communications. Data communications carried out in real time (i.e., multiple, concurrent accesses from terminals to a computer with practically instantaneous response) is called online processing.

A DPS 6 DSA system is a combination of hardware and software processing facilities. Hardware facilities include the central processing unit, peripherals, terminals, lines, and modems (or related devices). Software facilities include user application programs, a user access method, and a network control center.

NETWORK CONTROL CENTER

The network control center is a software product that runs under the MOD 400 Executive. The control center resides in a region of main memory and manages this region much like the Executive manages the machine environment: handling terminal and file I/O operations, dispatching concurrent processing activities, validating user requests for services, and performing other network-related functions. DPS T DSA software allows a MOD 400 application on one node to establish a session with an application on a different node to exchange application data.

Network software exchanges data between programs executing in the same or separate computers. The layer of network software that establishes connections, queues message text, and synchronizes the cooperating process is called session control.

This section summarizes and briefly describes the COBOL Session calls for use in Advanced COBOL programs. For complete information on Network application programming, see the Network Programmer's Guide.

The session control software provides two types of calls:

Workstation Administration Commands
COBOL Session Calls

The Workstation Administration commands create the networking environment: the workstation, mailboxes, and session type descriptions. COBOL Session Calls manage sessions. In addition to the COBOL Session Calls there are COBOL COPY files that are used to create and modify the session control I/O request blocks.

NETWORK ENVIRONMENT OF A PROCESS

Four types of entities compose the network environment of a process:

Workstation
Mailbox
Session Type Description
Task Group

The commands that define the data entities create parameter lists in the user's program space. These parameter lists do not control sessions; they provide input to the activating calls. These commands create data structures in network memory. Once the activating call completes, the parameter list in user's space is free. This structure can be modified and reused. Thus, you could create a single mailbox parameter list in program space, activate it, modify it to create a different mailbox, and then activate that mailbox.

WORKSTATION ADMINISTRATION COMMANDS

The Workstation Administration commands create and activate entities required by COBOL programs. For information on COBOL programming considerations, see the Network Programmer's Guide.

A brief description of the Workstation Administration Commands follows:

Activate Workstation	NWKS
Attach Task Group to Workstation	NATWS
Detach Task Group From Workstation	NDTWS
Activate Mailbox	NMBX
Activate Session Type Descriptor	NSTD
Print Expanded Network Structures	NPX
Modify Workstation	NMWKS
Deactivate Workstation	NDWKS
Deactivate Session Type Descriptor	NDSTD

Activate Spawning Mailbox EC	NMBXEC
Deactivate Mailbox	NDMBX

COBOL SESSION CONTROL I/O REQUEST BLOCK CALLS

Three data structures are used directly by the calls that manage the session. The data structure addresses are input parameters to the session calls, and data are returned to these structures when the session calls complete. These data structures are GCOS 6 standard extended input/output request blocks (IORBs) and are in user space. The session management request blocks are:

Session Initialization Request Block	NSIRB
Session Control Request Block	NSCRB
Session Recovery Request Block	NSRRB

Session initialization request blocks (NSIRBs) are used by the following COBOL session calls:

Accept Session
Initiate Session

Session control request blocks (NSCRBs) are used by the following COBOL session calls:

Cancel Enclosure
Receive
Send

Session recovery request blocks (NSRRBs) are used by the following COBOL session calls:

Send Interrupt
Receive Interrupt
Terminate Session
Test Session
Wait Session
Request Attributes

In COBOL I/O request block administration, you COPY the request block data division declarations from library files to create and modify data structures. Your program uses the copy file as a template.

For detailed descriptions of the COBOL session calls and contents of the COBOL copy files, see the Network Programmer's Guide.

COBOL SESSION CALLS

The COBOL session calls allow your program to control the establishment, management, and termination of sessions.

The COBOL session calls are described briefly below. The appropriate COBOL calls, IORB used by each call, and processing method of the call (synchronous or asynchronous is also specified).

For detailed descriptions of the COBOL session calls, see the Network Programmer's Guide.

Name	Call	IORB	Synchronism
Accept Session	ZACPTS	SIRB	Async
Cancel Enclosure	ZCANCL	SCRB	Sync
Initiate Session	ZINITS	SIRB	Async
Receive	ZTRECV	SCRB	Sync
Receive Interrupt	ZRINT	SRRB	Sync
Request Attributes	ZRAT	SIRB	Sync
Send	ZTSEND	SCRB	Sync
Send Interrupt	ZSINT	SRRB	Sync
Terminate Session	ZTERM	SRRB	Async
Test	ZTESTS	SIRB	Sync
Wait	ZWAITS	SIRB	Sync

Section 9
PATCH UTILITY

The Patch utility is used to apply patches to and remove patches from object units and bound units. Patches are identified by patch-ids. The Patch utility can also be used to list, by patch-id, all patches for an object unit or bound unit. The listing is written to the user-out file, terminal line screen, or printer for a hard copy.

The Patch utility, in modifying object or bound units, will extend the file space, as necessary. Insufficient file space will terminate Patch operations; therefore, you should ensure that sufficient space exists to accommodate the patch(es) on the medium (disk, etc.).

USING THE PATCH UTILITY

Patch execution is controlled by directives entered to Patch through the operator's terminal, user terminal, a card reader, or a sequential file. The Patch utility operates in batch mode or in interactive mode. Each mode is described separately below.

Batch Mode

In batch mode Patch processes directives and applies them to the file specified on the Patch command line. These directives allow patch application with or without verification of patches, the elimination of patches, and listing patches.

By using Patch directives, the user can:

- Manipulate shared and system attributes of bound units
- Create and maintain a version number facility for the bound unit
- Provide a method for patching all references to undefined external references in bound units
- Interrogate the current contents of a bound unit.

The Patch utility processes version number and attribute modification. It also interrogates directives as they are entered. Regardless of the input sequence of other directives, Patch processes directives in the order: eliminate patches, apply patches, and then list the patches.

Interactive Mode

By specifying the Patch command with the -IA argument, a bound unit can be patched in interactive mode. In interactive mode, Patch directives must be completed before they are applied; a directive is completed when the Patch utility reads a new directive. Only the file specified on the Patch command line can be patched with each invocation of the Patch utility.

Version number processing, manipulation of the shared or system attributes, and interrogation are always performed as the directives are keyed-in.

The Patch directives are listed and briefly defined below. Detailed descriptions for each Patch directive are provided later in this section.

Directive Name	Function
CLSY	Clear system bit
DP	Apply patch(es) to data section of bound unit or to common area of object file
EP	Eliminate named patch or all patches
GO	Process previous patch directive
GNSH	Set global share bit off
GSHR	Set global and root share bits on
HP	Apply hexadecimal patch(es) to specified file

Directive Name	Function
LDEF	Assign an address to an undefined external location reference
LN	List patches but do not exit from Patch
LP	List patches and exit from Patch if mode is batch
LS	List patches by name only and exit from Patch if mode is batch
NS	Set share bit off
Q	Process previous patch directives and exit from Patch
SD	Apply symbolic data patch(es)
SP	Apply symbolic patch(es)
SS	Set share bit on
STSY	Set system bit on
VDEF	Assign a value to an undefined external symbol
VN	Verify or change revision number of bound unit
WA	Interrogate bound unit
*	List a comment on the user-out file

LOADING PATCH

To load Patch, enter the PATCH command, as follows:

FORMAT:

```
PATCH filenm [ctl_arg]
```

ARGUMENTS:

filenm

Pathname of the object unit file or bound unit file to be patched. If an object unit is being patched, the last two characters of the pathname must be .O.

ctl_arg

The following control arguments may be entered:

-IA

Operate in interactive mode. Process one directive at a time; error messages (if any) immediately follow the applicable directive. If this argument is not specified, Patch operates in the batch mode.

-IN path

Pathname of the device through which Patch directives will be entered; can be the operator terminal, another terminal, a card reader, or a sequential file. Error messages are written to the error-out file. Patch error messages are described in the System Messages manual.

Default: Device specified in the in_path argument of the "Enter Batch Request" or "Enter Group Request" command.

-M6

Bound unit to be processed was created by the MOD 600 Linker.

This argument should not be used when patching an object file. (The .O at the end of the filenm on the command line identifies to Patch that the file is an object file.) Default: MOD 400.

-PROMPT
-PT

If input is from the operator terminal or another terminal, each time the PATCH utility program is ready to accept an input line, the typeout P? appears on the input device.

Default: No prompt.

-SI

Suppress the display of the sign-on message (i.e., PATCH, followed by the revision number and the date patch was created). Default: Patch sign-on message is displayed.

-SIZE n
-SZ

Create a patch work area of n 1024-word blocks of memory.

Default for n: 1.

Patch can operate on two types of files:

- Object units, which are variable sequential files created by the compilers
- Bound units created by the Linker.

SUBMITTING PATCH DIRECTIVES

Each Patch directive consists of only a directive name or a directive name followed by one or more values. Values must be separated by a delimiter. The delimiter can be a space, a comma, or a semicolon. However, on an interactive device (i.e., a terminal), the carriage return replaces the delimiter. Lines may neither begin nor end with a comma or semicolon. If directives are entered from a card reader, trailing blanks or column 80 replace the delimiter.

Multiple Patch directives may be specified during one execution of the Patch utility. To patch another bound unit or object unit, Patch must be re-executed.

For patching in the interactive mode:

- Patch directives are processed in the sequence in which they are entered.
- Patch directives can be entered in any order, except that Quit (Q) must be entered last.
- A patch directive must be complete before it is processed; it is complete when Patch reads a new directive.

For patching in the batch mode:

- The List Patches Now (LN) directive must be the first directive, otherwise, it is processed like an LP directive.
- Patches are first eliminated, then applied, and finally listed regardless of the sequence in which the associated directives are entered.

- The version number directives (VN), the share bit and systems bit directives (SS, STSY, CLSY, GSHR, GNSH, and NS) are always processed interactively in the order in which they are entered.
- The WA directive is processed when it is entered.

If directives are being entered through the operator terminal or another terminal, press RETURN at the end of each line. Each time RETURN is pressed, except after quit, the typeout P? is reissued if the prompt control argument was specified in the command line.

To enter Patch directives for a different file, you must reload Patch, specifying a different file in the filenm argument.

PATCHING TECHNIQUES

Techniques used when "Naming the Patch" and "Applying the Patch" are described in the following paragraphs.

Naming the Patch

Each patch has a patch-id by which it is identified. When you designate in Patch directives (DP, HP, SD, or SP) that one or more patches are to be applied to a specified object unit or bound unit, you must specify a patch-id. The patch-id identifies the patch(es) and designates whether the patch(es) are to be applied to an object unit, root, or overlay of a bound unit. To eliminate patches from an object unit or bound unit, you must specify in the Eliminate Patch directive the patch-id with which the patch(es) are associated. See "Data Patch Directive (DP)" for a description on how to designate patch-ids.

Applying the Patch

If an object unit is being patched, object records are created for the specified patches and appended to the end of the object file. When the object unit is processed by the Linker, existing values are replaced with the specified patch values. Locations that contain external references should not be patched; results are unspecified.

If a bound unit is being patched, each specified patch value is applied directly to the proper image record in the bound unit. The previous value, the patch-id, and the patch value are saved in a Patch history record that is written at the end of the file area allocated to the bound unit. This record is referred to each time a List Patch or Eliminate Patch directive is specified.

NOTE

Use caution when patching executing bound units. If a program or one of its overlays is loaded while in the process of being patched, results are unspecified.

PATCH DIRECTIVES

The Patch directives are described on the following pages in alphabetic order by directive function.

CLEAR SYSTEM BIT

CLEAR SYSTEM BIT

Turn off the system bit in the bound unit's attribute table. This directive prohibits the patched bound unit from executing in the system ($S) group. The CLSY directive is not allowed for object files.

FORMAT:

```
CLSY
```

NOTE

The system bit was initially set at link time by the SYS Linker directive.

COMMENT

List the accompanying text on the user-out file. The contents of the Comment directive are not saved.

FORMAT:

```
* comment-text
```

DATA PATCH

DATA PATCH

Apply (for bound units) one or more hexadecimal patches, by relative location, to the data section of the bound unit. The bound unit must have separate code and data sections, and have been created by the Linker when the -R Linker argument is specified.

For object files, the DP directive causes patches to be applied to common areas.

FORMAT:

For Bound and Load Units, Without Verification:

```
DP patch-id /addr patchval[ patchval...][ /addr
    patchval...]
```

For Bound and Load Units, With Verification:

```
DP patch-id /addr (verval patchval[ verval patchval...])
    [/addr (verval patchval[ verval patchval...])]
```

For Object Files, Without Verification -- Local Common Block:

```
DP patch-id /offset1 patchval[ /offset1 patchval]...
```

For Object Files, With Verification -- Local Common Block:

```
DP patch-id /offset1 (verval patchval)[ /offset1
    (verval patchval)]...
```

For Object Files, Without Verification -- Named Common Block -- One Blockname Per Directive:

```
DP patch-id blockname /offset patchval[ patchval...]/
    offset patchval [patchval...]...]
```

For Object Files, With Verification -- Named Common Block -- One Blockname Per Directive:

```
DP patch-id blockname /offset (verval patchval[ verval
    patchval]...)[/offset (verval patchval[verval
    patchval]...)...)]
```

ARGUMENTS:

patch-id

Patch-id of the patch(es) to be applied. A patch-id comprises eight to ten characters: the first six characters can be any ASCII characters except spaces. The last two to four characters must identify the root or overlay to which the patch(es) are being applied. If an object unit or the root of a bound unit is being patched, the patch-id is eight characters, the last two of which must be RT. If an overlay is being patched, the last two to four characters identify the hexadecimal overlay number; the first overlay is 00 for bound units created by the Linker, and subsequent overlays are numbered consecutively in ascending order. There may be no embedded blanks. Within the root and each overlay, patch-ids must be unique.

/addr

Relative location at which the first (or only) subsequent patch value will be applied. Each address must comprise one to six right-justified, hexadecimal characters, and must be preceded by the slash character (/). Subsequent patch values, if any, are applied to succeeding memory locations.

NOTE

Care must be taken in specifying an address to be patched. If the address of a location to be patched is identified when a bound unit is being executed, that memory address contains three possible factors:

1. The original address of the location in the bound unit relative to the beginning of the bound unit
2. The linking relocation factor
3. The loader relocation factor

If the address is identified at execution time and the bound unit is to be patched, the loader relocation factor must be subtracted from the address

identified in the executing bound unit. If the object unit is to be patched, both the linking and loader relocation factors must be subtracted. Object unit locations can also be obtained through examination of the listing produced during assembly of the object member.

offsetl

Non-negative offset from the beginning of $LCOMW.

patchval

Specify a value of one to six hexadecimal characters to insert into $LCOMW. Relocatable values are not permitted and only one patch value can be specified for each patch.

blockname

Symbolic name of the common block. The name can contain one to six characters.

offset

Offset from the symbol name of the common block.

/patchval

Value to be inserted at an address, replacing the contents of that location. The value must be specified as one of the following:

1. Data, represented by one to four hexadecimal characters.

2. Relocatable address, represented by one to six hexa- decimal characters, preceded by the character <.

verval

Verification value; one to six hexadecimal characters specifying value that should be in location before patch is applied.

NOTES

1. Each verval must be immediately followed by a patchval.

2. The verification value(s) and patch value(s) associated with each address must be enclosed within parentheses.

3. For consecutive locations, the old and new values can be included within one set of parentheses. The /addr field is adjusted internally by Patch.

4. Within a set of parentheses, the number of old values must equal the number of new values.

5. The IMA indicator cannot be used with an old value. IMA status is determined by Patch from the module or from the new value.

6. For SLIC or LAF IMAs, old value and new value can be up to six characters.

7. For SLIC or LAF IMAs, Patch allocates two words. For example, assume that the following directive applies to a SLIC module:

   ```
   DP patch-id,/100,(1111,<12345,ABC,DEF)
   ```

   If the contents of 100 and 101 are 001111, and the contents of 102 are ABC, the patch will be applied, and as a result the contents of the specified addresses will be:

Address	Contents
100	01
101	2345
102	DEF

8. Verified and nonverified patches can be included within one patch directive; however, if the verify fails, none of the addresses in the directive are patched.

9. A left parenthesis cannot immediately follow a right parenthesis. There must be a /addr field between them.

10. In a bound unit, an IMA may be patched to a non-IMA or a non-IMA is patched to an IMA.

11. In object modules, patches to areas that have no defined value cannot be verified.

12. In a bound unit, if the new value is not an IMA, the old value can be no more than four hexadecimal characters even if the old value is an IMA.

NOTE

SLIC means SAF/LAF independent code. MOD 400 bound units are usually LAF, but SLIC bound units may still be patched and executed.

ELIMINATE PATCH

Eliminate all patches associated with a specified patch-id from the designated object unit or bound unit. The patch(es) must have been previously applied by DP, HP, SD, or SP directives. To determine what patches have been applied, and their patch-ids, enter one of the list patch (LN, LP, LS) directives described later in this section.

FORMAT:

EP {patchid / ALL}

ARGUMENTS:

patchid

Patch-id of the patch(es) to be removed. A patch-id comprises eight to ten characters: the first six characters can be any ASCII characters except spaces; the last two to four characters must identify the root or overlay to which the patch(es) are being applied. If an object unit or the root of a bound unit is being patched, the patch-id is eight characters, the last two of which must be RT. If an overlay is being patched, the last two to four characters identify the hexadecimal overlay number, the first overlay is 00 for bound units created by the Linker, and subsequent overlays are numbered consecutively in ascending order. There may be no embedded blanks. Within the root and each overlay, patch-ids must be unique.

ALL

If the ALL option is used, all patches in the file are eliminated in the order that they were applied.

GO

Tell Patch that the previous directive is complete and is to be processed. This directive is effective only in the interactive mode. In the interactive mode, a new Patch directive signals the end of the previous one. The Go directive is used in circumstances in which the user would like to have a directive processed before entering any other directive.

FORMAT:

GO

HEXADECIMAL PATCH

Apply one or more individual patches, by relative location, to an object unit or bound unit.

If a bound unit is being patched, you can designate that specified patch(es) be applied only if specified location(s) currently contain specified value(s); these are called verification values. Within a single HP directive, verification values may be specified for some or all of the locations. If any of the verification values do not match the values currently at the locations for which verification values were specified, none of the patches specified in the HP directive are applied.

FORMAT:

Without Verification Values:

```
HP patch-id,[base,]/addr,patchval[,patchval...patchval]
   [,/addr,patchval[,patchval...patchval]]...
```

With Verification Values:

```
HP patch-id,[base,]/addr,(verval,patchval[,verval,
   patchval])[,/addr,(verval,patchval[verval,
   patchval)])]
```

NOTES

1. One or more lines of arguments may be specified. When two or more lines of arguments are entered for an HP directive, the last character on each line must be a valid hexadecimal character or right parenthesis. Individual fields, values, and addresses must not be split between lines. The entry of a Patch directive name (e.g., EP, LP) at the beginning of a line designates the end of the previous Patch directive.

2. A space may be used in lieu of a comma as a separator.

ARGUMENTS:

patchid

Patch-id of the patch(es) to be applied. A patch-id comprises eight to ten characters: the first six characters can be any ASCII characters except spaces; the last two to four characters must identify the root or overlay to which the patch(es) are being applied. If an object unit or the root of a bound unit is being patched, the patch-id is eight characters, the last two of which must be RT. If an overlay is being patched, the last two to four characters identify the hexadecimal overlay number; the first overlay is 00 for bound units created by the Linker, and subsequent overlays are numbered consecutively in ascending order. There may be no embedded blanks. Within the root and each overlay, patch-ids must be unique.

base

Optional argument allowed only for bound units. Base defines a value that is added to all locations; i.e., /addr specified in the associated DP, HP, SD, or SP directives and all IMA references. If this argument is omitted, the default value is zero. Base can be entered as a hexadecimal address of one to six characters or as a name that has been specified as an EDEF at link time and placed in the bound unit symbol table. If a symbol name is used, Patch finds the name in the symbol table and uses its address as the base value. The format for the symbol name as a base is +symname, where symname comprises 1 to 12 characters. If a hexadecimal address is used for base, the plus sign is not required.

For bound units created by the MOD 400 Linker the values specified for the /addr fields and IMA references (if any) must include the displacement of the root or overlay. The displacement is equal to the base address of the root or overlay as printed on the link map. The user may add the displacement to each /addr field and IMA, or achieve the same result by specifying the base parameter in the Patch directive. For example, if the first overlay of a bound unit is based at 1000 and a patch to locations 100 to 103 and 200 to 204 is to be made within the overlay, the following two patch directives are equivalent when applied to a LAF bound unit.

```
SP NUMBRA00;/1100/LDR $R1, 1500;STR $R,=$R2
    /1200/ADD $R1, 1600;JMP 1156
SP NUMBRA00;1000;/100;LDR $R1,500;STR $R1,$R2
    /200;ADD $R1,600;JMP 156
```

/addr

Relative location at which the first (or only) subsequent patch value will be applied. Each address must comprise one to six right-justified, hexadecimal characters, and must be preceded by the character /. Subsequent patch values, if any, are applied to succeeding memory locations.

- NOTE -

Care must be taken in specifying an address to be patched in either an object unit or a bound unit. If the address of a location to be patched is identified when a bound unit is being executed, that memory address contains three possible factors:

1. The original address of the location in the object unit relative to the beginning of the object unit
2. The linking relocation factor
3. The loader relocation factor

If the address is identified at execution time and the bound unit is to be patched, the loader relocation factor must be subtracted from the address identified in the executing bound unit. If the object unit is to be patched, both the linking and loader relocation factors must be subtracted. Object unit locations can also be obtained through examination of the listing produced during assembly of the object unit.

patchval

The value to be inserted at an address, replacing the contents of that location. The value must be specified as one of the following:

1. Data, represented by one to six hexadecimal characters

2. Relocatable address, represented by one to six hexadecimal characters, preceded by the character <.

verval

Verification value; one to four hexadecimal characters specifying value that currently should be in location at which subsequent patch will be applied. See the notes on verification that follow the DP directive.

Example 1:

```
HP PTCHIDRT,/1B2A,1FFF,1DFC,<2BFC,2D4E,<ABF2
```

This Hexadecimal Patch (HP) directive requests that the subsequent patches, identified by the name PTCHIDRT, be applied to the root. Patch values $1FFF_{16}$ through $<ABF2_{16}$ are to be inserted in successive locations, with the first patch value $1FFF_{16}$ to be located at address $1B2A_{16}$. The hexadecimal patches are to replace any previous values in these locations. The value to be inserted in address $1B2C_{16}$ is the two word addresss $2BFC_{16}$, which is to be relocated at load time; the relocatable address $ABF2_{16}$ is to be inserted in address $1B2F_{16}$.

Example 2:

```
HP VPATCH01,/1FEA,(1A1,1B7,1A7,1B8),/1E72,8900
```

This example illustrates the use of verification values in a Hexadecimal Patch (HP) directive requesting that specified patches, identified by the name VPATCH01, be applied to overlay 01. Patch will check location $1FEA_{16}$ for the value $1A1_{16}$, and location $1FEB_{16}$ for the value $1A7_{16}$; if the values are at those locations, then the contents of locations are changed as follows: location $1FEA_{16}$ will contain $1B7_{16}$, location $1FEB_{16}$ will contain $1B8_{16}$, and location $1E72_{16}$ will contain 8900_{16} . If either of the verification values is incorrect, none of the three locations will be changed.

INTERROGATE BOUND UNIT

INTERROGATE_BOUND_UNIT

Display the current contents of locations specified on the user-out file. This directive cannot be used to display locations in object files.

FORMAT:

```
WA,[ovly,]/addr1[,words][,/addr2...]
```

ARGUMENTS:

ovly

Overlay number in hex that the address references. If this field is omitted, the root is the default. The root can also be specified as RT. For -R type bound units, this field can be DP for data section or RT for code section as well as being an overlay number.

addr

Specify the hex address within specified root or overlay indicating where the display is to start.

LDEF

Assign a specified address to an undefined external location reference and change all locations that reference this name. This directive is not allowed for object files.

FORMAT:

LDEF;symname;[<]addr[;L]

ARGUMENTS:

symname

Name of the undefined external reference that will be assigned an address; can be from 1 to 12 characters in length.

addr

Address to which symname will be assigned.

[<]

Address specified is an IMA address. If this argument is not specified, the address is treated as P+DSP.

[;L]

List all changed external references to symname on the device specified as user-out.

Default: No list.

Undefined external references in a bound unit can only be changed one time. If you make a mistake, you must use HP patch directives to correct each location containing the wrong information.

NOTE

The user should be aware that there is no history kept of the changes that are made when the LDEF directive is used. It is wise, therefore, to utilize the L argument and retain the listing for future reference.

Example 1:

```
LDEF;EPPTR;50;L
```

This directive assigns address 50 to symbol EPPTR and lists all locations that are changed to the address 50.

Example 2:

```
LDEF;PK;<50;L
```

This directive assigns address 50 to symbol PK and changes all IMA references to external symbol PK to address 50.

LIST PATCHES

Produce a listing of all patches within the object unit or bound unit being patched. The listing is produced on the user-out file.

If a bound unit is being patched, the listing designates, for each patch, the following information in the order listed: full patch-id, address at which the patch was applied, contents of the location before the patch was applied, and the patch value.

NOTES

1. In the listing, the characters that identify the root or overlay appear first, and are separated from the other character constituting the id by spaces. When a bound unit is being patched in a common area, the letters CM are printed rather than RT.

2. If termination of the listing of patches is desired before normal completion of the list process, use the BREAK facility followed by a NEW_PROC command. The PATCH program must then be reloaded.

FORMAT:

```
LP
```

Example:

```
0001 NOHLT3 000002E2 00000000 00000F02
```

This printout is one line of a listing of patches applied to a bound unit being patched. The printout has the following meaning: a patch identified by the patch-id NOHLT3 was applied to overlay 01. The patch was applied to location 02E2; this location previously contained 0000, and now contains 0F02.

If an object unit is being patched, the listing designates, for each patch, the following information in the order listed: patch-id (excluding the last two characters, which identify the root), address at which the patch was applied, and the patch value.

Example:

```
    NUMBRF  00000162   00000444
            00000163   00000222
    NUMBRH  000001A6   00000333
            000001A7   00000444
            000001A8  <00000221
            000001AA   00000004
            000001AC  <00000321
```

This typeout is a listing of patches applied to an object unit being patched. The first line designates that patch 0444, whose patch-id is NUMBRF, was applied to location 0162. Note that the last two characters of the patch-id (e.g., RT) were omitted from the printout.

LIST PATCHES NOW

List all patches in the specified file and then allow more patches to be applied. This directive is effective only in the batch mode and can be applied only to bound unit files. It must be the first directive issued. If it is not the first directive, or if it is entered in the interactive mode, it is processed the same as an LP directive. The LN directive allows the current patches to be listed and additional patches to be applied without reloading Patch.

FORMAT:

LN

Example:

```
0000  CONRCT  000000A8  0005A4D  0005A4E
```

This printout is one line of a listing of patches applied to a bound unit being patched. The printout has the following meaning: a patch identified by the patch-id CONRCT was applied to overlay 00. The patch was applied to location 000000A8; this location previously contained 0005A4D, and now contains 0005A4E.

LIST PATCH NAMES

LIST PATCH NAMES

List the names (patch_ids) of the patches in the specified file. Addresses and values are not listed.

FORMAT:

LS

Example:

0000 CONRCT

The printout is one line of a listing of patches applied to a bound unit being patched. The printout has the following meaning: The patch identified by patch-id CONRCT was applied to overlay 00.

LIST_SPECIFIED_PATCH

List those patch ids specified. Up to five patch ids can be requested per run.

FORMAT:

```
LS patchid [;PATCH_id...]
```

Example:

```
LS NUMBRART; NUMBRB00
```

In this example, the directive will cause the entire patch NUMBRART and the entire patch NUMBRB00 to be listed.

QUIT

QUIT

Inform Patch that the last Patch directive has been entered, and initiate processing of the specified Patch directives. This directive should be preceded by at least one other Patch directive. When the directive(s) have been executed, execution of Patch terminates.

FORMAT:

```
Q
```

SET GLOBAL SHARE BIT OFF

Turn off the global share bit in the MOD 400 bound unit. The share bit of the root is not affected by this directive. This directive cannot be used in MOD 600 systems nor object unit files.

FORMAT:

GNSH

SET GLOBAL SHARE BIT ON

SET GLOBAL SHARE BIT ON

Set the global share bit of the root on in the bound unit. This directive cannot be used for MOD 600 bound unit or object files.

FORMAT:

GSHR

SET SHARE BIT OFF

Turn off the share bit of the root segment of a bound unit. Patch alters the status of the share bit only; it makes no check on the sharability of the module. This directive is not allowed for object files.

FORMAT:

NS

NOTE

This is the bit that is set on by the Linker directive SHARE.

SET SHARE BIT ON

SET SHARE BIT ON

Turn on the share bit of the root segment of a bound unit. Patch alters the status of the share bit only; it makes no check on the sharability of the module. This directive is not allowed for object files.

FORMAT:

SS

NOTE

This bit designates that the bound unit is sharable with a task group.

SET SYSTEM BIT ON

Turn on the system bit in the bound unit's attribute table. This directive must be employed if the patched bound unit is to execute in the system ($S) group. The STSY directive is not allowed for object files.

FORMAT:

STSY

NOTE

Before using this directive, consult with the person responsible for system building and determine the available memory. This Patch directive is equivalent to the Linker SYS directive.

SYMBOLIC DATA PATCH

SYMBOLIC DATA PATCH

Apply patches for object units and bound units created by the MOD 400 Linker. For bound units, the directive causes patches to be applied to the data portion of separated object units. For object units, the directive causes one or more Assembly language one-word symbolic instructions to be applied to common areas, i.e., to either named or local common blocks. You can verify the current contents of locations while patching.

FORMAT:

For Bound Units -- No Verification:

SD patch-id/off_1 $patchval_1$ [/off_2 $patchval_2$...]

For Bound Units -- With Verification:

SD patch-id/off_1 ($oldval_1$;$newval_1$)[/off_2 ($oldval_2$; $newval_2$)...]

For Object Units -- Named Common Block -- No Verification:

SD patch-id;blockname;/offs;$patchval_{11}$ [$patchval_{12}$ ... $patchval_{1n}$]

For Object Units -- Named Common Block -- With Verification:

SD patch-id;blockname;/offs;($oldval_1$;$newval_1$)[($oldval_2$;$newval_2$)...]

For Object Units -- Local Common Block -- No Verification:

SD patch-id;/offs;patchval

For Object Units -- Local Common Block -- With Verification:

SD patch-id;/offs;(oldval;newval)

NOTE

You can mix verification and nonverification patches. For example: SD NUMBRART;/135;(111;CMV $R7,8;2;STR $R6,*$R1);/150;ADD $R4,1000. Only the patches at locations 135 and 136 are verified.

ARGUMENTS:

patchid

Patch-id of the patch(es) to be applied. A patch-id comprises eight to ten characters: the first six characters can be any ASCII characters except spaces; the last two to four characters must identify the root or overlay to which the patch(es) are being applied. If an object unit or the root of a bound unit is being patched, the patch-id is eight characters, the last two of which must be RT. If an overlay is being patched, the last two to four characters identify the hexadecimal overlay number; the first overlay is 00 for bound units created by the MOD 400 Linker, and subsequent overlays are numbered consecutively in ascending order. There may be no embedded blanks. Within the root and each overlay, patch-ids must be unique.

offn

Non-negative offset from the beginning of the block.

oldval

Current contents of specified location. If the current contents are not oldvaln, all patches associated with patchid are not applied.

patchval (object units -- local common block)

Specify a value to insert into the block. Relocatable values ae not permitted, and only one patch value can be specified for each patch address.

patchval (object units -- named common block)

Value to be inserted at an address, replacing the contents of that location. The value must be specified as

opcode field1 [,field2] [,field3]

where opcode specifies an Assembly language instruction (except for I/O or floating point instructions); fieldn specifies either a register or a hexadecimal value.

blockname

Symbolic name of the common block. The name can contain one through six characters.

offs

Offset from the symbolic name of the common block.

patchval (bound units)

Value to be inserted at an address, replacing the contents of that location. The value must be specified as a symbolic instruction.

newval

Specify the patch value to be applied. See the appropriate description of patchval, above.

SYMBOLIC PATCH

Convert and apply one or more Assembly language symbolic instructions into the form of a hexadecimal patch. You can verify the current contents of the location while patching.

FORMAT:

Without Verification:

```
SP patch-id [;base] ;/addr1 ;instruction1
    [;instruction2...instructionn]
    [/addr2; instruction/[2...n]]
```

With Verification:

```
SP patch-id [;base] ;/addr;(oldval1;instruction1
    [;oldval2;instruction2...;oldvaln;instructionn])
```

NOTES

1. One or more lines of arguments may be specified. When two or more lines of arguments are entered in an SP directive, instructions and verification values must not be split between lines. No line may begin with a semicolon (;). Individual fields, values, and addresses must not be split between lines. The entry of a Patch directive name (e.g., EP, LP) at the beginning of a line designates the end of the previous patch directive. Hexadecimal patches are not permitted.

2. You can use a carriage return instead of a semicolon as a separator.

3. You can mix verification and nonverification patches. For example:

```
SP NUMBRDRT;/135;(111;LDV $R1,1;2;CL =
$R2);/150;STB $B2,400
```

   Only the patches at locations 135 and 136 are verified.

ARGUMENTS:

patch-id

Patch-id of the patch(es) to be applied. A patch-id comprises eight to ten characters: The first six characters can be any ASCII characters except spaces. The last two to four characters must identify the root or overlay to which the patch(es) are being applied. If an object unit or the root of a bound unit is being patched, the patch-id is eight characters, the last two of which must be RT. If an overlay is being patched, the last two to four characters identify the hexadecimal overlay number. The first overlay is 00 for bound units created by the MOD 400 Linker, and subsequent overlays are numbered consecutively in ascending order. There may be no embedded blanks. Within the root and each overlay, patch-ids must be unique.

base

Optional argument allowed only for bound units. Base defines a value that is added to all locations; i.e., /addr specified in the associated DP, HP, SD, or SP directives and all IMA references. If this argument is omitted, the default value is zero. Base can be entered as a hexadecimal address of one to six characters or as a name that has been specified as an EDEF at link time and placed in the bound unit symbol table. If a symbol name is used, Patch finds the name in the symbol table and uses its address as the base value. The format for the symbol name as a base is +symname, where symname comprises 1 to 12 characters. If a hexadecimal address is used for base, the plus sign is not required.

/addr

Relative location at which the first (or only) subsequent patch value will be applied. Each address must comprise one through six right-justified hexadecimal characters, and must be preceded by the character "/" Subsequent patch values, if any, are applied to succeeding memory locations.

NOTE

Object unit locations can be obtained by examining the listing produced during assembly of the object unit.

instructionn

Value to be inserted at an address, replacing the contents of that location. The value must be specified as:

```
opcode field1 [,field2] [,field3]
```

where opcode specifies an Assembly language instruction (except for I/O or floating point instructions); fieldn specifies either a register or a hexadecimal value.

oldvaln

Specify the current contents of the specified location. If the current contents are not oldvaln, all patches associated with patchid will not be applied.

NOTE

When using verification patches, specify oldvaln in hexadecimal notation, not as an Assembly language instruction.

VERIFY/SET PATCH REVISION NUMBER

Allow a revision number to be assigned to a bound unit patch. The revision number may be assigned unconditionally, or on condition that a specified number agrees with the revision number currently in the unit. The patch revision number is stored in the unit as an external value definition with the name ZPTREV.

FORMAT:

VN (str_1, str_2)

ARGUMENTS:

str_1

Character string from one through four hexadecimal digits that is compared with the current patch revision number. If the string does not match the current revision number, no change is made, and Patch terminates.

str_2

Character string from one through four hexadecimal digits to which the patch revision number may be set. If str_1 is omitted or if str_1 matches the current revision number, the patch revision number is set to the value of str_1. If str_2 is omitted and ZPTREV does not exist in the bound or load unit, an external value definition is created with a value of str_2. If str_1 is specified, str_1 and str_2 must be enclosed by parentheses.

NOTE

This directive should not be used when patching an object file.

VDEF

VDEF

Assign a specified value to an undefined external symbol and change all locations that reference this symbol to the specified value.

FORMAT:

VDEF;symname;value [;L]

ARGUMENTS:

symname

Name of the external reference that will be assigned a value; can be from 1 to 12 characters in length.

value

Value that is assigned to all references to symname.

[;L]

List all changed references to symname on the device specified as user-out.

Default: No list.

Example:

VDEF;VALZZ;50;L

Assign the value 50 to the undefined external symbol VALZZ and changes all locations that referenced VALZZ to 50.

NOTE

Undefined external references in a bound unit can be defined by a VDEF patch directive only one time. If you make a mistake you must use HP or DP directives to change each location containing the incorrectly defined value. No listing of the VDEF patch processing is kept, therefore, the L argument should be used.

VDEF is used for changing undefined value definitions. LDEF is used for changing undefined location definitions.

Appendix A
USING THE LINE EDITOR

This appendix provides information on using the Line Editor to create and modify files. When using the Line Editor, each directive or line of information entered must be followed by a carriage return. Throughout this text, all user entries shown in examples are shaded. This distinguishes user entries from system messages and prompts.

INITIATING A LINE EDITOR SESSION

To initiate a Line Editor session, enter the ED command followed by a space and a -PT as shown:

```
RDY:
ED -PT
Edit -REL -09/09/81
E?
```

The entry ED -PT causes display of an E? prompt. The E? prompt is caused by the -PT part of the entry and informs you that the Line Editor is ready to accept directives. The E? prompt also indicates that the Line Editor is ready for use.

Creating Work Files

In addition to the -PT argument, other arguments can be included with the ED command. One argument is the -SF argument. There may be times when you are working on file contents in a temporary work area known as the current buffer. If the system fails while working in the current buffer, the contents of the current buffer will be lost. The -SF option creates two permanent work files. If the system fails, current buffer contents are not lost.

The two files created by the -SF argument are the .EDWK1 and and .EDWK2 files. The format for the -SF argument is -SF, a space, and a name for the work files. The .EDWK1 and .EDWK2 suffixes are appended to the specified file name automatically. The file name can be from one to six characters long.

The following example uses the -PT and -SF arguments. Immediately following the -SF argument is a space and filename CSRC. This entry creates two permanent files named CSRC.EDWK1 and CSRC.EDWK2. As modifications are made to the current file, the permanent work files are updated alternately. If the system fails while you are working with the Line Editor, the files named CSRC.EDWK1 and CSRC.EDWK2 will be saved.

```
ED -PT -SF CSRC
Edit REL -09/09/81
```

Once the system is working again, you can sign on, invoke the Line Editor, and read in CSRC.EDWK1 or CSRC.EDWK2 to begin modifying the file again. Reading a file into the current buffer and working on the file are described later.

Two facts about the use of the -SF argument are very important:

- The permanent files with the .EDWK1 and .EDWK2 suffixes are updated alternately. You should check both files to determine which file has the latest version of the update file. Use the X directive (described under "Buffer Status") to determine file status. These files can be read in by the Line Editor only if there is a system failure. When you create the files with the -SF argument, you cannot examine those files. Also, if there is no system failure, the two files are released when you quit the Line Editor. The files are available for examination only after there is a system failure during use of the Line Editor.

- When you sign on and invoke the Line Editor after a system failure, the -SF argument should be followed by a file name that is different from the first backup file name; otherwise, the Line Editor will overwrite the old safe files.

Example:

Assume that you have signed on and invoked the Line Editor as shown below. Later, during the building or modification of a file through the current buffer, there is a system failure. Once the system is running again, sign on and invoke the Line Editor again. This time the -SF argument is followed by the file name CTEMP. As a result of this series of terminal entries, the permanent work files named CSRC.EDWK1 and CSRC.EDWK2 are available for examination and a new set of backup files named CTEMP.EDWK1 and CTEMP.EDWK2 are created. In this way, old files can be read into the current buffer for modification and two backup files are available in case there is another system failure. When the new update session is complete, the files named CSRC.EDWK1 and CSRC.EDWK2. You should release the files named CSRC.EDWK1 and CSRC.EDWK2. If there is no system failure, the files named CTEMP.EDWK1 and CTEMP.EDWK2 are released automatically.

```
RDY:
ED -PT -SF CTEMP
Edit REL -09/09/81
E?
```

Line Editor Modes

The Line Editor works in two modes: input mode and edit mode. Input mode is used for adding lines to an existing file or for building a new file. Edit mode is used for making changes to an existing file. In edit mode, deletion of lines, substitution, and printing of lines can be done.

Quitting the Line Editor

After invoking the Line Editor and finishing your editing session, you will want to quit, or exit, the Line Editor.

To exit the Line Editor, you must be in edit mode. The Line Editor prompt indicates the current mode. If the Line Editor prompt E? is displayed, the Line Editor is in edit mode.

If the prompt displayed in response to a terminal entry is I?, the Line Editor is in input mode. If the I? prompt is displayed and you want to quit the Line Editor, you must switch to edit mode. This can be done with the !F directive, as shown:

```
E? ABUILD A FILE
I?  !F
```

The I? prompt indicates that the Line Editor is in input mode. The !F entry causes the Line Editor to return to edit mode, as indicated by the E? prompt displayed. The uses of input mode and edit mode will be covered in detail throughout this section.

Once the Line Editor is operating in edit mode, exit from the Line Editor is accomplished with the Quit directive or Q. This directive causes the Line Editor to halt and returns you to command level.

Example:

The following example shows that the Line Editor is operating in edit mode due to the !F directive. In response to the E? prompt enter Q to return to the command level. This return is shown by the RDY: prompt. At this point, you can execute any command or reenter the Line Editor with the ED -PT directive.

```
I?!F
E?Q
RDY:
```

CREATING A FILE

To create a source file using the Line Editor, invoke the Line Editor (with the -PT and -SF arguments) and invoke input mode using the I directive as shown:

```
E?I
I?
```

Once in input mode, your lines of code are entered sequentially into your current buffer. The current buffer is allocated when the Line Editor is invoked, and a pointer is established to point to this buffer as the working buffer. The current buffer is a temporary work area that is established in your working directory and memory pool (allotment). You can build a new file or read a permanent file into the current buffer for additions or modifications.

When you quit the Line Editor, the current buffer is released. Buffer management will be discussed later. Each line of data entered starts in position 1 of the line and is terminated with a carriage return. When all lines of data have been entered, you terminate input mode with the !F directive. The following example shows directives used when entering data and terminating input mode:

```
E?I
I?IDENTIFICATION DIVISION.
I?PROGRAM-ID. PAYROLL3.
I?!F
E?
```

In this example, enter an I to switch to Input mode. Enter two lines of a COBOL program, pressing the carriage return after each entry. Enter !F to switch processing back to Edit mode.

As shown later, the A directive may also be used to build a new file.

ADDRESSING TECHNIQUES

After entering all the lines of data or source code, you may need to make corrections to a line (or lines) before saving the file. However, before you learn the directives for making corrections, it is necessary to understand basic addressing techniques. You must be in edit mode to address current buffer contents.

Addressing a Single Line

To address (specify for access) a single line you need to enter only the line number (assigned by the Line Editor as each line was entered) followed by a directive. Or, in the case of addressing the current line, you need to enter the directive only. The current line is established either as the last line addressed in edit mode or as the last line entered in input mode.

For example, assume that you want to address line 3 to view its contents. You enter 3 followed by the P directive to view the contents of the line. The following example uses these directives.

```
E?3P
AUTHOR. NAME.
E?
```

Notice that the P follows the 3. The sequence that you enter directives is very important. The syntax rule for entering directives is:

```
[adr1][,adr2] command
```

Since you are working in edit mode and line number 3 is the last line addressed, it is now the current line. To print line 3 again, as shown in the following example, you enter the P directive alone.

```
E?P
AUTHOR.   NAME.
E?
```

Addressing Multiple Lines

To print the contents of several lines in sequence use two line numbers separated by a comma. The comma informs the Line Editor that the line numbers are inclusive. An example is 10,12P, as shown:

```
E?10,12P
DATA DIVISION.
WORKING-STORAGE SECTION.
01  PNAME PIC X(5)
E?
```

In this example the contents of lines 10, 11, and 12 are listed at the terminal.

To print the contents of the entire current buffer, use the special character $ as the end-of-file character. When the $ follows the comma in line number addressing, all lines from the initial value to the end of the file will be affected. For example, if the addresses 7,$ are specified with the P directive, line 7 through the end of file will be listed. The following example shows a use of the $ with the P directive.

```
E?1,$P
IDENTIFICATION DIVISION.
AUTHOR.  NAME.
OTHER LINES
STOP RUN.  LAST LINE
E?
```

The entry shown in this example tells the Line Editor to start with line 1 and list to the end of file (the last entry made Input mode) the contents of all lines. The printing of the contents is caused by the P directive.

Printing Line Numbers

Note that when the P directive is used, only the contents of the line (or lines) specified is shown. To print specified contents of the current buffer and display the line numbers assigned to each line, specify !P.

The following example shows the use of the !P directive. Notice that the only difference in the listing is that line numbers are displayed at the terminal with the buffer contents.

```
E?1,$!P
1  IDENTIFICATION DIVISION.
2  PROGRAM-ID.  PROGNAME.
3  AUTHOR.  NAME.

E?
```

Use of Period (.) for Current Line

If you use multiple line addressing and want to start with the current line, use the period (.) character in the first address.

For example, your current line is 10 and you want to print through line 20. If the !P directive were used instead of the P directive in the following example, the line numbers 10 through 20 would be listed with the file contents.

```
E?.,20P
OBJECT-COMPUTER.  HIS-SERIES-60 LEVEL 6.
DATA DIVISION.
OTHER LINES
MOVE "DATUM" TO QNAME.
E?
```

Character String Addressing

The Line Editor can also address a line (or lines) by contents.

The contents, called a character string, are expressed using two delimiters. Delimiters are slashes (/) that precede and follow the designated search string. For example, the slashes in the expression /ABC/ delimit the string ABC. The two slashes are the delimiters and the characters between are the string that is searched for. The search begins with the line following the current line, continues to the end of the file, and then starts with line number one and searches to the current line. When the Line Editor finds a line containing the specified string it executes any directive specified with the search string and positions the current line pointer to that line.

In the following example, enter PROGRAM-ID as a character string with delimiters, with the P directive. As a result, the line containing PROGRAM-ID is printed, revealing the that the internal name of the program is PROG1. If the !P directive were used, the line number would have been printed with the line contents.

```
E?/PROGRAM-ID/P
PROGRAM-ID.  PROG1
E?
```

The search for the contents ends when the first line that contains the specified character anywhere in that line is found. The current line pointer is then positioned at that line. If no match is made with the search string, the current line pointer is positioned at the line that was the current line before the directive was executed. Also, if there is no character string match, the message SEARCH FAILED is displayed at the terminal.

SELECTIVE SPECIFICATION OF CHARACTER STRINGS

To be more selective when specifying search characters, make the string within the delimiters more specific. For example, you may want to print the line number and the contents of a line that contains the character X. Suppose there are two lines that contain X. One contains PIC X(5) and another PIC X(10). To specify the first, you designate the search string so that the Line Editor can determine that you want the line that contains PIC X(5). You could specify PIC X(5) as the character string. Or, you could specify only that portion of the string that distinguishes it from all other strings in the file. The string X(5 would be sufficient. If you enter /X(5/!P at the terminal, the only line that qualifies for printing is PIC X(5).

In the example only DATA DIVISION is printed because others, like PROCEDURE DIVISION, do not fit the search characteristics.

```
E?/A DIVISION/!P
11 DATA DIVISION.
E?
```

SPECIFYING INITIAL CHARACTER STRING

The Line Editor can also be told to search for a line beginning with a character string. This is done by preceding the character string with a circumflex (^). For example, assume that the specified character string TAG must occur as the first characters on a line. The following example shows specification of this search string:

```
E?/^TAG/!P
2 TAG EQU 10
E?
```

In this example the use of the circumflex (^) specifies the character string TAG must occur as the first three characters of the line searched for. The !P directive is used to print line contents with the line number.

SPECIFYING A CHARACTER STRING ENDING A LINE

A string occurring as the last character(s) on a line can be specified as a search string. You specify this with the $ character. To specify that you want to find the line that ends with the characters FILE, use the search string: /FILE$/.

The following example shows the use of the $ character to specify the FILE ending character string with the !P directive. The result is the terminal listing of FD INFILE with the line number 32.

```
E?/FILE$/!P
32 FD INFILE
E?
```

Remember that the results of the directive shown in the previous example are dependent upon the location of the current line. If the current line was after line 32 in the buffer (at the time of execution of the directive in the previous example), the terminal display might contain a different line number and contents, such as FD OUTFILE. Therefore, it is important to know the location of the current line when you initiate a character string search.

As shown in the addressing methods, certain characters represent special or control characters. The character ^ is used to specify a line beginning with a specified string. The $ is used to specify a string ending a line when it is used in a search for line contents. The $ is also used to specify the end of file when it is entered as the second address to designate multiple lines as in this example. The period (.), when used in line number specification, represents the current line. It has another use -- single character substitutions.

SPECIFYING A SINGLE CHARACTER SUBSTITUTION IN SEARCH STRINGS

When a period (.) is used in a character string search, it takes on a special meaning. When a period is used in a search string such as /A.C/ it means that any character between the characters A and C can be substituted. This means that ABC fits the search as does A1C or AZC.

In the following example, EDB is the first search match, so line 10 is printed.

```
E?/E.B/!P
10 LABEL EDB $B5, X'FFFF'
E?
```

USE OF ESCAPE CHARACTERS

It is possible that any of the special addressing characters, ($, !, or ^) are part of the data to be searched for. The preceding example contains such an example in the display of line 10.

To distinguish between a special character as data and a special character used to affect a search string, escape characters are used. For example, to specify that the character has its data meaning and not search meaning !C escape characters are used.

The escape characters remove the search meaning from the next character. For example, the search string /A1!C$/ contains !C, which removes the search meaning from the $. the Line Editor searches for three characters designated as A1$ rather than A1 at the end of a line. The following example shows another use of the escape characters.

```
E?/ION!C.$/1P
1  IDENTIFICATION DIVISION.
E?
```

In this example, !C precedes the period so that IDENTIFICATION DIVISION, is found. Also, the $ symbol insures that ION. will occur in the last four characters of the line that is found and listed.

SAVING FILE CONTENTS

All of the work done building a file in the current buffer is destroyed when you quit the Line Editor. The current buffer is a temporary working file. The contents of the current buffer must be stored in a permanent file if the buffer contents are to be saved after you quit the Line Editor.

If you are building a new file in the current buffer, you need to create a new permanent file to accept the contents of the current buffer. A new file can be created using the CR command at the command level before you call in the Line Editor. The current buffer contents can be copied into the previously created file. If the file you want to create is to be a sequential file, you can create that file at the same time that you copy current buffer contents by using the W directive.

As shown in the following example, a COBOL source program has been built in the current buffer. The file (source program) is saved to a permanent file called COBOLP.C. The file named COBOLP.C did not exist before the W directive was entered. When the W directive is entered, the file named COBOLP.C is created as a sequential file immediately subordinate to the working directory (however, a full or relative pathname could have been used). After the W directive creates the file, the same directive copies the contents of the current buffer to the designated permanent file, which is COBOLP.C in this case.

```
E?W  COBOLP.C
E?
```

Two other situations exist in which you may want to save the current buffer contents to a permanent file. One situation was already mentioned. The file may exist before the W directive is used because the file was created using the CR command. Another situation is the one in which you want to replace the contents of a file with the contents of the current buffer, as in the case of making modifications to a program. In either case, the W directive stores the contents of the current buffer in the designated file.

In the case of the file that was created using the CR command, the contents of the current buffer are copied into the existing permanent file. In the case of the permanent file that exists and contains a previously stored program or data, the old file contents are replaced by the contents of the current buffer. The format for copying the current file contents to the existing permanent file is the same as the one shown in the preceding example. Just be sure to designate the correct pathname for the existing permanent file that is to contain the current buffer file.

After preserving the contents of the new file you can quit the Line Editor, return to command level, and perhaps compile the program, if this is a source program. Remember, if you did not write the buffer file to a permanent file prior to quitting the Line Editor, all data from the editing session is lost.

READING FILE CONTENTS

This subsection describes the procedures used to modify the contents of a new file, or to modify a source or data file already in existence.

Invoke the Line Editor and be sure you are working in Edit mode.

If you want to modify the contents of an existing file, you must copy that file into the current buffer before you can use the file modification techniques.

If the file to be modified exists as a permanent file, you must copy the file contents into the Line Editor's current buffer. This is done with the Read (R) directive followed by the pathname of the file to be altered.

Note that in the following example, a full pathname is specified. You may use any of the pathname variations allowed.

```
E?R^VOLA>DIR1>COBOL.C
E?
```

The R directive is a read and append directive. That is, the contents of the file read are appended to the contents of the current buffer. If you want the contents of the file being read as the only data in the current buffer, first delete the contents of the current buffer and then perform the read. If the Line Editor was just invoked, the sequence is:

```
ED/EDPT/SP
E? R pathname
```

DELETING LINES IN CURRENT BUFFER

The D directive is used to delete lines from the current buffer. To delete lines, specify the line (or lines) to be deleted followed by the Delete directive (D). To delete one line, use the line number followed by D, as shown:

```
E?5D
E?
```

To delete the contents of the current line specify just the directive D, as shown:

```
E?D
E?
```

Deleting Multiple Lines

To delete multiple lines in sequence, specify the line numbers separated by a comma and followed by the D directive.

In the following example, 5,10D causes lines 5 through 10 to be deleted from the current buffer:

```
E?5,10D
E?
```

Deleting All Lines in Current Buffer

To delete all lines in the current buffer, use the character $ as the second address and line number 1 as the first address.

The following example shows the directive sequence (1,$D). This directive sequence is used to delete or clear the contents of the current buffer for the use of the R directive explained under "Reading Contents of Existing File". There are times when you will not want to clear the current buffer before using the R directive; these instances will be covered later in this appendix. Usually you will want a clean current buffer before you read a permanent file into it.

```
E?1$D
E?
```

The following example shows a typical clear-and-read sequence. The 1,$D directive clears the current buffer. Then the file named COBTEST, immediately subordinate to the working directory, is read into the current buffer.

```
E?1,$D
E?R COBTEST
E?
```

Avoiding Post-Deletion Problems

In the preceding examples, the use of line numbers to specify lines to be deleted was shown. After the line is deleted any remaining lines following the deleted line are automatically renumbered. This can cause problems in the deletion or modification of remaining lines.

For example, if you delete line number 10, what was line 11 is now line number 10 and line 12 is now line 11 and so on through the end of the file. If your next directive is to affect old line 15, you must remember that it is now line 14.

Example 2 shows the results of deleting lines 2 and 3 from the current buffer file shown in Example 1. Note that line 4 (containing the PROGRAM-ID) becomes line 2. All subsequent line numbers are affected in the same way. Notice that line numbers before the deleted line(s) are not affected; the line number for the IDENTIFICATION DIVISION statement, in this case does not change.

Example 1:

```
E?1,$!P
1  IDENTIFICATION DIVISION.
2  ****NOTE:  MAKE SURE THAT YOU CONFIRM INSTALLATION,*****
3  ****AND SECURITY BEFORE COMPLETING PROGRAM***********
4  PROGRAM-ID. COURSE.
5  AUTHOR. AMY SMITH.
6  INSTALLATION. LOMPOC, CA.
7  DATE WRITTEN, 05181
8  SECURITY. NONE.
9  ENVIRONMENT DIVISION.
10 CONFIGURATION SECTION.
11 SOURCE-COMPUTER. HIS-SERIES-60 LEVEL-6.
12 OBJECT-COMPUTER. HIS-SERIES-60 LEVEL-6.
E?
```

Example 2:

```
E?2,3D
E?1,$!P
1  IDENTIFICATION DIVISION.
2  PROGRAM-ID. COURSE.
3  AUTHOR. AMY SMITH.
4  INSTALLATION. LOMPOC, CA.
5  DATE-WRITTEN. 05181
6  SECURITY. NONE.
7  ENVIRONMENT DIVISION.
8  CONFIGURATION SECTION.
9  SOURCE-COMPUTER. HIS-SERIES-60 LEVEL-6.
10 OBJECT-COMPUTER. HIS-SERIES-60 LEVEL-6.
E?
```

Adding and Deleting Lines

To avoid confusion when adding or deleting lines, use one of the following methods. The first is to delete or add lines starting with the line nearest the end of file. For example, you must delete lines 10, 15, and 20 in a buffer containing 25 lines. If you start with line 20 first, lines 10 and 15 are unaffected because only the lines following 20 are renumbered. Then line 15 should be deleted so that line 10 will not be affected by the change. Finally, line 10 should be deleted.

The second method is to delete by contents of a line. For example, you know that only the line containing STOP RUN is to be deleted. So, you specify a search for only the line by its contents with the D directive.

The following example shows how a line can be deleted, regardless of its line number. Naturally, any number of lines may be deleted using this method, but a separate D directive must be used for each line to be deleted.

```
E?/STOP RUN/D
E?
```

Due to automatic resequencing of line numbers after line additions or deletions, it is helpful to list the file to determine the new line numbers before you attempt any subsequent modifications by line number.

CHANGING LINE CONTENTS

To change the contents of a line, use the Change (C) directive or the Substitute (S) directive. The C directive allows you to change the entire contents of a line. This directive is considered an input mode directive. After executing the C directive the Line Editor will be in input mode. This requires that you execute the !F directive to exit input mode.

In the following example, the whole new line had to be entered. You cannot use the C directive to change just portions of a line. Also, the use of the !F directive at the end of the entry to causes the Line Editor to return to edit mode.

```
E?10!P
10 MOM'S APPLE PIE.
E?10CSOURCE-COMPUTER. LEVEL-6. !F
E?
```

Changing Character Strings Within a Line

To change portions of a line, use the Substitute (S) directive. The S directive allows you to substitute one character string for another within a line. The S directive is an edit mode directive.

To make a substitution enter a line number, an S, a delimiter, the character string to search for, a delimiter, the substitute string, and a final delimiter.

To substitute a new character string for a specified character string on line 5, you could use this method:

```
E?5!P
5 PROGRAMMER-ID. ROGER.
E?5S/ROGER/GARY/
E?
```

In this method all occurrences in the line of the character string ROGER would be replaced by the character string GARY. This means that if ROGER occurs three times on line 5, three substitutions will be performed.

To specify that only one occurrence of a character string is to be altered, you must be more specific in the search character string. For example, the word ROGER occurs only once in the line shown in the preceding example. If ROGER occurs twice in the affected line and you want to change only the first occurrence to GARY, the search character string must be made more specific. This is described under "Selective Specification of Character Strings".

Changing All Occurrences of a String

The Line Editor can be told to search one line, multiple lines, or all lines for a specified character string and to substitute all occurrences of that string for the new string.

In the following example, all occurrences of IONFILE in the current buffer are changed to INFILE:

```
E?1,$S/IONFILE/INFILE/
E?
```

Techniques for substituting, printing, and deleting all occurrences of a specified string will be discussed below. These techniques for affecting all occurrences of a string can be helpful in program modification.

For example, a program that already exists might be almost perfect to meet a new data processing need, but the record or field names are wrong. With a single directive, all occurrences of the unwanted record name or identifier can be changed to the necessary entry.

Substituting Initial and Concluding Strings

As explained previously, the circumflex (^) and dollar sign ($) may be used in an address search. Those same rules apply here. To specify a character string that begins a line, use the ^ character.

The following example shows a directive used to replace the character string PROGRAM with the occurrence of PRGRAM at the beginning of line 3. PROGRAM replaces PRGRAM as the initial character string. To affect a string that ends the line, use the $ character.

```
E?3S/^PRGRAM/PROGRAM/
E?
```

The next example shows that the concluding character string OUTFILD at the end of line 7 will be replaced by the string OUTFILE., through the S directive. Of course, it is possible to replace all occurrences of an initial or concluding string with a specified character string.

```
E?7S/OUTFILD$/OUTFILE./
E?
```

In the following example, starting with line 1 through the end of the file, all occurrences of the # at the beginning of a line will be replaced by 7 spaces.

```
E?1,$S/^#/       /
E?
```

The technique shown in this example is commonly used in editing source programs that by convention have coding statements begin in specific columns of a line. Column 7 in FORTRAN and columns 8 and 12 in COBOL are examples of columns that are specified for coding purposes. For example, all statements that should start in column 8 might begin with a # character. As a result of the use of the directive in the example, all of those statements now begin in column 8 because of the preceding 7 spaces.

Deleting Character Strings

If you have entered a character string and later find that you do not need it, you can delete that string using the substitute directive. To delete a character string specify the substitu- tion field as blank.

In the following example, the substitution field is specified as //. This informs the Line Editor that there is no replacement string. Therefore, the string specified in the search field (DATA) in line 20 is to be deleted.

```
E?20S/DATA//
E?
```

Appending a New String to an Existing String

Expanding a character string on a line (or lines) can be done with the ampersand (&). The position of the & will dictate where the new character string will occur in the new line. For example, the character string L6 is on line 23, and that character string should be LEVEL-6. The following example shows how the use of &, by its position, changes the line in the correct location.

```
E?23!P
23 OBJECT-COMPUTER. HIS-SERIES-60 L6.
E?23S/L/&EVEL-/
E?23!P
23 OBJECT-COMPUTER. HIS-SERIES-60 LEVEL-6.
E?
```

The substitution causes the L in L6 to be followed immediately by EVEL-. The result is LEVEL-6. Notice that there are no spaces between delimiters and strings or between special characters (such as & or ^) and the strings. It is important to define the search strings and the modification strings accurately, or the result will be incorrect.

The escape characters (!C) cause the editing character to have no meaning to the Line Editor. For example, if the character & is to be used as a non-editing character in the second field of a Line Editor directive, the & should be preceded by the !C escape characters. Such an example would occur if you want to change the string $ION to &ION. The line entry would be S/!C$ION/!C&ION/.

Adding Lines to the Current Buffer

To add a new line to your current buffer, use either the Append (A) directive or the Insert (I) directive.

INSERTING LINES

Using the I directive you can insert a line (or lines) before the line specified in the address. To insert a line of code preceding line 15, enter 15I.

The result of the directives shown in the following example is to have a new line 15 and to have all subsequent line numbers incremented by 1. Therefore, the old line 15 becomes line 16, and so forth.

```
E?15I WORKING-STORAGE SECTION.↵!F
E?
```

Notice in this example that the E? prompt is displayed after you press the carriage return. In this case, because the !F directive follows the entry of the new line, the E? prompt signals that the Line Editor is ready to work in edit mode. If the !F directive were not entered, the I? prompt would be a request for another line of input. Remember, the I directive causes the Line Editor to work in input mode.

Two lines are inserted in the file in the current buffer in the next example. The first line added becomes line 15 because of the 15I directive. The I? prompt requests another line of input. The RBN-STATUS line is entered. It becomes line 16. The I? prompt requests another line of input. The !F response causes a switch to edit mode.

```
E?15I WORKING-STORAGE SECTION.↵
I? RBN-STATUS PIC X.↵
I?!F
E?
```

Remember, because lines 15 and 16 are inserted as new lines, the old line 15 becomes line 17 and all subsequent line numbers are incremented by 2.

APPENDING LINES

To append a new line to any point in a file, the A (Append) directive is used. Append will add a new line following the number specified.

The example below shows the use of the Append directive. This directive creates a new line 16 and all subsequent lines are renumbered. The new line follows line 15. The A directive allows you to add more than one line. If the !F directive was not used in this example, the Line Editor would be expecting you to enter another new line, that would have line number 17.

```
E?15A #WORKING-STORAGE SECTION.↵!F
E?
```

Note that when using the A directive, the !F directive is needed to exit input mode.

GLOBAL DIRECTIVES

Searching for lines that need to be modified or listed can be simplified by using the Global directive. The Global directive (G) will work only with the following directives: P, !P, D, and =.

With the Global directive only the lines that contain the specified character string will have the directive P, !P, D, or = applied to it.

Global Delete

Global Delete (GD) is used to remove a character string throughout a file. To issue a Global Delete, type the directive G followed by D followed by the character string to search for in delimiters. The following example shows a Global Delete directive.

```
E?GD/DATA/
E?
```

In this example, no line numbers are specified. Line numbers can be specified, but when they are absent, the directive defaults to start at line 1 and works to the end of the file. If line numbers are specified, only those lines specified are affected by the delete. For example, the entry 1,15GD/DATA/ removes all lines containing the string DATA from line 1 through 15.

Global Print

To print only lines that contain certain characters, specify the GP or G!P directive.

The GP directive prints just the contents of the lines that contain the character string and the G!P directive prints the contents and the line numbers associated with those contents.

The following example shows the terminal entry requesting the line number for each of the four COBOL divisions for a program that is in the current buffer. The G directive with the !P directive and the SION search directive is entered. The SION has been used as the search string because it is common to all four COBOL program divisions.

```
E?G!P/SION/
 3 IDENTIFICATION DIVISION.
 7 ENVIRONMENT DIVISION.
10 DATA DIVISION.
21 PROCEDURE DIVISION.
E?
```

The next example shows the G and = directives with the same search and current buffer file used in the previous example. Note that only line numbers are printed.

```
E?G=/SION/
 3
 7
10
21
E?
```

CURRENT AND AUXILIARY BUFFERS

Thus far, changes have been made to a permanent file through the use of the current buffer. In addition to the current buffer, there are five auxiliary buffers available to assist in manipulation of file contents.

For example, to repeat lines of coding in a program, move lines of coding from one location in a file to another location, or build a new file from coding lines of other files, auxiliary buffers are used.

Repeating Lines in a File

There are a number of file-building and file-modification functions that can be carried out through buffer management. The current buffer and up to five auxiliary buffers assist in creating or modifying file contents. The auxiliary buffers can have alphabetic or numeric names, the examples shown below will use single number names.

The next two examples illustrate buffer manipulation used to repeat lines in a file. The directives used in the manipulation of current and auxiliary buffers are the K and !B directives. K and !B are used for copying lines to a specified auxiliary buffer and for fetching lines from a specified auxiliary buffer.

The following example shows how the current buffer and an auxiliary buffer can be used to repeat lines of a file at the end of the file. The program stored in the file named COBPRG.C is read into the current buffer, and lines 50 through 63 are copied to an auxiliary buffer named 1 through the K1 directive. Note that if any lines exist in the auxiliary buffer at the time that the K directive is used to copy new lines to that buffer, the old lines in the buffer will be deleted before the new lines are copied to that buffer.

```
RDY:
ED -PT
E?R COBPRG.C
E?50,63K1
E?$A!B1!F
E?W COBPRG.C
E?1,$D
E?
```

After the copy is completed, the lines in buffer 1 are appended to the end of the file in the current buffer. The $A directive causes lines to be appended to the end of the file. The !B1 directive causes the lines in buffer 1 to be fetched for appending. The !F directive is used to change to edit mode (after the A directive is used). Then the W directive causes the current buffer contents to be copied to the file named COBPRG.C to replace the old contents in that file. Finally, the contents of the current buffer are deleted with the D directive to allow a new file to be read into the current buffer. If no more editing is to be done, the current buffer contents do not have to be deleted. The Q directive would be sufficient to quit the Line Editor.

The next example demonstrates two editing concepts important to buffer manipulation -- how to repeat lines within a file and how to use a different auxiliary buffer to save the contents in an existing auxiliary buffer.

```
E?R CFILE.C
E?9,17K2
E?45A!B2!F
E?W CFILE.C
```

A file named CFILE.C is read into the current buffer, and lines 9 through 17 are copied to an auxiliary buffer named 2. If there are lines in another auxiliary buffer, such as buffer 1, the copying of lines to buffer 2 will allow the lines to remain in buffer 1. The lines in buffer 2 are then appended to line 45 in the current buffer (through the 45A directive). The !B2 directive causes those lines to be fetched from buffer 2. Again the !F directive causes a return to edit mode. Finally, the changes must be made to the file named CFILE.C through the W directive.

Moving Lines in a File

The next example shows moving lines within a file. The first situation involves moving lines to a location near the end of the file.

The following example shows how lines 8 through 12 in a file can be moved to follow line 27. A file named FILEN is read into the current buffer. Lines 8 through 12 are copied to buffer 1. Then the lines are appended to line 27 in the file. Because the lines in the original location remain, they must be removed through the D directive. Then the current file contents are written back to FILEN. The current buffer contents are deleted to allow for additional editing of a different file.

```
RDY:
ED =PT
E?R FILEN
E?8,12K
E?27A!B1!
E?8,12D
E?W FILEN
E?1,$D
E?
```

When lines are moved to a location closer to the beginning of the file than their current location, the deletion of the old lines after the move presents a different problem.

The next example shows the steps taken to move lines 21 through 37 of a file to the beginning of the file. The file named RNGT is read into the current buffer, lines 21 through 37 are copied to buffer 1, and lines 21 through 37 in the current buffer are deleted.

The I directive is used to insert the contents of buffer 1 before line 1 of the file. The file is listed. The current buffer is written to RNGT.

The Line Editor keeps track of the actual line numbers of the lines in a buffer. You can add, delete, or rearrange lines and the Line Editor automatically renumbers to keep the count correct.

```
E?R RNGT
E?21,37K
E?21,37D
E?1I!B1!
E?L,$!
```

file is listed

```
E?W RNGT
E?
```

Using Existing Files

Sometimes a new program must be created and it contains logic elements that are similar to elements in one or more existing programs. In this case, you can call an existing program into the current buffer, delete the unnecessary coding from the current buffer, and build a new file around the useful coding. At other times, you may want to call portions of different programs into the current buffer and use those portions as a basis for building a new program. These Editor and buffer techniques can save program development time.

The following example shows how two programs can be used for developing a single new program. In this example only two buffers are used. It should be noted that all five can be used. Additionally, if necessary, the entire contents of the current buffer can be copied to an auxiliary buffer as different portions of the program are developed. This example shows the steps that can be used to develop the a program from two existing programs.

```
E?R ABPRG.C      Read in ABPRG.C.
E?1,8K1          Copy lines 1 through 8 to buffer 1.
E?59,67K2        Copy lines 59 through 67 to buffer 2.
E?1,$D           Delete contents of current buffer.
E?R GGPRG22.     C Read in GGPRG22.C.
E?29,32D         Delete lines 29 through 32.
E?26,27D         Delete lines 26 and 27.
E?1,12D          Delete lines 1 through 12.
E?1,$!P          List file with line numbers.
```

listing is produced

```
E?5A!B2!F        Append contents of buffer 2 after line 5.
E?1I!B1!F        Insert contents of buffer 1 before line 1.
E?1,$!P          List file with line numbers.
```

listing is produced

```
E?W NEWP.C       Write to a permanent file.
E?
```

The lines are deleted from the end of the file (29,32D) towards the beginning of the file (1,12D). The next step shown in this example is the listing of the current buffer at the terminal. This listing indicates line numbers for the insertion of lines from the two auxiliary buffers. The lines in buffer 2 are appended to line number 5 of the current buffer (5A!B2!F). If the lines from buffer 1 were inserted at the beginning of the file, the line numbers in the current buffer would change. Then you would have to list the contents of the current buffer again to see where the lines from buffer 2 should be appended. After the lines from buffer 2 are appended to line 5, the lines from buffer 1 can be inserted before line 1 of the current buffer.

To make sure that the current buffer contains all the lines in the proper sequence, list the file contents again. Then you can write the current buffer contents to a permanent file or continue working on the current buffer contents to create a program. The contents of the current buffer are saved to a permanent file named NEWP.C.

Buffer Status

The X directive is used to determine buffer size or current buffer status. Buffer status can be checked at any time during the editing process. The following example shows the use of the X directive:

```
E?X
15->(0)^VOL10>EDMOD> COBPRG.C
E?
```

The example indicates that there are 15 lines in buffer zero. The second field of the display information is an arrow pointing to the buffer that is the current buffer. In this example, only one buffer is shown; the following example shows two buffers.

```
E?X
15->(0)^VOL10>EDMOD>COBPRG.C
8   (1)
E?
```

This example indicates that there are two buffers, one with 15 lines, one with 8 lines. The buffer pointer is pointing at buffer 0, the current buffer.

If you were to add, delete, change, or substitute lines in the current buffer, MOD would appear, as shown:

```
E?X
15  MOD->(0)^VOL10>MOD>COBPRG.C
8         (1)
E?
```

The next field is the buffer name in parenthesis. The name for the first is 0 (the default current buffer), which was created when the Editor was invoked. The second is 1, which was created when data was moved into it.

The final field in the buffer status is the absolute pathname used in the last read or write operation using that buffer. Notice in the preceding examples, that the file with the absolute pathname ^VOL10>EDMOD>COBPRG.C has been read into the current buffer.

Saving Modified Buffer Contents

The following example shows part of a terminal session where a file has been modified using the Line Editor. You attempt to quit the Line Editor with the Q directive. Because you have not saved the current buffer contents with the W directive, the system displays the QUIT DEFERRED message. This message indicates that modifications have been made to the current buffer but the current buffer contents have not been saved.

```
E?Q
MODIFIED BUFFERS EXIST, QUIT DEFERRED
E?W TESTFL
E?Q
RDY:
```

In this case, you respond to the message with the entry W TESTFL and to the E? prompt with the Q directive to return to the command level of processing. If you responded to the QUIT DEFERRED message with the Q directive, the system would have accepted the directive, the current buffer contents would have been destroyed, and processing returned to command level.

USING SYSTEM COMMANDS IN THE EDITOR

The escape directive (E) allows you to use system commands while you are working with the Line Editor.

The Line Editor must be in edit mode. Then an E followed by any system command causes the Line Editor to pass that command to the Command Processor. After executing that command, the system returns control to the Line Editor.

Writing to Line Printer

To write the contents of the current buffer to the line printer, first reserve the line printer as the user output file. This is done with the FO command explained earlier. The following example shows the commands used to reserve a line printer.

```
E?EFO:LPT00
E?
```

After executing the command shown in the example, the Line Editor sends all output that would go to the screen (except for the ready display and errors) to the line printer. To get hard copy of the current buffer, type the directive line that prints the entire contents of the buffer.

The next example shows that once user output is directed to a printer any variation of the P directive will cause printing to take place at the printer. In this case, the entire file (1,$) is printed with line numbers (!P).

```
E?EFO !LPT00
E?1,$!P
E?
```

After printing the contents of the current buffer on the line printer, to change output back to the terminal (or default device) enter the FO command with no pathname.

The following example shows the Escape (E) directive is necessary to execute the FO command while the Line Editor is invoked.

```
E?1,$!P
E?EFO
E?
```

The execution of commands from the Line Editor is similar to the execution of commands at command level. Two of the differences include:

- The E? prompt instead of the RDY: prompt to indicate system readiness to execute a command or directive.
- The need for the E directive prefix when commands are executed from the Line Editor.

Date and Time

When the lines of the current buffer are listed on the line printer they are printed as is. There is no date or time displayed with the printing. Sometimes it is helpful or important to know when a listing of a file was made, particularly when various updates of files must be compared.

To display a date and time heading with your listing on the line printer, use the system command TIME after directing output to the line printer.

TIME is a system command. Therefore, the Escape (E) directive must be entered too.

The command ETIME will cause the system date and time to be displayed on the output device. This display becomes a header for the file listing which follows. For example, as a result of the following entries, the entire contents of the current buffer are listed at printer LPT00 with a date and time header.

```
E?EFO !LPT00
E?ETIME
E?1,$!P
E?
```

Important Considerations

When using the Escape (E) directive you can execute any system command. However, if the E prefix is omitted, certain problems can occur. For example, if the E prefix is not used, the Line Editor will not pass the entry to the Command Processor. As a result, the Line Editor will try to execute the entry as a Line Editor directive. For this reason, accidentally entering a command to the Line Editor without the E prefix can cause problems. Accidentally entering a command that begins with any of the following characters can be particularly problematical: W, LW, D, I, C, and A.

For example, an entry beginning with a C will change line contents. An I or an A will cause additions to a current buffer file. A W will copy the current buffer contents to a permanent file. Using the E directive prefix is important.

For instance, if LWD is entered without the preceding E, the Line Editor will cause a line feed and write the contents of the current buffer to a file called D under the working directory. The correct way to enter the LWD command is shown in the example:

```
E?ELWD
^VOL3>DIR23
E?
```

Appendix B
USING COBOL

This appendix describes procedures for using COBOL. The following information is provided:

- Explanation of the compile, link, and execute procedures for COBOL programming, including a sample program illustrating these steps
- Programming tips for communications via COBOL, including a sample program.

COBOL COMPILE, LINK, AND EXECUTE PROCEDURES

To compile a COBOL program, invoke the Advanced COBOL (COBOLA) compiler and the Linker. Input to the COBOL compiler consists of a source program written in COBOL and optional control information.

Output is a:

- COBOL object (.O) unit
- COBOL listing and diagnostics.

Input to the Linker is the relocatable object unit.

Output is a:

- Bound unit
- Link map.

Figure B-1 illustrates the compile and link operation, producing an executable module.

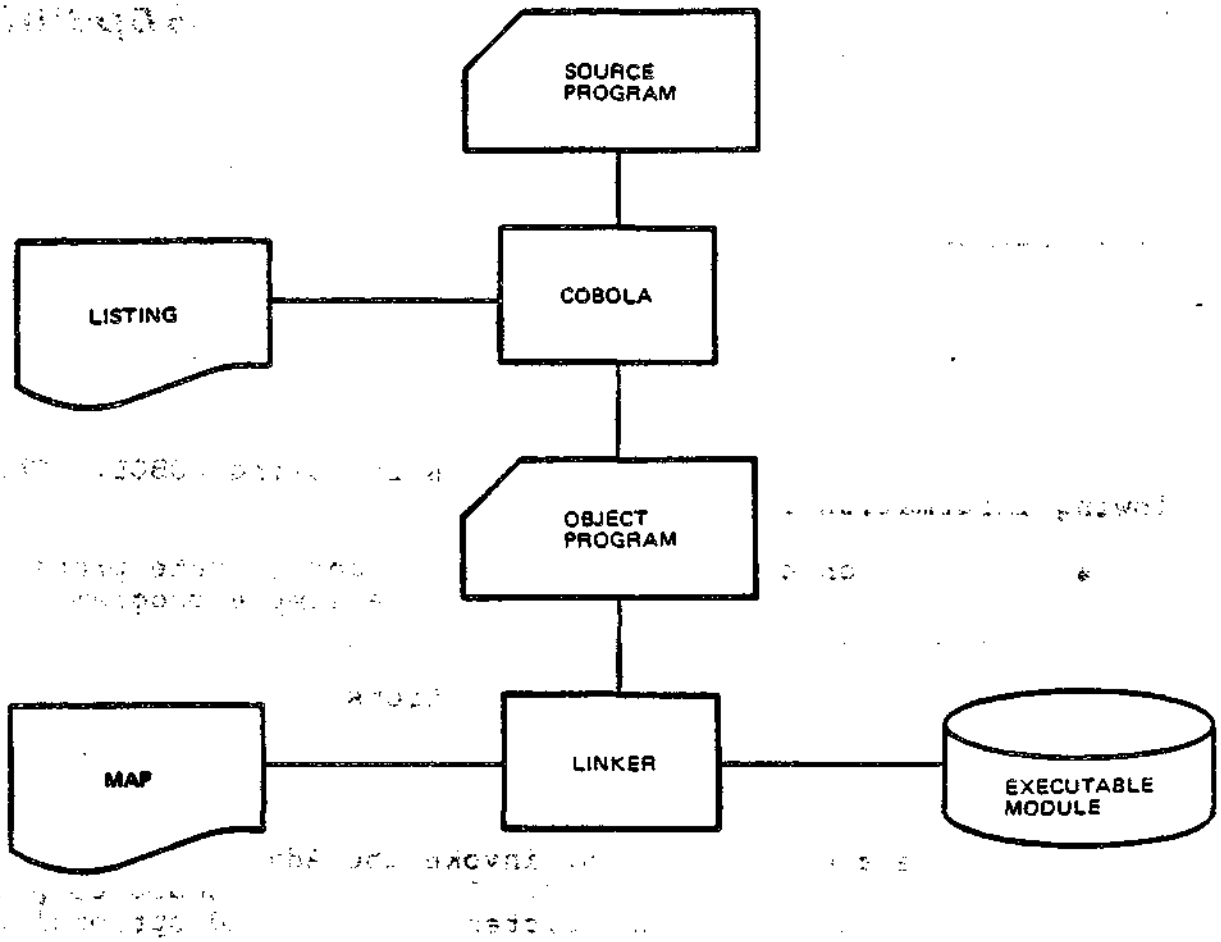


Figure B-1. Compiling and Linking a COBOL Program

Invoking the COBOL Compiler

The command used to invoke the COBOL compiler is:

```
COBOLA path [ctl_arg...]
```

where:

path

The pathname of the source file that is to be compiled by the Advanced COBOL compiler. If path does not have a suffix of .C, then one is assumed. However, the suffix .C must be the last component of the name of the source file.

ctl_arg

None or any number of control arguments. (See the Advanced COBOL Reference manual.)

For example, the source file might be PROG1.C shown in Figure B-2.

To compile PROG1.C, enter:

```
COBOLA PROG1
```

This causes compilation of the source file PROG1.C that, in this case, is in the current working directory.

The terminal dialog is:

```
COBOLA PROG1.                              Invoke the compiler
COBOLA 2.0 02/28/0057
NO FATAL ERRORS, 1 WARNING IN PROG1.       A warning is issued
    11              DATA DIVISION.
                    1
  ** 1 2-3   A PERIOD IS REQUIRED PRIOR TO THIS WORD
RDY:
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROG1.C
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. LEVEL-6.
OBJECT-COMPUTER. LEVEL-6.
SPECIAL-NAMES.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
I-O-CONTROL.
DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
01 WORD PICTURE X(72).
01 GOOD PICTURE X(72).
77 BLANKER PICTURE X(72) VALUE SPACES.
01 TEST1 PICTURE X(3).
01 TEST2 PICTURE X(4).
PROCEDURE DIVISION.
READING-ROUTINE.
    DISPLAY "WHAT WORD?".
    MOVE BLANKER TO WORD.
    ACCEPT WORD.
    MOVE WORD TO TEST2.
    IF TEST2 EQUAL TO "DONE" GO TO END-CARD; ELSE NEXT
        SENTENCE.
    DISPLAY "YOUR WORD WAS " WORD.
    DISPLAY "CORRECT?".
    ACCEPT GOOD.
    MOVE GOOD TO TEST1.
    IF TEST1 EQUAL TO "YES" GO TO READING-ROUTINE;
        ELSE NEXT SENTENCE.
    MOVE GOOD TO TEST2.
    IF TEST2 EQUAL TO "DONE" GO TO END-CARD;
        ELSE NEXT SENTENCE.
    DISPLAY "TOO BAD".
    GO TO READING-ROUTINE.
END-CARD.
    DISPLAY "THAT'S ALL FOLKS!".
    STOP RUN.
```

Figure B-2. COBOL Source Program PROG1.C

COBOL List File

Unless you specify otherwise, the Advanced COBOL compiler produces a list file for the program being compiled. Specify NOLIST on the COBOLA command line to delete the list file. The list file contains the source listing, the cross-reference listing, and the object map or object listing.

LIST HEADER

The list file begins with a header containing the program name, date and time of compilation, the compiler version used, and arguments, if specified, in the format shown in Figure B-3.

SOURCE LISTING

The source listing is a line-numbered, printable ASCII listing of the source program. The entire source line image for each line is presented, always in the fixed reference format. The line number shown with each line, referred to as the external line number, represents the relative position of that line in the source file (i.e., the relative record number). This is a two-part number if it represents a line from a COPY file. The first portion is the relative number of the COPY statement in the program (the first COPY statement in the program is 1, the next 2, etc.). The second portion, separated by a hyphen from the first, is the relative line number within the COPY file (e.g., 13-126). This is the line number permanently associated with the statement(s) on that line for the reporting of run-time errors.

SAMPLE LISTING

Figure B-3 shows the listing produced by compilation of PROG1.C.

```
GCOS MOD400-L2.1-09/24/1727    COBOLA  2.0
\cROG1     81/05/05    0910    PAGE 0001

COMPILATION LISTING OF PROG1.C                          List Header
COMPILED BY:  COBOLA VERSION 2.0  02/28/0057
COMPILED ON:  81/05/05 0910
    OPTIONS:  NONE

    1                IDENTIFICATION DIVISON.
    2                PROGRAM-ID.  PROG1.
    3                ENVIRONMENT DIVISION.
    4                CONFIGURATION SECTION.
    5                SOURCE-COMPUTER.  LEVEL-6.
    6                OBJECT-COMPUTER.  LEVEL-6.
    7                SPECIAL-NAMES.
    8                INPUT-OUTPUT SECTION.
    9                FILE-CONTROL.
    10               I-O-CONTROL.
    11               DATA DIVISION.
                     1
**  1 2-3   A PERIOD IS REQUIRED PRIOR TO THIS WORD .WARNING.
    Relative Line Number

    12               FILE SECTION.                                    SOURCE
    13               WORKING-STORAGE SECTION.                         PROGRAM
    14               01 WORD PICTURE X(72).
    15               01 GOOD PICTURE X(72).
    16               77 BLANKE
PICTURE X(72) VALUE SPACES.
    17               01 TEST1 PICTURE X(3).
    18               01 TEST2 PICTURE X(4).
    19               PROCEDURE DIVISION.
    20               READING-ROUTINE.
    21                   DISPLAY "WHAT WORD?".
    22                   MOVE BLANKER TO WORD.
    23                   ACCEPT WORD.
    24                   MOVE WORD TO TEST2.

*   1 5-148 RIGHT TRUNCATION OCCURS HERE  .EXPECTED WARNING. USING RELATIVE LINE NUMBERS
    Relative Line Number

    25                   IF TEST2 EQUAL TO "DONE" GO TO END-CARD;
    26                       ELSE NEXT SENTENCE.
    27                   DISPLAY "YOUR WORD WAS" WORD.
    28                   DISPLAY "CORRECT?".
    29                   ACCEPT GOOD.
    30                   MOVE GOOD TO TEST1.
                                      1

*   1 5-148 RIGHT TRUNCATION OCCURS HERE  .EXPECTED WARNING.
```

Figure B-3 Listing of PROG1.L

```
    31                    IF TEST1 EQUAL TO "YES" GO TO READING-ROUTIN
/CE;
    32                        ELSE NEXT SENTENCE.
    33                    MOVE GOOD TO TEST2.
                                    1
  *  1 5-148 RIGHT TRUNCATION OCCURS HERE  .EXPECTED WARNING.                SOURCE
                                                                            PORGRAM
    34                    IF TEST2 EQUAL TO "DONE" GO TO END-CARD;
    35                        ELSE NEXT SENTENCE.
    36                    DISPLAY "TOO BAD".
    37                    GO TO READING-ROUTINE.
    38                END-CARD.
    39                    DISPLAY "THAT'S ALL FOLKS".
    40                    STOP RUN.

GCOS6 MOD400-L2.1-09/24/1727      COBOLA  2.0
/cROG1       81/05/05    0910     PAGE 0002

                                                  SUMMARY OF
-> NO FATAL ERRORS; 1 WARNING                     COMPILER
EOF                                               MESSAGES.
RDY:
PR PROG1.M
```

Figure B-3 (cont). Listing of PROG1.L

Invoking the Linker

Once the source program is compiled, it can be linked. The command used to invoke the Linker is:

```
LINKER progname  -PT  [ctl_arg]
```

where:

progname

The bound unit pathname (simple, relative, or absolute) of the bound unit to be created (usually the program name, may be up to 62 characters in length).

-PT

Requests the Linker issue a prompt (L?) for input.

ctl_arg

Other valid control arguments for the Linker (see Section 6.

For example, to invoke the Linker for PROG1 (compiled above), enter:

```
LINKER PROG1 -PT
```

Figure B-4 shows the dialog used to link PROG1.

```
RDY:
LINKER PROG1 -PT              Invoke the Linker with prompt
LINKER -1982/06/18 0912:50.5
                              Linker responds with version and
                                date
L?                            Linker prompts for input
LIB>LDD>ZCART                 Link the run-time routines
L?                            Linker prompts for input
LINK PROG1                    Link the object program
L?                            Linker prompts for input
QUIT                          Quit the Linker
ROOT PROG1                    Linker responds with name of root
LINK DONE
```

Figure B-4. Linking PROG1

Executing a COBOL Program

To execute the compiled and linked COBOL program, type in the program name. Figure B-5 illustrates a sample manual execution of PROG1.

```
PROG1
WHAT WORD?
MARY
YOUR WORD WAS MARY
CORRECT?
YES
WHAT WORD?
MANUAL
YOUR WORD WAS MANUAL
CORRECT?
NO
TOO BAD
WHAT WORD?
DONE
THAT'S ALL FOLKS
RDY:
```

Figure B-5. Execution of PROG1

If data files are used they must be made available to the program by using the GET command before typing the program name. When execution terminates, use the REMOVE command to release the data files. For more information on GET or REMOVE, see the Commands manual.

PROGRAMMING TIPS FOR COMMUNICATIONS VIA COBOL

The File System interface provides the logical transfer between the COBOL program and an external device (terminal or another computer). The COBOL run-time routines issue File System macro calls according to the corresponding input/output statements in the compiled programs.

Interactive Devices and Files

The Executive defines communications devices and local TTY terminals in COBOL communications processing as "interactive."

Interactive devices are considered sequential files in COBOL. Data is read or written with the same COBOL read/write interface as for a file on a noninteractive device.

File System Considerations

Aside from the use of various COBOL I/O statements you should be aware of other considerations in using the File System within a communications environment. These considerations are detailed in the System Programmer's Guide, Volume I.

Source Program Entries in Communications

This subsection refers to certain COBOL source program entries in the context of COBOL communications. The Advanced COBOL Reference manual describes the COBOL source program language in detail.

SPECIFYING FILES IN THE SOURCE PROGRAM

You must describe every file with a separate SELECT statement in the FILE-CONTROL paragraph of the Environment Division. File organization and access mode must be stated as sequential.

Each file must have a unique name and, in the ASSIGN clause, be identified by a 2-character COBOL internal file name (IFN) consisting of a combination of the letters A through I and the digits 0 through 9; one letter must be included. The logical file number (LFN) is specified in the GET command (before execution) to connect the COBOL internal file name to the external file. This LFN is the same as the COBOL internal file name with letters A through I replaced by the digits 0 through 9. For example, a COBOL IFN of 0C would correspond to an LFN of 03 and an IFN of 0D to an LFN of 04, as in the commands.

```
GET 03 !VIP1
GET 04 !TTY1
```

USE OF GET COMMAND

In addition to connecting the internal file name to the external file, the GET command reserves the interactive file for processing until it is removed via the REMOVE command. GET guarantees exclusive use of the file prior to program execution and maintains use of the file until the corresponding REMOVE command.

ASSIGNING A FILE TO A DEVICE/TERMINAL

A device-type name of MSD used in the ASSIGN clause of the SELECT statement is the way to inform COBOL that the internal file is assigned to a terminal/device file.

For data entry applications (TTY or VIP) the file should be opened in INPUT mode.

For output-only terminals such as the receive-only printer (ROP) the file should be opened in OUTPUT mode. Bidirectional devices, such as the BSC 2780 can be opened in INPUT mode or OUTPUT mode but not for both INPUT and OUTPUT at the same time.

For interactive applications (TTY, VIP or BSC3780), the file can be opened in I-O mode allowing both input and output operations.

SELECT AND ASSIGN EXAMPLES

Figure B-6 shows an example of a FILE-CONTROL paragraph with SELECT and ASSIGN statements for the input file COMIN and the output file COMOUT. The internal file name for COMIN is 0C and for COMOUT is 0D. Before the program is executed, associate these files with the appropriate device(s) with either GET command. In this example, the commands could be:

```
GET 03   !TTY1
GET 04   !TTY1
```

Although these are different files, they can be associated with the same interactive device; i.e., TTY1, by matching the logical file numbers (03 and 04 for the device pathname !TTY1) with the internal file name 0C and 0D, respectively.

```
FILE-CONTROL.

     SELECT COMIN

          ASSIGN TO 0C-MSD
          ORGANIZATION IS SEQUENTIAL WITH VLR
          ACCESS MODE IS SEQUENTIAL
          FILE STATUS IS IN-STATE.

     SELECT COMOUT

          ASSIGN TO 0D-PRINTER
          ORGANIZATION IS SEQUENTIAL WITH VLR
          ACCESS MODE IS SEQUENTIAL
          FILE STATUS IS OUT-STATE.
```

Figure B-6. COBOL SELECT and ASSIGN Examples

CARRIAGE CONTROL

The print carriage control of some devices can be changed by the application program. If the device-type name is MSD, the application program controls the carriage directly by inserting a program-accessible control byte as the first character in each output record. This byte is the first character in each level-01 record description entry for the output file. It is counted as part of the record area and is directly accessible through statements in the COBOL application program.

PRINTER EMULATION

Printer emulation is the capability of assigning printer characteristics to other terminal devices. If the device-type name is PRINTER in the ASSIGN clause COBOL will automatically generate the carriage control byte as a result of an ADVANCING phase in the WRITE statement. This one byte print control character is inserted before each data record being written to the file. It is not counted as part of the record area and is not directly accessible to the application program.

SPECIFYING ASYNCHRONOUS OR SYNCHRONOUS READ AND WRITE EXECUTION

If the device is configured (see STTY directive) or modified (see STTY command) for synchronous I/O, READ and WRITE statements are always executed synchronously (i.e., the application is placed in the wait state until the read or write is complete).

If the device is configured (see STTY directive) or modified (see STTY command) for asynchronous I/O, READ and WRITE statements may be executed synchronously or asynchronously, as indicated through calls to the COBOL run-time routines ZCASYN (asynchronous execution) or ZCSYNC (synchronous execution). If neither call is specified, reads and writes are executed asynchronously.

A separate call to ZCSYNC or to ZCASYN is not necessary for each read or write, but when first issued, remains effective until changed by another call. However, if the same run unit is to execute several COBOL programs, each program must separately define its own synchronous or asynchronous condition.

NOTE

With either the ZCSYNC or ZCASYN call, the -CC argument must not be specified with the COBOLA command. The -CC argument causes CALL statements to refer only to overlays. (See the Advanced COBOL Reference manual.)

SYNCHRONOUS READ AND WRITE OPERATION (CALL "ZCSYNC")

In synchronous operation, the COBOL routine issues a read or write order without any file status checks. This puts the application program in the wait state until the read or write operation is complete, thus allowing other tasks to be executed.

The source language for synchronous read and write execution is:

```
CALL "ZCSYNC"
```

Synchronous operation is not useful for a program application that interacts with more than one terminal since each read from or write to a terminal must be satisfied before the next terminal can be processed.

ASYNCHRONOUS READ AND WRITE OPERATION (CALL "ZCASYN")

In asynchronous operation COBOL READ/WRITE runtime routines issue a test-file call prior to issuing a read or write order. For READ orders, a 9I return status is returned to the application if no data is available to be read. Likewise, for a WRITE order, a 9I status is returned to the application if the device is busy with the previous output. This permits the COBOL program to support terminal I/O without giving up control of the central processor until the I/O is complete. Note that this facility is only available for terminals which have been configured (see STTY directive) or modified (see STTY command) to allow asynchronous I/O.

WAIT FOR COMPLETION FOR ASYNCHRONOUS INPUT AND OUTPUT

For a multiterminal application, you can control asynchronous read and write operations by calling the COBOL runtime routines ZCWIN and ZCWOUT.

A call to ZCWIN results in a Wait File ($WIFIL) macro call, which waits until input is available from one or more of the specified terminals.

A call to ZCWOUT results in a Wait-File ($WOFIL) macro call, which waits until output is complete to one or more of the specified terminals.

NOTES

1. With Advanced COBOL programs, the -OC argument must not be specified with the COBOLA command for compilation when ZCWIN or ZCWOUT are called. ZCWIN and ZCWOUT must be called with dope vectors.

2. With the ZCWIN or ZCWOUT call, the -CC argument must not be specified with the COBOLA command. The -CC argument causes CALL statements to refer only to overlays.

3. See the Advanced COBOL Reference manual.

The System Programmer's Guide, Volume II describes the wait file macro calls, their format and arguments in detail. Note that the macro call arguments are similar to the values for the data-name description for the CALL statements (see below).

The source language to call ZCWIN or ZCWOUT is:

```
CALL {"ZCWIN"  } USING data-name
     {"ZCWOUT" }
```

Data-name is defined as follows:

```
01  data-name
    02  out-LFN  USAGE COMP-1.
    02  list-length  USAGE COMP-1.
    02  LFN-entry-1  USAGE COMP-1.
                  .
                  .
                  .
    02  LFN-entry-n  USAGE COMP-1.
```

The values for out-LFN, list-length, LFT-entry-1, and LFN-entry-n are identical to those for the wait file ($WIFIL and $WOFIL) macro calls, and are passed by the ZCWIN or ZCWOUT routine to the file system.

When CALL "ZCWIN" is specified, the list of LFNs may refer only to those devices for which READ statements have been issued. When call "ZCWOUT" is specified, the list of LFNs can refer only to those devices for which WRITE statements have been issued.

When an input/output operation is completed on any device in the list of LFNs, the application program resumes execution following the CALL statement. The LFN for the device for which input/output is complete is stored in the out-LFN data item.

Figure B-7 provides simplified program logic for processing multiple terminals. The call to "ZCWIN" stalls program execution until input is available from at least one of the terminals.

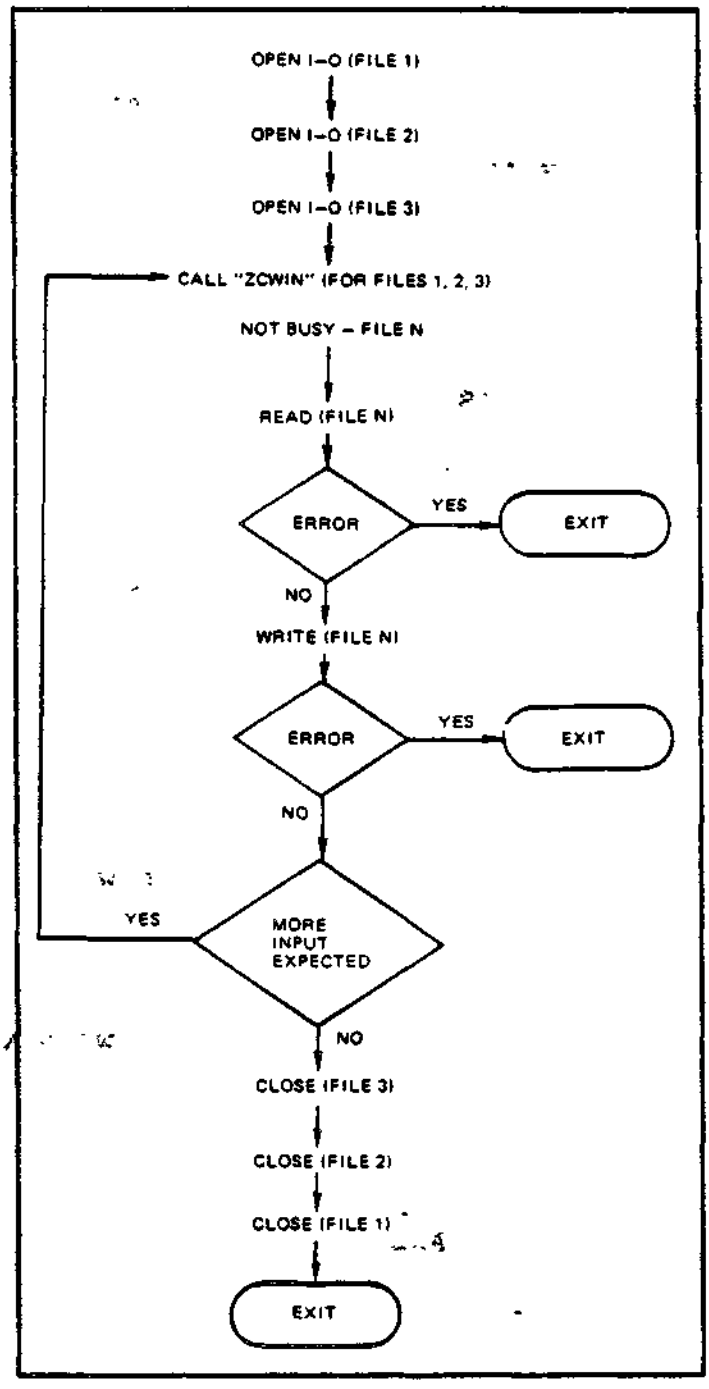


Figure B-7. Simplified COBOL Program Logic for Multiple Interactive Terminals (Asynchronous Input/Synchronous Output)

The following example illustrates portions of a COBOL program processing two terminals. This processing allows asynchronous input and synchronous output operations. The call to ZCWIN gives up control of the central processor unit input is available from one of the terminals.

```
      IDENTIFICATION DIVISION.
      PROGRAM-ID. TESTB.
      AUTHOR. HONEYWELL.
     *
     *                          ***COMMENTS***
     *
     *   THIS PROGRAM PERFORMS COMMUNICATIONS VIA
     *   COBOL.  TWO TERMINALS ARE TREATED AS
     *   SEQUENTIAL FILES.  DATA IS ASYNCHRONOUSLY
     *   READ FROM AN SYNCHRONOUSLY WRITTEN TO EACH
     *   OF THESE TERMINALS, DEPENDING UPON WHAT IS
     *   TYPED AT THE TERMINAL.
     *
     *   WHEN THE DATA IS READ - IT IS ECHOED BACK,
     *   EVALUATED, AND EITHER WRITTEN OUT AGAIN WITH A
     *   MESSAGE (I.E., "TERMINAL ONE:") OR PROCESSED
     *   THROUGH AN ERROR ROUTINE OR A HALT PROCEDURE.
     *
     *
      DATE-WRITTEN. DECEMBER 10, 1980.
      ENVIRONMENT DIVISION.
      CONFIGURATION SECTION.
      SOURCE-COMPUTER. LEVEL-6.
      OBJECT-COMPUTER. LEVEL-6.
      INPUT-OUTPUT SECTION.
      FILE-CONTROL.
          SELECT COM1
              ASSIGN TO 0C-MSD
              ORGANIZATION IS SEQUENTIAL WITH VLR
              ACCESS MODE IS SEQUENTIAL
              FILE STATUS IS C1-STAT.
          SELECT COM2
              ASSIGN TO 0D-MSD
              ORGANIZATION IS SEQUENTIAL WITH VLR
              ACCESS MODE IS SEQUENTIAL
              FILE STATUS IS C2-STAT.
      DATA DIVISION.
      FILE SECTION.
      FD  COM1
          LABEL RECORDS ARE OMITTED.
      01  COM1-REC.
          02  DA-TA1 PIC X(40).
          02  FILLER PIC X(40).
      FD  COM2
          LABEL RECORDS ARE OMITTED.
      01  COM2-REC.
          02  DA-TA2 PIC X(40).
          02  FILLER PIC X(40).
```

```
WORKING-STORAGE SECTION.
01  MES1.
    02  FILLER PIC X(01) VALUE "A".
    02  FILLER PIC X(13) VALUE "TERMINAL ONE:".
    02  INS1 PIC X(40) VALUE SPACES.
    02  FILLER PIC X(26) VALUE SPACES.
01  MES2.
    02  FILLER PIC X(01) VALUE "A".
    02  FILLER PIC X(13) VALUE "TERMINAL TWO:".
    02  INS2 PIC X(40) VALUE SPACES.
    02  FILLER PIC X(26) VALUE SPACES.
01  DONE.
    02  FILLER PIC X(04) VALUE "DONE".
    02  FILLER PIC X(76) VALUE SPACES.
01  LFN-LIST1.
    02  WHICH PIC 99 USAGE COMP-1
          VALUE ZEROES.
    02  NUMB USAGE COMP-1
          VALUE IS 02.
    02  LFN1 USAGE COMP-1
          VALUE IS 03.
    02  LFN2 USAGE COMP-1
          VALUE IS 04.
01  LFN-LIST2.
    02  WHICH2 PIC 99 USAGE COMP-1
          VALUE ZEROES.
    02  NUMB2 USAGE COMP-1
          VALUE IS 01.
    02  LFN1A USAGE COMP-1
          VALUE IS 03.
01  LFN-LIST3.
    02  WHICH3 PIC 99 USAGE COMP-1
          VALUE IS ZEROES.
    02  NUMB3 USAGE COMP-1
          VALUE IS 01.
    02  LFN2A USAGE COMP-1
          VALUE IS 04.
01  MSG.
    02  FILLER PIC X(01) VALUE "A".
    02  FILLER PIC X(11) VALUE "ENTER DATA":.
    02  FILLER PIC X(28) VALUE SPACES.
01  ECHO.
    02  ECHO1 PIC X(01) VALUE "A".
    02  ECHO2 PIC X(39) VALUE SPACES.
01  C1-STAT PIC X(02) VALUE SPACES.
01  C2-STAT PIC X(02) VALUE SPACES.
PROCEDURE DIVISION.
 BEGIN.
* THIS CALL CAUSES THE SYNCHRONOUS OPENING OF
* THE TWO FILES (TERMINALS).
  CALL "ZCSYNC"
  OPEN I-O COM1.
  OPEN I-O COM2.
```

```
SCR1.
    MOVE MSG TO DA-TA1.
*   ENTER DATA MESSAGE IS WRITTEN ON SCREEN 1.
    WRITE COM1-REC.
    MOVE SPACES TO COM1-REC.
SCR2.
    MOVE MSG TO DA-TA2.
*   ENTER DATA MESSAGE IS WRITTEN ON SCREEN 2.
    WRITE COM2-REC.
    MOVE SPACES TO COM2-REC.
R-D.
*   "ZCASYN" CAUSES ALL I/O TO BE ASYNCHRONOUS.
*   "ZCWIN" CAUSES A WAIT UNTIL INPUT IS AVAILABLE
*   FROM 1 OR MORE FILES (TERMINALS).  THE FIRST
*   TIME THROUGH THE LOOP IT IS WAITING FOR THE
*   ANTICIPATORY READ FROM AT LEAST ONE OF THE
*   OPEN STATEMENTS TO BE COMPLETED.
*
    CALL "ZCASYN".
    CALL "ZCWIN" USING LFN-LIST1.
    READ COM1 AT END GO TO EOF1.
    READ COM2 AT END GO TO EOF2.
    IF C1-STAT IS EQUAL TO "9I" GO TO R-D2.
    IF C1-STAT IS EQUAL TO "00" GO TO WR1.
    IF C1-STAT IS EQUAL TO "10" GO TO EOF1.
    GO TO ER-ROR.
R-D2.
    IF C2-STAT IS EQUAL TO "00" GO TO WR2.
    IF C2-STAT IS EQUAL TO "10" GO TO EOF2.
    GO TO ER-ROR.
WR1.
    MOVE DA-TA1 TO ECHO2.
    MOVE ECHO TO DA-TA1.
    WRITE COM1-REC.
* THE WRITE IS ASYNCHRONOUS BUT THE CALL
* TO "ZCWOUT" CAUSES THE I/O TO BE PERFORMED
* SYNCHRONOUSLY BECAUSE IT CAUSES A WAIT UNITL
* THE WRITE IS COMPLETE.
    CALL "ZCWOUT" USING LFN-LIST2.
    IF ECHO2 IS EQUAL TO DONE
        GO TO EOP1.
    MOVE ECHO2 TO INS1.
    WRITE COM1-REC FROM MES1.
    CALL "ZCWOUT" USING LFN-LIST2.
    MOVE SPACES TO INS1, DA-TA1, ECHO2.
    GO TO R-D.
WR2.
    MOVE DA-TA2 TO ECHO2.
    MOVE ECHO TO DA-TA2.
    WRITE COM2-REC.
    CALL "ZCWOUT" USING LFN-LIST3.
    IF ECHO2 IS EQUAL TO DONE
        GO TO EOP2.
```

```
      MOVE ECHO2 TO INS2.
      WRITE COM2-REC FROM MES2.
      CALL "ZCWOUT" USING LFN-LIST3.
      MOVE SPACES TO INS2, DA-TA2, ECHO2.
      GO TO R-D.

    ER-ROR.
      DISPLAY "ERROR IN COMM PROCESSING.".
      CLOSE COM1.
      CLOSE COM2.
      STOP RUN.
    EOP1.
      DISPLAY "TERMINAL ONE TERMINATES PROGRAM".
      DISPLAY "BY ONE MESSAGE.".
      CLOSE COM1.
      CLOSE COM2.
      STOP RUN.
    EOP2.
      DISPLAY "TERMINAL TWO TERMINATES PROGRAM".
      DISPLAY "BY DONE MESSAGE.".
      CLOSE COM1.
      CLOSE COM2.
      STOP RUN.
    EOF1.
      DISPLAY "END OF FILE 1 REACHED.".
      CLOSE COM1.
      CLOSE COM2.
      STOP RUN.
    EOF2.
      DISPLAY "END OF FILE 2 REACHED.".
      CLOSE COM1.
      CLOSE COM2.
      STOP RUN.
  END COBOL (GENERATED)
```

Before program execution, specify these commands to connect the LFNs to the specific terminal files.

```
GET 3  !TTY1 (for IFN 0C-MSD)
GET 4  !TTY2 (for IFN 0D-MSD)
```

BINARY SYNCHRONOUS COMMUNICATION (BSC) WITH COBOL

Binary Synchronous Communication (BSC), operating in 2780 or 3780 mode, permits a COBOL program to transmit data over communications lines from one DPS 6/Level 6 system to another DPS 6/Level 6, to a Level 66 system, or to a non-Honeywell host system.

BSC DATA TRANSMISSION CONVENTIONS

BSC Data Codes

Data can be in alphanumeric ASCII, alphanumeric EBCDIC, or binary format. In communication between DPS 6/Level 6 and remote

host, each system must use the same code set (either ASCII or EBCDIC). When EBCDIC is used, the application programs must know whether transmission is nontransparent or trnsparent (i.e., BSC control characters are interpreted as data).

BSC Data Transmission Modes

There are two BSC transmission modes: basic and advanced.

In basic transmission mode there is no control byte. The absence of a control byte limits the functionality of the protocol, e.g., an application cannot send or receive two message blocks or cannot initiate a reverse interrupt (RVI) sequence.

In advanced transmission mode there is a control byte which is the first byte in the program's input or output buffer. The control byte is used to control the tranmission of data and is used to convey information concerning the receipt of data. With the control byte, the application has complete control over the transmission and reception of data to a remote host.

BSC Multi-block Transmission

The BSC multi-block feature allows an application to send or receive from one to seven message blocks (records) in a single transmission. It is available in both BSC 2780 and BSC 3780, and in both basic and advanced transmission modes. To use this feature, the application must:

- Specify this feature during system build (refer to the System Programmer's Guide, Volume I)
- Select this feature at connect time.
- Organize the data buffer.

Unique rules apply to the usage of this feature. Refer to the sections dealing with multi-block trnsmission in the System Programmer's Guide, Volume I. For example: if the multi-block feature is used during 2780 basic mode transmission, a word must be allocated at the beginning of the application's buffer (for the control byte), even though a control byte is not used in basic mode.

BSC 2780 and BSC 3780

BSC 2780 is a subset of BSC 3780. Technical differences between the two protocols can be summarized as a set of extensions to the 2780 protocol which are:

- Ability to receive a conversational reply without a preliminary bid sequence

- Ability to receive and transmit selected BSC control characters

The differences between the two protocols can be summarized as:

- BSC 2780

  Specified at system building time by the BSC device directive.

  Operates in basic or advanced mode.

  Supported with bidirectional use of BSC 2780 communication line. A CLOSE/OPEN sequence must be initiated prior to the reversal of the communication line.

- BSC 3780

  Specified at system building time by the XBSC directive.

  Operates only in advanced mode.

  Supported only in receive mode. Transmits must be performed in single-block mode (refer to the System Programmer's Guide, Volume I).

  Supported with interactive usage of the BSC 3780 communication line. To terminate a transmission the application must initiate an EOT sequence by setting the appropriate bit within the control byte.

An ETX message transmission sequence can also be terminated if the other application sends a conversational reply. Receipt of a conversational reply is indicated by a bit setting within the transmit control byte. Receipt of a conversational reply forces the application to issue a read order to receive the conversational response. The termination of a read seqeunce is indicated by the AT END condition.

BSC 2780 IN BASIC TRANSMISSION MODE

The following conditions apply in the use of binary synchronous communications in basic data transmission mode:

- An application cannot send an RVI (reverse interrupt) control character through the file system.

- BSC devices in basic transmission mode cannot initiate double (ITB) block transmissions (see "BSC 2780/3780 Line Protocol Handler".)

- An application can send only the ETB (end of transmission block) BSC control character, not the ETX (end of text) BSC control character.

- An application cannot detect the receipt of a DLE EOT (switched line disconnect). Refer to "BSC 2780/3780 Line Protocol Handler".

- An application can send data in either transparent or non-transparent mode.

- An application can send EOT (end of transmission) control characters by a CLOSE call.

- BSC operation assumes that the detab option is set off.

Figure B-8 illustrates the necessary logic to support a BSC 2780 application in basic transmission mode.

BSC 2780 IN ADVANCED DATA TRANSMISSION MODE

In the BSC advanced data transmission mode, the first byte of the application program's input or output buffer is a control byte that controls or supplies information about read/write operations. This byte can indicate, for example, whether data is to be transferred in transparent or nontransparent mode, or whether an ETB (end of transmission block) or ETX (end of text) control character is to be sent or received or if data is to be sent in single or multi-block mode (see the System Programmer's Guide, Volume I).

It is not necessary to send EOT control characters through the control byte since the user must close the file in output mode before attempting to read. Closing the file forces BSC if not in idle mode, to send an EOT control character.

BSC 3780 IN ADVANCED DATA TRANSMISSION MODE

The first byte of the application program's input or output buffer is a control byte. The control byte controls or supplies information about read/write operations. As with 2780 in advanced mode, the control byte controls whether data is to be sent in transparent mode, how the data is to be blocked (ETB or ETX), of if the data is to be sent in single-block or in multi-block streams.

The following conventions apply in using 3780 binary synchronous communication in advanced data transmission mode:

- The receipt of an optional conversational reply is indicated by a bit setting in the transmit control byte. (This can occur if the application has transmitted the last (ETX) block of a message.) The application must issue a read to receive the conversational response.

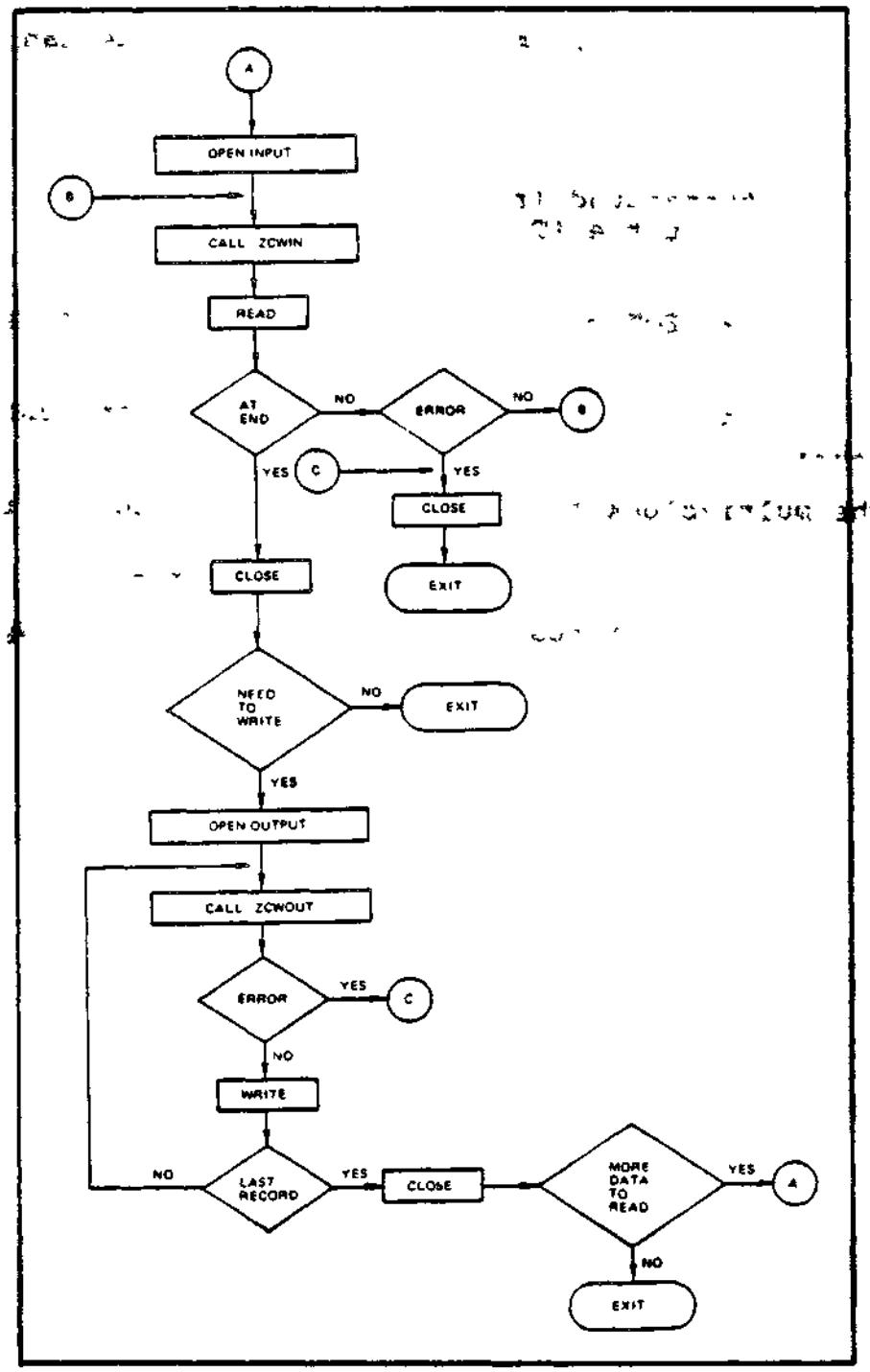


Figure B-8. Simplified Program Logic for BSC 2780

- The termination of a transmit sequence is signaled (via control byte) by the transmission of an EOT control character following the last block of a message. Once this has been done a read macro call will be needed to receive transmissions from the remote system. (It is not necessary to close and reopen the file to turn the line around.)

- The termination of a receive seqeunce is indicated by the AT END condition. A transmission sequence can be reinitiated by issuing another write macro call. (It is not necessary to close and reopen the file to turn the line around.)

- A line turnaround (receipt of an EOT or DLE EOT) is indicated at the AT END condition. At this point the application can use the line for data transmission by issuing another write request. It is also possible to receive an EOT control character that indicates the abortion of the current transmission seqeunce by the remote host. Such an occurrence is indicated by an AT END condition. If this occurs the application must close the line.

- The multi-block feature is only supported for receive operations. Transmit operations must be in single-block mode (refer to the System Programmer's Guide, Volume I).

Figure B-9 illustrates the necessary logic to support a BSC 3780 application.

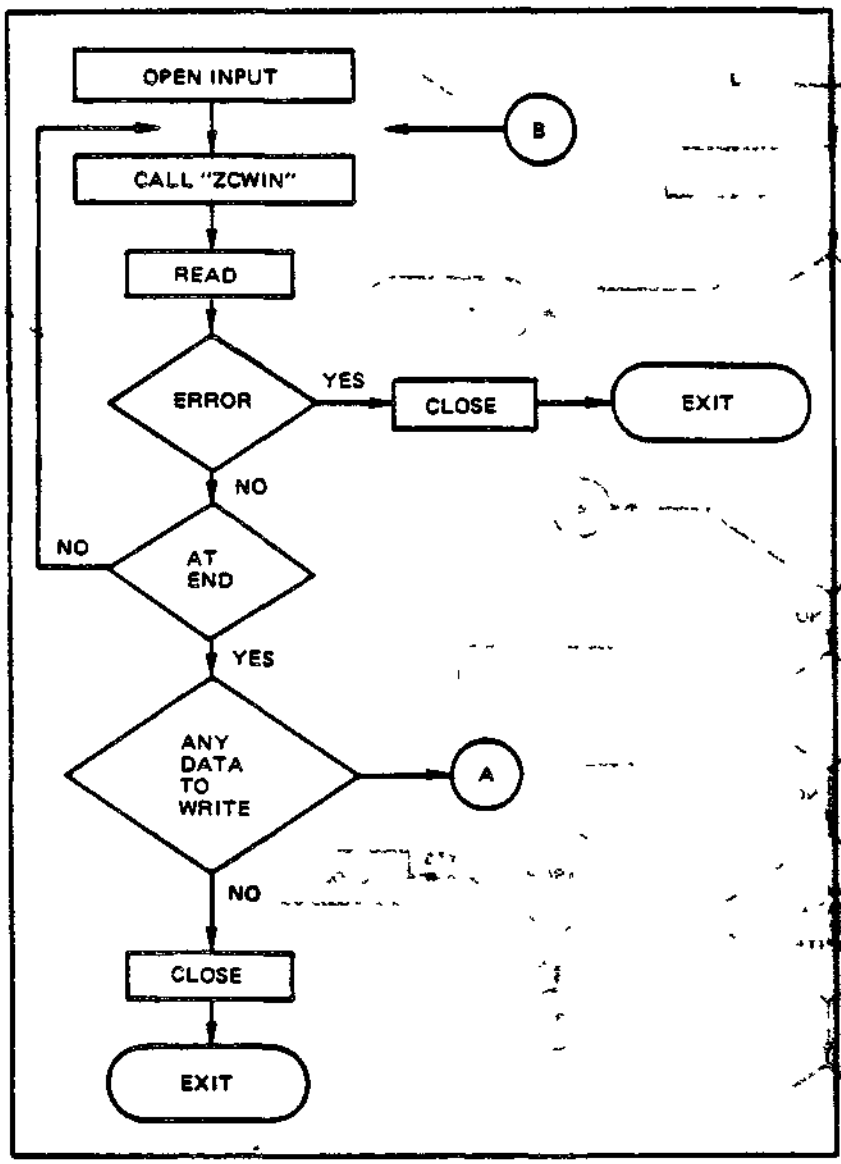


Figure B-9. Simplified Program Logic for BSC 3780

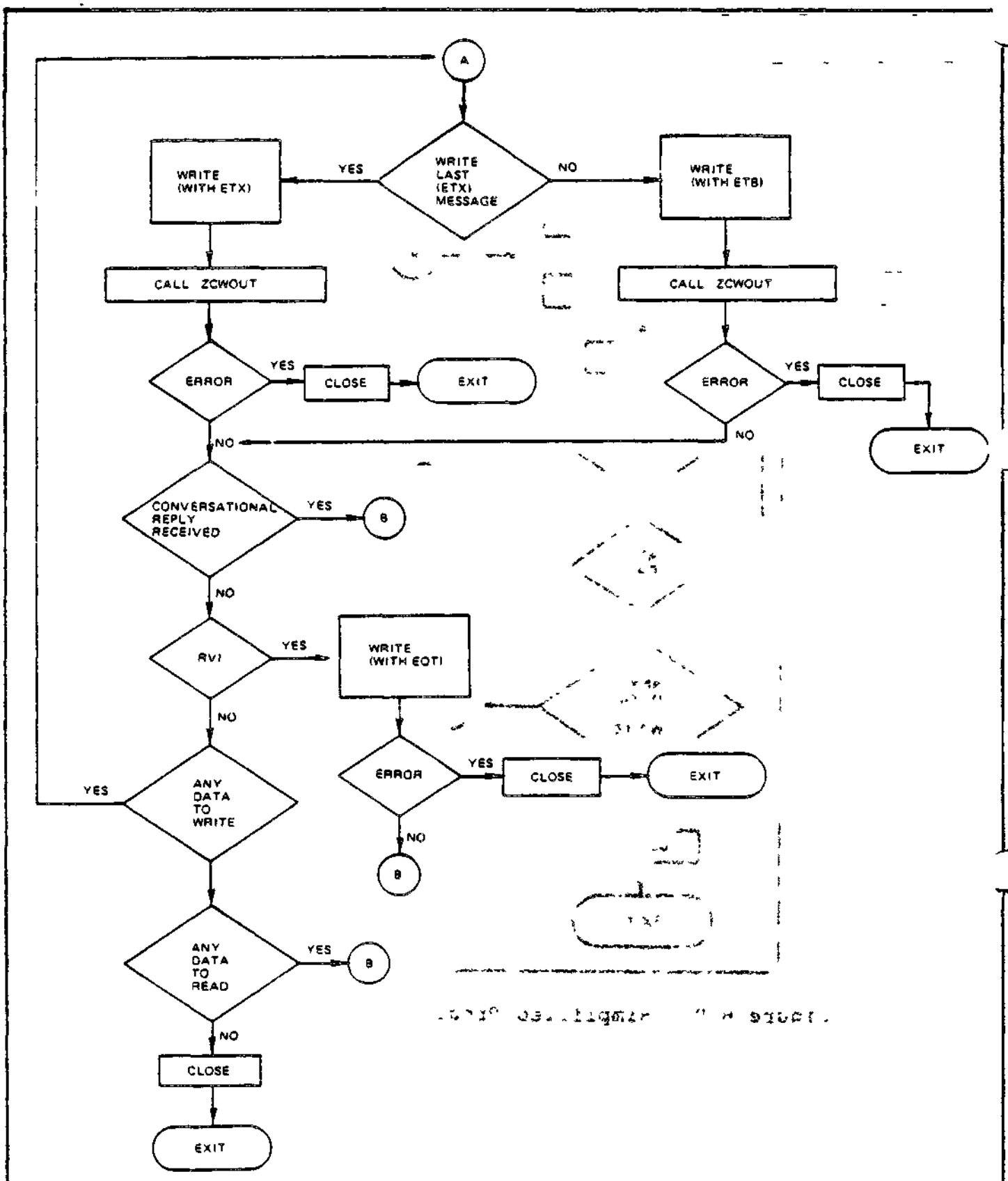


Figure B-9 (cont). Simplified Program Logic for BSC 3780

COBOL TTY OR VIP APPLICATION EXAMPLE

The COBOL source program listing in Figure B-10 is an example of an interactive application that involves either VIP or TTY devices.

This program (named CARCOM) processes commands entered from the operator terminal, and includes input/output operations to three communications terminals (TTY or VIP). An input and output file is assigned to each device. The program uses the operator terminal for entering commands and for receiving error messages. Input/output processing messages are displayed on the line printer.

COMMANDS IN THE COBOL EXAMPLE

The program processes the following interactive commands received from the operator terminal. The command COMND is entered from terminal 1, 2, or 3. (See "File Assignments" below.)

Command	Program Action
OPEN filename	Opens the file
CLOSE filename	Closes the file
ROUTE	Routes terminal output to other terminals as input
GO	Exits command mode, looks for input from terminals
COMND (entered from a terminal)	Exits terminal input mode; returns to operator terminal in command mode
STOP	Stops execution

FILE ASSIGNMENTS IN COBOL EXAMPLE

The program uses the following file names and corresponding logical file numbers (LFNs):

File Name	LFN	Device
COM1IN	03	Input terminal 1
COM1OT	04	Output terminal 1
COM2IN	05	Input terminal 2
COM2OT	06	Output terminal 2
COM3IN	07	Input terminal 3
COM3OT	08	Output terminal 3
PRINTFILE	01	Printer

```
 GCOS6 MOD400-S100-12/01/1413    COBOL   0200    01/01/01    0002    PAGE 0001
SOURCE  PROGRAM

    1           IDENTIFICATION DIVISION.
    2           PROGRAM-ID. CANCOM.
    3          *        COBOL COMMUNICATIONS
    4           ENVIRONMENT DIVISION.
    5           CONFIGURATION SECTION.
    6           SOURCE-COMPUTER. HIS-SERIES-60 LEVEL-6.
    7           OBJECT-COMPUTER. HIS-SERIES-60 LEVEL-6.
    8          *
    9           INPUT-OUTPUT SECTION.
   10           FILE-CONTROL.
   11               SELECT COM1IN
   12                   ASSIGN TO 0C-MSD,
   13                   ORGANIZATION IS SEQUENTIAL WITH VLR,
   14                   ACCESS MODE IS SEQUENTIAL,
   15                   FILE STATUS IS IN1-STAT.
   16               SELECT COM1OT
   17                   ASSIGN TO 0D-MSD,
   18                   ORGANIZATION IS SEQUENTIAL,
   19                   ACCESS MODE IS SEQUENTIAL,
   20                   FILE STATUS IS OT1-STAT.
   21               SELECT COM2IN
   22                   ASSIGN TO 0E-MSD,
   23                   ORGANIZATION IS SEQUENTIAL WITH VLR,
   24                   ACCESS MODE IS SEQUENTIAL,
   25                   FILE STATUS IS IN2-STAT.
   26               SELECT COM2OT
   27                   ASSIGN TO 0F-MSD,
   28                   ORGANIZATION IS SEQUENTIAL,
   29                   ACCESS MODE IS SEQUENTIAL,
   30                   FILE STATUS IS OT2-STAT.
   31               SELECT COM3IN
   32                   ASSIGN TO 0G-MSD,
   33                   ORGANIZATION IS SEQUENTIAL WITH VLR,
   34                   ACCESS MODE IS SEQUENTIAL,
   35                   FILE STATUS IS IN3-STAT.
   36               SELECT COM3OT
   37                   ASSIGN TO 0H-MSD,
   38                   ORGANIZATION IS SEQUENTIAL,
   39                   ACCESS MODE IS SEQUENTIAL,
   40                   FILE STATUS IS OT3-STAT.
   41               SELECT PRINTFILE
   42                   ASSIGN TO 0A-PRINTER,
   43                   ORGANIZATION IS SEQUENTIAL,
   44                   ACCESS MODE IS SEQUENTIAL,
   45                   FILE STATUS IS PRT-STAT.
   46          *
   47           DATA DIVISION.
   48          *
   49           FILE SECTION.
   50           FD  COM1IN
   51               BLOCK CONTAINS 1 RECORDS,
   52               LABEL RECORDS ARE OMITTED.
   53          *
   54           01  IN1-REC   PIC   X(80).
   55          *
   56           FD  COM1OT
   57               BLOCK CONTAINS 1 RECORDS,
   58               LABEL RECORDS ARE OMITTED.
```

Figure B-10. COBOL TTY or VIP Application Example

```
 GCOS6 MOD400-S100-12/01/1413    COBOL   0200
SOURCE  PROGRAM

  59          *
  60           01  OUTCOM1-REC.
  61               02  OT1-REC  PIC  X(80).
  62          *
  63           FD  COM2IN
  64               BLOCK CONTAINS 1 RECORDS,
  65               LABEL RECORDS ARE OMITTED.
  66           01  IN2-REC   PIC   X(80).
  67          *
  68           FD  COM2OT
  69               BLOCK CONTAINS 1 RECORDS,
  70               LABEL RECORDS ARE OMITTED.
  71           01  OUTCOM2-REC.
  72               02  OT2-REC  PIC  X(80).
  73          *
  74           FD  COM3IN
  75               BLOCK CONTAINS 1 RECORDS,
  76               LABEL RECORDS ARE OMITTED.
  77           01  IN3-REC    PIC    X(80).
  78          *
  79           FD  COM3OT
  80               BLOCK CONTAINS 1 RECORDS,
  81               LABEL RECORDS ARE OMITTED.
  82           01  OUTCOM3-REC.
  83               02  OT3-REC   PIC   X(80).
  84          *
  85           FD  PRINTFILE
  86               BLOCK CONTAINS 1 RECORDS,
  87               LABEL RECORDS ARE OMITTED.
  88           01  PRT-REC  PIC  X(120).
  89          *
  90           WORKING-STORAGE SECTION.
  91           01  TITLE.
  92               02  FILLER  PIC  XX  VALUE SPACES.
  93               02  FILLER  PIC  X(15)  VALUE "COBOL COMM TEST".
  94           01  CCMND1.
  95               02  FILLER  PIC  XX  VALUE SPACES.
  96               02  FILLER  PIC  X(27)  VALUE "TYPE FILE COMMANDS, THEN GO".
  97               02  FILLER  PIC  XX  VALUE SPACES.
  98           01  CCMND2.
  99               02  FILLER  PIC  XX  VALUE SPACES.
 100               02  FILLER  PIC  X(8)  VALUE "COMMAND?".
 101           01  HEAD1.
 102               02  FILLER  PIC  X(52)  VALUE SPACES.
 103               02  FILLER  PIC  X(15)  VALUE "COBOL COMM TEST".
 104               02  FILLER  PIC  X(53)  VALUE SPACES.
 105           01  HDR2.
 106               02  FILLER   PIC   X(6)   VALUE SPACES.
 107               02  FILLER   PIC   X(27)   VALUE "****  INPUT MSG FILE:  ".
 108               02  HDR2FIL   PIC   X(6)   VALUE SPACES.
 109           01  HDR3.
 110               02  FILLER   PIC   X(6)   VALUE SPACES.
 111               02  FILLER   PIC   X(28)   VALUE "****  OUTPUT MSG FILE:  ".
 112               02  HDR3FIL   PIC   X(6)   VALUE SPACES.
 113           01  LOADCOMP.
 114               02  FILLER  PIC  XX  VALUE SPACES.
 115               02  FILLER  PIC  X(13)  VALUE "LOAD COMPLETE".
 116           01  CONIN.
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

```
 GCOS8 MOD800-S100-12/01/1813    COBOL   0200
SOURCE  PROGRAM

 117              02  CMDFLD.
 118                  03  GOLFLD  PIC  X(2)  VALUE SPACES.
 119                  03  FILLER  PIC  X(3)  VALUE SPACES.
 120              02  FILLER  PIC  X  VALUE SPACES.
 121              02  FILFLD  PIC  X(8)  VALUE SPACES.
 122              02  FILLER  PIC  X(4) VALUE SPACES.
 123              02  FILFLD2  PIC  X(6) VALUE SPACES.
 124          01  CONINI REDEFINES CONIN.
 125              02  FILLER  PIC X(5).
 126              02  FILFLD1 PIC X(6).
 127          01  DSR-REC.
 128              02  ITEMNUM PIC  XXX  VALUE SPACES.
 129              02  FILLER  PIC  XX  VALUE SPACES.
 130              02  DESCFLD PIC  X(20)  VALUE SPACES.
 131              02  FILLER  PIC  XX  VALUE SPACES.
 132              02  QTYMED  PIC  9999  VALUE ZERO.
 133              02  FILLER  PIC  X(30) VALUE SPACES.
 134          01  IN1-STAT    PIC   XX    VALUE SPACES.
 135          01  OT1-STAT    PIC   XX    VALUE SPACES.
 136          01  IN2-STAT    PIC   XX    VALUE SPACES.
 137          01  OT2-STAT    PIC   XX    VALUE SPACES.
 138          01  IN3-STAT  PIC  XX  VALUE SPACES.
 139          01  OT3-STAT  PIC  XX  VALUE SPACES.
 140          01  PRT-STAT    PIC   XX    VALUE SPACES.
 141          01  RDR-STAT    PIC   XX    VALUE SPACES.
 142          01  INVF-STAT  PIC  XX  VALUE SPACES.
 143          *
 144          77  WKEY-I    PIC  999  VALUE ZERO.
 145          77  GO-IT     PIC  XX  VALUE "GO".
 146          77  OPNFIL     PIC  X(4)  VALUE "OPEN".
 147          77  CLSFIL     PIC  X(5)  VALUE "CLOSE".
 148          77  ALLDON   PIC  X(4)  VALUE "QUIT".
 149          77  STAR   PIC  X  VALUE "*".
 150          77  GO-ON   PIC  X(4)  VALUE "NEXT".
 151          77  LOADF  PIC  X(4)  VALUE "LOAD".
 152          77  ENDFM  PIC  X(4)  VALUE "EOF".
 153          77  IN1     PIC  X(6)  VALUE "COMIIN".
 154          77  OT1     PIC  X(6)  VALUE "COMIOT".
 155          77  IN2     PIC  X(6)  VALUE "COM2IN".
 156          77  OT2     PIC  X(6)  VALUE "COM2OT".
 157          77  IN3   PIC  X(6)  VALUE "COM3IN".
 158          77  OT3   PIC  X(6)  VALUE "COM3OT".
 159          77  ECHO  PIC  X(4)  VALUE "ECHO".
 160          77  NECHO PIC  X(5)  VALUE "NECHO".
 161          77  RDRF  PIC  X(6)  VALUE "CARDIN".
 162          77  INVF  PIC  X(8)  VALUE "INVFIL".
 163          77  RMO-ORD  PIC  9  VALUE ZERO.
 164          77  RMO-ERR  PIC  9  VALUE ZERO.
 165          77  FILCOUNT     PIC  99  VALUE ZERO.
 166          77  SDCOUNT     PIC  9999  VALUE ZERO.
 167          77  SD1COUNT    PIC  9999  VALUE ZERO.
 168          77  SD2COUNT    PIC  9999  VALUE ZERO.
 169          77  SD3COUNT    PIC  9999  VALUE ZERO.
 170          77  SD4COUNT    PIC  9999  VALUE ZERO.
 171          77  SD1MSG    PIC  9999  VALUE ZERO.
 172          77  SD2MSG    PIC  9999  VALUE ZERO.
 173          77  SD3MSG    PIC  9999  VALUE ZERO.
 174          77  SD4MSG    PIC  9999  VALUE ZERO.
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

```
GCOS6 MOD400-S100-12/01/1813    COMOL   0200
SOURCE  PROGRAM
                                                                    SEQ
175          77  MSGCOUNT   PIC  9999  VALUE ZERO.
176          77  ECHFLG1    PIC   9    VALUE ZERO.
177          77  ECHFLG2    PIC  9  VALUE ZERO.
178          77  ECHFLG3    PIC  9  VALUE ZERO.
179          77  RT1FLG     PIC  99  VALUE ZERO.
180          77  RT2FLG    PIC  99  VALUE ZERO.
181          77  RT3FLG    PIC  99  VALUE ZERO.
182          77  ROUTE     PIC  X(5)  VALUE "ROUTE".
183          77  RE-SET      PIC  X(5)  VALUE "RESET".
184          77  COMDNM     PIC  X(5)  VALUE "COMND".
185          77  SEND-DATA PIC  XX  VALUE "SD".
186          77  SEND-TEXT  PIC  XX  VALUE "ST".
187          77  KEYER      PIC  X(13)  VALUE "RELATIVE KEY ".
188          77  KDAYNM     PIC  X(13)  VALUE "INVALID KEY= ".
189          77  ORDERCMD  PIC  XX  VALUE "O ".
190          77  UPDATCMD  PIC  XX  VALUE "U ".
191          77  DISPITM   PIC  XX  VALUE "D ".
192          77  CCCHAR     PIC  X  VALUE "A".
193          77  NOTIFY     PIC  9999  VALUE 9999.
194          77  SWITCH1     PIC  99  VALUE ZERO.
195          77  SWITCH2     PIC  99  VALUE ZERO.
196          77  SWITCH3     PIC  99  VALUE ZERO.
197          77  SD1-FLG  PIC  9  VALUE ZERO.
198          77  SD2-FLG  PIC  9  VALUE ZERO.
199          77  SD3-FLG  PIC  9  VALUE ZERO.
200          77  SD4-FLG  PIC  9  VALUE ZERO.
201          77  INVSWTCH  PIC  99  VALUE ZERO.
202          77  TRNSWTCH  PIC  99  VALUE ZERO.
203          77  STATIN1     PIC  99  VALUE ZERO.
204          77  STATOT1     PIC  99  VALUE ZERO.
205          77  STATIN2     PIC  99  VALUE ZERO.
206          77  STATOT2     PIC  99  VALUE ZERO.
207          77  STATIN3    PIC  99  VALUE ZERO.
208          77  STATOT3    PIC  99  VALUE ZERO.
209          77  ERSUM1IN  PIC  99  VALUE ZERO.
210          77  ERSUM1OT  PIC  99  VALUE ZERO.
211          77  ERSUM2IN  PIC  99  VALUE ZERO.
212          77  ERSUM2OT  PIC  99  VALUE ZERO.
213          77  ERSUM3IN   PIC  99  VALUE ZERO.
214          77  ERSUM3OT   PIC  99  VALUE ZERO.
215          77  QTYSUR  PIC  S9999  VALUE ZERO.
216          77  NMCHRSLT  PIC  9  VALUE ZERO.
217          77  MAXNUM  PIC  9999  VALUE ZERO.
218          77  MAXITMNO  PIC  999  VALUE 200.
219          77  MAXQTY  PIC  9999  VALUE 1000.
220          77  CHRNUM  PIC  9999  VALUE ZERO.
221          *
222          01  INSPECTI.
223              02  INCMD  PIC  X(5)  VALUE SPACES.
224              02  FILLER  PIC  X(75)  VALUE SPACES.
225          01  COMCMD REDEFINES INSPECTI.
226              02  CMDIND  PIC  X.
227              02  CMDTYP  PIC  XX.
228              02  FILLER  PIC  X.
229              02  LFNNUM  PIC  99.
230              02  FILLER  PIC  X.
231              02  NUMSENDS  PIC  9999.
232              02  FILLER  PIC  X.
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

```
GCOS6 M0D400-S100-12/01/1413    COBOL   0200
SOURCE  PROGRAM

233             02  TEXT-MSG  PIC  X(68).
234         01  OPDSPL.
235             02  FILLER  PIC  XX  VALUE SPACES.
236             02  OFLNAM  PIC  X(6)  VALUE SPACES.
237             02  FILLER  PIC  XX  VALUE SPACES.
238             02  FILLER  PIC  X(6)  VALUE "OPENED".
239         01  OPERMSPL.
240             02  FILLER  PIC  XX  VALUE SPACES.
241             02  FILLER  PIC  X(19)  VALUE "OPEN ERROR   FILE: ".
242             02  OFLNER  PIC  X(6)  VALUE SPACES.
243             02  FILLER  PIC  X(6)  VALUE SPACES.
244             02  FILLER  PIC  X(8)  VALUE "STATUS= ".
245             02  KEYERR  PIC  XX  VALUE SPACES.
246         01  RDERMSG.
247             02  FILLER  PIC  XX  VALUE SPACES.
248             02  FILLER  PIC  X(19)  VALUE "READ ERROR   FILE: ".
249             02  RDERFIL  PIC  X(6)  VALUE SPACES.
250             02  FILLER  PIC  X(6)  VALUE SPACES.
251             02  FILLER  PIC  X(8)  VALUE "STATUS= ".
252             02  RDERSTAT  PIC  XX  VALUE SPACES.
253         01  WRERMSG.
254             02  FILLER  PIC  XX  VALUE SPACES.
255             02  FILLER  PIC  X(19)  VALUE "WRITE ERROR  FILE: ".
256             02  WRERFIL  PIC  X(6)  VALUE SPACES.
257             02  FILLER  PIC  X(6)  VALUE SPACES.
258             02  FILLER  PIC  X(8)  VALUE "STATUS= ".
259             02  WRERSTAT  PIC  XX  VALUE SPACES.
260         01  CLDSPL.
261             02  FILLER  PIC  XX  VALUE SPACES.
262             02  CFLNAM  PIC  X(6)  VALUE SPACES.
263             02  FILLER  PIC  XX  VALUE SPACES.
264             02  FILLER  PIC  X(6)  VALUE "CLOSED".
265         01  CLERMSG.
266             02  FILLER  PIC  XX  VALUE SPACES.
267             02  FILLER  PIC  X(19)  VALUE "CLOSE ERROR  FILE: ".
268             02  CFLNER  PIC  X(6)  VALUE SPACES.
269             02  FILLER  PIC  X(6)  VALUE SPACES.
270             02  FILLER  PIC  X(8)  VALUE "STATUS= ".
271             02  CKEYERR  PIC  XX  VALUE SPACES.
272         01  BADFIL.
273             02  FILLER  PIC  XX  VALUE SPACES.
274             02  FILLER  PIC  X(16)  VALUE "ILLEGAL FILENAME".
275         01  BADCMD.
276             02  FILLER  PIC  XX  VALUE SPACES.
277             02  FILLER  PIC  X(15)  VALUE "ILLEGAL COMMAND".
278         01  NOTFSUM.
279             02  FILLER  PIC  XX  VALUE SPACES.
280             02  FILLER  PIC  X(6)  VALUE "FILE: ".
281             02  FRR9I   PIC   X(6)   VALUE SPACES.
282             02  FILLER  PIC  X(6)  VALUE SPACES.
283             02  FILLER  PIC  X(10)  VALUE "STATUS= 91".
284         01  STOPCOB.
285             02  FILLER  PIC  XX  VALUE SPACES.
286             02  FILLER  PIC  X(10)  VALUE "STOP COBOL".
287         01  KEY-MSG.
288             02  FILLER  PIC X(16)  VALUE "FILE KEY STATUS ".
289             02  BAD-KEY PIC XX   VALUE SPACES.
290             02  FILLER   PIC X(12)  VALUE " TEST FAILED".
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

```
 GCOS6 MOD400-S100-12/01/1813    COBOL    0200
SOURCE   PROGRAM

 291          01  DATAMSG.
 292              02  MSGNUM  PIC  9999  VALUE ZERO.
 293              02  FILLER  PIC  X(6)  VALUE "456789".
 294              02  FILLER  PIC  X(10) VALUE "0123456789".
 295              02  FILLER  PIC  X(10) VALUE "0123456789".
 296              02  FILLER  PIC X(10) VALUE "0123456789".
 297              02  FILLER  PIC  X(10) VALUE "0123456789".
 298              02  FILLER  PIC  X(20) VALUE "01234567890123456789".
 299              02  FILLER  PIC  X(20) VALUE "01234567890123456789".
 300          *
 301          PROCEDURE DIVISION.
 302          *
 303          PHEADS.
 304              DISPLAY CTITLE.
 305              OPEN OUTPUT PRINTFILE.
 306              MOVE HEAD1 TO PRT-REC.
 307              WRITE PRT-REC AFTER ADVANCING PAGE.
 308          PCMD1.
 309              DISPLAY CCMND1.
 310              MOVE SPACES TO CONIN.
 311              ACCEPT CONIN.
 312              IF CMDFLD IS EQUAL TO OPNFIL GO TO OPENIT.
 313              IF CMDFLD IS EQUAL TO CLSFIL GO TO CLOSIT.
 314              IF CMDFLD IS EQUAL TO GO-ON GO TO PCMD2.
 315              DISPLAY BADCMD.
 316              GO TO PCMD1.
 317          PCMD2.
 318              DISPLAY CCMND2.
 319              MOVE SPACES TO CONIN.
 320              ACCEPT CONIN.
 321              IF CMDFLD IS EQUAL TO OPNFIL GO TO OPENIT.
 322              IF CMDFLD IS EQUAL TO CLSFIL GO TO CLOSIT.
 323              IF CMDFLD IS EQUAL TO ROUTE GO TO SETROUTE.
 324              IF CMDFLD IS EQUAL TO RF-SET GO TO DEMOUTF.
 325              IF CMDFLD IS EQUAL TO ECHO GO TO SETECHO.
 326              IF CMDFLD IS EQUAL TO NECHO GO TO NO-ECHO.
 327              IF CMDFLD IS EQUAL TO DO-IT GO TO READ1.
 328              IF CMDFLD IS EQUAL TO ALLDON GO TO DONEIT.
 329              DISPLAY BADCMD.
 330              GO TO PCMD2.
 331          OPENIT.
 332              IF FILFLD1 IS EQUAL TO IN1 GO TO OPIN1.
 333              IF FILFLD1 IS EQUAL TO OT1 GO TO OPOT1.
 334              IF FILFLD1 IS EQUAL TO IN2 GO TO OPIN2.
 335              IF FILFLD1 IS EQUAL TO OT2 GO TO OPOT2.
 336              IF FILFLD1 IS EQUAL TO IN3 GO TO OPIN3.
 337              IF FILFLD1 IS EQUAL TO OT3 GO TO OPOT3.
 338              DISPLAY BADFIL.
 339              IF FILCOUN1 GREATER THAN 1 GO TO PCMD2.
 340              GO TO PCMD1.
 341          OPIN1.
 342              OPEN INPUT COMIN.
 343              IF IN1-STAT = "00" OR IN1-STAT = "95";
 344                  MOVE 1 TO STATIN1;
 345                  MOVE 1 TO SWITCH1;
 346                  MOVE IN1 TO OFLNAM;
 347                  GO TO OPMSG.
 348              MOVE IN1 TO OFLNER.
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

```
 GCOS6 MOD400-S100-12/01/1813     COBOL    0200
SOURCE   PROGRAM

 349                 MOVE IN1-STAT TO KEYERR.
 350                 GO TO OPERMG.
 351             OPOUT1.
 352                 OPEN OUTPUT COM1OUT.
 353                 IF OT1-STAT = "00" OR OT1-STAT = "95":
 354                     MOVE 1 TO STATOT1;
 355                     MOVE OT1 TO OFLNAM;
 356                     GO TO OPMSG.
 357                 MOVE OT1 TO OFLNER.
 358                 MOVE OT1-STAT TO KEYERR.
 359                 GO TO OPERMG.
 360             OPIN2.
 361                 OPEN INPUT COM2IN.
 362                 IF IN2-STAT = "00" OR IN2-STAT = "95":
 363                     MOVE 1 TO STATIN2;
 364                     MOVE 1 TO SWITCH2;
 365                     MOVE IN2 TO OFLNAM;
 366                     GO TO OPMSG.
 367                 MOVE IN2 TO OFLNER.
 368                 MOVE IN2-STAT TO KEYERR.
 369                 GO TO OPERMG.
 370             OPOT2.
 371                 OPEN OUTPUT COM2OT.
 372                 IF OT2-STAT = "00" OR OT2-STAT = "95":
 373                     MOVE 1 TO STATOT2;
 374                     MOVE OT2 TO OFLNAM;
 375                     GO TO OPMSG.
 376                 MOVE OT2 TO OFLNER.
 377                 MOVE OT2-STAT TO KEYERR.
 378                 GO TO OPERMG.
 379             OPIN3.
 380                 OPEN INPUT COM3IN.
 381                 IF IN3-STAT = "00" OR IN3-STAT = "95":
 382                     MOVE 1 TO STATIN3;
 383                     MOVE 1 TO SWITCH3;
 384                     MOVE IN3 TO OFLNAM;
 385                     GO TO OPMSG.
 386                 MOVE IN3 TO OFLNER.
 387                 MOVE IN3-STAT TO KEYERR.
 388                 GO TO OPERMG.
 389             OPOT3.
 390                 OPEN OUTPUT COM3OT.
 391                 IF OT3-STAT = "00" OR OT3-STAT = "95":
 392                     MOVE 1 TO STATOT3;
 393                     MOVE OT3 TO OFLNAM;
 394                     GO TO OPMSG.
 395                 MOVE OT3 TO OFLNER.
 396                 MOVE OT3-STAT TO KEYERR.
 397                 GO TO OPERMG.
 398             OPMSG.
 399                 DISPLAY OPOSPL.
 400                 ADD 1 TO FILCOUNT.
 401                 IF FILCOUNT GREATER THAN 1 GO TO PCMD2.
 402                 GO TO PCMD1.
 403             OPERMG.
 404                 DISPLAY OPERRSPL.
 405                 IF FILCOUNT GREATER THAN 1 GO TO PCMD2.
 406                 GO TO PCMD1.
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

```
 GCOS6 M00400-S100-12/01/1415     COMOL    0200
SOURCE  PROGRAM

407             CLOSIT.
408                 IF FILFLD IS EQUAL TO IN1 GO TO CLIN1.
409                 IF FILFLD IS EQUAL TO OT1 GO TO CLOT1.
410                 IF FILFLD IS EQUAL TO IN2 GO TO CLIN2.
411                 IF FILFLD IS EQUAL TO OT2 GO TO CLOT2.
412                 IF FILFLD IS EQUAL TO IN3 GO TO CLIN3.
413                 IF FILFLD IS EQUAL TO OT3 GO TO CLOT3.
414                 DISPLAY BADCMD.
415                 IF FILCOUNT GREATER THAN 1 GO TO PCMD2.
416                 GO TO PCMD1.
417             CLIN1.
418                 CLOSE COMIIN.
419                 IF IN1-STAT = "00";
420                     MOVE ZERO TO SWITCH1;
421                     MOVE ZERO TO STATIN1;
422                     MOVE IN1 TO CFLNAM;
423                     GO TO CLOPMSG.
424                 MOVE IN1 TO CFLNER.
425                 MOVE IN1-STAT TO CREYERR.
426                 GO TO COPERMG.
427             CLOT1.
428                 CLOSE COM1OT.
429                 IF OT1-STAT = "00";
430                     MOVE ZERO TO STATOT1;
431                     MOVE OT1 TO CFLNAM;
432                     GO TO CLOPMSG.
433                 MOVE OT1 TO CFLNER.
434                 MOVE OT1-STAT TO CREYERR.
435                 GO TO COPERMG.
436             CLIN2.
437                 CLOSE COM2IN.
438                 IF IN2-STAT = "00";
439                     MOVE ZERO TO SWITCH2;
440                     MOVE ZERO TO STATIN2;
441                     MOVE IN2 TO CFLNAM;
442                     GO TO CLOPMSG.
443                 MOVE IN2 TO CFLNER.
444                 MOVE IN2-STAT TO CREYERR.
445                 GO TO COPERMG.
446             CLOT2.
447                 CLOSE COM2OT.
448                 IF OT2-STAT = "00";
449                     MOVE ZERO TO STATOT2;
450                     MOVE OT2 TO CFLNAM;
451                     GO TO CLOPMSG.
452                 MOVE OT2 TO CFLNER.
453                 MOVE OT2-STAT TO CREYERR.
454                 GO TO COPERMG.
455             CLIN3.
456                 CLOSE COM3IN.
457                 IF IN3-STAT = "00";
458                     MOVE ZERO TO SWITCH3;
459                     MOVE ZERO TO STATIN3;
460                     MOVE IN3 TO CFLNAM;
461                     GO TO CLOPMSG.
462                 MOVE IN3 TO CFLNER.
463                 MOVE IN3-STAT TO CREYERR.
464                 GO TO COPERMG.
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

```
GCOS6 MUD400-S100-12/01/1413     COBOL    0200
SOURCE  PROGRAM

465             CLOT3.
466                 CLOSE COMSOT.
467                 IF OTS-STAT = "00";
468                     MOVE ZERO TO STATOT3;
469                     MOVE OT3 TO CFLNAM;
470                     GO TO CLOPMSG.
471                 MOVE OT3 TO CFLNER.
472                 MOVE OT3-STAT TO CRETERR.
473                 GO TO COPERMG.
474             CLOPMSG.
475                 DISPLAY CLOSPL.
476                 SUBTRACT 1 FROM FILCOUNT.
477                 IF FILCOUNT GREATER THAN 1 GO TO PCMD2.
478                 GO TO PCMD1.
479             COPERMG.
480                 DISPLAY CLERMSG.
481                 IF FILCOUNT GREATER THAN 1 GO TO PCMD2.
482                 GO TO PCMD1.
483             SETROUTE.
484                 IF FILFLD IS EQUAL TO IN1 AND STATIN1 = 1 GO TO SET1.
485                 IF FILFLD IS EQUAL TO IN2 AND STATIN2 = 1 GO TO SET2.
486                 IF FILFLD IS EQUAL TO IN3 AND STATIN3 = 1 GO TO SET3.
487             RTEERR.
488                 DISPLAY BADCMD.
489                 GO TO PCMD2.
490             SET1.
491                 IF FILFLD2 IS EQUAL TO OT2 AND STATOT2 = 1;
492                     MOVE 2 TO RT1FLG;
493                     GO TO PCMD2.
494                 IF FILFLD2 IS EQUAL TO OT3 AND STATOT3 = 1;
495                     MOVE 3 TO RT1FLG;
496                     GO TO PCMD2.
497                 GO TO RTEERR.
498             SET2.
499                 IF FILFLD2 IS EQUAL TO OT1 AND STATOT1 = 1;
500                     MOVE 1 TO RT2FLG;
501                     GO TO PCMD2.
502                 IF FILFLD2 IS EQUAL TO OT3 AND STATOT3 = 1;
503                     MOVE 3 TO RT2FLG;
504                     GO TO PCMD2.
505                 GO TO RTEERR.
506             SET3.
507                 IF FILFLD2 IS EQUAL TO OT1 AND STATOT1 = 1;
508                     MOVE 1 TO RT3FLG;
509                     GO TO PCMD2.
510                 IF FILFLD2 IS EQUAL TO OT2 AND STATOT2 = 1;
511                     MOVE 2 TO RT3FLG;
512                     GO TO PCMD2.
513                 GO TO RTEERR.
514             DEMROUTE.
515                 IF FILFLD IS EQUAL TO IN1;
516                     MOVE ZERO TO RT1FLG;
517                     GO TO PCMD2.
518                 IF FILFLD IS EQUAL TO IN2;
519                     MOVE ZERO TO RT2FLG;
520                     GO TO PCMD2.
521                 IF FILFLD IS EQUAL TO IN3;
522                     MOVE ZERO TO RT3FLG;
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

```
 GCOS6 MOD400-S100-12/01/1413    COBOL   0200
SOURCE  PROGRAM

523                 GO TO PCMD2.
524             DISPLAY BADCMD.
525             GO TO PCMD2.
526         ND-ECMD.
527             IF FILFLD IS EQUAL TO IN1 MOVE 1 TO ECMFLG1;
528                 GO TO PCMD2.
529             IF FILFLD IS EQUAL TO IN2 MOVE 1 TO ECMFLG2;
530                 GO TO PCMD2.
531             IF FILFLD IS EQUAL TO IN3 MOVE 1 TO ECMFLG3;
532                 GO TO PCMD2.
533             DISPLAY BADCMD.
534             GO TO PCMD2.
535         SETECMD.
536             IF FILFLD1 IS EQUAL TO IN1 MOVE ZERO TO ECMFLG1;
537                 GO TO PCMD2.
538             IF FILFLD1 IS EQUAL TO IN2 MOVE ZERO TO ECMFLG2;
539                 GO TO PCMD2.
540             IF FILFLD1 IS EQUAL TO IN3 MOVE ZERO TO ECMFLG3;
541                 GO TO PCMD2.
542             DISPLAY BADCMD.
543             GO TO PCMD2.
544         READ1.
545             IF FILCOUN1 = ZERO GO TO PCMD1.
546             IF SWITCH1 = ZERO GO TO READ2.
547             MOVE SPACES TO IN1-REC.
548             READ COM1IN AT END GO TO DONEIT.
549             IF IN1-STAT = "00" GO TO GOODR1.
550             IF IN1-STAT = "91";
551                 GO TO READ2.
552             MOVE IN1-STAT TO RDERSTAT.
553             MOVE IN1 TO RDERFIL.
554             DISPLAY RDERMSG.
555             ADD 1 TO ERSUM1IN.
556             IF ERSUM1IN NOT LESS THAN 4 GO TO CLIN1.
557         READ2.
558             IF SWITCH2 = ZERO GO TO READ3.
559             MOVE SPACES TO IN2-REC.
560             READ COM2IN AT END GO TO DONEIT.
561             IF IN2-STAT = "00" GO TO GOODR2.
562             IF IN2-STAT = "91";
563                 GO TO READ3.
564             MOVE IN2-STAT TO RDERSTAT.
565             MOVE IN2 TO RDERFIL.
566             DISPLAY RDERMSG.
567             ADD 1 TO ERSUM2IN.
568             IF ERSUM2IN NOT LESS THAN 4 GO TO CLIN2.
569             GO TO READ3.
570         READ3.
571             IF SWITCH3 = ZERO GO TO CHKSD.
572             MOVE SPACES TO IN3-REC.
573             READ COM3IN AT END GO TO DONEIT.
574             IF IN3-STAT = "00" GO TO GOODR3.
575             IF IN3-STAT = "91" GO TO CHKSD.
576             MOVE IN3 TO RDERFIL.
577             DISPLAY RDERMSG.
578             ADD 1 TO ERSUM3IN.
579             IF ERSUM3IN NOT LESS THAN 4 GO TO CLIN3.
580         CHKSD.
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

```
 GCOS6 MOD400-S100-12/01/1815    COMOL    0200
SOURCE  PROGRAM

581                 IF SD1-FLG IS EQUAL TO 1 PERFORM SD1.
582                 IF SD2-FLG IS EQUAL TO 1 PERFORM SD2.
583                 IF SD3-FLG IS EQUAL TO 1 PERFORM SD3.
584                 GO TO READ1.
585             GOODR1.
586                 MOVE ZERO TO ERSUM1IN.
587                 PERFORM PRTIN1 THRU CHK9IPT1.
588                 MOVE IN1-REC TO INSPECTT.
589                 IF CMDIND IS EQUAL TO STAR GO TO WATLOU.
590                 IF INCMD IS EQUAL TO COMDNM GO TO PCMD2.
591                 IF RT1FLG IS EQUAL TO 2;
592                     MOVE IN1-REC TO OT2-REC;
593                     GO TO WRITE2.
594                 IF RT1FLG IS EQUAL TO 3;
595                     MOVE IN1-REC TO OT3-REC;
596                     GO TO WRITE3.
597                 IF ECHFLG1 IS NOT EQUAL TO ZERO GO TO READ2.
598                 MOVE IN1-REC TO OT1-REC.
599                 GO TO WRITE1.
600             PRTIN1.
601                 MOVE IN1 TO HDR2FIL.
602                 MOVE HDR2 TO PRT-REC.
603                 WRITE PRT-REC.
604                 MOVE SPACES TO PRT-REC.
605                 MOVE IN1-REC TO PRT-REC.
606             CHK9IPT1.
607                 WRITE PRT-REC.
608                 IF PRT-STAT = "91" GO TO CHK9IPT1.
609             WRITE1.
610                 PERFORM WR11 THRU WEXIT1.
611                 GO TO READ2.
612             WRT1.
613                 WRITE OUTCOM1-REC.
614                 IF OT1-STAT = "00";
615                     PERFORM WRT10R;
616                     GO TO WEXIT1.
617                 IF OT1-STAT = "91" GO TO WRT1.
618                 MOVE OT1-STAT TO WRERSTAT.
619                 MOVE OT1 TO WRERFIL.
620                 DISPLAY WPERMSG.
621                 ADD 1 TO ERSUM1OT.
622                 IF ERSUM1OT NOT LESS THAN 4 GO TO CLOT1.
623             WEXIT1.
624                 EXIT.
625             WRT10R.
626                 MOVE ZERO TO ERSUM1OT.
627                 PERFORM PRTOT1 THRU CHK9IP01.
628             PRTOT1.
629                 MOVE OT1 TO HDR3FIL.
630                 MOVE HDR3 TO PRT-REC.
631                 WRITE PRT-REC.
632                 MOVE SPACES TO PRT-REC.
633                 MOVE OT1-REC TO PRT-REC.
634             CHK9IP01.
635                 WRITE PRT-REC.
636                 IF PRT-STAT = "91" GO TO CHK9IP01.
637             GOODR2.
638                 MOVE ZERO TO ERSUM2IN.
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

```
GCOS6 MOD400-S100-12/01/1413    COBOL   0200
SOURCE  PROGRAM

639             PERFORM PRTIN2 THRU CHK9IPI2.
640             MOVE IN2-REC TO INSPECTI.
641             IF CMDIND IS EQUAL TO STAR GO TO WATCMD.
642             IF INCMD IS EQUAL TO COMDNM GO TO PCMD2.
643             IF RT2FLG IS EQUAL TO 1;
644                 MOVE IN2-REC TO OT1-REC;
645                 GO TO WRITE1.
646             IF RT2FLG IS EQUAL TO 3;
647                 MOVE IN2-REC TO OT3-REC;
648                 GO TO WRITE3.
649             IF ECHFLG2 IS NOT EQUAL TO ZERO GO TO READ3.
650             MOVE IN2-REC TO OT2-REC.
651             GO TO WRITE2.
652         PRTIN2.
653             MOVE IN2 TO HDR2FIL.
654             MOVE HDR2 TO PRT-REC.
655             WRITE PRT-REC.
656             MOVE SPACES TO PRT-REC.
657             MOVE IN2-REC TO PRT-REC.
658         CHK9IPI2.
659             WRITE PRT-REC.
660             IF PRT-STAT = "91" GO TO CHK9IPI2.
661         WRITE2.
662             PERFORM WRT2 THRU WEXIT2.
663             GO TO READ3.
664         WRT2.
665             WRITE OUTCOM2-REC.
666             IF OT2-STAT = "00";
667                 PERFORM WRT2OK;
668                 GO TO WEXIT2.
669             IF OT2-STAT = "91" GO TO WRT2.
670             MOVE OT2-STAT TO WRERSTAT.
671             MOVE OT2 TO WRERFIL.
672             DISPLAY WRERMSG.
673             ADD 1 TO ERSUM2OT.
674             IF ERSUM2OT NOT LESS THAN 4 GO TO CLOT2.
675         WEXIT2.
676             EXIT.
677         WRT2OK.
678             MOVE ZERO TO ERSUM2OT.
679             PERFORM PRTOT2 THRU CHK9IP02.
680         PRTOT2.
681             MOVE OT2 TO HDR3FIL.
682             MOVE HDR3 TO PRT-REC.
683             WRITE PRT-REC.
684             MOVE SPACES TO PRT-REC.
685             MOVE OT2-REC TO PRT-REC.
686         CHK9IP02.
687             WRITE PRT-REC.
688             IF PRT-STAT = "91" GO TO CHK9IP02.
689         GOODR3.
690             MOVE ZERO TO ERSUM3IN.
691             PERFORM PRTIN3 THRU CHK9IPI3.
692             MOVE IN3-REC TO INSPECTI.
693             IF CMDIND IS EQUAL TO STAR GO TO WATCMD.
694             IF INCMD IS EQUAL TO COMDNM GO TO PCMD2.
695             IF RT3FLG IS EQUAL TO 1;
696                 MOVE IN3-REC TO OT1-REC;
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

```
GCOS6 MOD400-S100-12/01/1413    COBOL   0200
SOURCE  PROGRAM

697                 GO TO WRITE1.
698             IF RT3FLG IS EQUAL TO 2;
699                 MOVE IN3-REC TO OT2-REC;
700                 GO TO WRITE2.
701             IF ECHFLG3 IS NOT EQUAL TO ZERO GO TO READ1.
702             MOVE IN3-REC TO OT3-REC.
703             GO TO WRITE3.
704         PRTIN3.
705             MOVE IN3 TO HDR2FIL.
706             MOVE HDR2 TO PRT-REC.
707             WRITE PRT-REC.
708             MOVE SPACES TO PRT-REC.
709             MOVE IN3-REC TO PRT-REC.
710         CHK9IPT3.
711             WRITE PRT-REC.
712             IF PRT-STAT = "9I" GO TO CHK9IPT3.
713         WRITE3.
714             PERFORM WRT3 THRU WEXIT3.
715             GO TO READ1.
716         WRT3.
717             WRITE OUTCOM3-REC.
718             IF OT3-STAT = "00";
719                 PERFORM WRT3OK;
720                 GO TO WEXIT3.
721             IF OT3-STAT = "9I" GO TO WRT3.
722             MOVE OT3-STAT TO WRERSTAT.
723             MOVE OT3 TO WRERFIL.
724             DISPLAY WRERMSG.
725             ADD 1 TO ERSUM3OT.
726             IF ERSUM3OT NOT LESS THAN 4 GO TO CLOT3.
727         WEXIT3.
728             EXIT.
729         WRT3OK.
730             MOVE ZERO TO ERSUM3OT.
731             PERFORM PRTOT3 THRU CHK9IP03.
732         PRTOT3.
733             MOVE OT3 TO HDR3FIL.
734             MOVE HDR3 TO PRT-REC.
735             WRITE PRT-REC.
736             MOVE SPACES TO PRT-REC.
737             MOVE OT3-REC TO PRT-REC.
738         CHK9IP03.
739             WRITE PRT-REC.
740             IF PRT-STAT = "9I" GO TO CHK9IP03.
741         WATCMD.
742             IF CMDTYP IS EQUAL TO SEND-DATA OR CMDTYP
743                 IS EQUAL TO SEND-TEXT GO TO GEN-DATA.
744             GO TO CERR1.
745         GEN-DATA.
746             IF LENNUM IS NOT NUMERIC GO TO CERR2.
747             IF NUMSENDS IS NOT NUMERIC GO TO CERR2.
748             IF NUMSENDS NOT GREATER THAN ZERO GO TO CERR2.
749             IF LENNUM IS LESS THAN 1 OR LENNUM IS
750                 GREATER THAN 3 GO TO CERR1.
751             MOVE ZERO TO MSGCOUNT.
752             MOVE NUMSENDS TO SDCOUNT.
753             IF CMDTYP IS EQUAL TO SEND-TEXT;
754                 GO TO ST1 ST2 ST3 DEPENDING ON LENNUM.
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

```
 GCOS6 MOD400-S100-12/01/1413     COBOL    0200
SOURCE  PROGRAM

 755              IF LFNNUM IS EQUAL TO "01" AND SD1COUNT IS EQUAL TO ZERO;
 756                  MOVE 1 TO SD1-FLG;
 757                  MOVE SDCOUNT TO SD1COUNT;
 758                  MOVE MSGCOUNT TO SD1MSG;
 759                  GO TO READ1.
 760              IF LFNNUM IS EQUAL TO "02" AND SD2COUNT IS EQUAL TO ZERO;
 761                  MOVE 1 TO SD2-FLG;
 762                  MOVE SDCOUNT TO SD2COUNT;
 763                  MOVE MSGCOUNT TO SD2MSG;
 764                  GO TO READ1.
 765              IF LFNNUM IS EQUAL TO "03" AND SD3COUNT IS EQUAL TO ZERO;
 766                  MOVE 1 TO SD3-FLG;
 767                  MOVE SDCOUNT TO SD3COUNT;
 768                  MOVE MSGCOUNT TO SD3MSG;
 769                  GO TO READ1.
 770          CERR1.
 771              DISPLAY BADCMD.
 772              GO TO READ1.
 773          CERR2.
 774              DISPLAY BADCMD.
 775              GO TO READ1.
 776          SD1.
 777              ADD 1 TO SD1MSG.
 778              IF SD1MSG = SD1COUNT;
 779                  MOVE ZERO TO SD1-FLG;
 780                  MOVE ZERO TO SD1COUNT.
 781              MOVE SD1MSG TO MSGNUM.
 782              MOVE DATAMSG TO OT1-REC.
 783              PERFORM WRT1 THRU WEXIT1.
 784          SD2.
 785              ADD 1 TO SD2MSG.
 786              IF SD2MSG = SD2COUNT;
 787                  MOVE ZERO TO SD2-FLG;
 788                  MOVE ZERO TO SD2COUNT.
 789              MOVE SD2MSG TO MSGNUM.
 790              MOVE DATAMSG TO OT2-REC.
 791              PERFORM WRT2 THRU WEXIT2.
 792          SD3.
 793              ADD 1 TO SD3MSG.
 794              IF SD3MSG = SD3COUNT;
 795                  MOVE ZERO TO SD3-FLG;
 796                  MOVE ZERO TO SD3COUNT.
 797              MOVE SD3MSG TO MSGNUM.
 798              MOVE DATAMSG TO OT3-REC.
 799              PERFORM WRT3 THRU WEXIT3.
 800          INCRMSG.
 801              ADD 1 TO MSGCOUNT.
 802              MOVE MSGCOUNT TO MSGNUM.
 803          ST1.
 804              MOVE TEXT-MSG TO OT1-REC.
 805              PERFORM WRT1 THRU WEXIT1 SDCOUNT TIMES.
 806              GO TO READ2.
 807          ST2.
 808              MOVE TEXT-MSG TO OT2-REC.
 809              PERFORM WRT2 THRU WEXIT2 SDCOUNT TIMES.
 810              GO TO READ3.
 811          ST3.
 812              MOVE TEXT-MSG TO OT3-REC.
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

```
 GCOS6 M00400-S100-12/01/1413    COBOL   0200
SOURCE  PROGRAM

 813              PERFORM #RT3 THRU #EXIT3 SDCOUNT TIMES.
 814              GO TO READ1.
 815          DONEIT.
 816              DISPLAY STOPCOB.
 817              STOP RUN.
 818          END COBOL



FILE   MAP
LINE   LFN  IFN              NAME                    RECORD   SIZE (DEC.)

  11   03   0C=MSD           COM1IN                  023A     80
  16   04   0D=MSD           COM1OT                  0262     80
  21   05   0E=MSD           COM2IN                  02MA     80
  26   06   0F=MSD           COM2OT                  02H2     80
  31   07   0G=MSD           COM3IN                  02DA     80
  36   08   0H=MSD           COM3OT                  0302     80
  41   01   0A=PRINTER       PRINTFILE               032H     120
```

Figure B-10 (cont). COBOL TTY or VIP Application Example

ERROR MESSAGES IN COBOL EXAMPLE

When appropriate, the example program displays error messages in the formats:

```
{OPEN }                 {COM1IN}
{CLOSE}                 {COM1OT}
{READ } ERROR FILE      {COM2IN}   zz - FILE STATUS
{WRITE}                 {COM2OT}
                        {COM3IN}
                        {COM3OT}
```

Program actions that would occur with these messages are:

OPEN or CLOSE message:

Returns control to the operator terminal.

READ or WRITE message:

Tries the I/O operation four times; then closes the file and returns control to the operator terminal.

STATUS CODES IN COBOL EXAMPLE

The program CARCOM includes checks that verify operation of COBOL error returns and information status returns. The check codes are:

91 - For a read operation, indicates there is no data. For a write operation, indicates that the device is busy.

95 - Record length error.

EXECUTION OF COBOL TTY or VIP PROGRAM EXAMPLE

When the program begins to execute, the operator terminal displays the message:

TYPE COMMANDS, THEN GO.

At least two files on the same device must be open to proceed to the next level of command input. At this level, the program displays the message:

COMMANDS?

You may then enter commands to: (1) open files, (2) close files, (3) route (message switch), (4) activate the read/write loop, or (5) stop.

NOTE

Activating the read/write loop deactivates command input from the console and causes the application to check open terminals for input.

To return to command level, type COMND from an active terminal.

A typein from a remote terminal is echoed back to that terminal and displayed on the second terminal.

COBOL BSC APPLICATION EXAMPLE

The source program listing in Figure B-11 is an example of a COBOL communications program to test BSC file transmission by:

1. Generating records
2. Transmitting the records over one communication line
3. Reading them back over another communication line for comparison

The program name is BSCTST. When executed, it displays the following error messages, as appropriate:

Error format 1:

```
BSC TEST FILE- INPUT     PROBLEM- OPEN     STATUS - zz
               OUTPUT             CLOSE
                                  READ
                                  WRITE
```

zz=9I - Device busy
zz=00 - Program may read or write

Program action: Issues reads and writes four times; then the file is closed and the program terminated.

Error format 2:

BSC - TEST - NO MATCH RECORD nnnn

Program action: Reading application does not receive the expected record; records out of sequence or garbled.

File is closed and the program terminated.

```
#GCOS6 MDU400-S120-09/11/1535    COHOL1  0210    -L0
SOURCE  PROGRAM

    1          IDENTIFICATION DIVISION.
    2          PROGRAM-ID.  BSCTST.
    3         * THIS IS A PROGRAM WHICH TESTS BSC FILE TRANSMISSION -
    4         * IT DOES SO BY GENERATING RECORDS , SENDING THEM OUT
    5         * AND BRINGING THEM BACK IN FOR COMPARISON
    6         * FOR A MORE DETAILED DESCRIPTION REFER TO THE GCOS 6.1
    7         * TEST SPECIFICATION FOR COBOL COMMUNICATIONS
    8          ENVIRONMENT DIVISION.
    9          CONFIGURATION SECTION.
   10          SOURCE-COMPUTER. HIS-SERIES-60   LEVEL-6.
   11          OBJECT-COMPUTER. HIS-SERIES-60 LEVEL-6.
   12         *
   13          INPUT-OUTPUT SECTION.
   14          FILE-CONTROL.
   15         *
   16               SELECT T-OUTPUT
   17               ASSIGN TO 01,
   18               ORGANIZATION IS SEQUENTIAL WITH VLR,
   19               ACCESS IS SEQUENTIAL,
   20               FILE STATUS IS OUT-STAT.
   21               SELECT T-INPUT
   22               ASSIGN TO A0,
   23               ORGANIZATION IS SEQUENTIAL WITH VLR,
   24               ACCESS IS SEQUENTIAL,
   25               FILE STATUS IS IN-STAT.
   26         *
   27          DATA DIVISION.
   28         *
   29          FILE SECTION.
   30          FD  T-OUTPUT
   31               BLOCK CONTAINS 1 RECORDS,
   32               LABEL RECORDS ARE STANDARD.
   33          01  OUT-REC   PIC X(80).
   34         *
   35          FD  T-INPUT
   36               BLOCK CONTAINS 1 RECORDS,
   37               LABEL RECORDS ARE STANDARD.
   38          01  IN-REC   PIC X(80).
   39         *
   40          WORKING-STORAGE SECTION.
   41         *
   42          77  IN-STAT   PIC XX     VALUE SPACES.
   43          77  OUT-STAT  PIC XX           VALUE SPACES.
   44          77  MAX-LVL           PIC 9999          VALUE 1001.
   45          77  W-INPUT           PIC X(6)          VALUE "INPUT ".
   46          77  W-OUTPUT          PIC X(6)          VALUE "OUTPUT".
   47          77  W-OPEN            PIC X(5)          VALUE "OPEN ".
   48          77  W-CLOSE  PIC X(5)  VALUE "CLOSE".
   49          77  W-READ            PIC X(5)          VALUE "READ ".
   50          77  W-WRITE  PIC X(5)          VALUE "WRITE".
   51          01  TEST-REC.
   52               02  FILLER  PIC X(12)  VALUE "TEST RECORD ".
   53               02  IN-CNT  PIC 9999  VALUE ZERO.
   54               02  FILLER  PIC X(54)  VALUE SPACES.
   55               02  FILLER PIC X(10)   VALUE "**********".
   56          01  EOF-REC.
   57               02 FILLER   PIC X(3)   VALUE "EOF".
   58               02  FILLER  PIC X(77)    VALUE SPACES.
```

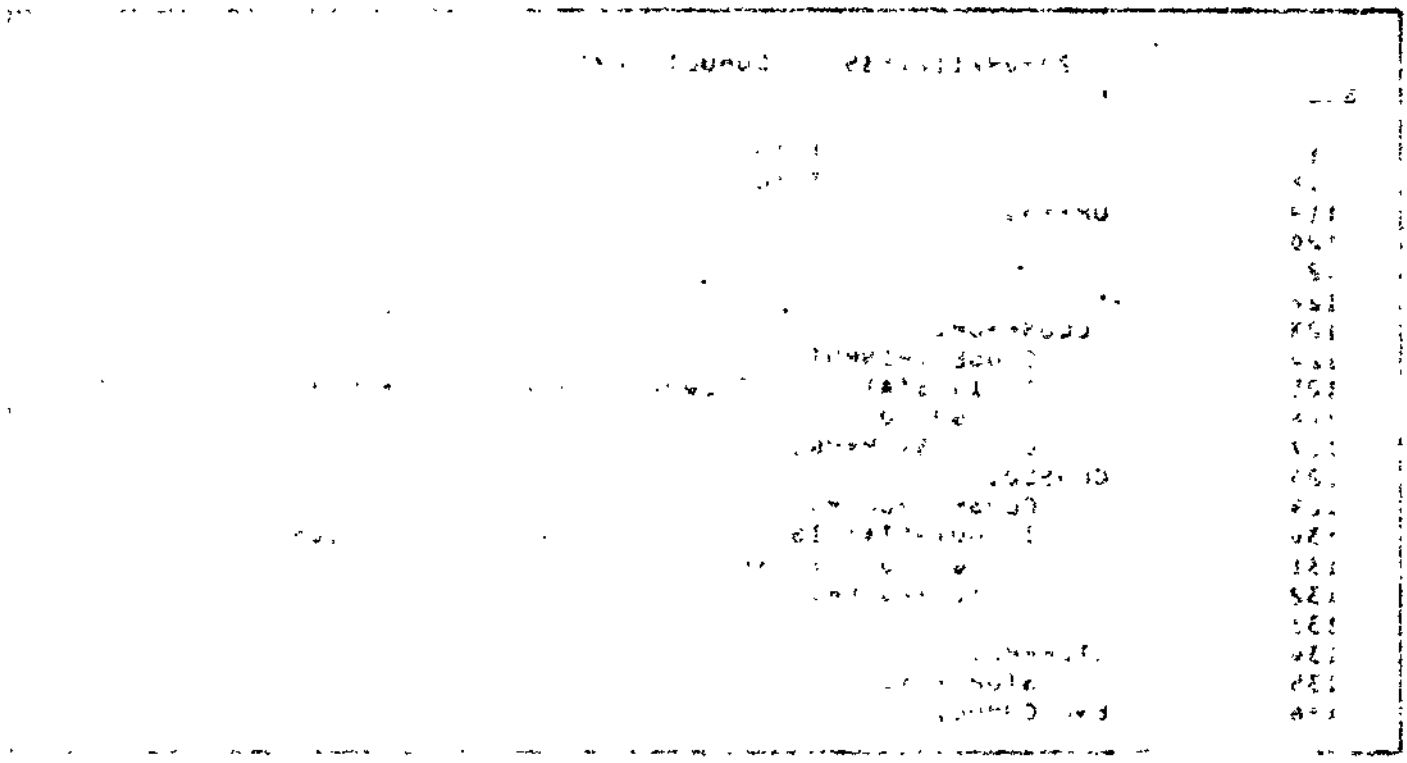
Figure B-11. COBOL BSC Application Example

```
=GCOS6 MU0400-3120-09/11/1535    COBOL2  0210    -LO                    MSGTST
SOURCE  PROGRAM

 59          01  ER-MSG1.
 60              02  FILLER   PIC X(16)   VALUE "BSC TEST- FILE- ".
 61              02  E-FILE   PIC X(6)    VALUE SPACES.
 62              02  FILLER   PIC X(10)   VALUE " PROBLEM- ".
 63              02 E-TYPE   PIC X(5)    VALUE SPACES.
 64              02  FILLER    PIC X(9)    VALUE " STATUS- ".
 65              02  E-STAT    PIC XX      VALUE SPACES.
 66          01  ER-MSG2.
 67              02  FILLER  PIC X(28)  VALUE "BSC TEST- NO MATCH, RECORD- ".
 68              02  BAD-REC   PIC 9(4)    VALUE ZEROES.
 69          01  EOJ-MSG.
 70              02 FILLER PIC X(9) VALUE "BSC EOJ- ".
 71              02 FINAL-CNT PIC 9(4) VALUE ZEROES.
 72              02 FILLER PIC X(20) VALUE " RECORDS TRANSMITTED".
 73          *
 74          PROCEDURE DIVISION.
 75          MSGREAD.
 76              MOVE ZEROES TO IN-CNT.
 77          OPEN-UP.
 78              OPEN INPUT I-INPUT.
 79              IF IN-STAT NOT EQUAL "00"; MOVE "OPEN" TO E-TYPE;
 80                  GO TO I-O-ERR.
 81              OPEN OUTPUT T-OUTPUT.
 82              IF OUT-STAT NOT EQUAL "00"; MOVE "OPEN" TO E-TYPE;
 83                  GO TO OUT-ERR.
 84          *
 85          *
 86          MASTER.
 87              ADD 1 TO IN-CNT.
 88              MOVE TEST-REC TO OUT-REC.
 89              IF IN-CNT = MAX-CNT; MOVE EOF-REC TO OUT-REC.
 90          READ1.
 91              READ I-INPUT AT END; MOVE IN-CNT TO FINAL-CNT;
 92                  DISPLAY EOJ-MSG; GO TO CLOSE-UP.
 93              IF I-O-STAT = "00"; GO TO COMPARE.
 94              IF IN-STAT = "91"; GO TO WRITE1.
 95              MOVE "READ" TO E-TYPE.
 96              GO TO IN-ERR.
 97          WRITE1.
 98              WRITE OUT-REC.
 99              IF OUT-STAT = "00"; GO TO COMPARE.
100              IF OUT-STAT = "91"; GO TO READ1.
101              MOVE "WRITE" TO E-TYPE.
102              GO TO OUT-ERR.
103          COMPARE.
104              IF I-O-STAT = "91"; GO TO READ1.
105              IF IN-REC IS EQUAL TO TEST-REC; GO TO MASTER.
106              IF OUT-REC = EOF-REC; GO TO CLOSE2.
107              MOVE IN-CNT TO BAD-REC.
108              DISPLAY ER-MSG2.
109              GO TO STOP-PG.
110          *
111          *
112           IN-ERR.
113              MOVE "INPUT" TO E-FILE.
114              MOVE IN-STAT TO E-STAT.
115              GO TO OP-MSG.
116          OUT-ERR.
```

Figure B-11 (cont). COBOL BSC Application Example

```
=GCOS6 MOD400-S120-09/11/1535    COMOL1  0210    =LO
SOURCE  PROGNAM

 117              MOVE A-OUTPUT TO E-FILE.
 118              MOVE OUT-STAT TO E-STAT.
 119           OP-MSG.
 120              DISPLAY ER-MSG1.
 121              GO TO STOP-PG.
 122           *
 123            CLOSE-UP.
 124              CLOSE I-INPUT.
 125              IF IN-STAT IS NOT EQUAL "00"; MOVE A-CLOSE TO E-TYPE;
 126                  GO TO IN-ERR.
 127              GO TO STOP-PG.
 128           CLOSE2.
 129              CLOSE I-OUTPUT.
 130              IF OUT-STAT IS NOT EQUAL "00"; MOVE A-CLOSE TO E-TYPE;
 131                  GO TO OUT-ERR.
 132              GO TO MASTER.
 133           *
 134            STOP-PG.
 135              STOP RUN.
 136            END COMOL.
```

Figure B-11 (cont). COBOL BSC Application Example



Appendix C
USING FORTRAN

This appendix describes procedures for using FORTRAN. The following information is provided:

- An explanation of the compile, link, and execute procedures for Advanced FORTRAN programs, including a sample program illustrating these steps.
- Programming tips for communications via FORTRAN, including a sample program.

INTRODUCTION

FORTRAN programs are compiled with the Advanced FORTRAN compiler, linked with standard Linker directives, and executed by optionally specifying the GET command and then specifying the program name.

FORTRAN COMPILE, LINK, AND EXECUTE PROCEDURES

To compile a FORTRAN program, invoke the Advanced FORTRAN (FORTRANA) compiler. Input to the FORTRAN compiler consists of a source program written in FORTRAN and optional control information.

Output is:

- A FORTRAN object (.O) unit
- A FORTRAN listing and diagnostics.

To link a compiled FORTRAN program, invoke the Linker. Input to the Linker consists of the relocatable object program. Output is a:

- Bound unit
- Link map.

Figure C-1 illustrates the compile and link operation, producing an executable module.

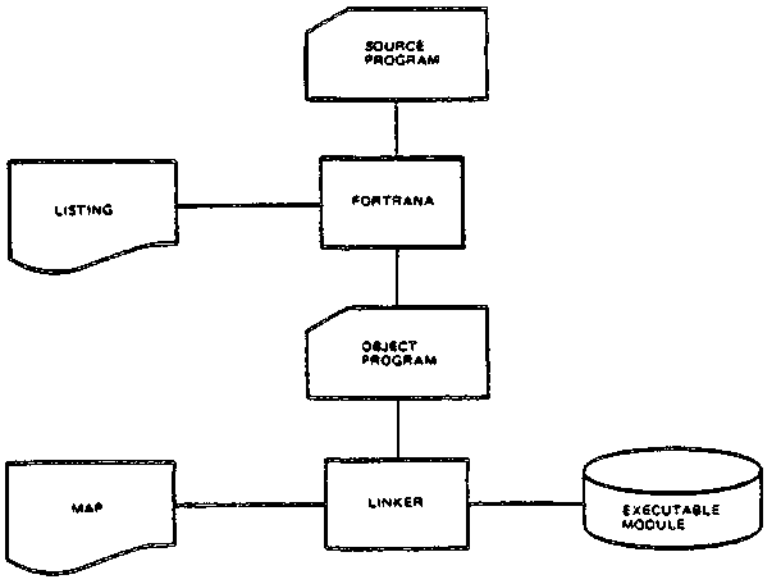


Figure C-1. Compiling and Linking a FORTRAN Program

Invoking the Advanced FORTRAN Compiler

The command used to invoke the Advanced FORTRAN compiler is:

```
FORTRANA path [ctl_args]
```

where:

path

The pathname of the source file to be compiled. The file must have the suffix .F; but this should be omitted from the path.

[ctl_args]

None or any number of control arguments. (See the Advanced FORTRAN Reference manual.)

For example, the source file might be TEST.F, shown in Figure C-2. To compile TEST.F, enter: FORTRANA TEST

The terminal dialog is:

```
RDY:
FORTRANA TEST                                  Invoke the compiler

FORTRANA 2.0 11/21/0712
000/000 W/E COUNT TEST                         There are no errors

RDY:
```

```
      PROGRAM TEST
      CHARACTER WORD*10, GOOD*10,DONE*4,YES*3
      DONE='DONE'
      YES='YES'
   50 WRITE (6,100)
  100 FORMAT('ΔWHAT WORD?')
      READ(5,200)WORD
  200 FORMAT(A10)
      IF(WORD.EQ.DONE) GOTO 600
      WRITE (6,300)WORD
  300 FORMAT('ΔYOUR WORDΔWAS ',A10,'ΔCORRECT?')
      READ(5,400)GOOD
  400 FORMAT(A4)
      IF(GOOD.EQ.YES) GOTO 50
      IF(GOOD.EQ.DONE) GOTO 600
      WRITE(6,500)
  500 FORMAT('ΔTOO BAD')
      GOTO 50
  600 STOP ' THAT IS ALL FOLKS'
      END
```

Figure C-2. FORTRAN Source Program TEST.F

Sample FORTRAN Listing Format

The compiler generates the following listings:

- Source and diagnostic
- Allocation
- Cross reference
- Object
- Called subprograms
- Statement label
- Line number listing
- Summary.

The source and diagnostic listing includes a sequential line number, source image, and interspersed diagnostics.

The allocation listing includes the name, class, type, size, and location of each variable. Allocation errors are listed.

The cross reference listing includes all labels and symbolic names in alphabetic order, and the line number of each reference. The line number is followed by an asterisk (*) if the reference is a possible modification. The line number is followed by a slash (/) if the reference appears on a specification statement or the reference appears as the definition of a label.

The object listing is produced only for the verification of object code by the developers and maintainers. It includes the location, symbolic opcode, and operands of each instruction. It may not represent valid input to the assembler but is adequate for analysis.

The called subprogram listing includes the name, number, and type of arguments, and class for each referenced subprogram, intrinsic function, and runtime routine. Subprogram units are classed as intrinsic, runtime, function, block data, and subroutine. The function class is used to describe only user-defined function subprograms.

The statement label listing includes the label, location, and type of use of all statement labels.

The line number listing includes each line number and its location.

The summary listing includes the number of errors, the number of warnings, the program size, and the data size.

STATEMENT ERROR DIAGNOSTICS

During compilation, statements which violates the syntactic or semantic rules of the language are recognized, and error messages are printed.

There are two levels of statement diagnostics: warnings and errors. Warning messages are issued for minor errors where the compiler can make an assumption as to what is to be done and compile the statement. Error messages are issued indicating that more serious source problems exist. In the case of errors, compilation proceeds as if the statement was never encountered. The statement label, if any, remains defined. If an error exists in an executable statement and that statement is executed, program execution terminates and you are notified that an attempt was made to execute a statement with a source error. The line number of the statement is displayed.

For each error or warning, one character of the statement is marked with a currency symbol ($), output directly beneath the erroneous character. For example:

```
ZATA = X + Y *
             $
```

The * character is marked as an error.

In the case of a syntax error, the marked character itself is unacceptable, as in the example above. In the case of a semantic error, an identifier or other construct is in error, the mark indicates the last character of the construct. For example, in the line

```
      COMMON ALPHA, BETA, ALPHA, GAMMA
                           $
```

the mark indicates that the identifier ALPHA is misused.

The compiler attempts all interpretations of statement type before discarding a statement. The marked position indicates the greatest amount of correct information found under the most logical assumption of statement type.

A comment specifying the reason for the failure is printed directly after the marked line. There may be more than one diagnostic per line and more than one diagnostic per mark. The marks are numbered from left to right with the number of a mark preceding the associated comment for the diagnostic. Each diagnostic is followed by a sequence of characters: *E*E*... or *W*W*W... indicating error or warning, respectively. The last diagnostic for a line is followed by the line number of the previous line with a diagnostic, if any.

SAMPLE LISTING

Figure C-3 shows the listing produced by compilation of TEST.F.

```
TEST          GCOS6 MOD400-12.1-09/24/1727 FORTRANA 2.0 11/21/07/12
        1981/05/05 0919:17.500     SLJC  PAGE 0001

     1          PROGRAM TEST
     2          CHARACTER WORD*10,GOOD*10,DONE*4,YES*3
     3          DONE='DONE'
     4          YES='YES'
     5       50 WRITE(6,100)
     6      100 FORMAT(1X,'WHAT WORD?')
     7          READ(5,200)WORD
     8      200 FORMAT(A10)
     9          IF(WORD.EQ.DONE) GOTO 600
    10          WRITE(6,300)WORD
    11      300 FORMAT(1X,'YOUR WORD WAS',1X,A10,1X,'correct?')
    12          READ(5,400)GOOD
    13      400 FORMAT(A4)
    14          IF(GOOD.EQ.YES) GOTO 500
    15          IF(GOOD.EQ.DONE) GOTO 600
    16          WRITE(6,500)
    17      500 FORMAT(1X,'TOO BAD')
    18          GOTO 50
    19      600 STOP 'THAT IS ALL FOLKS'
    20          END
```

```
TEST            GCOS6 MOD400-L2.1-09/24/1727 FORTRANA 1.0 11/21/0712
          1981/05/05 0919:17.500      SLIC  PAGE 0002

     SCALAR ALLOCATION

     LOCN     WORDS     CLASS     TYPE          NAME

     000C       2       SCALAR    CHARACTER*4   DONE
     000E       1+      SCALAR    CHARACTER*3   YES
     0010       5       SCALAR    CHARACTER*10  WORD
     0015       5       SCALAR    CHARACTER*10  GOOD

     PROGRAM COMPILED WITH FOLLOWING COMMAND LINE PARAMETERS:

TEST            GCOS6 MOD400-L2.1-09/24/1727 FORTRANA 1.0 11/21/0712
          1981/05/05 09/19:17.500     SLIC  PAGE 0003

     STATEMENT LABELS

     LABEL      LOCN      USE             LABEL     LOCN      USE          LABE
     \cL        LOCN      USE             LABEL     LOCN      USE

     50           0042                      100       0000      FORMAT       200
     \c         0009      FORMAT          600       00C9
     300          000C      FORMAT            400       0021      FORMAT       500
     \c         0023      FORMAT

     STATEMENT LOCATIONS

     LINE       LOCN            LINE      LOCN            LINE      LOCN
     \c         LINE      LOCN            LINE      LOCN

     3            0030            4         0039            5         0042
     \c         6         0052            7         0053
     8            006A            9         006A            10        0075
     \c         11        008C            12        008C
     13           00A3            14        00A3            15        00AE
     \c         16        0086            17        00C7
     18           00C7            19        00C9            20        00CF


FINAL SUMMARY

PROGRAM SIZE = Z'00CF' WORDS
DATA SIZE = Z'006D' WORDS
COMPILATION COMPLETE
0 WARNINGS
0 ERRORS
```

Figure C-3. Listing of TEST.F

```
TEST            GCOS6 MOD400-L2.1-09/24/1727 FORTRANA 1.0 11/21/0712
          1981/05/05 0919:17.500      SLIC  PAGE 0002

    SCALAR ALLOCATION

    LOCN     WORDS     CLASS     TYPE                NAME

    000C       2       SCALAR    CHARACTER*4         DONE
    000E       1+      SCALAR    CHARACTER*3         YES
    0010       5       SCALAR    CHARACTER*10        WORD
    0015       5       SCALAR    CHARACTER*10        GOOD

    PROGRAM COMPILED WITH FOLLOWING COMMAND LINE PARAMETERS:

TEST            GCOS6 MOD400-L2.1-09/24/1727 FORTRANA 1.0 11/21/0712
          1981/05/05 09/19:17.500     SLIC  PAGE 0003

    STATEMENT LABELS

    LABEL      LOCN      USE          LABEL     LOCN      USE          LABE
    \cL      LOCN      USE          LABEL     LOCN      USE

    50         0042                   100       0000      FORMAT       200
    \c       0009      FORMAT       600       00C9
    300        000C      FORMAT         400       0021      FORMAT       500
    \c       0023      FORMAT

    STATEMENT LOCATIONS

    LINE       LOCN           LINE      LOCN           LINE      LOCN
    \c       LINE      LOCN         LINE      LOCN

    3          0030           4         0039           5         0042
    \c       6         0052         7         0053
    8          006A           9         006A           10        0075
    \c       11        008C         12        008C
    13         00A3           14        00A3           15        00AE
    \c       16        0086         17        00C7
    18         00C7           19        00C9           20        00CF


FINAL SUMMARY

PROGRAM SIZE = Z'00CF' WORDS
DATA SIZE = Z'006D' WORDS
COMPILATION COMPLETE
0 WARNINGS
0 ERRORS
```

Figure C-3 (cont). Listing of TEST.F

Invoking the Linker

Once the source program is compiled, it can be linked. The command used to invoke the linker is:

```
LINKER progname  -PT [ctl_arg]
```

where:

progname

The bound-unit-pathname (simple, relative, or absolute) of the bound unit to be created (usually the program name), may be up to 62 characters in length.

-PT

Requests the Linker issue a prompt (L?) for input.

ctl_arg

Other valid control arguments for the Linker (see Section 5 above).

For example, to invoke the Linker for TEST (compiled above), enter:

```
LINKER TEST -PT
```

Figure C-4 shows the Linker dialog:

```
RDY:
LINKER TEST -PT -MAP     Invoke the Linker
LINKER 1982/06/18 0912:50.5
                         Linker responds with version and date
L?                       Linker prompts for input
LIB >LDD>ZF1RT           Give pathname to runtime library
L?                       Linker prompts for input
LINK TEST                Link the object program
L?                       Linker prompts for input
QUIT                     Quit the Linker
ROOT TEST                Linker responds with root name
LINK DONE
RDY:
```

Figure C-4. Linking TEST

Executing a Program

To execute the compiled and linked FORTRAN program, type in the program name. Figure C-5 illustrates a sample manual execution of TEST.

```
TEST
WHAT WORD?
FORTRANA
YOUR WORD WAS FORTRANA     CORRECT?
YES
WHAT WORD?
ADVANCED FORTRAN
YOUR WORD WAS ADVANCED F CORRECT?
NO
TOO BAD
WHAT WORD?
DONE
STOP   THAT IS ALL FOLKS
RDY:
TEST
WHAT WORD?
VERSION
YOUR WORD WAS VERSION     CORRECT?
YES
WHAT WORD?
DONE
STOP   THAT IS ALL FOLKS
RDY:
```

Figure C-5. Sample Execution of TEST

If data files are used, they may be made available to the program by using the GET command before typing the program name. When execution terminates, use the REMOVE command to release the data files. For more information on GET or REMOVE, see the Commands manual.

PROGRAMMING TIPS FOR USING COMMUNICATION DEVICES VIA FORTRAN

The File System interface provides the logical transfer of data between the FORTRAN program and an external device (terminal or another computer). The FORTRAN runtime routines issue File System macro calls according to the corresponding input/output statements in the compiled programs.

Interactive Devices and Files

The Executive defines communications devices and local TTY terminals for processing as "interactive." Interactive devices can be considered as sequential files in FORTRAN. Data is read or written with the same FORTRAN read/write interface as for a file on a noninteractive device.

ASSIGNING INTERACTIVE DEVICES AT EXECUTION

Before the compiled FORTRAN program can be executed, you may specify the actual interactive device for the specified file, using the command GET (get file). The logical file number (LFN) specified in the command must be the same as the unit specifier (u) that was included in the control information list (cilist) in the FORTRAN input/output statement READ, WRITE, or PRINT for that file. You may also use an OPEN statement with a FILE=argument to connect the actual device. See the Advanced FORTRAN Reference manual for descriptions of FORTRAN statements and the unit specifier. See the MOD 400 Commands manual for description of the GET and other commands.

CHANGING TERMINAL'S FILE CHARACTERISTICS

Using the Set Terminal Characteristics (STTY) command or $STTY macro call, you can reset the following terminal file characteristics: line length or record size, detabbing, and device type.

SYNCHRONOUS INPUT/OUTPUT

If the device is configured (see STTY directive) or modified (see STTY command) for synchronous I/O, then an input order to the device is only issued when the application issues a read, and output is only performed when the application issues a write. The application is placed in the wait state until the read or write operation is complete. Synchronous I/O is not useful for an application which processes more than one device since each read from or write to a device must be satisfied before the next device can be processed.

ASYNCHRONOUS INPUT

If the device is configured (see STTY directive) or modified (see STTY command) for asynchronous input, then the File System issues anticipatory reads into a system buffer. This is effectively double buffering, since the application can be processing one input record while the system is reading the next one. It also allows the application to process multiple devices efficiently, since it can test each device for input and thus not have to wait for input from one device before being able to process another device.

The FORTRAN subroutine Z1STIN allows the application program to check the status of the input communications device (file) before issuing a READ statement. Note that the device must have been configured or modified for asynchronous input in order for this check to be meaningful.

If the device is configured (see STTY directive) or modified (see STTY command) for asynchronous output, then when an application issues a write, the File System moves the application data to a system buffer, queues it for writing, and returns immediately to the application. This is effectively double buffering, since the application can be constructing one output record while the system is writing the previous one. It also allows the application to process multiple devices efficiently since it does not wait for output to one device to finish before being able to process another device. The application can test each device before performing the write operation, to see if the previous output is complete.

The FORTRAN subroutine Z1STOT allows the application program to check the status of the output communications device (file) before issuing a WRITE statement. Note that the device must be configured or modified for asynchronous output in order for this check to be meaningful.

FORTRAN File Status Check (ZFSTIN and ZFSTOT)

The FORTRAN OPEN statement must precede any other input/output statement to a file that is a communications device.

When the program issues an I/O request statement (a READ or WRITE), it waits until that request is completed.

The FORTRAN subroutines Z1STIN and Z1STOT, when called before an I/O request is issued, check the availability of the communications device (file), and can prevent the problem of program inactivation or program termination due to file or device unavailability.

The subroutine Z1STIN checks the status of the input file, Z1STOT checks the output file. Their use monitors the status of the files without loss of program control and prevents the imposition of file system waits.

A CALL statement to either subroutine should be issued before the application issues any I/O requests to ascertain (1) whether the file (device) is available, and (2) any device error status.

The subroutine Z1STIN or Z1STOT, when called, issues a request to the file system, which in turn (without waiting for any pending I/O request to be completed) returns status information about the file's availability. When the file is not busy, the File System will return status information about the previous I/O request.

CALL STATEMENT FOR Z1STIN or Z1STOT

The CALL statement for subroutine Z1STIN or Z1STOT is specified as:

```
CALL  {Z1STIN}  (lfn,arg)
      {Z1STOT}
```

lfn

The logical file number, in a GET command, that identifies the unit specifier (u) for the file to be checked.

arg

The symbolic integer variable into which the File System will return one of the following status values:

000_{10}

File is available (READ or WRITE can be issued). The last request, if a READ or WRITE, was successful.

512_{10}

Request rejected; undefined LFN was used, or the file system is not available.

516_{10}

File is busy (READ or WRITE in progress). If Z1STIN, then a READ is in progress and not yet complete. If Z1STOT, the previous WRITE is not yet complete.

519_{10}

File is not open; last request was not successful. Another READ or WRITE will result in an error return.

A call to Z1STIN or Z1STOT made to a noncommunications file always results in a 000 (not busy) status return. Such a call allows you to debug the application program by first using noncommunications files, then write the program so that it can use either communications or noncommunications files.

The FORTRAN subroutine Z1STIN, when called before issuing a READ request, checks for the availability of input. It prevents the loss of program control until data is available in a file system buffer. When Z1STIN indicates that the file is not busy, then a READ can be issued to move the data just read from the File System to the application program area.

The FORTRAN subroutine Z1STOT, when called before issuing a WRITE request, checks to see if previous output is complete and the terminal is free to accept more data. When Z1STOT indicates that the file is not busy, then a WRITE can be issued to move data from the application program area to a File System buffer and schedule it to be written to the terminal.

Z1STIN and Z1STOT Programming Examples

The following are examples of (1) coding that causes the program to stall when input from a terminal is not completed before a second READ is issued, and (2) a call to subroutine Z1STIN to check the file status before the second READ is issued. Note that in each case the first FORTRAN statement is OPEN.

Example 1:

```
      OPEN(UNIT=8)
      READ(8,100)IN
      READ(8,100)IN
100   FORMAT(I2)
```

Example 2:

```
      OPEN(UNIT=8)
      READ(8,200)IN
 50   CALL Z1STIN(8,ISTAT)
      IF(ISTAT .EQ. 0) GO TO 100
      IF(ISTAT .EQ. 512) GO TO 900
      IF(ISTAT .EQ. 519) GO TO 900
      GO TO 50
100   READ(8,200)IN
           .
           .
           .
200   FORMAT(I5)
900   WRITE(4,910)
910   FORMAT(' ERROR FOUND')
```

FORTRAN Application Example for TTY

The FORTRAN source program (program name FORCL4) listing shown in Figure C-6 is an example of a FORTRAN application program involving a TTY remote device.

The program processes eight message groups before terminating. It first issues four data messages to the remote terminal and to the operator terminal. It issues the write requests from alternate data buffers to ascertain the status of the interfaces among the File System, source program, and the communications subsystem. When the four initial message groups are complete, the program requests input data from the operator terminal.

After you enter a message, the operator terminal displays the message and an acknowledgement message. When the fourth message is received, the application program terminates.

Every input message, which is preceded by a blank or NUL character that is not displayed, may have up to 59 ASCII characters.

The system continually monitors the status register, displaying error condition codes or status messages on the operator terminal. For example, a condition indicating no data available (buffer busy) at the remote device, lasting more than 20 seconds, causes a status return code of 516. The program continues the read attempt since that status is not an error condition. The read statement is issued only after a status code 0 is returned to indicate that data is available (buffer not busy).

```
 1 C     FORTRAN COMMUNICATION PROGRAM - FORCL4
 2 C
 3 C        ILLUSTRATES USE OF Z1STIN AND Z1STOT
 4 C
 5 C        WRITES 4 MESSAGES TO THE OPERATOR'S TERMINAL (LFN 4)
 6 C        AND SEND TO A REMOTE DEVICE (IF. TTY) ON LFN 9 VIA MLCP
 7 C        FOLLOWED BY A READ OF 4 MESSAGES FROM THE SAME REMOTE
 8 C        DEVICE (IF. TTY) ON LFN 8. ALL MESSAGES ARE DISPLAYED
 9 C        ON THE OPERATOR'S CONSOLE, AND RECEIVED MESSAGES ARE
10 C        ACKNOWLEDGED ON THE REMOTE DEVICE
11 C        DEVICE STATUS IS REPORTED USING,
12 C             CALL 7FSTIN(I,J) FOR INPUT, AND
13 C             CALL ZFSTOF(I,J) FOR OUTPUT.
14 C
15        PROGRAM FORCI 4
16        CHARACTER *48 CW3,CW4
17        CHARACTER CR1(60),CR2(60)
18        DATA CW3/'THIS IS COMM. OUTPUT TO THE TTY - MESSAGE NUMBER'/
19 C
20        J = 0
21        N = 0
22        K = 9
23        CW4 = CW3
24        OPEN(UNIT=8)
25        OPEN(UNIT=9)
26        GO TO 20
27   15   K = 8
28 C
29 C      CHECK COMMUNICATION DEVICE STATUS
30 C      USING ZFSTIN OR ZFSTOT ROUTINE
31 C
32    20  N = N + 1
33    25  J = 0
34    30  IF(K.EQ.8)CALL ZFSTIN(K,ISTAT)
35        IF(K.FQ.9)CALL ZFSTOT(K,ISTAT)
36        IF(ISTAT .EQ. 0) GO TO (70,90,70,90,100,120,100,120),N
37        IF(ISTAT - 516)50,40,50
38    40  J = J + 1
39        IF(J .LT. 10000) GO TO 30
40    50  WRITE(4,60)N,ISTAT
41    60  FORMAT(1X,'STATUS RTN MESSAGE NO.',I2,' STATUS TYPE',I4)
42        IF(ISTAT .FQ. 516) GO TO 25
43        GO TO 140
```

Figure C-6. FORTRAN Application Example for TTY

```
44 C
45 C     OUTPUT MESSAGES TO REMOTE DEVICE (LFN 9)
46 C     4 MESSAGES ISSUED TO DEVICE AND LFN4
47 C     FROM ALTERNATING BUFFERS
48 C
49   70      WRITE(9,80)CW3,N
50   80      FORMAT(1X,A48,I2)
51           WRITE(4,80)CW3,N
52           GO TO 20
53   90      WRITE(9,80)CW4,N
54           WRITE(4,80)CW4,N
55           IF(N .EQ. 4) GO TO 15
56           GO TO 20
57 C
58 C     INPUT FROM REMOTE DEVICE (LFN 8)
59 C     4 MESSAGES ALLOWED
60 C
61 C          SPACE 1 CHARACTER AND TYPE UP TO 59 CHARACTERS
62 C          FOLLOWED BY A CARRIAGE RETURN
63 C          TYPE SECOND MESSAGE WHEN DEVICE   TYPES
64 C          "MESSAGE X RECD"
65 C
66  100      READ(R,110)CR1
67  110      FORMAT(1X,60A1)
68           WRITE(4,110)CR1
69  112      CALL Z1STOT(9,ISTAT)
70           IF(ISTAT .EQ. 0)GO TO 114
71           GO TO 112
72  114      WRITE(9,115)N
73  115      FORMAT(1X,'MESSAGE ',T2,' RECD')
74           IF(N .NE. 8)GO TO 20
75           GO TO 130
76  120      READ(8,110)CR2
77           WRITE(4,110)CR2
78  121      CALL Z1STOT(9,ISTAT)
79           IF(TSTAT .EQ. 0)GO TO 125
80           GO TO 121
81  125      WRITE(9,115)N
82           IF(N .NE. 8)GO TO 20
83 C
84 C     CLOSE UNITS AND EXIT
85 C
87  130 CALL Z1STOT(9,ISTAT)
87           IF(ISTAT .EQ. 0) GO TO 140
88           GO TO 130
89  140      CLOSE(UNIT=8)
90           CLOSE(UNIT=9)
91           STOP
92           END
  0      DIAGNOSTICS
```

Figure C-6 (cont). FORTRAN Application Example for TTY

Appendix D
USING BASIC

This appendix describes procedures for using BASIC. The following information is provided:

- An explanation of the compile, link, and execute procedures for BASIC programs.
- A sample program illustrating the compile, link, and execute procedures for BASIC programs.

INTRODUCTION

BASIC programs may be processed in two ways:

- Create and run interactively. These programs are not compiled or linked. They may be saved as source files, read into BASIC, and run from within BASIC.
- Create and compile under BASIC, then link and execute. These programs are created, normally tested and debugged, and compiled under BASIC. They may be saved as source files. Once compiled, an object unit exists. This object unit must be linked using the BASIC Linker EC, LNKBPRG. After successful linking, they are executed by specifying the program name.

INVOKING THE BASIC INTERPRETER/COMPILER

To create a BASIC program, invoke BASIC. Figure D-1 shows the dialog used to invoke BASIC and create and save source program PROG1.B.

```
BASIC                                   Invoke BASIC
BASIC2.0-01/09/1400C                    BASIC responds with version
                                           and date

READY                                   BASIC prompt
100 INPUT "WHAT WORD", WORD$            Create a source program
200 IF WORD$ = "DONE" GOTO 900
300 PRINT "YOUR WORD WAS "; WORD$
400 INPUT "CORRECT", GOOD$
500 IF GOOD$ = "YES" GOTO 100
600 IF GOOD$ = "DONE" GOTO 900
700 PRINT "TOO BAD"
800 GOTO 100
900 PRINT "THAT'S ALL FOLKS"
1000 END
SAVE PROG1                              Save the source program
READY
```

Figure D-1. BASIC Source Program PROG1.B

If you leave BASIC using the QUIT command, reenter BASIC using the BASIC command shown above and issue the OLD command to bring your BASIC program into memory:

```
OLD PROG1
```

If you have not left BASIC, skip this step. PROG1.B has been saved in the File System, and is also still in memory.

At this point, you can run PROG1 interactively to find and correct errors. This is done within BASIC.

EXECUTING BASIC INTERACTIVELY

You can execute a program within BASIC without compiling and linking. In this case, no object unit or executable module is produced.

In the example below, assume you have entered BASIC, retrieved PROG1.B, and are ready to execute PROG1 interactively. Figure D-2 shows the execution dialog:

```
RUNNH           Execute the current program interactively
WHAT WORD     ?  BASIC
YOUR WORD WAS BASIC
CORRECT       ?  YES
WHAT WORD     ?  COMPILER
YOUR WORD WAS COMPILER
CORRECT       ?  NO
TOO BAD
WHAT WORD     ?  DONE
THAT'S ALL FOLKS
END AT 1000
READY
```

Figure D-2. Interactive Execution of PROG1

The Interactive Execution looks identical to a compile and link execution. The only difference is the ready message. The interactive execution issues the BASIC prompt, READY, while a compile and link execution results in the system prompt, RDY:.

BASIC PROGRAMS

BASIC programs are created, debugged interactively, and compiled within BASIC. Programs with an interactive call can not be debugged interactively; they are linked using the supplied BASIC Linker EC, LNKBPRG. Input to the BASIC compiler consists of a source program written in BASIC and resident in memory. Output is:

- A BASIC object (.O) unit
- Diagnostic messages that appear at the terminal during compilation.

Input to the Linker consists of the relocatable object program. Output is:

- An executable module
- A link map.

Figure D-3 illustrates the compile and link operation, producing an executable module.

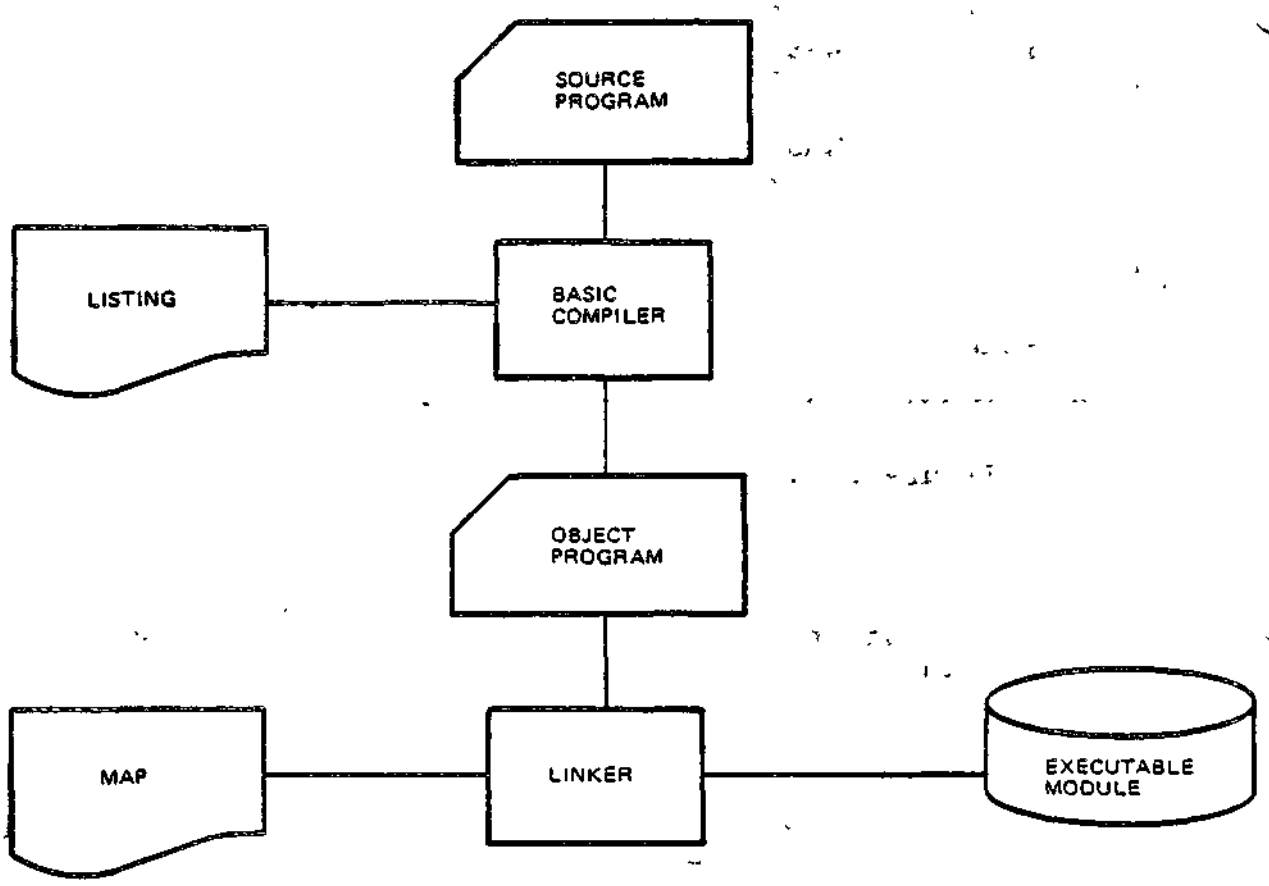


Figure D-3. Compiling and Linking a BASIC Program

Compiling a BASIC Program

After successful interactive execution, the program can be compiled. Figure D-4 shows the BASIC session continuing to compile the program, then quitting.

```
COMPILE PROG1    Compile
READY
QUIT             Leave BASIC
RDY:             System prompt
```

Figure D-4. Compiling PROG1 and Quitting BASIC

Programming Considerations

The following points are provided to aid programming in BASIC.

Making Procedure Calls

Object units produced by other language processors may be called from object units produced by the BASIC compiler. When making procedure calls, the following programming considerations should be reviewed.

1. The length of a string variable or element of a string array is not constant throughout a BASIC program but is, at any given point in the program, established by the value last given to the string by an assignment or read statement. A string variable or array element may have a zero length if its last usage was in a CLEAR statement or if the last assignment results in a null string.

2. The elements of a string array are not necessarily equal in length.

3. Because string variables and string array elements are not of constant length, they are not allocated to fixed positions in memory; i.e., the addresses of such entities are variable during the execution of a BASIC program. Elements of string arrays are not necessarily allocated in contiguous memory or in any specific order. For this reason, it is not possible to use an entire string array as an argument of a CALL statement.

Resequencing Line Numbers

For convenience in resequencing line numbers within a BASIC source program, a free-standing Resequence utility program is provided. All statements are renumbered using a constant increment; reference line numbers within the source program are adjusted accordingly. The format of the Resequence utility is:

```
RESEQ pathname [options]
```

where:

pathname

The pathname of the BASIC source file to be resequenced.

options

One or more of the following:

-START ssss

Starts resequencing with the specified line number ssss; otherwise, the initial line number is set to 10

-INC iiii

Uses a resequencing increment of iiii; otherwise, the increment is 10

-RANGE L1 L2

Resequences a range (L1 through L2) of current line numbers; otherwise, the entire program is resequenced

-NEW path

Gives this pathname to the resequenced program and leaves the original source unaltered. Otherwise, the original source is replaced with the resequenced version

-NO_LIST

-NL

Does not produce a listing of the resequenced program

-COUT path

Directs any output to the specified pathname

-SIZE nn

-SZ nn

Uses this size memory for line number table, default is 4K

-LE lists errors only

Controlling Screen Processing

GCOS 6 screen management utilities (forms processing) may be used from BASIC compiled programs. When such programs are terminated by execution of a STOP or END statement, the usual BASIC End-of-Program message is displayed on the screen. To avoid this, terminate your programs by chaining to the program ZBENDT. For information on forms processing, see the Display Formatting and Control manual.

Controlling Common Areas

The BASIC generated common area, B$COMM, should not be shared with other programming languages if the COMMON contains string variables or string arrays.

Linking a BASIC Program

An EC (LNKBPRG.EC) is provided as a model to aid in linking programs compiled by BASIC. In most cases, you can use it as supplied, but for some applications, specialization may be required. The format is:

```
EC >SYSLIB2>LNKBPRG pathname [s]
```

where:

pathname - The name of the program to be linked. The bound unit produced also has this name.

s - Optional; specifies that BASIC run-time procedures in the bound unit BRTNUC configured into the system, are to be used at execution time.

The LINK EC assumes that the run-time library directories ZBRT (containing the real run-time routines) and ZBRTS (containing the dummy run-time modules necessary to prevent link errors when using BRTNUC) exist on the directory LDD under the root volume directory.

If you use the Linker directly to link your program, be aware that you may have undetected symbol resolution errors during the link which appear during execution of the program.

The following command is used to invoke the Linker for previously compiled BASIC program BPROG:

```
EC >SYSLIB2>LNKBPRG BPROG
```

A link map is produced. If you only want to link and execute your program, you may select to put the link map aside. The link map and its extended uses are described in Section 6.

Executing a BASIC Program

To execute the compiled and linked BASIC program, type in the program name. Figure D-5 illustrates a sample manual execution of BPROG.

```
BPROG
WHAT WORD    ?    BASIC
YOUR WORD WAS BASIC
CORRECT      ?    YES
WHAT WORD    ?    COMPILER
YOUR WORD WAS COMPILER
CORRECT      ?    NO
TOO BAD
WHAT WORD    ?    DONE
THAT'S ALL FOLKS
END AT 1000
RDY:
```

Figure D-5. Execution of BPROG

Appendix E
USING THE MULTI-USER DEBUGGER (SYMBOLIC MODE)

This appendix guides you through a sample session using the debugger in symbolic mode. You can follow along at a terminal step-by-step. Section 7 of this manual provides a detailed description of the debugger.

The original source program could be written in either Advanced FORTRAN or Advanced COBOL. Before you continue, create an Advanced COBOL or Advanced FORTRAN program. For more information on creating programs, refer to Section 4 on the screen editor or Section 5 on the line editor and Appendix B (COBOL) or Appendix C (FORTRAN) describing the compile, link, and execute procedures.

COMPILING A PROGRAM FOR USE WITH THE DEBUGGER

Compile your program with either the Advanced FORTRAN or Advanced COBOL compiler using the -SYMBOL argument. The -SYMBOL argument creates a symbol table file, name.Z, where name is the name of your source program. The compiler commands are described next.

FORMAT:

```
{COBOLA   } name   -SYMBOL [ctl_arg]
{FORTRANA }
```

ARGUMENTS:

name	Name of your source program.
ctl_arg	Other control arguments you wish to use. See the Commands manual for the complete command description.

If necessary, correct any compilation errors and recompile. Procede with the next step when you have an error-free compilation.

Sample Compilation Dialogs

To compile the Advanced FORTRAN program NFTYPM, the compilation dialog might be:

FORTRANA NFTYPM -SYMBOL	Invoke the Advanced FORTRAN compiler specifying that object code be listed, a special symbol table be created, and 4K of memory be used.
FORTRANA 2.0 07/09/1302	The compiler is invoked.
000/000 W/E COUNT NFTYPM	There are no warnings or errors.
RDY:	Control returns to command level.

To compile the Advanced COBOL program COMPTV, the compilation dialog might be:

COBOLA COMPTV -SYMBOL	Invoke the Advanced COBOL compiler specifying that a special symbol table be created.
COBOLA 3.4 07/15/0813	The compiler is invoked.
NO FATAL ERRORS OR WARNINGS IN COMPTV	There are no errors.
RDY:	Control returns to command level.

LINKING AN OBJECT UNIT WITH THE DEBUGGER

Link the object unit resulting from successful compilation using the -SYMBOL option. -SYMBOL creates a separate link file named buname.V, where buname is the name of the bound unit created by the link.

The Linker commands description follows:

FORMAT:

```
LINK   buname -SYMBOL [ctl_arg]
```

ARGUMENTS:

buname	Name of the bound unit to be created.
ctl_arg	Other control arguments. See the Commands manual for the complete command description.

For more information on the Linker, see Section 6. After you have linked successfully, go on to the next step.

Sample Linker Dialogs

To link object unit NFTYPM, compiled by the Advanced FORTRAN compiler above, the Linker dialog might be:

```
LINKER NFTYPM -SYMBOL -PT          Invoke the Linker specifying creation of a special debug link map and a prompt for the user.
LINKER 1982/06/18 0912:50.5        The Linker is invoked.
L?                                 You are prompted for a directive.
LIB >LDD>ZF1RT                     Include the standard Advanced FORTRAN runtime library, ZF1RT.
L?
LINK NFTYPM                        Link the object unit.
L?
QUIT                               Terminate the Linker.
ROOT NFTYPM                        The bound unit is created.
LINK DONE                          The Linker is finished.
RDY:                               Control returns to command level.
```

To link object unit COMPTV, compiled by the Advanced COBOL compiler above, the Linker dialog might be:

```
LINKER COMPTV -SYMBOL -PT                Invoke the Linker, specifying
                                         a special debug link map be
                                         created and a prompt be given.
LINKER 1982/06/18 0912:50.5              The Linker is invoked.
L?                                       You are prompted for a
                                         directive.
LIB >LDD>ZCART                           Include the standard Advanced
                                         COBOL runtime library, ZCART.

L?
LINK COMPTV                              Link the object unit.

L?
QUIT                                     Terminate the Linker.
ROOT COMPTV                              The bound unit is created.
LINK DONE                                The Linker is finished.
RDY:                                     Control returns to command
                                         level.
```

INVOKING THE DEBUGGER

To initiate the debugger and set breakpoints in the program to be debugged, issue the Debug command.

FORMAT:

```
DEBUG buname
```

ARGUMENTS:

buname

Name of the bound unit to be debugged.

The debugger responds with a prompt (the greater-than sign, >). Set breakpoints using the At directive. You can set up to 32 breakpoints. They will be numbered from 31 to 0 in descending order. During initialization you can also list the breakpoints with the List directive and clear erroneously set breakpoints with the Clear directive.

When you are satisfied with the breakpoints you have set, issue the Sleep (SP) directive to temporarily suspend the debugger and return to the command processor. Now you can begin execution of your program with the debugger.

Sample Initialization Dialog

The debugger initialization dialog for NFTYPM might look like this:

```
DEBUG NFTYPM                    Invoke the debugger for the bound
                                unit named NFTYPM.
>                               The debugger responds with its
                                prompt, the greater-than sign.
AT 12                           Set a breakpoint at line 12.
BP 31 SET                       The breakpoint id is given in
                                response.
>
LIST *                          List all breakpoints.
BP 31 BU=NFTYPM CU=NFTYPM LINENO=12
                                The debugger responds with a list of
                                all current breakpoints by ID.
>
AT 19                           Set a breakpoint at line 19.
BP 30 SET
>
AT 24                           Set a breakpoint at line 24.
BP 29 SET
>
CL 29                           Clear breakpoint 29, at line 24.
BP 29 DELETED                   The debugger acknowledges the
                                deletion.
>
AT 25                           Set a breakpoint at line 25.
BP 29 SET
>
LIST *
BP 31 BU=NFTYPM CU=NFTYPM LINENO=12
BP 30 BU=NFTYPM CU=NFTYPM LINENO=19
BP 29 BU=NFTYPM CU=NFTYPM LINENO=25
>
SP                              Temporarily suspend the debugger and
                                return to the command processor.
```

Debugging Multiple Bound Units

The debugger can be invoked for one bound unit. After that, the debugger must be turned off before other bound units can be debugged. To turn off the debugger, issue the command sequence:

```
DEBUG
QT
```

Then another bound unit can be debugged by re-invoking the debugger with DEBUG new_bound_unit_name.

EXECUTING YOUR PROGRAM WITH THE DEBUGGER

Execute your program by entering the bound unit name. Execution is suspended at the first breakpoint set during debugger initialization. The At directive allows you to specify a list of debugger directives (a request list) to be executed when the specified breakpoint is reached. If you included a request list when you set the breakpoint, the request list executes. If not, the debugger enters interactive mode and issues the greater-than prompt (>). You can then enter any valid debugger directives to check out the program.

SAMPLE EXECUTION DIALOG

Assume you initialized the debugger using the previous example, your execution then might look like this:

NFTYPM	Begin execution of the program NFTYPM.
*BRKPT 31 AT LINE 12	Execution stops at line 12, the first breakpoint set.
>	The debugger prompts for a directive.
DUMP YY	Print the current value of variable YY.
YY 81	The value is printed.
>	
CHANGE YY = %3834	Change the value of YY to ASCII 84 (hexadecimal 3834).
>	
DUMP YY	Check that the change was made.
YY 84	
>	
GO	Continue execution. If you found an error, you could issue the Quit (QT) directive to terminate the debugger.
*BRKPT 30 AT LINE 19	Execution resumes until the next breakpoint is encountered.
>	The debugger prompts for a directive.

```
IF YY  = 81
```

Check the value of YY and terminate if YY is not equal to 81.
YY was not equal to 81, so you return to the Command Processor, which issues the standard ready message.

```
TRACE

>

GO

*TR AT LINE 52

>

    RDY:
```

Appendix F
USING EXECUTION COMMAND (EC) FILES

The EC file is a user-created file that contains a sequence of frequently used commands. For example, an EC file may contain the commands used to compile, link, and execute a program.

EC FILE ADVANTAGES

The EC file provides the following advantages:

- Less time spent in interactive dialog
- Fewer typing errors
- Less confusion over what to do next
- More control over the processing environment.

EC file capabilities and guidelines for generating and using EC files in the applications programming environment are presented below.

EC FILE FEATURES

You can create an EC file using the screen editor or the line editor. Each line within the file contains a command (or series of commands) that instruct the command processor to perform processing according to the arguments supplied in the command line.

EC control directives (described later) are written into an EC file to maintain control over file execution.

Active functions (described later) can be included in the command line to specify values in the argument string.

EC file names can be up to 12 characters long. The name must include the suffix .EC (e.g., FILEA.EC).

EXECUTING AN EC FILE

Enter the Execute command (EC) and the name of your EC file (without the .EC suffix) to process the EC file.

DEVELOPING A SIMPLE EC FILE

Working in the application development environment, assume that you are writing a FORTRAN program called AREA. The commands that you will use most often are:

ED (to invoke the Editor)
FORTRANA (to invoke the FORTRAN compiler)
LINKER (to invoke the Linker)
AREA (to execute your program)

A simple EC file that can take you through the above program development stages into program execution is shown in Figure F-1.

```
ED -PT
FORTRANA AREA -LE
LINKER AREA -IN LNKDR
DPRINT AREA.M
AREA
```

Figure F-1. Sample EC File: Command-Only

To perform the program development stages shown above, you create the source code using the Editor. When you enter the Editor directive "Q", the command processor automatically reads the next command in the file; i.e., FORTRANA AREA, and begins compiling your program.

When compilation is completed, the command processor invokes the Linker that reads its directives from a file called LNKDR. Upon a successful link, the program AREA executes.

The EC file used above contains only commands and is the simplest form of EC file. More control over your applications can be gained by using active strings within your EC file.

ACTIVE STRINGS

An active string is part of a command line that is evaluated during command interpretation. Any MOD 400 command or any active function (described later) can be used in an active string. The command processor substitutes the resulting value(s) for the active string(s) in the command line. The value(s) is then interpreted as the control argument(s) for that command. For example, if you are working under a directory other than your home directory and you want to return to your home directory, you could issue (or have included in your EC file) the command

```
CP [LHD] > ** ==
```

This command causes all files in the user's home directory to be copied to the current working directory.

Note that all active strings are bound by left and right brackets.

Active strings can be nested. For example:

```
LS -P [CWD [LHD]] -BF
```

causes the command processor to:

1. Interpret the active string [LHD] and insert the correct character string into the command line. At this point, the command line reads:

   ```
   LS -P [CWD ^ZSYS51>PROGS>JSMITH] -BF
   ```

2. Interpret the active string:

   ```
   [CWD ^ZSYS51>PROGS>JSMITH]
   ```

   The command line now reads:

   ```
   LS -P ^ZSYS51>PROGS>JSMITH -BF
   ```

3. Execute the LS command. A brief listing of the directory will be written to the current user-out file.

Command-only EC files allow minimal control over command processing. After practice with small command-only EC files (their creation and execution), you can add active strings or active strings in combination with EC control directives (described later in this section) to increase control of your processing environment.

ACTIVE FUNCTIONS

An active function is a command explicitly designed for use within an active string. Just as single (or multiple) commands are evaluated in command-only active strings, each active function is evaluated and its resulting value is substituted in the command line prior to execution. EC directives and active functions within your EC file control the sequence of processing. EC directives must begin with an ampersand (&) and must be followed by a space or a tab character. Active functions and their use in EC files are defined below.

Using EC Active Functions

Active functions can have arguments of their own. For example:

```
[MINUS a b]
```

Subtracts the value of b from the value of a. The result is returned to the command line and the command is processed.

Nested Active Functions

Active functions can be nested. For example, to find the value of:

```
3(2(5+1)-6)
```

the active string reads:

```
[TIMES 3 [MINUS [TIMES 2 [PLUS 5 1]] 6]]
```

If active functions are nested, the innermost pair of brackets is evaluated first, then the next pair of brackets out from those, etc. In this example, the value 18 is returned.

Multiple Active Functions

Multiple active functions can be included in a single active string. In this case, each active function is separated from the next by a semicolon. Only one pair of brackets encloses the entire string. The resulting value is the concatenation of the separate values of each active function. For example, if the active string:

```
[SUBSTR EXECUTE 1 6]
```

returns the value EXECUT, and the active string:

```
[SUBSTR DRIVER 3 3]
```

returns the value IVE, the active string:

```
[SUBSTR EXECUTE 1 6;SUBSTR DRIVER 3 3]
```

returns the value EXECUTIVE

Using Active Functions as Commands

The following active functions can be used as commands:

- CVD
- EQUAL
- NOW
- USER
- LWD
- LHD
- VALIDCKPT
- WH

If an active function (or active string) is used as the only entry on a command line, enter the active function without the enclosing brackets.

Groups of Active Functions

Active functions are divided into eight groups:

1. Arithmetic
2. Checkpoint
3. Date/Time
4. Directory
5. Logical
6. Question
7. String
8. User

The following alphabetic list of MOD 400 active functions indicates the group to which each active function belongs. Active functions preceded by an asterisk (*) can be entered as commands.

Active Function	Group
AND	Logical
BDATE	Date/Time
*CVD	Date/Time
DATE	Date/Time
DIVIDE	Arithmetic
*EQUAL	Logical
EXISTS	Logical
EXSW	Logical
GREATER	Logical
INDEX	String
LENGTH	String
LESS	Logical
*LHD	Directory
*LWD	Directory
MINUS	Arithmetic
NOT	Logical
*NOW	Date/Time
OR	Logical
PLUS	Arithmetic
*QUERY	Question
RESPONSE	Question
RETCODE	Logical
SUBSTR	String
TIME	Date/Time
TIMES	Arithmetic
*USER	User
*VALIDCKPT	Checkpoint
*WH	Directory

Arithmetic Active Functions

The arithmetic active functions perform arithmetic operations on their arguments. The arithmetic active functions are:

- PLUS
- MINUS
- TIMES
- DIVIDE

The values of the arguments can range from -32767 (decimal) to +32767 (decimal). A character string is returned as the result. If the operation produces an overflow result, the character string OVFL is returned. Note that division by 0 produces an overflow.

Example:

```
&IF [EQUAL[PLUS 3 2][MINUS 9 4]] &THEN &ELSE &G FINISH
```

In this example, the value returned for both arithmetic functions (PLUS, MINUS) is 5.

CHECKPOINT ACTIVE FUNCTION

The checkpoint active function returns a character string that specifies whether or not a valid restartable checkpoint was found in a specified pair of checkpoint files. If a valid checkpoint was found, the character string TRUE is returned; otherwise FALSE is returned. The checkpoint active function is:

VALIDCKPT

When used in conjunction with the EC control directive &IF (described later), if a valid restartable checkpoint is found, the VALIDCKPT active function can activate a restart when a task group invocation has abnormally terminated.

Example:

```
[VALIDCKPT  CPOINT]
```

If either the CPOINT.1 or CPOINT.2 checkpoint file contains a valid checkpoint, TRUE is returned; otherwise FALSE is returned.

DATE/TIME ACTIVE FUNCTIONS

Date and time active functions return a character string that represents the date and time. The date and time active functions are:

- BDATE
- CVD
- DATE
- NOW
- TIME

Example:

```
&IF [EQUAL [TIME] 09:05] &THEN &ELSE &G FINISH
```

If the time is 9:05, the value returned for the TIME active function is 09:05.

DIRECTORY ACTIVE FUNCTIONS

A directory active function returns a character string that represents information about an entry in the directory hierarchy. Directory active functions are:

- LHD
- LWD
- WH

Example:

```
GET [WH DATA] 8
```

The value returned is the full pathname ^PAYROLL>OVERTIME>DATA.

LOGICAL ACTIVE FUNCTIONS

These active functions are used in conjunction with the &IF,&THEN,&ELSE directives of the Execute command (EC). (These directives are described later in this section.)

Logical active functions return character strings TRUE or FALSE. The argument strings are compared character by character according to their ASCII code value. The first instance of an unequal ASCII code value returns FALSE. Thus, the sequence:

```
[EQUAL  4  04]
```

although arithmetically equal, returns FALSE because the first ASCII comparison is between 4 and 0.

Unequal string lengths also return FALSE.

Logical active functions are:

- AND
- EQUAL
- EXISTS
- EXSW
- GREATER
- LESS
- NOT
- OR
- RETCODE

QUESTION ACTIVE FUNCTIONS

Question active functions return a TRUE/FALSE representation of user-supplied answer to a specified question. The question active functions are:

- QUERY
- RESPONSE

Example:

```
&IF [EQUAL [QUERY "DO YOU WANT TO CONTINUE?"] TRUE]
&THEN &ELSE &G FINISH
```

If your response to the question "DO YOU WANT TO CONTINUE?" is YES, the character string TRUE is returned to the active string.

STRING ACTIVE FUNCTIONS

String active functions return the results of operations performed on a character string. String active functions are:

- INDEX
- LENGTH
- SUBSTR

Example:

```
LS [SUBSTR AB* 3 1;SUBSTR ABC.EC 4 3]
```

The first active function returns the string found beginning at the third position of the character string AB*, and of length 1. The result is *. This value is concatenated with the string found beginning at character 4 of the string ABC.EC and of length 3 (.EC). The returned value for the entire active string is *.EC.

USER ACTIVE FUNCTION

The user active function returns information about the current user of the system. The user active function is:

```
USER
```

Depending on the argument supplied, selected information can be retrieved from system data base.

Example:

```
[USER NAME]
```

This example returns the name of the current system user.

CREATING A MORE COMPLEX EC FILE

With careful planning you can create EC files that control the type of processing to be done; i.e., as specified in the command line. The simple EC file described earlier in this section controls processing with a step-by-step procedure. The EC file shown in Figure F-1, earlier, allows you to develop, compile and link a FORTRAN program. This EC file assumes that there are no errors in your source code and that the compilation of the source code is error-free. If there are errors in your source code, how can you check to make sure that your object code will enter the Linker session with a "clean compile"? This can be accomplished by using EC control directives explained below.

EC Control Directives

You can control certain operational aspects of the command processor to provide a degree of control over the logic of command execution by using EC control directives within your EC files.

These directives begin with an ampersand (&) and are followed by a space or a tab character. Each EC control directive is described briefly below. (For more detailed information on each directive, see the Commands manual.)

&

Specifies a comment line that is not processed. The line is visible only on the listing of the EC file.

&IF,&THEN,&ELSE

Specifies a series of conditional execution directives of the form:

```
&IF [active_function]
&THEN then_clause
&ELSE else_clause
```

The active function in an &IF control directive is evaluated. If the value of the active function is the string TRUE, then_clause is executed, otherwise else_clause is executed.

NOTES

1. The &IF, &THEN, and &ELSE directives cannot be used independently of each other.

2. The &ELSE else_clause directive is optional.

3. Then_clause is optional. Without it the next command is executed. An &ELSE else_clause if present, is skipped as usual.

4. Then_clause and else_clause can be any command line or control directive except &L, &IF, &THEN or &ELSE.

5. The EC processor terminates execution if any problem is encountered in an &IF statement. This includes improper syntax, an error from the active function, or an error from the then_clause or else_clause. Major commands, e.g., ASSEM, MACROP, and TRAN should not be used as a then_clause or else_clause because they have a higher likelihood of errors. The &G directive is recommended instead.

&A[pathname]

Changes the current user-in file to the specified pathname. If the pathname is not specified, the current command-in, i.e., the EC file is used as the user-in file.

&D

Restores the user-in file to what it was when the EC file was first invoked.

NOTE

This directive should be used before each &Q in your EC file to return the system to the state in which you began your session.

&N

Turns on the printing of command lines to the user-out file before the commands are passed to the command processor.

&F

Turns off the printing of commands to the user-out file. This is the default.

&G label

Provides a "go to" capability. This directive is used in conjunction with the &L and &IF,&THEN,&ELSE directives, or can be used alone. The next command that is processed is the first command (or EC directive) after the first &L directive that defines the label.

Example:

```
&IF [EQUAL A B] &THEN &ELSE &G END
                .
                .
                .
&L END
&D
&Q
```

In this example, if A is <u>not</u> equal to B, the EC directive routine goes to the label END. The next command read is &D, the first command after the label END.

&L label

Defines a label that may be the object of an &G directive. (See the &G example above.)

&P

Prints a line. Any character string entered after &P is printed on the current user-out.

Example:

```
&P LINKER SESSION BEGINS
```

In this example, the character string LINKER SESSION BEGINS is written to user-out.

&Q

Stops execution of the current EC file.

CREATING A GENERALIZED EC FILE

Every program you design may not be written in FORTRAN, and it is unlikely to name all bound units AREA. A convenient way of tailoring your program development EC file is by using substitutable parameters in argument lists.

Substitutable Parameters

A substitutable parameter in the command-in file is an ASCII character string whose first character is an ampersand (&), followed by one or more digits. The value indicates the position in the argument list of the data element to be substituted. For example, the format of the EC command is:

EC path arg_1 arg_2 ...

where the substitutable parameter for path is &0, the substitutable parameter for arg_1 is &1, etc.

To further illustrate, assume you want to create, compile, and link a COBOL source program. You can still use the basic outline of the EC file shown in Figure F-2, but with some minor modifications.

```
&CREATE, COMPILE, AND LINK A FORTRANA PROGRAM
&P BEGIN EDITOR SESSION
ED -PT
&P COMPILATION BEGINS
&IF [EQUAL [RETCODE] 0000] &THEN &ELSE &G ERROR1
&P LINKER SESSION BEGINS
&A LINKDR
LINKER
&IF [EQUAL [RETCODE] 0000] &THEN &ELSE &G ERROR1
&P LINK COMPLETE
&G FINISH
&L ERROR1
&P ERROR ENCOUNTERED IN DEVELOPMENT SEQUENCE
&P EC TERMINATED
&D
&Q
&L FINISH
&D
&Q
```

Figure F-2. Sample Complex EC

You can initially create (or subsequently modify) the EC file to accommodate all programming languages. Figure F-3 shows a generalized program development EC file. To execute the EC file for a COBOL program development session, you invoke the EC file by entering:

```
EC PROG_DEV COBOLA  PAYROLL
```

The path PROG_DEV is substituted for all occurrences of &0 (there are none in this EC file); COBOLA is substituted for all occurrences of &1; and PAYROLL is substituted for all occurrences of &2. The changed lines in the generalized EC file now read:

```
& CREATE, COMPILE, AND LINK A COBOLA PROGRAM
                    .
                    .
                    .
COBOLA PAYROLL
                    .
                    .
                    .
LINKER PAYROLL
                    .
                    .
                    .
```

Similar EC command formats can be used for each programming language.

If you are reading in system software directives (e.g., Linker directives) from the EC file, you must enter the &A directive before the command line that calls the component (see Figure F-3). This attaches the user-in file to the command-in file (i.e., the EC file itself). It is recommended that you enter an &D directive before each &Q directive in your EC file. The &D directive returns user-in to what it was before you entered the EC command.

```
    & CREATE, COMPILE, AND LINK A &1 PROGRAM
   &P BEGIN EDITOR SESSION
   ED -PT
   &P COMPILATION BEGINS
  &1 &2
   &IF [EQUAL [RETCODE] 0000] &THEN &ELSE &G ERROR1
   &P LINER SESSION BEGINS
   &A
   LINKER &2
  LINK &2
  QT
   &IF [EQUAL [RETCODE] 0000] &THEN &ELSE &G ERROR1
   &P LINK COMPLETE
   &G FINISH
   &L ERROR1
   &P ERROR ENCOUNTERED IN DEVELOPMENT SEQUENCE
   &P EC TERMINATED
   &D
   &Q
   &L FINISH
   &D
   &Q
```

Figure F-3. Sample Generalized EC File: Application Development

Appendix G
BACKUP AND RECOVERY

MOD 400 supports facilities that enable you to save and restore disk files, preserve the execution environment during a power failure, perform file recovery at the recovery level, and restart a program from a previously established point.

The save/restore facility allows you to preserve selected disk files and directories on magnetic tape or another disk volume and, when later required for processing, to restore the files, directories, and associated structures to disk.

The power resumption facility uses the memory save and auto-restart unit to preserve the memory image through a power failure lasting up to two hours. If power is restored during this time, the power resumption facility reconnects the previously online peripheral and communication devices and restarts the tasks that were running when the power failure occurred. If the power failure lasts more than two hours, the memory image is destroyed and the power resumption facility disabled. When power is restored, the user can reinitialize the system and use the file recovery and checkpoint facilities to restart the system from a previously established restart point.

File recovery enables you to dynamically save record images before they are updated and, if necessary, later write the images back to the file, thereby returning the file to its unaltered state. File recovery provides file integrity in the event of a system failure.

File recovery is provided through three distinct functions:

- "Before image" recording, which preserves a record prior to its being updated.
- "Cleanpoint" or "checkpoint" declarations, which are issued in your program and define a point at which all updates are complete. When the updates are complete, the associated before images are destroyed.
- "Rollback" or "restart" functions, which return the files to their unaltered state by applying all before images that have been recorded since the last cleanpoint.

The cleanpoint and rollback functions should be used to provide file recovery in a transaction-oriented environment. They are best suited for applications in which a single transaction causes a number of record updates. In a batch processing environment, the checkpoint and restart procedures should be used for file recovery and program restart.

The checkpoint restart facility enables you to establish a point in the program to which you can return at a later time and continue processing. The return point (checkpoint) is used to save the current status of the task group. You issue a checkpoint call in the program when you reach a point in processing where the program could be restarted. A restart can be performed at the most recently completed checkpoint at any time during processing. If the task group is abnormally terminated for any reason, it can be restarted at the most recent valid checkpoint.

DISK FILE SAVE AND RESTORE

The Save and Restore programs allow you to save and restore disk files and directories. Save is used to save disk files and directories on a disk or magnetic tape volume for later restoration by Restore.

The Restore program reconstructs the file structures copied by the Save program. If a file being restored already exists on the volume (or volumes), the Restore program replaces the current file contents with the file data saved by the Save program. (The access list is not altered.) If a file being restored does not exist on the volume, the Restore program creates the file and loads the saved data. (Access is set as defined in the saved file.)

POWER RESUMPTION

Power resumption is an optional facility that allows the system execution environment to be automatically restarted after a power interruption. The Level 6 central processor must have the memory save and autorestart unit. This unit can preserve the memory image through a power failure lasting up to two hours.

(It cannot, however, preserve the state of the I/O controllers nor ensure that no operational changes have been made to the mounted volumes.)

If fewer than two hours have elapsed when power is returned to the central processor, the power resumption facility will perform the following functions:

- Reinitialize the system software.
- Reconnect peripheral devices.
- Reconnect communication devices serviced by the asynchronous terminal device (ATD) line protocol handler or the teleprinter (TTY) line protocol handler (see the System Building and Administration manual and System Programmer's Guide, Volume I for information on line protocol handlers).
- Restart application tasks that were active at the time of the failure if these are display formatting and control facility tasks or are tasks containing user-written code to handle power failure/power resumption.

Implementing the Power Resumption Facility

The power resumption facility must be included in the MOD 400 Executive at system building. The Level 6 central processor must contain a memory save and autorestart unit that has been activated by the operator (see the System User's Guide for activation procedures).

When power resumption is specified in the system building dialog, all peripheral devices and all communication devices associated with the ATD and TTY line protocol handlers are designated as reconnectable and will be automatically reconnected when power is restored. If any ATD/TTY-associated device is not to be automatically reconnected, you must edit the CLM file to remove the -RECONNECT argument from the STTY directive generated for the device.

Power Resumption Procedures

The power resumption facility automatically performs the following functions:

- Restarts the device drivers, clock, communications subsystem, and display formatting and control facility.
- Reconnects all peripheral devices that were online at the time of the failure.

- Reconnects ATD/TTY-associated communication devices that were online at the time of the failure, except for those devices designated as not reconnectable.
- Restarts the screen forms on reconnected terminals controlled by the display formatting and control facility.
- Resets the system date and time if the date/time clock has a separate battery backup unit.
- Reloads the memory management unit (if any).
- Reestablishes the integrity of mounted volumes.
- Restarts application tasks that were active when the power failure occurred if they are display formatting and control facility tasks or tasks containing user-written code to handle power failure/power resumption.

In order for an application task to be notified when a power resumption has occurred, it must connect its own trap handler and enable trap 53. Trap 53 condition will be signaled when the task becomes active and is issuing its own instructions (not executing Executive instructions). See "Trap Handling" in the MOD 400 System Concepts manual.

After a power resumption has occurred, peripheral devices and reconnectable ATD/TTY-associated devices that were online at the time of the failure are again brought online. The system operator may be required to initialize certain peripheral devices. A terminal user may be required to reenter the input line if he had not pressed the RETURN or XMIT key when the failure occurred. See the System User's Guide for details.

FILE RECOVERY

File recovery enables you to save record images from a file before it is updated and to later write these images back to the file, eliminating the alterations made during the updating. Every time a record is updated, a copy of the record, as it exists before the update, is written to a system-created file. The system-created file is called a recovery file; the records it contains are called before images. The system uses the recovery files to bring data files to a consistent state following a software failure or a system failure such as that caused by a loss of power. When the before images are applied in reverse chronological order to the data files, the data files are rolled back to a previously established state.

Designating Recoverable Files

File recovery is optional. You can designate a file as recoverable through the -RECOVER argument of the create file (CR) command. Files not created as recoverable can be made recoverable by specification of the -RECOVER argument of the modify file attribute (MFA) command.

Recoverable files can be made non- recoverable through the specification of the -NORECOVER argument in the MFA command.

Recovery File Creation

Each task (or task group in some cases) having a data file designated as recoverable has associated with it a recovery file. The recovery file is created by the system when the first before image for a recoverable file is about to be written.

If the tasks in a task group have only sharable files, only one recovery file exists for the group. If any task in a task group has an exclusive file, one recovery file is created for each task in the group.

All recovery files are created subordinate to your working directory. The names of the files are recorded in the RECOVERY directory, which is positioned under the root directory of the system volume. This directory is maintained by the system. Each recovery file is assigned a name of the form:

```
$$RECOV.ggtt
```

where:

gg - Group identifier
tt - Task identifier

File Recovery Process

The system recovers a data file (i.e., erases the updates made to it) by writing the before images back to the file.

You can declare points in your processing (called clean-points) at which all file updates are considered valid. When a cleanpoint is declared, all before images taken up to that point are invalidated. New before images are written when you begin to update the file.

You can perform a rollback at any time during processing. When a rollback is requested, the before images are written to the file, wiping out updates made since the last cleanpoint.

Use of the cleanpoint and rollback functions is recommended in a transaction-oriented environment.

TAKING CLEANPOINTS

When you consider the data in your file to be consistent and valid, you declare a cleanpoint in your program. Cleanpoints are established by CALL "ZCLEAN" statements in COBOL programs or $CLPNT macro calls in assembly language programs.

When a cleanpoint is declared, the system performs the following actions:

- Writes all modified buffers to disk
- Updates all directory records
- Invalidates the recovery file before images that have been taken for the data file
- Unlocks all records previously locked by the user (tasks waiting for these records are activated).

Note that the file system performs a cleanpoint when a recoverable file is closed.

REQUESTING ROLLBACK

Rollback initiates the recovery of a file to the condition in which it was at the last cleanpoint. If programming in COBOL, you request a rollback by coding a CALL "ZCROLL" statement. If programming in assembly language, you request a rollback by coding a $ROLBK macro call. When a rollback is requested, the system performs the following actions:

- Takes before images from the recovery file and writes them to the data file, thereby wiping out updates made since the last cleanpoint.
- Invalidates the before images on the recovery file.
- Unlocks all records previously locked by the user. (Tasks waiting for these records are activated.)

The file system performs a rollback when a task group terminates abnormally.

RECOVERING AFTER SYSTEM FAILURE

When the system is reinitialized following a system failure, it checks for the existence of recovery files. If recovery files do not exist, files had not previously been declared as recoverable or updates had not previously been made to recoverable files. If recovery files do exist, the system failure occurred while updates were being made to a file that had the recover attribute. If recovery files exist, the operator should issue the Recover command so that the system will perform a rollback of all recoverable data files. See the System User's Guide for details.

CHECKPOINT RESTART

The checkpoint restart facility allows you to provide a file recovery and program restart capability in a batch processing environment. Through checkpoint restart you can establish a point in your program to which you can return at any time and continue processing. This return point (called a checkpoint) is used to save the current status of the task group request. You can perform a restart to the most recently completed checkpoint after the abnormal termination of the task group request or at any point during the processing of the task group request. A restart cannot be performed from an earlier checkpoint, nor can it be performed after the normal termination of a task group request.

Checkpoint restart does not support the use of the listener secondary login facility.

Checkpoint

When a task requests a checkpoint, the system records the current contents of your memory and the current state of tasks, files, and screen forms onto a checkpoint file previously assigned. The system then takes a cleanpoint so that recoverable files are synchronized with that checkpoint. See "File Recovery" earlier in this section for a description of recoverable files and cleanpoints.

The system supports one checkpoint task and any number of other tasks that are dormant or are waiting on requests placed against other tasks in the task group. (Thus, a single active command executing under the command processor and/or any number of nested ECs can be checkpointed.)

Checkpoint File Assignment

You can enable the checkpoint restart facility for your task group and designates where its checkpoint images are to be recorded by issuing the checkpoint file assignment (CKPTFILE) command.

Checkpoints are written alternately to each of a pair of checkpoint files. This technique ensures the availability of the previous valid checkpoint if a failure occurs during the process of taking a checkpoint. The system locates and uses only the most recently completed successful checkpoint from the pair of checkpoint files that you have specified.

When designating the checkpoint file, you specify a single pathname (the last element of which can be a maximum of 10 characters). The system appends the suffixes .1 and .2 as appropriate. If the system cannot find one or both of the specified checkpoint files, it creates it/them.

TAKING A CHECKPOINT

When a checkpoint is taken, the system writes a checkpoint image and performs a cleanpoint for all recoverable files. If programming in Advanced COBOL, you request a checkpoint by coding a CALL "ZXCKPT" statement or using the RERUN clause in the I-O-CONTROL paragraph. If programming in assembly language, you request a checkpoint by coding a $CKPT macro call.

Your task group must be in a "checkpointable" state when it requests a checkpoint. A task group is in a checkpointable state when each task that is part of the group has requested a checkpoint, is waiting on a request issued to another task in the task group, or is dormant (i.e., there are no current requests for the task).

Once a checkpoint is recorded by a task group, it remains available as a restart point until the next checkpoint request is completed, the current checkpoint file is disassigned (by the -DISASSIGN argument of the CKPTFILE command), or the task group request is terminated normally.

The lead task of the group may be waiting on both another task, which is a member of the group and a "break" request.

CHECKPOINT PROCESSING

When a task group takes a valid checkpoint, the system records the following information on the checkpoint file established for that group.

1. Executive information, including data structures, user pool memory blocks, data segments of bound units linked with separate code and data, and floatable overlays.

2. Status and pathnames of the standard I/O files and of nonsharable bound units.

3. Memory locations and pathnames of sharable bound units.

4. Current state of screen forms for terminals operating under the display formatting and control facility.

5. Status and position of all active files (i.e., files that have been associated, reserved, or opened).

When your file information has been recorded, the checkpoint image is completed and a cleanpoint is taken. You must ensure that files to be synchronized with the checkpoint restart process have been designated as recoverable. Since the file system performs a cleanpoint when a recoverable file is closed, you may have to take a checkpoint prior to closing the file to keep checkpoint restart synchronized with the state of the recoverable file. (Temporary files cannot be designated as recoverable.)

Checkpoints cannot be taken while an active local mail message group exists (i.e., a checkpoint cannot be taken in the period between message initiation or acceptance and message termination).

Checkpoints are not made automatically obsolete by the normal termination of the task under which they were issued. To invalidate a previous checkpoint (taken during the execution of one command) before processing a new command, you must take a checkpoint immediately prior to the termination of that command.

Restart

You can perform a restart at the following times:

- During the processing of the task group request that issued the checkpoint restart.
- During the processing of a task group request that was scheduled after the abnormal termination of the task group request in which the checkpoint was taken.
- When the system is reinitialized following a system failure.

When a restart request is issued, the task group issuing the request is terminated abnormally and the task group request recorded on the checkpoint file is again put into effect.

The system locates the most recently completed checkpoint and reads the checkpoint image from the file, rebuilding the Executive data structures and memory blocks, reloading bound units, and repositioning active files.

Procedural code and workspace must occupy the same physical memory locations that were used when the checkpoint was taken. In general, task groups that are to be restarted must be the sole users of exclusive memory pools. Sharable bound units referred to by these groups must be permanently loaded (through the Load command in the system startup EC file). The configuration under which the restart is performed must be identical to that which existed when the checkpoint was taken.

REQUESTING A RESTART

To restart from the last completed checkpoint (and to abort the current task group request if restarting during the session), you issue the Restart command. The operator can restart an existing task group that has a valid checkpoint by using the -GROUP argument of the Restart command. If the memory blocks required to effect the restart are not available, the restart will be aborted. Specification of the -WTMEM argument of the restart command will cause the system to wait until the specific memory blocks required to perform the restart become available.

If this is a restart following a system failure, the Recover command must have been issued by the operator or through an EC file to perform a system-wide rollback of all recoverable files.

If a restart is performed during a session, the abort (termination) of the group request will cause a rollback of all recoverable files in your task group. The abnormal termination of the group request causes the last completed checkpoint image to be retained as a valid checkpoint. The Abort Group and Abort Group Request commands force an abnormal termination; the Bye command causes a normal termination. The normal termination of the command processor with a nonzero value in the $R2 register is treated as an abnormal termination for checkpoint file purposes.

RESTART PROCESSING

When the Restart command is issued, the system performs the following steps:

1. Locates the most recently completed checkpoint.

2. Validates that the restart is being performed under the same user id as that used when the checkpoint was taken.

3. Rebuilds Executive data structures.

4. Reads nonsharable bound units, data segments, floatable overlays, and memory blocks that were obtained by get-memory operations from the checkpoint image into the same memory locations they occupied at the time the checkpoint was taken.

5. Reloads sharable bound units in the system memory pool. Only the code segment is reloaded if the bound unit was linked with separate code and data. Unless it was linked with the restart relocatable attribute (Linker RR directive), the code segment is reloaded at the same system pool memory locations occupied when the checkpoint was taken.

6. Associates, gets, opens, and positions active user files recorded on the checkpoint image. Rollback should have been performed already; see "Requesting a Restart" above.

7. Restores the screen content of terminals that were operating under the display formatting and control facility and were active at the time of the checkpoint.

8. Reissues the break request if such a request had been issued by the lead task at the time of the checkpoint.

9. Turns on the task that issued the checkpoint request at the next sequential instruction after the checkpoint.

The checkpointed state of the standard I/O files is reestablished at restart time. Modifications made to files (e.g., EC files) between the checkpoint and the restart must be restricted to those that do not invalidate the repositioning of the files. A command being restarted must remain in the same position in the file; only those commands that follow the restarted command have any effect on the restarted task group request.

Sharable bound units being used by a checkpointed task group are reloaded and not restored from a checkpointed memory image (except for the data segments of bound units linked with separate code and data). Thus, all such bound units should contain only code. All sharable bound units in use by a restarting task group must be identical to the versions that existed at the checkpoint. They cannot be relinked. If an overlay area table (OAT) is in use for such a bound unit, no overlay area can be reserved at the time the checkpoint is taken.

If the application programs that issue physical I/O orders for communication devices, you must reissue connects to those devices before issuing read and write orders to them.

Appendix H
REQUESTING AND USING MEMORY DUMPS

This appendix provides procedures for requesting memory dumps, as well as procedures for analyzing, interpreting, and resolving errors using memory dumps. The following memory dump utilities are described:

- MDUMP
- DPEDIT

MDUMP UTILITY

The MDUMP is a stand alone utility that allows you to obtain a memory dump with no requirement for system functions. MDUMP may be used when it is not possible or practical to use the debug utility dump facility.

MDUMP Requirements

To use MDUMP, you need a disk that contains an MDUMP bootstrap record on sector 0, and a file (DUMPFILE) large enough to contain the complete memory image. The Create Volume command is used to prepare this disk (see "Preparing to Execute MDUMP", below).

To dump memory to the disk file, bootstrap the prepared disk as described under "Procedure for Using MDUMP," below. This procedure loads and executes MDUMP. When MDUMP terminates, an image of memory is contained in DUMPFILE.

This file can be edited and printed using the Dump Edit utility, also described later in this section.

Preparing to Execute MDUMP

Before loading the program for which a memory dump is required, enter the Create Volume command:

```
CV path {-MDUMP nn} [{-BOOT  X'hhhh'}]
        {-MD    nn} [{-BT  X'hhhh' }]
```

ARGUMENTS:

path

Designates the pathname to the disk volume being prepared for MDUMP. The pathname may be !sympd or !sympd>volid. If >volid is specified, the volume label is checked. The volume must have been previously formatted via a Create Volume command. (This command is described in detail in the Commands manual.) The volume can contain other data.

{-MDUMP nn}
{-MD nn }

Writes the MDUMP bootstrap record to the volume specified in the path argument and allocates a file (DUMPFILE) large enough to contain nn 4K word modules to be dumped. The resulting dump volume may be used for any configuration of memory less than or equal to the value nn x 4K words.

{-BOOT X'hhhh'}
{-BT X'hhhh' }

Creates bootstrap records and intermediate loader records and writes them to disk sectors 0 through 6. The optional X'hhhh' field defines certain available bootstrap options. See the Commands manual for details.

NOTE

This argument can be used in conjunction with the -MDUMP argument to obtain a combination bootstrap/MDUMP (described below).

Procedure for Using MDUMP

Once an executing program encounters a problem or a halt occurs, you can obtain a memory dump by taking the following actions:

1. Bootstrap MDUMP, which then sends the memory dump to the disk file DUMPFILE.

2. Rebootstrap the system.

3. Use the Dump Edit utility program (DPEDIT) to print all or a portion of the memory dump from the disk volume that contains MDUMP's output.

Procedure for Bootstrapping MDUMP

To bootstrap the MDUMP bootstrap record into memory, perform the procedure shown below. MDUMP then transfers to the disk file (DUMPFILE) the amount of memory image specified in the -MDUMP argument of the Create Volume command.

1. Mount the disk containing the MDUMP bootstrap routine on the device to be used in bootstrapping.

2. Press Stop and CLear.

3. Set the P-register to 0004 .

4. Enter the channel number of the bootstrap device (i.e., the disk mounted in step 1) in register R1.

   If -BT was specified when creating the MDUMP dump device, bit 12 must be on in the R1 value (i.e., R1 is set to CCC8, where CCC is the channel number). This causes the MDUMP bootstrap record to be selected.

5. Enter the initial address of the memory area into which MDUMP is to be held in register B1. MDUMP requires one sector of the disk device type on which it is stored. The initial address of B1 should be greater than 100 to ensure that hardware dedicated locations are not overlayed.

6. Press Load, then Execute. MDUMP is read into the memory location specified in step 5 above, and dumps the amount of memory image that fills DUMPFILE. The dump is complete when an end-of-job halt occurs (see Table H-1).

NOTE

The size of DUMPFILE is limited by the capacity of the storage device. A maximum of 120K of memory can be stored on a diskette file.

MDUMP Halts

No messages are issued during execution of MDUMP. If a halt occurs during execution, the contents of the P-register and R6 register must be displayed to determine the significance of the halt, as indicated in Table H-1.

Table H-1. MDUMP Halts

Register Contents			
P-Register	R6 Register	Condition	Operator Action
003E	=0	End of job	No operator action required. For information only.
003E	^=0	Disk error	Reboot MDUMP. (R6 contains the disk status word.)
03nn	[illegible]	Trap handler error has occurred.	For a description of trap messages, see the "Trap Handling" section of the System Programmer's Guide, Volume I.

Address relative to the initial address of MDUMP as stored in memory.

DUMP EDIT UTILITY (DPEDIT)

Dumps produced by the Dump Edit utility are written to the user out file, which must be capable of receiving a 132-character line.

There are two sources of dumps:

- Files created by the previous execution of the MDUMP utility. (All or selected portions of the file can be dumped.)
- Main memory. (A dump of main memory allows you to determine the configuration under which Dump Edit is executing.)

Dumps produced by Dump Edit may be logical (edited format) dumps or physical (memory image format) dumps. Control arguments in the DPEDIT command (described later in this section) allow you to request either a logical or physical dump. If these control arguments are omitted, execution of Dump Edit produces a full logical dump followed by a full physical dump.

Logical and physical dumps are printed in both hexadecimal and ASCII notation. Duplicate lines, if any, are suppressed. Suppressed lines are designated as described under "Dump Edit Line Format".

Page Header

The page heading contains the following information:

- Indicates whether the dump is from main memory or a dump file
- The date and time of the edit
- The version of DPEDIT used
- The version of the system DPEDIT is executing on
- Indicates the pool and group currently being dumped for a logical dump
- The page number

Dump Edit Line Format

The format of a basic dump edit line for both logical and physical dumps is as follows:

Columns	Content
1-6	Six hexadecimal digits designating the starting physical (real) address of the line of dump information. The hexadecimal digit in print position 6 is always 0. This forces the dump line to agree with the template printed at the heading of each page.
7	Slash (/)
8	Blanks
9-14	Six hexadecimal digits designating the starting virtual address of the line of dump information.
18-98	Sixteen consecutive words. Each word is represented by four hexadecimal digits and is followed by a space.
99-100	Blanks
101-132	ASCII representation of the previous group of 16 consecutive words. A byte that is not printable is designated by a period (.).
1-11	Blanks
12-93	* * * * * * * * * * * * (indicates one or more duplicate lines)
94-132	Blanks

Physical Dumps

In a physical dump, the leftmost six columns of data designate real memory addresses. When the Memory Management Unit (MMU) is in use, there may be ranges of invalid virtual addresses (columns 9-14; discontinuities) in a physical dump from main memory. When an invalid virtual address is encountered, a message within the physical dump contains the physical address for which no valid virtual address exists.

The virtual address is displayed whenever possible. If it does not appear, it means that the virtual and physical addresses are the same (in low memory), or that DPEDIT could not discover the virtual address corresponding to a given physical address. When the physical dump resumes, the valid virtual address is known and the left column continues to designate real memory addresses as if the discontinuity did not exist.

A physical dump from an external dump file does not display invalid virtual address messages, and the left column of addresses is an uninterrupted continuum of physical addresses.

A physical memory dump in Figure H-1 was produced by Dump Edit in response to the command:

```
DPEDIT ^DMPVOL>DUMPFILE - NL - TO X'0731'
```

Logical Dumps

A logical dump can be tailored by selecting (or suppressing) task group information on a group identification basis. File system information can also be suppressed. Logical dump tailoring is specified using DPEDIT command control arguments.

The main addresses in a logical dump are virtual addresses (columns 9-14). The leftmost six columns of data are physical addresses, and will be displayed whenever they differ from the virtual addresses. This applies to dumps of disk files as well as to dumps of main memory. For disk files, Dump Edit calculates the virtual address in the same way as the Memory Management Unit would under the same conditions.

The arrangement of information in a logical dump is described in the following paragraph and illustrated in Figure H-1.

The information contained in a logical dump includes:

- Location and contents of hardware-dedicated main storage
- System time of dump
- Time of system boot
- Time of power-fail restart (if it occurred)

- Hardware configuration
- Location and contents of System Control Block (SCB)

  - Model number of central processor

  - Presence (or absence) of the Commercial Instruction Processor, the Scientific Instruction Processor, and the Memory Management Unit

  - Value of the real-time clock scan cycle

  - Presence (or absence) of an operator's terminal

  - High address of virtual memory

  - High address of physical memory

- Software Configuration

  - Name and version of operating system
  - Presence (or absence) of the error message library
  - Size of trap save area (TSA)
  - Size of interrupt save area (ISA)
  - Number of indirect request blocks (IRBs) in IRB pool
  - Presence (or absence) of the batch task group.

```
MDUMP FILE  COMP     1982/06/28 1532149.8      DUMPEDIT- 3.0-04/22/1713        GCOS4 MOD400-L3.0-04/02/1430          PAGE 0001

 REAL   VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F        ASCII REPRESENTATION

HARD AND  DEDICATED LOCATIONS
        0000000/  0BC0 FFFF 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
        0000100/  0000 0100 0000 0000 0004 0000 0000 0000  0000 01AD 0000 0000 0000 0000 0000 0000  ................................
        0000200/  0000 0000 0000 0001 0000 0101 0000 0103  0000 0105 0000 0107 0000 0109 0000 010B  ................................
        0000300/  0000 010D 0000 010F 0000 0111 0000 0113  0000 0115 0000 0117 0000 0119 0000 011B  ................................
        0000400/  0000 011D 0000 011F 0000 0121 0000 0123  0000 0125 0000 0127 0000 0129 0000 012B  ..............!...#...%...'...)...+
        0000500/  0000 012D 0000 012F 0000 0131 0000 0133  0000 0135 0000 0137 0000 0139 0000 013B  ...-.../...1...3...5...7...9...;
        0000600/  0000 013D 0000 0139 0000 0141 0000 0143  0000 0145 0000 0147 0000 0149 0000 014B  ...=...?...A...C...E...G...I...K
        0000700/  0000 014D 0000 014F 0000 0151 0000 0153  0000 0155 0000 0157 0000 0159 0000 2475  ...M...O...Q...S...U...W...Y...$u
        0000800/  0000 0182 0000 055e 0000 056a 0000 06A1  0000 0503 0000 0008 0000 0002 0000 0000  ........^...h...................
        0000900/  0000 0000 0000 EFAA 0000 BF3E 0000 99F0  0000 9B0E 0000 9C2C 0000 9D4A 0000 9E68  ............>...........,...J...h
        0000A00/  0000 A0D2 0000 A03D 0000 A15e 0000 A279  0000 A5aC 0000 A6DA 0000 A7F8 0000 A916  ........=...^...y...............
        0000B00/  0000 AAF3 0000 AF03 0000 B3A6 0000 B18A  0000 B2DE 0000 AA34 0000 0000 0000 0000  ...............4..........^.....
        0000C00/  0000 0000 0000 0000 0000 0000 0000 D6C6  0000 0000 0000 0000 0000 0000 0000 0000  ................F...............
        0000D00/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
                    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
        0001000/  0000 0180 3C33 B2D3 3E20 3302 0004 ABCD  007A AF83 CFE2 DBC0 0034 9805 D051 D570  .....8..> 3.......8.......4...0.P

TIME OF SYSTEM COMP     1982/05/06 1059:11.8
TIME OF SYSTEM STOP     1982/05/06 1404:11.3

SYSTEM CONTROL BLOCK     0001A0
        0001A00/  0000 0000 2020 0001 0C08 8585 5348 5334  3431 3032 312C 434F 5059 5249 4748 542C  ....  .......8N8021828,COPYRIGHT,
        0001B00/  2843 2929 484F 4E45 5957 454C 4C49 4E46  4F52 4D41 5449 4F4E 5359 5354 4540 532C  (C),HONEYWELLINFORMATIONSYSTEMS,
        0001C00/  494E 432E 2C31 3930 3000 2056 392E 3020  3034 2F30 322F 3134 3330 0000 0002 0000  INC.,1980.-V9.0-04/02/1430......
        0001D00/  0912 0000 0DA8 0004 DBC3 0000 0000 0000  0090 0003 1064 0000 0BA0 0005 0FE3 0140  ...................d...........@
        0001E00/  FC7A 0003 B0C3 0255 ADDB FDA8 0032 0000  0000 86F0 0000 0000 0003 152C 000F FFFF  .z.....U.....2.............,....
        0001F00/  0000 0043 0005 0F83 0000 0090 0000 8703  0000 4D42 0025 FA7F FFFC 1000 0000 0001  ...C.............MB.%..........
        0002E00/  0000 0000 0000 0000 0CA1 0000 0000 0000  0000 2442 FFFE 0000 F4AA 2800 0000 1C4A  ..................$B........(....J
        0002F00/  0000 0480 0000 2409 000F FFFF 0004 E1E3  0000 0000 0000 0000 0000 0000 E000 0000  ......$.........................
        0003000/  0000 0000 0000 0000 0000 0000 0005 0E23  0000 0000 0427 1096 0003 C45A 0000 0000  ...............'.....Z.....
        0003100/  0005 0003 0000 2520 000A 0000 2969 0000  0000 0000 0000 0005 05A6 0000 9807 0819  ......%-....)!..................
        0003200/  0005 1CA3 3000 0000 C888 0000 0001 748D  0000 0000 0801 0000 0000 0000 0000 0000  ....0.....t.....................
        0003300/  0000 0000 0000 0006 0000 0000 0001 339F  0001 3667 0001 3823 0080 0001 3FD0 0025  ..............3...6g..8#....?..%
        0003400/  000A 0000 0001 0001 37F2 0000 2AA2 0000  2ACA 0001 9698 0000 2078 0000 0E92 0000  ........7...*...*...*...v.......
        0003500/  0E78 0000 0458 0000 0000 0000 0000 2020  2020 0004 E2F0 0000 0000 0000 0000 0000  .....X..........  ..............
        0003600/  0000 0001 3006 0005 0438 0000 0000 0000  1385 0000 27F5 0000 27FE 0000 0000 0000  ....0....8..........'5....'~....
        0003700/  0000 0000 0000 0001 0000 0000 0000 454E  0000 0001 7465 0000 0000 0000 55EA 0255  ..............EN....te........U..U
        0003800/  ACCD E624 0000 0000 0000 0000 045C 0005  1800 0004 0000 0001 0003 1873 0003 1040  ...)..........\...............s...@
        0003900/  0001 0000 0000 0000 7FFF 0001 EE18 000e  2CE3 0001 9C56 0000 0000 0000 0000 0000  ................,....V..........
        0003A00/  0000 0000 FFFF 0003 6674 0004 F2C3 0004  F523 0004 F238 0402 0003 8685 0000 0000  ........ft.......#...8..........

COMMERCIAL PROCESSOR:  YES
SCIENTIFIC PROCESSOR:  YES
MEMORY MANAGEMENT UNIT IN USE:  YES
TIME (MILLI-SEC) BETWEEN REAL TIME CLOCK INTERRUPTS:  0032
OPERATOR'S TERMINAL:  YES
HIGH PHYSICAL MEMORY ADDRESS:  0FFFFF
HIGH VIRTUAL  MEMORY ADDRESS:  0FFFFF
EXTERNAL DUMPFILE'S CAPACITY:  0FFFFF

OPERATING SYSTEM:  MOD400/L3.0
OPERATING SYSTEM VERSION:  -V9.0-04/02/1430
```

Figure H-1. Memory Dump Example

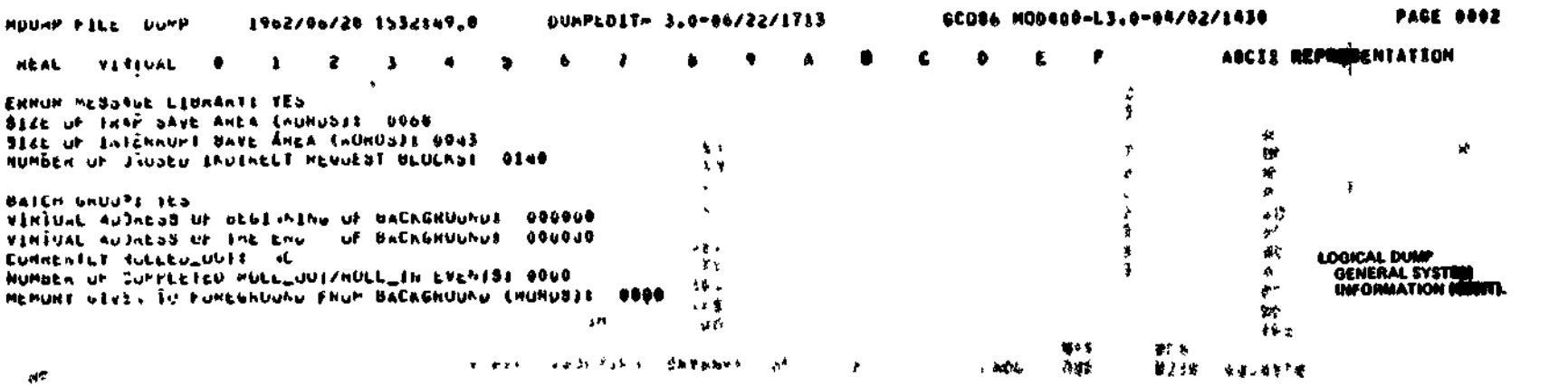


Figure H-1 (Cont). Memory Dump Example

REAL VISUAL 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII REPRESENTATION

MEMORY POOL DEFINITIONS

POOL NAME	START ADDRESS	END ADDRESS	SIZE (WORDS)	PHYSICAL START
SS	031540	0517FF	020CC0	031540
SM	051800	0BFFFF	06E800	051800
L0	051800	0BFFFF	06E800	051800
L1	051800	0BFFFF	06E800	051800
L2	051800	0BFFFF	06E800	051800
L3	051800	0BFFFF	06E800	051800
L4	051800	0BFFFF	06E800	051800
L5	051800	0BFFFF	06E800	051800
L6	051800	0BFFFF	06E800	051800
L7	051800	0BFFFF	06E800	051800
L8	051800	0BFFFF	06E800	051800
L9	051800	0BFFFF	06E800	051800
LA	051800	0BFFFF	06E800	051800
LB	051800	0BFFFF	06E800	051800
LC	051800	0BFFFF	06E800	051800
LD	051800	0BFFFF	06E800	051800
ZZ	051800	0BFFFF	06E800	051800
CC	051800	0BFFFF	06E800	051800
AB	051800	0BFFFF	06E800	051800
SP	051800	0BFFFF	06E800	051800
I2	060000	06FFFF	010000	0C0000
I1	070000	07FFFF	010000	0D0000
BAICM	080000	09FFFF	020000	0E0000

MEMORY POOLS STATUS

TOTAL_UNUSED (WORDS)	MAXIMUM_UNUSED CONTIGUOUS (WORDS)	NUMBER_OF FRAGMENTS	NUMBER OF_USERS
00DAA0	00A580	000048	000002
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
046900	032800	00000E	000007
010000	010000	000001	000000
006560	0064C0	000004	000001
020000	020000	000001	000001

NUMERIC INFORMATION ON ALL POOLS

MEMORY POOL ATTRIBUTES

POOL NAME	PROTECTED	CONTAINED	OVERLAPPED	ISOLATED	SWAPPED	SERIALLY USED	QUEUED MEMORY MANAGEMENT	USER'S_WRITE RING_NUMBER	PRIVILEGED
SS	YES	NO	NO	NO	NO	NO	NO	0	YES
SM	YES	YES	NO	NO	YES	NO	YES	3	NO
L0	YES	YES	NO	NO	YES	NO	YES	3	NO
L1	YES	YES	NO	NO	YES	NO	YES	3	NO
L2	YES	YES	NO	NO	YES	NO	YES	3	NO
L3	YES	YES	NO	NO	YES	NO	YES	3	NO
L4	YES	YES	NO	NO	YES	NO	YES	3	NO
L5	YES	YES	NO	NO	YES	NO	YES	3	NO
L6	YES	YES	NO	NO	YES	NO	YES	3	NO
L7	YES	YES	NO	NO	YES	NO	YES	3	NO
L8	YES	YES	NO	NO	YES	NO	YES	3	NO
L9	YES	YES	NO	NO	YES	NO	YES	3	NO
LA	YES	YES	NO	NO	YES	NO	YES	3	NO
LB	YES	YES	NO	NO	YES	NO	YES	3	NO
LC	YES	YES	NO	NO	YES	NO	YES	3	NO
LD	YES	YES	NO	NO	YES	NO	YES	3	NO
ZZ	YES	YES	NO	NO	YES	NO	YES	3	NO
CC	YES	YES	NO	NO	YES	NO	YES	3	NO
AB	YES	YES	NO	NO	YES	NO	YES	3	NO
SP	YES	YES	NO	NO	YES	NO	YES	3	NO
I2	YES	YES	NO	NO	NO	NO	NO	2	NO
I1	YES	YES	NO	NO	NO	NO	NO	0	NO
BAICM	YES	YES	NO	NO	NO	NO	NO	3	NO

PROPERTIES OF ALL POOLS

Figure H-1 (Cont). Memory Dump Example

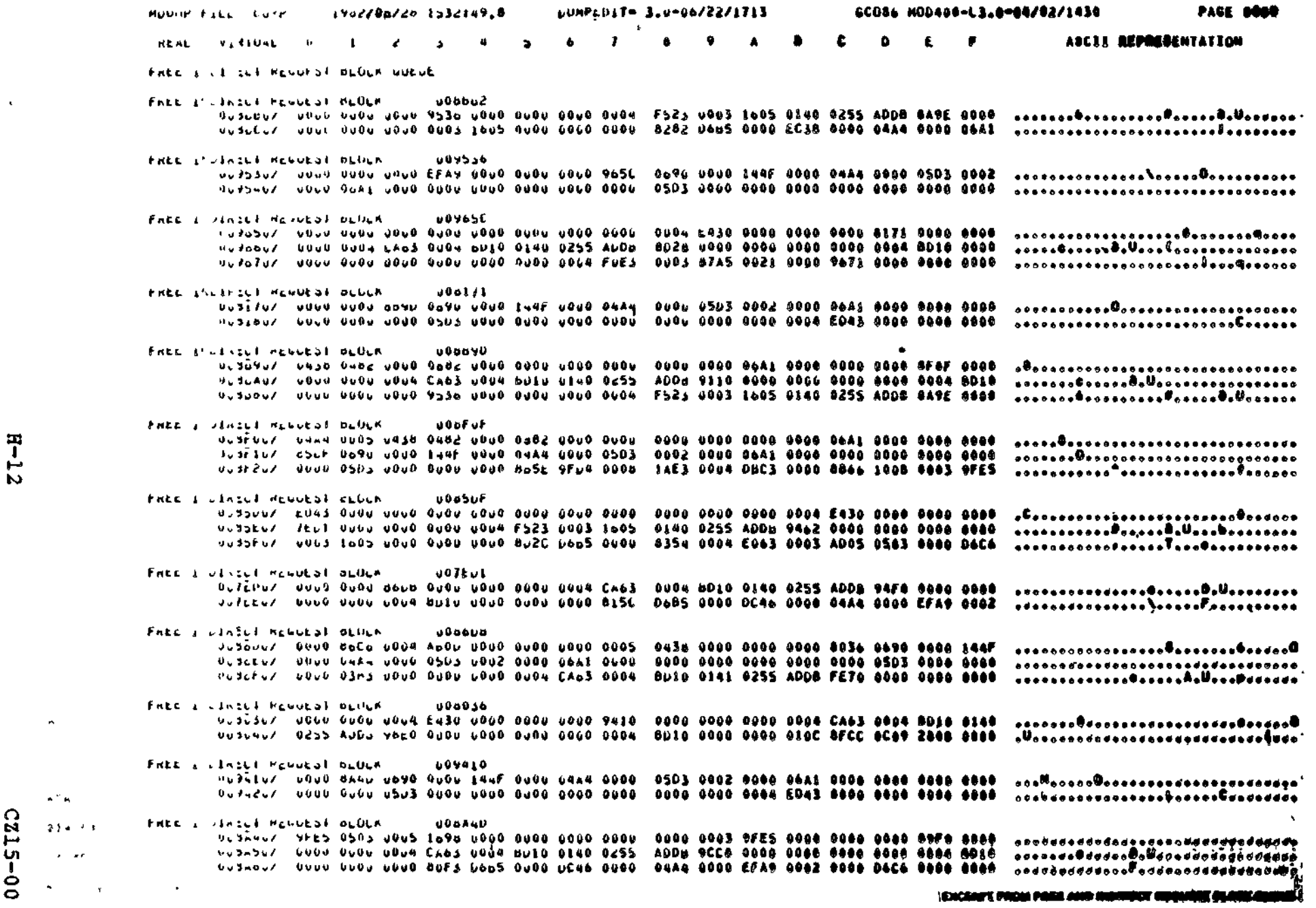
```
DUMP FILE FOR    1982/08/28 15:52:49.8      DUMPEDIT- 3.0-06/22/1713          GCOS6 MOD400-L3.0-06/02/1430          PAGE 0004

 REAL    VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F        ASCII REPRESENTATION

SINGLE SEG FOR SEGMENT POOL 03

MEMORY POOL DESCRIPTION 031040
        031040/  0001 0000 0000 0000 0000 0001 0000 0000  0000 0310 0002 0003 1045 0003 0FC3 0004  ..............................E.....
        031050/  AEE5 0003 0003 1180 001E 1033 0027 0FDE  1880 0000 0000 0000 0000 0000 1773 0186  ..........3.'.................0...
        031060/  0944 0E3F 004A 0051 0003 1A8E 2420 0003  1049 0000 0000 0000 0000 0000 0000 0001  .D.?.K.Q....$$...I..............

BIT MAP  031101
        031100/  FFFF FFFC 01C0 0000 0000 3F80 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ..........?.....................
        031140/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
                  *    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
        031180/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0E00 0070 0000 0000  ........................p.......
        0311C0/  0000 0000 0000 0000 0000 0000 0000 FF00  0000 0000 0000 0000 0007 FC01 FFFF 8000  ................................
        031100/  0000 0000 0003 FF80 07FF FFFF FFFF FFFF  F800 0000 0000 0000 0000 03FF FFFF FFFF  ................................
        031140/  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ................................
                  *    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
        031200/  C7FF F1FF FFF1 F8E0 7E00 0200 0000 2070  00FF 1E00 0000 00FE 00F0 0000 0000 03FF  ..............? ..............
        031240/  FFC0 FFFF C7FC 7FF0 3FC0 FFC7 07FF 1FF1  FF00 FFC0 FC11 C000 FC00 0000 0000 0000  .........?......................
        031280/  0000 F000 1C00 0000 0000 01C0 040F 8700  E016 1E80 3878 E200 0000 0030 0007 E0E0  .....?.......0.......0x.....0.....
        0312C0/  0008 0100 0200 C000 0000 0010 0000 0000  0800 0040 1000 0000 0000 0000 0010 0200  ..................@.............
        031300/  0000 2000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0004 0000 0002 0000  ..  ..............?.............
        031340/  0000 0000 0000 0FFF FFFF FFFF FFFF FFFF  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ................................
```

SEGMENT DESCRIPTORS

SEGMENT	BASE	SIZE	READ	WRITE	EXECUTE
0.0	000000	002000	3	0	3
0.1	001000	002000	3	0	3
0.2	002000	002000	3	0	3
0.3	003000	002000	3	0	3
0.4	004000	002000	3	0	3
0.5	005000	002000	3	0	3
0.6	006000	002000	3	0	3
0.7	007000	002000	3	0	3
0.8	008000	002000	3	0	3
0.9	009000	002000	3	0	3
0.A	00A000	002000	3	0	3
0.B	00B000	002000	3	0	3
0.C	00C000	002000	3	0	3
0.D	00D000	002000	3	0	3
0.E	00E000	002000	3	0	3
0.F	00F000	002000	3	0	3
1.0	010000	020000	3	0	3
2.0	020000	020000	3	0	3
3.0	030000	020000	3	0	3
4.0	040000	011800	3	0	3
5.0	050000	001800	3	0	3

```
SINGLE SEG FOR MEMORY POOL 04

MEMORY POOL DESCRIPTION 031073
        031070/  0000 0000 0000 000C 0007 0003 106F 0003  0EC8 0004 C503 0000 0000 0000 0510 0007
```


Figure H-1 (Cont). Memory Dump Example



```
MODIP FILE  CURR     1982/08/20 13:32:49.8      DUMPEDIT- 3.0-06/22/1713          GCO86 MOD400-L3.0-06/02/1430          PAGE 0000

 REAL   VIRTUAL   0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F          ASCII REPRESENTATION

FREE [illegible] REQUESTED BLOCK 000E6

FREE [illegible] REQUESTED BLOCK      000BB2
        0035B0A7  0000 0000 0000 953E 0000 0000 0004 0004  F523 0003 1605 0140 0255 ADDB 8A9E 0000  ........&.........#.....B.U.....
        0035C0A7  0001 0000 0000 0003 1605 0000 0000 0000  8282 D6B5 0000 EC38 0000 04A4 0000 06A1  .....................8..........

FREE [illegible] REQUESTED BLOCK      000536
        0035D0A7  0000 0000 0000 EFA9 0000 0000 0000 9651  0690 0000 144F 0000 04A4 0000 05D3 0002  ..............\.....O...........
        0035E0A7  0000 06A1 0000 0000 0000 0000 0000 0000  05D3 0000 0000 0000 0000 0000 0000 0000  ................................

FREE [illegible] REQUESTED BLOCK      000651
        0035F0A7  0000 0000 0000 0000 0000 0000 0000 0006  0004 E430 0000 0000 0000 8171 0000 0000  ..................0........q....
        003600A7  0000 0004 CA63 0004 BD10 0140 0255 ADDB  8028 0000 0000 0000 0000 0004 BD10 0000  .....c.....B.U...(.............
        003610A7  0000 0000 0000 0000 0000 0000 0004 F0E3  0003 87A5 0021 0000 9671 0000 0000 0000  ..................3...!....q......

FREE [illegible] REQUESTED BLOCK      000171
        003620A7  0000 0000 0000 0690 0000 144F 0000 04A4  0000 05D3 0002 0000 06A1 0000 0000 0000  ..........O.....................
        003630A7  0000 0000 0000 05D3 0000 0000 0000 0000  0000 0000 0000 0004 E043 0000 0000 0000  ..............................C......

FREE [illegible] REQUESTED BLOCK      000690
        003640A7  0430 0482 0000 0482 0000 0000 0000 0000  0000 0000 06A1 0000 0000 0000 8F8F 0000  .0..............................
        003650A7  0000 0000 0004 CA63 0004 BD10 0140 0255  ADDB 9110 0000 0006 0000 0000 0004 BD10  .......c.....B.U................
        003660A7  0000 0000 0000 9536 0000 0000 0000 0004  F523 0003 1605 0140 0255 ADDB 8A9E 0000  .......&.........#.....B.U.....

FREE [illegible] REQUESTED BLOCK      000FUF
        003F00A7  0484 0003 0438 0482 0000 0382 0000 0000  0000 0000 0000 0000 06A1 0000 0000 0000  .....0..........................
        003F10A7  C5CF 0690 0000 144F 0000 04A4 0000 05D3  0002 0000 06A1 0000 0000 0000 0000 0000  ..........O.....................
        003F20A7  0000 05D3 0000 0000 0000 8D5E 9FD4 0008  1AE3 0004 DBC3 0000 8866 1008 0003 9FE5  ..........^.............f.......

FREE [illegible] REQUESTED BLOCK      000650
        003D00A7  E043 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0004 E430 0000 0000 0000  .C..........................0......
        003D10A7  7EB1 0000 0000 0000 0004 F523 0003 1605  0140 0255 ADDB 9462 0000 0000 0000 0000  ..........B.....B.U....b........
        003D20A7  0003 1605 0000 0000 0000 0000 8D2C D6B5  0000 8354 0004 E063 0003 AD05 0583 0000  .............,....T....0........

FREE [illegible] REQUESTED BLOCK      007E01
        003E00A7  0000 0000 0000 0000 0000 0000 0004 CA63  0004 BD10 0140 0255 ADDB 94F0 0000 0000  ...............c.....B.U........
        003E10A7  0000 0000 0004 BD10 0000 0000 0000 8150  D6B5 0000 DC46 0000 04A4 0000 EFA9 0002  ..............\.....F....q......

FREE [illegible] REQUESTED BLOCK      000600
        003E20A7  0000 E0C0 0004 A000 0000 0000 0000 0005  0438 0000 0000 0000 0036 0690 0000 144F  ..................8.....6.....O
        003E30A7  0000 04A4 0000 05D3 0002 0000 06A1 0000  0000 0000 0000 0000 0000 05D3 0000 0000  ................................
        003E40A7  0000 03F3 0000 0000 0000 0004 CA63 0004  BD10 0141 0255 ADDB FE70 0000 0000 0000  ..........c.....A.U....p........

FREE [illegible] REQUESTED BLOCK      000036
        003E50A7  0000 0000 0004 E430 0000 0000 0000 9410  0000 0000 0000 0004 CA63 0004 BD10 0140  .......0..................c.....@
        003E60A7  0255 ADDB 9BE0 0000 0000 0000 0004 0004  BD10 0000 0000 010C 8FCC 0C09 2000 0000  .U..............................

FREE [illegible] REQUESTED BLOCK      009410
        003E70A7  0000 8A4E 0690 0000 144F 0000 04A4 0000  05D3 0002 0000 06A1 0000 0000 0000 0000  ...N.....O.....................
        003E80A7  0000 0000 05D3 0000 0000 0000 0000 0000  0000 0000 0004 E043 0000 0000 0004 0000  ..............................C.....

FREE [illegible] REQUESTED BLOCK      00BA4D
        003E90A7  9FE5 05D3 0005 1690 0000 0000 0000 0000  0000 0003 9FE5 0000 0000 0000 89F0 0000  ................................
        003EA0A7  0000 0000 0004 CA63 0004 BD10 0140 0255  ADDB 9CC8 0000 0000 0000 0000 0004 BD10  .......c.....@.U................
        003EB0A7  0000 0000 0000 80F3 D6B5 0000 DC46 0000  04A4 0000 EFA9 0002 0000 D4C6 0000 0000  .............F....q.............

[EXCERPT FROM PAGE AND MEMORY REQUIRED BLANK SPACE]
```

Figure H-1 (Cont). Memory Dump Example

REAL VIRTUAL 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII REPRESENTATION

SYSTEM SYMBOL TABLE

MODULE	SYMBOL	VALUE	
ZBEXEC			
ZWIDUM			
	ZUIMAX	009750	LOCATION
	ZGIMGL	007761	LOCATION
ZUEXEC			
	ZQESCB	00C524	LOCATION
	ZGEGCB	00C526	LOCATION
	ZUECIT	00C528	LOCATION
	ZGEMIT	00C52A	LOCATION
	ZUEMSD	00C52C	LOCATION
	ZUELSD	00C52E	LOCATION
	ZUEMLD	00C530	LOCATION
	ZGELLD	00C532	LOCATION
	ZGEMPD	00C534	LOCATION
	ZGELPD	00C536	LOCATION
	ZGEMND	00C538	LOCATION
	ZUELND	00C53A	LOCATION
	ZUEINC	00C53A	LOCATION
	ZUEXIT	00C575	LOCATION
	ZGEIN1	00C5B4	LOCATION
	ZGEMAP	00C5A2	LOCATION
	ZGEIN2	00C642	LOCATION
	ZUE001	00D354	LOCATION
	ZUE002	00D358	LOCATION
	ZUE003	00D361	LOCATION
	ZQE004	00D367	LOCATION
	ZGEICP	00C8AE	LOCATION
	ZGEREC	00C8EE	LOCATION
	ZGETIM	00C9E8	LOCATION
	ZUEUPD	00CFE3	LOCATION
	ZGEUPS	00CFE9	LOCATION
	ZQEUPX	00CFF0	LOCATION
	ZUEUPZ	000003	LOCATION
	ZUEPDU	00D07E	LOCATION
	ZUEPMU	00D09A	LOCATION
	ZUEDUG	00D1D5	LOCATION
	ZUEPST	00D0AB	LOCATION
	ZQEIN3	00D0CE	LOCATION
	ZUEINB	00D1FF	LOCATION
	ZQEGET	00D0E1	LOCATION
	ZQEPUT	00D10D	LOCATION
	ZGEENG	00D157	LOCATION
	ZQEDEQ	00D16D	LOCATION
	ZQENXT	00D1DC	LOCATION
	ZQEUNQ	00D249	LOCATION
	ZQEBKG	00D220	LOCATION
	ZGEBKK	00D228	LOCATION
	ZUEVNT	00D1FC	LOCATION
	ZQPLKI	00C52E	LOCATION
	ZUELKI	00D20C	LOCATION
	ZUEKUB	00C779	LOCATION
	ZUEKAP	00D2A7	LOCATION
	ZUEDLP	00D268	LOCATION
	ZQEPUC	0002FF	LOCATION

[illegible] THE SYSTEM SYMBOL TABLE

Figure B-1 (Cont). Memory Dump Example

```
 REAL   VIRTUAL  0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F        ASCII REPRESENTATION

RECORD LOCKING POOL CONTROL BLOCK    04E1E3
        04E1E0/  0001 0000 0000 0003 C223 0003 C225 0000  0000 001E 001E 0000 001E 001E 0000 0000  ..........#...%.................
        04E1F0/  0000 0000 0255 ADDA 71C2 0069 0001 E1F4  0004 E1F4 0004 E1F4 0000 0000 0000 0001  .....U..q..i....................

TREE OF VOLUME, DIRECTORY, AND FILE DESCRIPTION BLOCKS

                 VOLUME DESCRIPTION BLOCK
                 DEPTH  00       LOCATION  000BA8
                 LOGICAL RESOURCE NUMBER  0001
                 ******************************
        0003E0/  0000 0000 0000 0000 0000 0000 0000 0000  0000 98EF 0000 0000 0000 0C00 0000 0000  ................................
        0003F0/  0000 0000 0000 0001 0000 0000 0042 0000  0000 0000 2453 0000 0000 0001 0000 0000  ..............B.....$S..........
        000400/  4359 4C4F 4E20 2020 2020 2020 2020 0020  0100 0000 0008 0005 0000 0000 00A4 0000  CYLON         & . ............
        000410/  FFFD 0005 0005 0005 00A0 0000 0098 009C  0000 0000 0000 0000 0013 0000 00A8 0000  ................................
        000420/  0001 00A5 0000 0801 0100 0001 0000 0028  0100 0000 0028 0000 0000 0000 0008 0000  ...............(.....(..........
        000430/  0000 0003 7663 0000 0000 0000 4643 4D30  3020 2020 2020 2020 0000 000E 1100 0003  ....vc......FCM00         ........
        000440/  1920 0255 7CFC 88C6 0005 1263 0000 08A8  0000 0000 0000 0000 0000 0000 0000 0001  .*.U.|.....c....................

                 FILE DESCRIPTOR BLOCK
                 DEPTH  01       LOCATION  000C04
                 ******************************
        000450/  1920 0255 7CFC 88C6 0005 1263 0000 06A8  0000 0000 0000 0000 0000 0000 0000 0001  .*.U.|.....c....................
        000460/  0000 0000 0000 0100 0000 0000 4C37 0000  0000 0001 0000 0000 5A33 4950 4548 5350  ............L7..........Z3EXECUT
        000470/  4956 4556 0782 0100 0100 0000 0011 008C  0000 0000 00AE 0000 FFFD 008C 0001 0000  IVEL............................
        000480/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0014 00AF 0000 0801  ................................
        000490/  0100 0001 0000 045C 0100 0009 04A0 0000  0000 0000 0000 0000 0000 0000 0000 0000  .......\........................
        0004A0/  0000 0000 2020 C000 0003 0C6D 03D6 0100  0000 0125 0000 0000 0000 0000 0000 0000  .... ....m.....U................
        0004B0/  0000 0000 0000 0000 0000 0C04 0000 0000  8012 0000 0001 0000 0000 0000 0000 0000  ................................

                 DIRECTORY DESCRIPTOR BLOCK
                 DEPTH  01       LOCATION  051263
                 ******************************
        051260/  0003 0000 0000 0005 0FE3 0000 08A8 0005  1203 1800 0000 0000 0000 0000 0001 0000  ................................
        051270/  0000 0000 0008 0000 0000 4343 0000 0000  0001 0000 0000 404C 2020 2020 2020 2020  ..........CC..........@L
        051280/  2020 2620 0020 0100 0000 0AF8 0001 1808  0001 00AF 0000 FFFD 0002 0001 0000 0000  & . ............................
        051290/  0000 0000 0008 0000 0000 0000 0000 0005  0000 0000 0000 0010 0000 0000 0001 0100  ................................
        0512A0/  0001 0000 0009 00A0 0000 0010 0000 0000  0000 0000 0000 0000 0005 1743 0000 0000  ...........................C....
        0512B0/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................

                          FILE DESCRIPTOR BLOCK
                          DEPTH  02       LOCATION  051203
                          ******************************
        0511F0/  0003 0000 0000 0005 11A3 0005 1263 0000  0000 1800 0000 0000 0000 0000 0001 0000  ..............c.................
        051200/  0000 0000 0000 0000 0000 4343 0000 0000  0001 0000 0000 404C 4649 4C45 2020 2020  ..........CC..........@LFILE
        051210/  2020 4FB3 00FC 0200 0000 0AFC 0116 0001  0000 0020 0000 FFFD 0116 0002 0000 0000  .O.|.....|.....|.....|.....|....
        051230/  0000 0005 0000 0000 041E 0000 0002 0000  0000 0040 0001 0000 0000 0000 0001 0100  ...........|.....|.....|.....|....
        051240/  0002 0000 0458 0200 0000 0458 0000 0000  0005 136F 0000 0000 0005 1101 0000 1102  .....|.....|.....|.....|.....|....
        051250/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................

                                                                                EXCERPT FROM THE FILE SYSTEM INFORMATION
```

Figure H-1 (Cont). Memory Dump Example

```
MDUMP FILE  DUMP     1982/06/28 1532149.8      DUMPEDIT- 3.0-06/22/1713          GCOS6 MOD400-L3.0-06/02/1430          PAGE 0053

 REAL   VIRTUAL   0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F          ASCII REPRESENTATION

CHAIN OF PUBLIC BUFFER POOLS
 ********************************************************

BUFFER POOL NAME: SYS256
NUMBER OF BUFFERS: 0032
BUFFER SIZE (BYTES): 0100
NUMBER OF USERS (FILES): 0002

 BUFFER POOL HEADER FOR POOL SYS256  031CA3
        031CA0/  0003 0000 0000 0003 3003 0022 5359 5332  3536 0100 0032 0032 0002 0019 0000 0000  .!.....#..."SYS256....2.2.........
        031CB0/  0000 0003 2076 0003 2226 0000 0000 0000  0000 33A7 000B 6735 0000 0000 000C F316  .... v...".&.....3...g5..........
        031CC0/  0000 FF82 0000 DA4D 0000 0000 0000 1280  0000 999D 0000 0000 0000 0000 0000 0000  ........M.......................

 BUFFER CONTROL BLOCKS FOR POOL  SYS256  031CCE
        031CC0/  0000 FF82 0000 DA4D 0000 0000 0000 1280  0000 999D 0000 0000 0000 0000 0000 0000  ........M.......................
        031CD0/  0000 0000 0000 0000 0020 0003 23E3 0100  0000 001F 0000 0000 0001 0000 0000 0000  ........ ...#...................
        031CE0/  0000 909E 0041 0E01 0003 23E3 0100 C75F  0000 0000 0000 0000 0003 1E5A 0003 1E7E  .....A....#....._.........Z....~
        031CF0/  0000 0000 0000 0000 0000 0000 0000 0000  0027 0003 2463 0100 0000 0086 0000 0000  ..................'...$c........
        031D00/  0001 0000 0000 0000 0000 837E 0041 0E01  0003 2463 0100 CA36 0000 0000 0000 0000  ..............~.A....$c...6......
        031D10/  0003 1D3A 0003 1DA6 0000 0000 0000 0000  0000 0000 0000 001C 0003 24E3 0100 0000  ...:..........................$c...
        031D20/  0000 001C 0000 0000 0001 0000 0000 0000  0000 86F0 0041 0E01 0003 24E3 0100 C75B  ..........................A....$c...[
        031D30/  0000 0000 0000 0000 0003 208E 0003 2382  0000 0000 0000 0000 0000 0000 0000 0000  .......... ...#.................
        031D40/  0028 0003 2563 0100 0000 0027 0000 0000  0001 0000 0000 0000 0000 9360 0041 0E01  .(..%c.....'.............h.A...
        031D50/  0003 2563 0100 CA37 0000 0000 0000 0000  0003 202E 0003 1CF2 0000 0000 0000 0000  ..%c...7.......... .............
        031D60/  0000 0000 0000 0000 001E 0003 25E3 0100  0000 001D 0000 0000 0001 0000 0000 0000  ..............%.................
        031D70/  0000 821F 0041 0E01 0003 25E3 0100 C75D  0000 0000 0000 0000 0003 1E7E 0003 20BE  .....A....%....]..............~ .
        031D80/  0000 0000 0000 0000 0000 0000 0000 0000  003A 0003 2663 0100 0000 0039 0000 0000  ...................:..&c.....9....
        031D90/  0001 0000 0000 0000 0000 0431 0041 0E01  0003 2663 0100 0079 0000 0000 0000 0000  ...........1.A....&c...y........
        031DA0/  0003 200A 0003 1FE6 0000 0000 0000 0000  0000 0000 0000 0000 0026 0003 26E3 0100  .. ...............&.....&c...
        031DB0/  0000 002C 0000 0000 0001 0000 0000 0000  0000 8282 0041 0E01 0003 26E3 0100 CA35  ...,.....................A....&c...5
                                                        ***   REFER TO PHYSICAL DUMP   ***
        0323D0/  0003 1D1C 0003 1F0E 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................

 BUFFERS FOR POOL  SYS256  0323E3
        0323E0/  00C9 0000 0000 0100 0000 0000 0020 0000  801F 00FF 647C 1CFF 64AB C600 8587 64AF  ..............  .....d|...d.....d.
        0323F0/  E48C C7FF FE8D 44FF FE6C 03D3 8010 0000  0D09 0000 0000 8765 CC01 1000 002B 0CAB  ......D..l..........e.....+..
        0324D0/  C600 13AF C600 05CB C600 28A8 C600 0587  64AF E4F8 7010 60FF 648C C7FF FE8D 44FF  ..........+.....d...p.`.d.....D.
        0324E0/  FE6C 02D3 8010 0000 0D09 0000 0000 9EB0  0000 1000 0021 0CFF 4600 21CB C600 28AC  .l.....................!...F.!...+.
        0324F0/  C600 0587 64AF E48C C7FF FE8D 44FF FE6C  0103 8010 0000 0D09 0000 0000 C6DE 4D90  ....d.......D..l..............M.
        0325D0/  1000 000F 0CF8 4600 2198 C000 0A83 8010  0000 0D09 0000 0000 C744 CA16 1000 0027  ......F.!..............D.....'
        0325E0/  0C00 0000 0002 0C00 5000 0600 0AFF FAFF  F000 02F8 4600 2183 C000 4BCB C600 282C  ........P...........F.!...K...+,
        0325F0/  009B 8010 0000 090D 0000 02E3 1000 0015  0CAF 649F E48C C7FF FE8D 44FF FE6C 0103  ..............d.........D..l..
        0326D0/  0010 0040 0509 0100 0000 0000 0027 1000  000D 0900 0000 0083 C5A7 9210 0000 110C  ...@.........'..................
        0326E0/  05C0 0001 CBC6 000A 2C00 9B80 1000 0009  0D00 0003 3310 0000 150C AF64 9FE4 8CC7  ..........,.....3.......d......
        0326F0/  FFFE 8D44 FFFE 6C01 0380 1000 0009 0008  0011 9810 0000 110C 7901 FFEF C8C6 000A  ...D..l.................V.......
        0327D0/  2C00 9B80 1000 0009 0D00 0003 4A10 0000  150C AF64 9FE4 8CC7 FFFE 8D44 FFFE 6C01  ,.............J.....d.........D..l.
        0327E0/  0380 1000 0009 0D00 0000 00B3 C5A7 9210  0000 0D0C CBC6 000A 2C00 9B80 1000 0009  ..............,.....
        0327F0/  0D00 0003 EA10 0000 150C AF64 9FE4 8CC7  FFFE 8D44 FFFE 6C01 0380 1000 000D 0900  ..........d.....D..l.........
        0328L0/  0000 00C0 DE4D 5A10 0000 210C 9FC6 0DB3  C8C6 000A ACC6 8003 8764 AFE4 8CC7 FFFE  .....MZ...!.............d.......
        0328D0/  8D44 FFFE 6C01 0380 1000 000D 0900 0008  0061 C267 DA10 0040 110C C8C6 000A ACC6  .D..l..........a.B...@..........
                                                        ***   REFER TO PHYSICAL DUMP   ***
        0331E0/  69AE 6564 20A0 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ined m..........................

BUFFER POOL NAME: SYS512                                                                    BUFFER POOL INFORMATION
```

Figure H-1 (Cont). Memory Dump Example

```
 REAL   VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F        ASCII REPRESENTATION

 GLOBALLY SHAREABLE BOUND UNITS
 ******************************

                        BOUND UNIT DESCRIPTION   03B0C8
 0330C0/  0010 0000 0000 0005 0F03 0001 8001 0004  DEB3 0083 82B6 0063 0000 0003 C343 0000  ..............................
 0330D0/  0000 0000 0000 0003 C3A3 0000 0000 0000  0000 0000 0000 4445 4255 4720 0000 C84D  ......................DEBUG ...M

                        BOUND UNIT ATTRIBUTES    03B0D8
 0330D0/  0000 0000 0000 0003 C3A3 0000 0000 0000  0000 0000 0000 4445 4255 4720 0000 C84D  ......................DEBUG ...M
 0330E0/  0000 0800 0000 0000 8801 0005 0000 0627  0000 0000 A901 0013 04EF 0000 0000 0000  ...............'..............
 0330F0/  1101 002A 0000 009F 0000 0000 1101 0028  0000 00FD 0000 0000 1101 002A 0000 012F  ...*.............(.........../
 0331D0/  0000 0000 1101 0020 0000 0051 0000 0000  1101 002E 0000 00D8 0000 0000 1101 0030  ...... ...Q.............0
 0331110/ 0000 00CF 0000 0000 1101 0032 0000 009F  0000 0000 1101 0034 0000 004A 0000 0000  .......2.......4...J.....
 0331120/ 1101 0035 0000 006D 0000 0000 1101 0036  0000 0062 0000 0000 1101 0037 0000 00C1  ...5...m.......b.......7....
 0331130/ 0000 0000 1101 0039 0000 013B 0000 0000  1101 003C 0000 00C0 0000 0000 1101 003E  .......9...;.......<.......>
 0331140/ 0000 0075 0000 0000 1101 003F 0000 013C  0000 0000 1101 0042 0000 0000 0000 0000  ...u.......?...<.....B......
 0331150/ 1101 0044 0000 0057 0000 0000 1101 0045  0000 00C0 0000 0000 1101 0047 0000 0087  ...D...W.......E.........G....
 0331160/ 0000 0000 1101 0049 0000 0048 0000 0000  1101 004A 0000 009A 0000 0000 1101 004C  .......I...H.......J.........L
 0331170/ 0000 0085 0000 0000 1101 004E 0000 0060  0000 0000 1101 004F 0000 0058 0000 0000  .........N...`.....O...X.....
 0331180/ 1101 0056 0000 0135 0000 0000 1101 0053  0000 00C2 0000 0000 1101 0055 0000 00D0  ...V...5.........S.....U.....
 0331190/ 0000 0000 7101 0057 0000 0420 0000 0000  1101 0043 0000 0108 0000 0000 7101 0048  .....W... .........C.........H
 0331A0/ 0000 00D4 0000 0000 7101 0048 0000 009C  0000 0000 1101 006E 0000 00DE 0000 0000  ........H.......n...........
 0331B0/ 1101 0070 0000 00C3 0000 0000 1101 0072  0000 0045 0000 0000 7101 0073 0000 1C01  ...p.........r...E....q..s....
 0331C0/ 0000 0000 1101 00AE 0000 0109 0000 0000  1101 0081 0000 00AC 0000 0000 7101 0083  ..............................q....
 0331D0/ 0000 0107 0000 0000 1101 0087 0000 00D6  0000 0000 1101 00B9 0000 0137 0000 0006  ..............................7....
 0331E0/ 1101 008C 0000 0091 0000 0000 1101 008E  0000 00A8 0000 0000 1101 00C0 0000 0129  ...............................)
 0331F0/ 0000 0000 1101 00C3 0000 0046 0000 0000  7101 00C4 0000 0118 0000 0000 1101 00C6  ...........F....q...........F
 0332D0/ 0000 0082 0000 0000 1101 00CA 0000 0028  0000 0000 1101 00CB 0000 0128 0000 0000  ...............(.....(.....(....
 0332E0/ 1101 00CF 0000 0133 0000 0000 1101 00D2  0000 0138 0000 0000 1101 00D5 0000 0135  ...3.............8.....5.....5
 0332F0/ 0000 0000 1101 00D8 0000 0102 0000 0000  1101 00DB 0000 0130 0000 0000 1101 00DE  ...............0.....0.....0
 0333D0/ 0000 0087 0000 0000 1101 00E1 0000 00FE  0000 0000 1101 00E3 0000 010F 0000 0000  ..............................
 0333E0/ 7101 00E6 0000 006A 0000 0000 1101 00E8  0000 00EC 0000 0000 1101 00EB 0000 00E5  q.....j.....l...................
 0333F0/ 0000 0000 1101 00ED 0000 013B 0000 0000  1101 00F0 0000 00AE 0000 0000 7101 00F2  ...............;..............q....
 0334D0/ 0000 00F0 0000 0000 1101 00F5 0000 0139  0000 0000 1101 00F8 0000 0007 0000 0000  ...............9..............
 0334E0/ 7101 00FA 0000 0007 0000 0000 1101 00FD  0000 0025 0000 0000 1101 00FE 0000 013D  q.....%.....................=
 0334F0/ 0000 0000 1101 0101 0000 013A 0000 0000  1101 0104 0000 0112 0000 0000 1101 0107  ...............:..............
 0335D0/ 0000 009A 0000 0000 1101 0109 0000 0108  0000 0000 1101 010C 0000 0070 0000 0000  ...............p.....p.....p....
 0335E0/ 1101 010D 0000 0134 0000 0000 1101 0110  0000 013F 0000 0000 1101 0113 0000 003E  .......4.....?.....>.....>
 0335F0/ 0000 0000 1101 0114 0000 002C 0000 0000  95C9 0020 0000 0000 FFC1 AD29 FFFF FFFF  ...............,.... .....)....

                        BOUND UNIT:    DEBUG   03C343
 0336A0/  0045 0000 0000 5A33 4558 8543 0000 0000  DFE7 EFE7 9FE7 EF67 1981 8096 ECF8 FFED  .E....Z3EXC.....................
 0336B0/  ECC6 0034 08F8 0000 0DC6 0002 0901 008C  E8C6 0016 AFE7 8F67 DF67 8FE7 CF67 80C8  ...4.............g.g.....g...
 0336C0/  FFE2 0985 82C6 0007 8000 0505 3D01 890E  0F81 0968 3D01 0901 0005 AF67 CFE7 9058  .............h=........g.....X
 0336D0/  2C02 D3CE 00A7 CCF7 A877 ACC3 0008 D852  5906 82C2 0004 8000 0501 0854 580E 8756  ,.....w.........R.....TX..V
 0336E0/  ACC3 000C 8002 0982 ACD3 CDC2 000A 0396  CBC4 FFFF 8AD6 0FFA 6891 ABC2 800F 2F04  ..............h.....h...../.
 0336F0/  A8A2 E842 0002 688A 8256 ACC3 000A A892  ADD6 0280 8AD1 80D6 8FF9 ACC2 0000 3C0A  ...B..h..V..............<.
 0337D0/  580F 9CC6 0031 8981 0968 3909 82F9 05FE  93C8 083F 19F7 D958 0920 0FF8 3C0A 9CC6  X....1...h9....X. ....<.....
 0337E0/  002D 5803 9CC6 0033 8981 0909 3907 82F9  05FE 93C8 082E 191A 0FF3 3C0A 98C8 FF84  .-X....3.....9.....<.....
 0337F0/  1C03 C65A C911 0983 19FD 0F8C 9CC6 0028  5803 9CC6 0836 0968 0908 3C0A 0CF9 05FE  ...Z.....X....6.h..<.....
 0338D0/  0F85 C877 8CF7 A855 0F88 C877 8CF7 DFE7  D3CE 886C 0CF7 A899 8085 0886 C804 8077  ...w...U...w.....l.....w...
 0338E0/  8877 ACF7 8751 E877 9CF7 ECF7 DCF7 8385  3008 8C06 0837 8081 1842 8788 8781 C888  .w...Q.w.....7...B.....
 0338F0/  C1DF 0988 4D20 0903 2D06 027A 8751 C843  0003 8381 0F88 83C8 0401 0F10 0F0F 3008  ....M ..-..z.Q.C..............0.
```

Figure H-1 (Cont). Memory Dump Example

```
MDUMP FILE  DUMP     1982/06/28 153249.8      DUMPEDIT- 3.0-06/22/1715          GC086 MOD400-L3.0-04/02/1630  SS       PAGE 0058

REAL   VIRTUAL   0    1    2 — 3    4    5 — 6    7 — 8    9    A — B    C    D — E    F ——     ASCII REPRESENTATION

*****************
*   POOL  SS    *
*****************


SHAREABLE BOUND UNITS
*********************

                        BOUND UNIT DESCRIPTION   04AEC3
044EC0/  0002 0000 0000 0000 0000 0001 8801 0004  D6F3 0004 AEEA 0004 AED8 0004 AE83 0000  ................................
044ED0/  0000 0000 0000 0004 AE83 0000 0000 0000  0000 0000 0000 4147 5220 2020 0000 0000  ....................AGR     ....

                        BOUND UNIT ATTRIBUTES    04AED8
044EE0/  0000 8000 0000 0004 AE83 0000 0000 0000  0000 0000 0000 4147 5220 2020 0000 0000  ....................AGR     ....
044EF0/  0000 0029 0000 0000 8801 0001 0002 0029  0000 0000 6102 0000 7D06 0000 6101 0000  ...)...........)....a.....a...

                        BOUND UNITS     AGR      04AE83
044E80/  0002 0000 0000 0000 0000 ACF8 FFED C807  4D02 0384 9870 1702 0F9D 0905 A842 0002  ................M....p.......B..
044E90/  0F81 0006 9CC7 0303 C871 4984 9870 1712  0F91 4D02 0904 9870 1704 0F8C A801 A970  .........qI..p....M....p.......p
044EA0/  2453 0984 9870 1705 0F85 E870 0831 0001  0D07 A851 0001 0103 0602 0000 0400 0000  $S...p.....p.i.....q............

                                                                                                CURRENT MEMORY POOL:
                                                                                                  SHARABLE BOUND UNITS FOR POOL 05
```

Figure B-1 (Cont). Memory Dump Example

```
MDUMP FILE  DUMP     1982/06/28 1532149,8      DUMPEDIT- 3.0-06/22/1713          SCO86 MOD400-L3.0-04/02/1430  88  88   PAGE 0000

 REAL   VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F         ASCII REPRESENTATION

 TASK GROUP STRUCTURES FOR GROUP $$
 **********************************

 PERSON  IDENTIFICATION: OPERATOR
 ACCOUNT IDENTIFICATION: SYSTEM
 MODE    IDENTIFICATION: OPR
 BASE NAME/NAME LEVEL:  0000
 NAME OF MEMORY POOL USED BY GROUP:  $$
 BIT MAP OF EXTERNAL SWITCHES: 0001
 NUMBER OF OUTSTANDING REQUESTS THIS GROUP HAS MADE TO THE SYSTEM GROUP: 0000
 DIRECTORY DESCRIPTOR BLOCK FOR CURRENT_WORKING_DIRECTORY IS LOCATED AT: 0008A8      SIMPLE NAME IS  CYLON
 FILE CONTROL BLOCK FOR ERROR_OUT FILE IS LOCATED AT:  051433      SIMPLE NAME IS  CONSOLE
 FILE CONTROL BLOCK FOR  USER_OUT FILE IS LOCATED AT:  051433      SIMPLE NAME IS  CONSOLE
 SYSTEM TASK GROUP

 GROUP CONTROL BLOCK      000AA4
        000AA0/  C000 0000 0000 0000 0000 0000 2453 0000  0006 0004 A938 0000 0000 0001 1049 0003  ..............$S........8.......I..
        000AB0/  0E59 0000 0502 0000 0001 0000 0000 0000  0000 0000 2424 A031 8DA8 AD70 0000 C7A7  .Y..............$$.1...p....
        000AC0/  CAC1 0000 0000 AD3E 0000 0000 0BA8 0005  1433 0005 1433 0000 0000 0000 0000 0001  ........>.........3...3.........
        000AD0/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
        000AE0/  0000 0000 0000 0000 0005 00E0 0005 0800  0000 0800 0000 0000 0000 0000 082D 083F  .............................-.?
        000AF0/  0000 012C 0000 0000 0000 3191 0000 005A  0000 0001 0000 0000 0000 0000 0000 0005  ... .....1....Z.................

 LOGICAL RESOURCE TABLE  03UE59
        03UE50/  00D6 1965 0870 0140 0070 000C 0F81 FF47  0038 0000 D48F 0000 0827 0000 F197 0000  ...e.p.M.p.....6.8.......'.......
        03UE60/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 9F29 0000  ..............................).
        03UE70/  9993 0000 9A01 0000 9BCF 0000 9CED 0000  9E08 0000 9875 0000 9FE0 0000 A0FE 0000  ..........................u..........
        03UE80/  A21C 0000 0000 0000 A33A 0000 A3F1 0000  A4A8 0000 A55F 0000 A67D 0000 A798 0000  ..........:.....1.....K....=....
        03UE90/  46C9 0000 AD96 0000 AEA6 0000 B048 0000  B12D 0000 B281 0000 A9D7 0000 0000 0000  ..............K....-.....P......
        03UEA0/  0713 0000 08A9 0000 0A07 0000 0865 0000  DCC3 0000 DE21 0000 DF7F 0000 E0DD 0000  ..............e.....!...........
        03UEB0/  E23B 0000 E399 0000 E4F7 0000 E655 0000  E7B8 0000 E919 0000 EE4D 0000 EE8A 0000  .;...........U...+.......M.......
        03UEC0/  E7B3 0000 E911 0000 EA6F 0000 EBCD 0000  0000 0000 0000 8000 301F 8010 301F 8020  .........o..............0...0..

 LOGICAL FILE TABLE      000502
        0004F0/  0000 012C 0000 0000 0000 3191 0000 005A  0000 0001 0000 0000 0000 0000 0000 0005  ... .....1....Z.................
        000500/  0AC3 000F 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
```


Figure H-1 (Cont). Memory Dump Example

```
MDUMP FILE  DUMP     1982/06/28 153249.8      DUMPEDIT- 3.0-06/22/1713          GCO36 MOD400-L3.0-06/02/1430  SS  SS   PAGE 0060

 REAL   VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F          ABCII REPRESENTATION

 RESOURCE CONTROL TABLE INFORMATION FOR THE SYSTEM TASK GROUP


LOGICAL RESOURCE NUMBER: 0000
HARDWARE CHANNEL NUMBER: C0U4
DEVICE IDENTIFICATION:   F200
DRIVER INDICATOR FLAGS:  0200
LAST DEVICE STATUS WORD: 0000
TASK CONTROL BLOCK ADDRESS:  000F3E
ERROR LOG    BLOCK ADDRESS:
 RESOURCE CONTROL TABLE  000A0F
        000470/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
                   *    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
        000480/  0000 0000 003E 0000 CAFC 0000 04A4 0000  11F0 0000 BE41 BE61 4C45 0000 0F3E C00A  .....>..............A.aLE...>..


LOGICAL RESOURCE NUMBER: 0001
HARDWARE CHANNEL NUMBER: 0800
DEVICE IDENTIFICATION:   23B5
DRIVER INDICATOR FLAGS:  1440
LAST DEVICE STATUS WORD: 8000
TASK CONTROL BLOCK ADDRESS:  000882
ERROR LOG    BLOCK ADDRESS:  000000
 RESOURCE CONTROL TABLE  000827
        000700/  0004 8F00 0000 04A4 0000 0000 1461 0000  183C 0000 2A0A 0000 183C 01D5 001F 0002  .........1..*...<..*....<.......
        000710/  E64F 0000 14A5 0004 8F58 0000 04A4 0025  0006 0000 0000 183C 0000 2DF5 0000 11D9  .........X.....X.......<..-.....
        000720/  0000 105F 0000 0F3E 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ...._....>......................
        000800/  0000 0000 0000 0000 0000 0000 0000 0006  0000 0000 0000 06A1 0000 0F2A 0000 0D58  ..............................*...X
        000810/  0A17 0310 0000 0000 20F5 0000 000E 0000  0827 0217 0000 FFFE 0000 0000 0000 0000  ........-........'...............
        000820/  0000 0000 0000 0000 1308 0000 0862 0800  2385 1440 8000 0000 0000 0003 18FD 99CD  ..............#..B.............


LOGICAL RESOURCE NUMBER: 0002
HARDWARE CHANNEL NUMBER: 0025
DEVICE IDENTIFICATION:   0000
DRIVER INDICATOR FLAGS:  0000
LAST DEVICE STATUS WORD: 0001
TASK CONTROL BLOCK ADDRESS:  000F14A
ERROR LOG    BLOCK ADDRESS:
 RESOURCE CONTROL TABLE  000F197
        000F40/  0000 F14A 0025 0100 0000 0000 0000 0000  0000 5792 0000 FFFF 0703 0000 3586 6003  ...J.W.............W.........5.`.
        000F50/  0000 57F4 0000 01AD 0000 3584 0000 D2F9  0000 0000 0000 F144 0000 F139 0028 FC43  ..W.........5.............D...9.|.C
        000F60/  0009 0000 07AA 0000 002C 0025 FF00 FF00  FF00 FF03 FF20 FF00 FF00 0000 0000 0000  .........*.,.%..... ............
        000F70/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
                   *    *    *    *    *    *    *    *    *    *    *    *    *    *    *    *
        000F90/  0000 0000 0000 0000 0000 0000 F14A 0025  0000 0000 0001 0A60 0000 A841 000F 6933  ............J.%.......h....A...i3


LOGICAL RESOURCE NUMBER: 000A
HARDWARE CHANNEL NUMBER: 0800
DEVICE IDENTIFICATION:   2380
DRIVER INDICATOR FLAGS:  1440
LAST DEVICE STATUS WORD: 0000
TASK CONTROL BLOCK ADDRESS:  000882
ERROR LOG    BLOCK ADDRESS:  000000
 RESOURCE CONTROL TABLE  009F29
```

Figure B-1 (Cont). Memory Dump Example

```
REAL VIRTUAL  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F        ASCII REPRESENTATION

          TASK UNIT (TASK) NAME: AGR
          TASK UNIT (TASK) RELOCATION FACTOR: 04AE83
          TASK UNIT (TASK) DEFAULT START ADDRESS: 04AE85
          HARDWARE LEVEL: 0025
          LOGICAL RESOURCE NUMBER: FFFF
          BIT MAP OF ENABLED TRAP NUMBERS: 0000000000000000
          RESERVED OVERLAY AREA IS LOCATED AT: 000000
          CURRENT OVERLAY AREA IS LOCATED AT: 000000
          FILE CONTROL BLOCK FOR COMMAND_IN FILE IS LOCATED AT: 000000     SIMPLE NAME IS
          FILE CONTROL BLOCK FOR   USER_IN FILE IS LOCATED AT: 000000     SIMPLE NAME IS
          CLEARANT

          TASK CONTROL BLOCK     04A938    AGR
04A8E0/ 0005 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0004 A938 0025 0000 0006  ................8.%.....
04A8F0/ 0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0004 A8F8 0005  ................................
04A900/ 13E3 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
04A910/ 0000 0000 0000 0000 0116 0016 1180 0006  0000 0000 0000 0000 0004 AEC3 0004 AE85  ................^...^.....
04A920/ FFFF FFFF FFFF 0004 A898 0004 04A4 0000  0004 A927 0004 A927 0000 0000 0004 A938  ................^...^...^...8
04A930/ 0025 0100 0000 0003 0FC3 0000 0000 5D42  0000 FFFF 8703 0000 3586 6003 0004 AC9A  .%..........]B....5.`.....
04A940/ 0000 01A0 0000 3584 0000 0000 0000 0000  0004 A932 0004 A927 0000 818A C804 003A  ......5.........2...^.....:
04A950/ 0002 0000 030F 0025 FF00 FF00 FF00 FF03  FF20 FF00 FF00 0000 0000 0000 0000 0000  .......%..... ..... ........
04A960/ 0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
04A970/ 0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................

          TRAP SAVE AREA      005D42
          INSTRUCTION WHICH TRAPPED IS AN MCL AT LOCATION  002FAF.  FUNCTION CODE IS 0103.
          INSTRUCTION: 0001    P_COUNTER: 002F81  I'S 3F20   Z: 80C1     A: 002FAF  R3: 000F   B3:000708
04B040/ 0000 2495 0000 6702 3F20 000F 0001 80C1  0000 2FAF 0000 2F81 0000 6708 0004 ABE9  ..%...g.? ......../.../...g.....
04B050/ 0000 676A 0000 01A0 0000 04A4 0000 0335  003A 0002 FF00 0003 0000 0000 0000 0000  ..gj.....  ...5.:.g...........
04B060/ 0000 0000 0000 0000 0000 0000 0000 0004  A8C3 0000 0000 0000 0004 0001 0000 0000  ................C...........
04B070/ 0000 030F 0000 0008 0000 04A4 0000 04A4  0000 0000 0000 0030 3033 3046 0000 0000  ...............0030F....
04B080/ 3030 3030 3030 0000 0000 0012 0012 4141  4752 3A20 3137 2030 3330 4620 2832 3529  000000.........AAGR: 17 030F (25)
04B090/ 0000 04A4 818A C804 003A 0002 0802 030F  0000 2807 0000 0004 04F8 0000 04A4 0004  ....a...H....:.....(.....$...
04B0A0/ 4938 C804 818A 003A 0000 2003 0000 209A  0000 2080 0000 0000 3F01 005E 0001 80C1  I8.....:.....^.....?..^....

          MCL WORK SPACE      04A8E9
04B0B0/ 0006 0000 0000 0000 0000 0000 0000 0004  AC0F 00FC 0000 0000 0000 0005 0000 0000  ................,.|.....
04B0C0/ 030F 0000 0000 0000 04A4 0000 04A4 0000  0000 0000 0030 3033 3046 0000 3031 3030  ..........$.....0030F..0100
04B0D0/ 3030 3030 0000 0000 0035 0014 4153 5447  3A20 3137 2030 3330 4620 2832 3529 2020  0000.....5..ASTG: 17 030F (25)
04B0E0/ 5245 4645 5245 4E43 4520 544F 2055 4E41  5641 494C 4142 4C45 2052 4553 4F55 5243  REFERENCE TO UNAVAILABLE RESOURC
04B0F0/ 4545 2052 4553 4F55 5243 4563 0005 0000  1631 0001 0000 0000 1559 0004 AE80 0000  EE RESOURCEc.....1.....Y.....
04B100/ 04A4 0005 0005 05C3 0000 49E2 0005 05C3  0004 0000 2A0A 0000 0000 2A0A 0000 41D9  ..........C...I.....*.....*...A.
04B110/ 0005 1103 0005 1403 0005 17D3 0030 0000  2A0A 0000 41D9 0005 1423 0000 9807 0005  ..........C...*...A.....0.....
04B120/ 0000 4941 0000 0005 004A 0000 3807 0005  1423 0000 3A90 0000 4AA6 0000 388A F020  ..IA.....J.....C...*...J...8..
04B130/ 0000 35C0 0004 A4E6 0004 A4E6 0000 0000  4941 0000 003F 0000 1631 0001 0000 0000  ..5.....$...$...IA...?...1.....
04B140/ 1559 0004 A4C0 0000 04A4 0035 0000 3643  0000 183C 0116 0016 0002 F2AD 431A 0000  .Y.....$...6C...4.....C.....
04B150/ 0000 0000 0000 2666 0004 04F8 0000 04A4  818A C804 003A 0002 0802 030F 0000 2807  ......&.....$.....H...:.....(.
04B160/ 0000 0004 04F8 0000 04A4 0004 A938 C804  818A 003A 0000 2DF5 0000 209A 0000 2D8D  .....$.....$.....I8.H...:..5...^...-.
04B170/ 0006 0000 0000 0000 0000 0005 0F83 0000  0000 1000 0000 0000 0000 0000 0001 0000  ................................

                                                                                TASK STRUCTURES FOR AGR
```

Figure H-1 (Cont). Memory Dump Example

REAL VIRTUAL 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII REPRESENTATION

CHAIN OF WORK SPACE BLOCKS FOR THE CURRENT TASK GROUP
****

```
          WORK SPACE BLOCK        0500E0
053UE0/   0001 0005 06C0 0005 00EF 0000 95C9 00C5  0001 0004 CA2D 0005 0438 0000 0000 0000  ..........................-...8.......
053UF0/   0000 0001 FE02 0000 0000 0001 36C0 0004  CA23 0000 0000 0030 DBF9 0000 0000 2020  ..........#.....#.....0.....#.....0.....

          WORK SPACE BLOCK        0506C0
0536C0/   0001 0005 00C0 0005 06CF 0000 95B4 00C5  0001 0004 EBAD 0005 0438 0000 0000 0000  ..........................-...8.......
0536D0/   0000 0001 FE02 0000 0000 0004 FA63 0004  EBA3 0000 0000 0100 B0A1 0000 0000 2020  ..........#.....#.....0.....#.....0.....

          WORK SPACE BLOCK        0500C0
053JC0/   0001 0005 06C0 0005 00CF 0000 8FF6 00C5  0001 0004 CB2D 0005 0438 0000 0000 0000  ..... ..............-...8.......
053JD0/   0000 0001 FE02 0000 0000 0001 3503 0004  CB23 0000 0000 574F 5248 5320 5354 5220  ..........#.....#.....WORKS STR

          WORK SPACE BLOCK        0506D0
0536D0/   0001 0004 FAC0 0005 06DF 0000 9719 00C5  0001 0004 C9ED 0005 0438 0000 0000 0000  .........-.....................8.......
0536E0/   0000 0001 FE02 0000 0000 0004 D7C3 0004  C9E3 0000 0000 2020 2020 2020 2020 2020  ..........#.....#.....#.....#.....#.....

          WORK SPACE BLOCK        04FAC0
04FAC0/   0001 0005 0A60 0004 FADF 0000 66C6 00C5  0001 0004 A800 0005 0438 0000 0000 0000  .....`.....................8.......
04FAD0/   0000 0001 FE02 0000 0000 0001 3719 0004  A803 0000 0000 0000 0000 0000 0000 0000  ..........#.....#.....7.....#.....#.....

          WORK SPACE BLOCK        050A60
053A60/   0001 0004 CD00 0005 0A6F 0000 809F 00C5  0001 0005 032D 0005 0438 0000 0000 0000  .........#.....#.....-...8.......
053A70/   0000 0001 FE02 0000 0000 0001 34AA 0005  0323 0000 0000 0100 E3D5 0000 0000 2020  ..........#.....#.....#.....#.....

          WORK SPACE BLOCK        04CD00
04CD00/   0003 0004 F7C0 4120 4449 5220 2020 2020  2032 3536 2020 2020 2020 3230 2020 2020  .....A DIR      256      20
04CD10/   2020 3230 2020 2020 2030 0000 0000 0000  0048 0001 0001 0003 0003 5359 5332 3536    20      0.......H.........SYS256
04CD20/   0022 0032 0100 0032 0001 5359 5335 3132  0022 0032 0200 0032 0001 4449 5220 2020  .".2...2..2...2..SYS512."."2...2..DIR
04CD30/   01C0 0014 0100 0014 0000 0000 0004 CD34  0004 CD34 0004 CD34 0000 0000 0001 0001  . ..............#...#...#.....#.....
04CD40/   0004 0000 0004 0000 0000 0001 0001 0000  0000 0000 0100 0000 0402 0000 0000 0000  ..........#.....#.....#.....#.....
04CD50/   0000 0100 0000 0000 0000 0000 0000 0000  0000 0000 0000 0059 00AE 0000 0001 0100  ..........#.....#.....Y.....#.....

          WORK SPACE BLOCK        04F7C0
04F7C0/   0001 0004 FCE0 0004 F7DF 0000 9185 00C5  0001 0004 D54D 0005 0438 0000 0000 0000  .....................M...8.......
04F7D0/   0000 0001 FE02 0000 0000 0000 82AC 0004  D543 0000 0000 2020 2020 2020 2020 2020  ..........#.....#.....C.....#.....#.....

          WORK SPACE BLOCK        04FCE0
04FCE0/   0002 0003 80A0 0000 82AC 0001 0011 0000  0000 0000 3031 0000 0000 2300 0000 0000  ..........#.....#.....01....#.....
04FCF0/   0000 0000 0000 0000 0000 0100 8001 0000  0000 0001 0000 0000 0000 0000 0000 0000  ..........#.....#.....#.....#.....
04FD00/   0000 0000 0000 0000 2458 472E 2442 2020  2020 2020 0000 0000 0000 0000 0000 0000  ........$XG.$B          ............
04FD10/   0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ..........#.....#.....#.....#.....#.....

          WORK SPACE BLOCK        0300A0                                                    WORK SPACE BLOCKS USED BY SYSTEM GROUP 00
```

Figure H-1 (Cont). Memory Dump Example

REAL VIRTUAL 0 1 2 3 4 5 6 7 8 9 A B C D E F ABCII REPRESENTATION

```
**************
*  POOL L2   *
**************


SHAREABLE BOUND UNITS
**********************

                    BOUND UNIT DESCRIPTION   04C583
        04C580/ 0002 0000 0000 0005 12E3 0001 A801 0004  C4B3 0004 C580 0008 C598 000A 0000 0000  ................................
        04C5A0/ 0000 0000 0000 000A 0000 0008 E0A3 0000  0000 0000 0000 4142 4252 4556 0000 C000  ........................ABBREV....

                    BOUND UNIT ATTRIBUTES    04C598
        04C5B0/ 0000 0000 0000 000A 0000 0004 E0A3 0000  0000 0000 0000 4142 4252 4556 0000 C000  ........................ABBREV....
        04C5C0/ 0000 05CD 0000 0000 A801 0001 0000 023A  0000 0000 A801 0007 0000 05CD 0000 0000  ...............:................

                    BOUND UNITS    ABBREV  0A0000
05B200/ 0A3000/ C8C0 04FC 0001 0A00 1981 040F A870 0001  0001 0A01 1981 040F A870 0031 0001 0A01  ..............P.........P.1.....
05B210/ 0A3010/ 1981 04D3 A870 0005 0001 0A01 1981 04CD  4302 CBC6 006E 0001 1408 CBC6 0085 0001  .....P.........C.....P...........
05B220/ 0A3020/ 1400 2C00 0004 1C2E 9AC3 0004 00FF 92EF  1D2E 09FE 1C50 97EF 1C46 97EF 1C20 97EF  ..,..........P...F... ..
05B230/ 0A3030/ AF5E A883 2C00 B6C6 006E 3C00 92FF 1D20  09FE 1C3E 97FB 8AD3 0008 69FF C8C6 0085  .^..,....n<.... ...>........i.....
05B240/ 0A3040/ 0001 1020 9970 0209 099E C8C6 0010 0001  1030 1981 01E7 9846 0010 9F46 002E 9F46  ... .p.........g.....F....F....F
05B250/ 0A3050/ 002E C8C6 0002 0001 1050 1981 01D8 0001  1055 1981 01D7 C8C6 0028 0001 1025 1981  ..H.........P.....U.....H...(...%..
05B260/ 0A3060/ 0101 0FC6 1981 01C6 9846 0005 9F46 002E  9F46 0028 C8C6 002E 0001 1050 1981 01C2  .........F...F...F.H.....P.....B.
05B270/ 0A3070/ 8E46 0663 0002 0501 0473 8846 0068 0001  0501 FFF7 82C6 0068 0260 0581 004A 0001  .F.k.....s.F.k.....k.`...J..
05B280/ 0A3080/ 0506 82C6 0068 0040 0506 82C6 0068 0020  0515 0FA6 9846 0067 1EE8 82C6 0068 0200  .....k.@.....k. .....F.g.....k..
05B290/ 0A3090/ 0502 1EF3 1041 8ED1 C8C6 003E C894 5C14  D870 0014 0001 0504 0F93 9846 0067 1EF6  .....A.....>..\..P.........F.g..
05B2A0/ 0A30A0/ 82C6 0063 0200 0582 1EF5 1041 8ED1 C8C6  003E C894 5C09 D870 0009 0001 0505 82C6  ...k.........A.....>..\..P.........
05B2B0/ 0A30B0/ 0068 0200 0596 C846 0094 D846 0095 2C00  8046 0094 84D5 9846 0067 1EF6 1041 8ED1  .k.....F...F.....F.....F.g....A...
05B2C0/ 0A30C0/ C8C6 003E C894 5C09 0870 0009 0001 0505  ACF8 FFED 82C2 0004 0040 0596 8846 0068  ...>..\..P.................B...F.k
05B2D0/ 0A30D0/ 0400 0509 9846 006A 9570 003F 1012 0903  1D1A 098A C8C6 003E E846 0067 7C00 0001  .....F.j.p.?.........>.F.g.....
05B2E0/ 0A30E0/ 0903 1981 0402 A8C0 042A 2C00 88C6 011D  3C00 E870 0100 0008 69FF C8C6 011D E870  .........B.....*..P.....i.....p
05B2F0/ 0A30F0/ 0100 7C00 0001 0802 0D46 0069 9970 021F  0901 0045 1981 03E9 0001 0506 8D46 0094  .........F.i.P.....E.........F..
                                                        ***   REFER TO PHYSICAL DUMP   ***
0507C0/ 0A35C0/ 4A47 4849 4A4B 4C4D 4E4F 5051 5253 5455  5657 5859 5A7B 7C7D 7E7F 2041 5247 5320  FGHIJKLMNOPQRSTUVWXYZ.....-ARGS

                    BOUND UNIT DESCRIPTION   0512E3
        0512E0/ 0002 0000 0000 0004 F363 0003 8801 0005  0ED3 0005 1310 0005 12F8 000A 0000 0000  .........s......................
        0512F0/ 0000 0000 0000 000A 0000 0005 1183 0000  0000 0000 0000 414D 5420 2020 0000 0001  ......................AMT     ....

                    BOUND UNIT ATTRIBUTES    0512FB
        0512F0/ 0000 0000 0000 000A 0000 0005 1183 0000  0000 0000 0000 414D 5420 2020 0000 0001  ......................AMT     ....
        051300/ 0000 00E9 0000 0000 8801 0001 0000 00E9  0000 0000 1001 0003 0018 0105 0000 0000  ..................Y.............
        051310/ 0000 0100 0000 0000 0000 0000 0000 0000  0000 8018 0090 0001 0000 0000 0000 0000  ................................

                    BOUND UNITS    AMT     0A0000
055F00/ 0A3000/ EC80 0000 0018 9846 0040 0001 0000 8943  FFFF 00C0 0305 9870 0C01 8CF8 FFED 8CC3  .........F.@.....C.........P.........
055F10/ 0A3010/ 0030 80D3 0904 89C3 FFF8 0908 F870 006E  6C00 2C00 0001 0402 190C 8870 0017 F851  .0.............Pn!.,.........P....8
055F20/ 0A3020/ E870 0000 0001 0F00 A870 FFFF 0001 0103  9804 8057 C8C0 008A 0001 0A00 A870 0000  .P.........P.........W.........P...
```


Figure B-1 (Cont). Memory Dump Example

```
MDUMP FILE  DUMP     1982/06/28 153249.8      DUMPEDIT- 3.0-06/22/1713          GCOS6 MOD400-L3.0-04/08/1430  L2  L2   PAGE 0002

 REAL   VIRTUAL   0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F        ASCII REPRESENTATION

 TASK GROUP STRUCTURES FOR GROUP L2
 ********************************


 PERSON  IDENTIFICATION: EISENBERG
 ACCOUNT IDENTIFICATION: SDS
 MODE    IDENTIFICATION: INT
 BASE PROGRAMMER LEVEL: 0025
 NAME OF MEMORY POOL USED BY GROUP: L2
 BIT MAP OF EXTERNAL SWITCHES: 0000
 NUMBER OF OUTSTANDING REQUESTS THIS GROUP HAS MADE TO THE SYSTEM GROUP: 0002
 DIRECTORY DESCRIPTOR BLOCK FOR CURRENT_WORKING_DIRECTORY IS LOCATED AT: 04B6E3      SIMPLE NAME IS  LINK
 FILE CONTROL BLOCK FOR ERROR_OUT FILE IS LOCATED AT: 0601F3      SIMPLE NAME IS  TTY03
 FILE CONTROL BLOCK FOR  USER_OUT FILE IS LOCATED AT: 0601F3      SIMPLE NAME IS  TTY03


 GROUP CONTROL BLOCK     060BC3
       060BC0/  0003 0000 0000 0004 B523 4C32 0025 0004  0003 AD05 0003 AAC5 0003 1013 0006 00A8  .........#L2.E.%.........(....
       060BD0/  0006 C127 0002 0001 0000 0000 0000 0000  0000 4C32 6DA7 6E5E 6F03 0000 BE1F 0000  ...'............L2m.n^o.........
       060BE0/  0000 0000 A770 0000 0004 B6E3 0006 01F3  0006 01F3 0006 01C3 0000 00E8 0000 0000  .....p..........................
       060BF0/  0000 0000 0000 0004 F4E9 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................
       060C00/  0000 0000 0000 0006 0700 0008 0340 283C  8800 0000 0000 0000 0000 0000 0000 0000  ..............@(<................
       060C10/  0554 0000 1F01 0000 146A 0000 0108 0000  0000 0000 0000 0000 0000 0000 0000 0000  .2.......j......................


 LOGICAL RESOURCE TABLE  060A84
0579A0/ 0603A0/  0007 0000 0000 003C 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  .......<........................
0579B0/ 0603B0/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0000 0000  ................................

                  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *
057A10/ 060410/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0001 0000  ................................


 LOGICAL FILE TABLE      060127
057A10/ 060410/  0000 0000 0000 0000 0000 0000 0000 0000  0000 0000 0000 0000 0000 0000 0001 0000  ................................
057A20/ 060420/  0000 0000 0000 0000 0000 0000 0028 0000  0000 0000 0000 0000 0000 0000 0000 0000  .............(..................
```


Figure H-1 (Cont). Memory Dump Example

REAL VIRTUAL 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII REPRESENTATION

```
SOUND UNIT (TASK) NAME:  SYSSAV
SOUND UNIT (TASK) RELOCATION FACTOR: 0A0000
SOUND UNIT (TASK) DEFAULT START ADDRESS: 0A0000
MAINTAINE LEVEL:  0026
LOGICAL RESOURCE NUMBERS: FFFF
BIT MAP OF ENABLED TRAP NUMBERS: 0060000000000000
RESERVED OVERLAY AREA IS LOCATED AT: 000000
COMMENT  OVERLAY AREA IS LOCATED AT: 000000
FILE CONTROL BLOCK FOR COMMAND_IN FILE IS LOCATED AT:  060193      SIMPLE NAME IS  TTY03
FILE CONTROL BLOCK FOR    USER_IN FILE IS LOCATED AT:  060193      SIMPLE NAME IS  TTY03
COMMAND_IN FILE IS INTERACTIVE
DEFAULT
```

SEGMENT DESCRIPTORS

SEGMENT	BASE	SIZE	READ	WRITE	EXECUTE
6.0	057900	000900	1	0	1
8.0	06A300	001D00	1	1	1
A.0	050800	003000	1	1	1
B.0	07E200	002100	1	1	0

SEGMENT 6.0 MEMORY CONTROL BLOCK 04CEC3 GDB

```
04CEC0/ 0003 0000 0000 2000 0004 0004 0000 0000  0000 0000 0009 0001 0000 0000 0002 0579  ..... ...................Y
04CED0/ 0008 0001 0000 0000 0000 0000 8A38 0001  0003 0000 0000 0000 0000 0000 0000 0000  0.............8...................
04CEE0/ 0000 0000 0001 0000 0000 0000 0394 0001  0004 CEE2 0003 AA04 0000 0000 0005 0000  ................................
```

SEGMENT 8.0 MEMORY CONTROL BLOCK 04C863 GNB

```
04C8C0/ 0003 0000 0000 2008 0004 0004 0000 0002  0000 0008 0018 000A 0000 0000 0002 06A3  ..... ..................
04C8D0/ 0014 0001 0000 0000 0000 0000 93A7 0001  0003 0000 0000 0000 0000 0000 0000 0000  ................................
04C8E0/ 0006 0000 0001 0000 0000 0000 0014 0001  0004 C882 0003 AA04 0000 0000 0005 0006  ................................
```

SEGMENT A.0 MEMORY CONTROL BLOCK 050843 BU

```
0508A0/ 0001 0000 0000 2010 0001 0001 0000 0000  0000 0000 0038 0000 0000 0000 0A08 0588  ..... ..............8...........
0508B0/ 0037 0001 0000 0000 0000 0000 82EB 0001  0001 0000 0000 0000 0000 0000 0000 0000  .7..............................
0508C0/ 0001 0000 0000 2020 0001 0001 0000 0000  0000 0000 0021 0000 0000 0000 0800 07E2  ..... ..............!...........
```

SEGMENT B.0 MEMORY CONTROL BLOCK 050843 USER

```
0508C0/ 0001 0000 0000 2020 0001 0001 0000 0000  0000 0000 0021 0000 0000 0000 0800 07E2  ..... ..............!...........
0508D0/ 0C20 0001 0000 0000 0000 0000 080A 0001  0001 0000 0000 0000 0000 0000 0000 0000  . ..............................
0508E0/ 0002 0000 0000 0000 0000 0005 0A69 0001  0001 0051 0000 034A 4C4F 474F 5554 0000  ..................i...JLOGOUT..
```

TASK CONTROL BLOCK 034005 SYSSAV

```
034C00/ 0009 0000 0000 0003 A005 0026 8000 301F  8010 301F 8020 301F 8030 301F 8040 301F  ..........8...0...0...0... 0...00..00.
034C50/ 8050 301F 8060 301F 8070 301F 8080 301F  8090 301F 80A0 301F 80B0 301F 80C0 301F  .P0...`0...p0...0...0...0...0...0...0.
034C00/ 80D0 301F 80E0 301F 80F0 301F 8100 31FF  8200 31FF 8300 31FF 8400 3117 8800 3017  ..0...0...0...0...0...1...1...0...0.
034C70/ 0579 3008 0000 0000 86A3 001A 0000 0000  8588 0037 07E2 0C20 0000 0000 0000 0000  .Y0.....................7..... ........
```

TASK STRUCTURES FOR SYSSAV (CONT).:

H-24

CZ15-00

Figure H-1 (cont). Memory Dump Example

```
MDUMP FILE  DUMP      1982/06/28 1932:49.8      DUMPEDIT- 3.0-06/22/1713          GC086 MOD400-L3.0-06/02/1430  L2  L2   PAGE 0101

 REAL    VIRTUAL   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F        ASCII REPRESENTATION

         034C80/ 0000 0000 0000 0000 0004 CEC3 0000 0000 0004 C843 0000 0000 0005 0843 0005 0863 ..................C.....C...c
         034C90/ 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 ................................
         034CA0/ 0300 1A00 0003 8321 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 000D ........!....................
         034CB0/ 0000 0000 0000 0000 0000 0003 ACFF 0003 ACF4 0007 0026 0000 0008 0000 0000 0000 ..........................&...........
         034CC0/ 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0003 ACC8 0006 8223 0000 0000 ........................H....#....
         034CD0/ 0000 0000 0000 0000 0000 0006 0193 0006 0193 0006 0193 0060 0000 0000 0000 000A ........................`........
         034CE0/ 023E 0000 0000 9479 8DC7 0008 DBC3 0000 0000 0006 0303 000A 0000 0000 0000 FFFF .>.....y........................
         034CF0/ 0003 86E5 0004 DBC3 0000 0003 ACF4 0003 ACF4 0000 0000 0003 AD05 0026 0300 0020 ..........................&...
         034D00/ 0003 AC86 0000 0000 690A 0000 FFFF 8703 0000 3586 6003 0000 696C 0000 01AD 0000 ....F....i.........5.`...il......
         034D10/ 3584 0008 1AC3 0000 0000 0003 ACFF 0003 ACF4 0000 431A 0000 0007 0000 0000 0000 5.....................C...........
         034D20/ 0026 FF00 FF00 FF00 FF03 FF20 FF00 FF00 000B 08A9 0000 0000 0000 0000 0000 0000 .&.........  ..... ..... ....
         034D30/ 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 ................................
         034D40/ 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 ................................

                          TRAP SAVE AREA      0069UA
                 INSTRUCTION WHICH TRAPPED IS AN MCL AT LOCATION  002FAF.  FUNCTION CODE IS 0103.
                 INSTRUCTION: 0001    P_COUNTER: 002FB1  I': 3F24   Z: 80C1     A: 002FAF  R3: 000F   B3:006430
         005900/ 0005 0B03 0000 68A8 0004 FA9E 000C 0004 0004 000C 0000 642A 3F24 000F 0001 80C1 .......h.............d*?$........
         005910/ 0000 2FAF 0000 2FB1 0000 6430 0000 692C 0000 6492 0000 01AD 0004 DBC3 00C9 0008 ../.../...d0..i,..d.............
         005920/ 0007 0000 FF00 0003 0004 0000 0000 0000 0002 0008 FFFF FF00 0000 0000 0000 0000 ................................
         005930/ 0008 1A03 0000 2008 0000 0000 0009 0000 0000 030F 0000 0000 0000 0000 0000 0000 ..... ...........              ....
         005940/ 0000 0000 0000 0030 3033 3046 0000 0000 3030 3030 3030 0000 0000 0000 0003 AAC5 .......0030F....000000.........E
         005950/ 0000 8DC7 0004 DBC3 0000 431A 0000 0000 2834 0000 0000 8DC9 0000 8687 0000 282A ..........C.....C.....+4.....I...(*
         005960/ 1006 0008 1AC3 0003 AAC5 0004 DBC3 0003 AD05 0000 431A 0007 0000 2DF5 0000 2D9A ..........E.....C.....C.....-...-.
         005970/ 0000 2D80 0000 0000 3F20 FFFF 0001 8001 800A 0142 000A 0144 0008 0042 0000 4994 ..-.....? .........B...D...B..I.

                          TRAP SAVE AREA      006A2A

 ***  VIRTUAL ADDRESS IS INVALID:   1D0008

 ***  THE REAL VIRTUAL ADDRESS IS AT  006A3A
                 INSTRUCTION: 8F87    P_COUNTER: 002F93  I': 3104   Z: 0002     A: 0003C4  R3: 0008   B3:1D0008

 ***  VIRTUAL ADDRESS IS INVALID:   C90008

 ***  THE REAL VIRTUAL ADDRESS IS AT  006A42
                 POSSIBLE CONTEXT
                   R1, R2, R4, R5, R6, R7:  1060  0000  0008  0007  0002  0010
                   B1, B2, B4, B5, B6, B7:  C90008  000000  0003C8  0A1458  000398  000380
         005420/ 0004 BF2C 0000 63C8 0000 6C04 4000 0002 FFFF 0000 0000 0000 3104 0008 8F87 0002 ...C...c...l.d.............1.......
         005430/ 0008 03C4 0C0A 2F93 001D 0008 0000 644C 0008 038D 0008 8398 000A 145B 0008 03C8 ....../.......dL.............[....
         005440/ 0000 0000 00L9 0008 0004 0010 0002 0007 0006 0000 1060 FF00 0000 0000 0008 0000 ..................`...........
         005450/ 0008 1A03 0000 0008 0000 0000 0001 0002 0000 FFFF 0000 8000 8000 0000 49C5 0000 ..........................I...
         005460/ 0000 000B 0372 001D 8D09 00FF 0000 0000 0000 0026 0000 4539 0003 1CA3 0000 0000 .....r.............&...E9........
         005470/ 0000 0C0C 41C1 009B 000B 0002 53C4 0005 0000 0000 1631 0001 0000 0000 1559 0004 ....A.......$.........1.......Y..
         005480/ 00A0 0000 0404 0000 0000 0000 0000 24A9 0008 09B1 0000 642A 0000 0000 0000 0000 ..............$.......d*........
         005490/ 0000 000U 0000 0000 3F13 2020 0001 80C1 000A 0083 000A 0085 0004 CFE9 0000 64B4 ........?.  .................d.

                          GROUP UNIT DESCRIPTION   060303
 057C00/ 063300/ 0002 0000 0000 0000 0000 0001 2101 0006 0000 0006 032A 0006 031B 000A 0000 0000 .............!.........*...........
 057C10/ 063310/ 0000 0000 0000 000A 0000 0005 0843 0000 0000 0000 0000 5359 5353 4156 0000 0000 ..............C.........SYSSAV....

                                                                                                                    TASK STRUCTURES FOR SYSSAN
```

Figure H-1 (cont). Memory Dump Example

- Batch Group Data (shown if batch group is present)

  - Virtual address of beginning of background
  - Virtual address of the end of background
  - Rollout status (currently rolled out or not)
  - Number of completed rollout/rollin events
  - Size of background memory given to foreground

- Memory Pool Data*

  - Pool identification
  - Starting address of pool
  - End address of pool
  - Total size of pool
  - Physical start address
  - Total available space
  - Maximum contiguous available space
  - Number of available fragments (pieces) of pool space
  - Number of users
  - Table of attributes for each pool

- Additional pool information

  - Memory pool descriptor
  - Bit map (unless it is a queue managed pool)
  - Segment descriptors

- System Symbol Table

  The names and values of all symbols that have an entry in the system symbol table are displayed. Symbols are grouped according to the bound unit(s) in which they occur.

File System Structures

The logical dump displays the location and content of the following file system structures:

- Record locking pool control block
- Volume descriptor blocks (VDBs)
- Directory descriptor blocks (DDBs)
- File descriptor blocks (FDBs)

---

*Supplied for each memory.

An "X" appears beside a pool name that can cause the batch group to be rolled out.

The pool name for the batch group is BATCH.

- Currency control blocks
- Remote extent blocks
- Wait control blocks
- User control blocks
- Semaphore control blocks
- Record locking control blocks

- Device descriptor blocks (DDBs)
  - Buffer control blocks
- Public buffer pool headers (BPHs)
- Buffer control blocks (BCBs)
- Buffers.

The hierarchy of these structures is indicated by the dump as shown in Figure H-2, which is an abridged section of a logical dump. Each block is assigned an integer that corresponds to the level of the block in the hierarchy. The headings of all blocks are indented according to the depth of the block. This makes it easy to see which files belong to volume major directories and which belong to subordinate directories.

The display of the tree of file system structures may be suppressed by the -NF argument.

- Free indirect request block queue (only when editing a dump file)

- Globally sharable bound units

  - Bound unit description
  - Bound unit attributes
  - Bound unit

The preceding logical dump information is obtained from the operating system area of memory and occurs once within a logical dump. The following information can be repeated more than once depending on the number of active pools, task groups, and tasks. This information is presented in the following order:

1. Memory pools (as allocated at CLM time) if there are task groups assigned to them.

2 Task groups within a memory pool.

3 Tasks within a task group.

MEMORY POOL STRUCTURES

The following information is repeated for each pool with assigned task groups:

- Sharable Bound Units

  - Bound unit description

- Bound unit attributes
- Bound unit.

TASK GROUP STRUCTURES

The following information is repeated for each task group in a pool.

- Edited Task Group Information
  - User name, account id, and mode
  - Assigned memory pool
  - Bit map switches
  - Outstanding requests to system group
  - Address and name of control block for current working directory, error-out, and user-out
- Group control block
- Logical resource table
- Logical file table
- Task structures (detailed below)
- File control blocks (if there are active files)
- Work space blocks.

NOTES

For the system task group, IRBs (and hence also RBs) are displayed only when DPEDIT is processing a dump file; i.e., the display is suppressed when the input is from current main memory.

Work space blocks and FCBs for the batch task group are not displayed when the batch group is rolled out.

TASK STRUCTURES

The following information is repeated for each task in a group:

- Edited Task Information
  - Bound unit name, location, and start address
  - Hardware level
  - Logical resource number
  - Enabled trap bit map
  - Reserved and current overlay area locations
  - Control block name and address for user-in and command-in

- Segment descriptor table (swap pool only)
- Memory control block for each segment (swap pool only)
- Task control block
- Trap save area

  - MCL word space (for an MCL trap)

- Bound unit description
- Bound unit attributes
- Bound unit
- Overlay areas (if an overlay area table was used).

The firmware-defined fields (instruction, P-counter, I', Z, A, R3, and B3) for each trap save area (TSA) are displayed. If the instruction is a monitor call, the function code is also displayed.

In addition, a possible context of the remaining data and address registers (R1, R2, R4, R5, R6, R7, B1, B2, B4, B5, B6, and B7) is displayed for each trap save area. This context, which is extracted from the work space area of the trap save area, may not be valid in all cases but in general, is correct due to internal conventions of the Executive.

DPEDIT Command

The DPEDIT command loads the Dump Edit utility program. Immediately after Dump Edit begins executing, a message is issued to the error-out file giving the unique version number in the following format: DPEDIT-nnnn-mm/dd/hhmm. The message "DUMP COMPLETE" is issued to the error-out file immediately before the execution of Dump Edit terminates. The format for the DPEDIT command is:

```
DPEDIT [path] [ctl_arg]
```

ARGUMENTS:

path

Pathname of the memory dump file to be printed. Either the path argument or the -MEMORY control argument must be specified.

ctl_arg

Control arguments; zero, one, or more of the following control arguments may be entered, in any order:

{-NO_LOGICAL
-NL}

No logical dump of system control structures produced.

Default: Logical dump produced.

{-NO_PHYSICAL
-NP}

No physical dump of memory produced.

Default: Physical dump produced.

{-FROM X'address'
-FM X'address'}

Low-memory address of area that will appear in physical dump; must be specified in hexadecimal. The specified address must be a virtual address if processing memory, and a physical address if processing a dump file.

Default: Absolute 0.

-TO X'address'

High-memory address (up to five hexadecimal digits) of area that will appear in physical dump; must be specified in hexadecimal. The specified address must be a virtual address if processing memory, and a physical address if processing a dump file.

Default: High memory address of the dump file.

{-MEMORY
-MEM}

Produces a dump of main memory. If both the path argument and this argument are specified, an error message appears at the terminal. If the -FROM (and/or -TO) control argument is used in conjunction with the -MEMORY control argument, then the address that is specified must be a virtual address.

Default: A dump is produced of the file specified in the path argument.

{GROUP}
{-GP } group id [group-id] ...

Requests the logical dump to contain task group-related information for the specified group(s) only.

Default: Task group information for all groups is included in the logical dump.

{-NO_FILES}
{-NF }

No tree of file management structures is produced.

Default: A tree of file management structures is produced.

-ME

Dump only the group in which DPEDIT is running in the logical dump. Suppress all system information. This is equivalent to: DPEDIT -MEM -NO_SYS -NP -GP my_group-id

-NS

Do not dump the sharable or globally sharable bound units in the logical dump.

-NO_SYS

Do not dump the system area in the logical dump.

-PSYS

Limit the physical dump to the system area.

-FORCE

If the error "DUMPFILE IS INCOMPLETE" (defined below) appears, this argument causes DPEDIT to ignore this condition and to try to process the file anyway. Note that since part of the memory image is missing, it may not be possible to get a logical dump.

NOTE

Either the path argument or the -MEMORY control argument must be specified.

Example 1:

```
DPEDIT ^DMPVOL>DUMPFILE -NL -TO X'3000'
```

This command loads the Dump Edit utility and requests only a physical dump of the first 12K locations of the specified dump file.

Example 2:

```
DPEDIT -MEM
```

This command loads the Dump Edit utility and requests a logical and physical dump of current main memory.

Example 3:

```
DPEDIT -MEM -GROUP $S $D -NP -NF
```

This command loads the Dump Edit utility and requests a logical dump of only the System and Debugger groups from current main storage. This command suppresses display of the file management structures.

Example 4:

```
DPEDIT -MEM -GROUP XX -NP -NF
```

By specifying a group that does not exist (i.e., XX) this command requests an abbreviated logical dump consisting of only the System Summary of the currently executing system.

Operating Procedure for Dump Edit

The following steps must be performed before the Dump Edit program can be executed.

1. Mount the disk volume containing Dump Edit.

2. If Dump Edit is being used to print MDUMP output, mount the disk volume that contains the memory image obtained from the MDUMP memory dump.

3. Execute Dump Edit by specifying the DPEDIT command described previously.

DPEDIT processing can be stopped at any time by pressing the "BREAK" key. A **BREAK** message appears on the user's terminal display when processing stops. A GCOS 6 command may be specified at this point. If the Unwind (UW) command is specified, the end-of-processing details are automatically handled and control returns to the command processor with a successful subtask completion status. If the Start (SR) command is specified, DPEDIT resumes processing.

If DPEDIT appears to be looping, the loop can usually be broken and DPEDIT can be made to recover by forcing a **BREAK** and entering the Program Interrupt (PI) command. Note, however, that it is normal for DPEDIT to run for five or ten minutes while dumping a large memory or dump file.

DPEDIT Error Messages

Fatal errors terminate DPEDIT processing, return control to the command processor, and post an unsuccessful subtask completion status. Fatal errors include logical I/O errors and physical I/O errors as well as DPEDIT-specific errors. Fatal error messages are written to the error-out file. Error messages specific to DPEDIT are listed below. Additional information on error messages can be obtained in the System Messages manual.

Immediately after execution of DPEDIT begins, and immediately before execution terminates, a message is written to the error-out file. These messages are explained in the description of the DPEDIT command.

Informational messages that generally reflect some condition peculiar to the data within the dump file may be interspersed with the dump information in the user-out file. These messages are provided to facilitate analysis of the dump and are listed below. A brief explanation of each message is provided. "^" in a message indicates that a parameter is supplied.

-MEM AND PATHNAME NOT ALLOWED ON SAME INVOCATION

Memory and dump file can not both be processed during a single invocation of DPEDIT.

ARGUMENT NOT RECOGNIZED

An invalid argument was given in the DPEDIT command line.

ATTEMPT TO INCREMENT A VIRTUAL ADDRESS BEYOND FFFFF

An internal error has occurred; the memory block dump routine has incremented beyond the largest virtual address.

DPEDIT CONTINUES AFTER A PI OR TRAP. P: ^ I: ^ LOAD ADR: ^

DPEDIT has trapped or a break, program interrupt has been executed. The P-register, I-register and load address at the time of the interruption are displayed and DPEDIT recovers.

DPEDIT MUST EXECUTE IN THIS POOL TO DUMP THIS STRUCTURE FROM MEMORY

Because DPEDIT is executing in a different memory pool, it does not have visability to the structure. Either execute DPEDIT from the current pool or take an MDUMP.

DUMPFILE IS INCOMPLETE

Either MDUMP did not complete properly or the dump file was too small to hold the complete memory image (see the -FORCE argument).

DUMPFILE IS INCORRECT FILE TYPE

The dump file must be a non-UFAS relative file

ILLEGAL NUMBER OF ARGUMENTS

Too many group names follow the -GROUP argument.

LAST VALID DUMP LOCATION REFERENCED: ^

Indicates the last valid dump address processed before an invalid dump address was found.

NEED MOD400 REL2.1 DPEDIT TO PROCESS THIS DUMPFILE

A release 3.0 version of DPEDIT has accessed a release 2.1 (or earlier) MDUMP file.

NULL BUD POINTER IN THE TCB

The pointer to the bound unit description in the task control block is null.

NULL LINK IN THE ^QUEUE

A null link was found in the specified hardware queue.

PHYSICAL ADDRESS IS NOT IN PHYSICAL MEMORY: ^

DPEDIT has encountered a physical address which is higher than the highest physical address of the system being dumped.

REQUIRED ARGUMENT MISSING

The address has not been specified for the -TO or -FROM argument.

THE BAD VIRTUAL ADDRESS IS AT ^

An invalid virtual pointer was encountered by DPEDIT at the specified address.

THE SEGMENT IS ^ IN MEMORY

DPEDIT has found an invalid segment descriptor and the segment is or is not currently in memory, as indicated.

THERE WERE ERRORS DURING THE EDIT

If the output of DPEDIT was directed to a file, errors that tend to appear frequently are only written on the file. If the errors occurred during the dump, this error is issued to the user's terminal.

THIS ADDRESS DOES NOT FALL WITHIN THE DUMP FILE: ^

The specified address is not within the scope of the dump file.

THIS BOUND UNIT WAS PREVIOUSLY DUMPED IN ^

The bound unit was previously dumped in the specified group or pool.

THIS SWAP POOL STRUCTURE CANNOT BE DUMPED FROM MEMORY

DPEDIT does not have visability to the current structure. An MDUMP is required.

VIRTUAL ADDRESS EXCEEDS PHYSICAL MEMORY: ^

The specified virtual address represents a physical address which exceeds the highest physical address in the system being dumped.

VIRTUAL ADDRESS IS INVALID: ^

The specified virtual address exceeds FFFFF.

VIRTUAL ADDRESS NOT FOUND FOR ^. DUMP FILE IS SUGGESTED.

During a physical dump of memory, the specified physical address could not be translated into a valid virtual address for DPEDIT. An MDUMP is needed.

VIRTUAL ADDRESS OFFSET EXCEEDS SEGMENT SIZE: ^

The specified virtual address exceeds the segment size in the corresponding segment descriptor.

VIRTUAL ADDRESS REFERENCES INVALID SEGMENT: ^

The segment descriptor for this specified virtual address is invalid.

INTERPRETING AND USING MEMORY DUMPS

This subsection describes significant locations in memory dumps, how to interpret the contents of locations on memory dumps, and how to use memory dumps to perform the following procedures:

- Finding the location in memory of your code
- Determining where a trap occurred

- Determining the state of execution of your code.

A trap is a special software- or hardware-related condition that may occur during the execution of a task. Many traps are caused by an error, but a few, such as the Monitor Call, are not. The above procedures may have to be performed if a trap message is issued. Traps and trap messages are described in detail in the System Programmer's Guide, Volume I.

SIGNIFICANT LOCATIONS ON MEMORY DUMPS

Table H-2 describes memory locations on the dump that may be useful to refer to during debugging. It is assumed that you are familiar with the data structures referenced. Brief definitions of these data structures are contained in the glossary of the System Concepts manual. Figure H-2 illustrates a map of systems data structures.

Table H-2. Significant Locations on Memory Dump

Memory Address	Meaning
0010/0011	Head of queue of available trap save areas (TSAs).
0018/0019	Pointer to system control block (SCB). This is the key to locating all system data structures.
0020-0023	Level activity flags for levels 0 through 63. Bits ON indicate which levels are ready to execute; the lowest (numerically) of these levels is the level currently executing (i.e., the active level). The level 63 bit always is on. The clock level bit (4) may be on, and the debug level bit is on if the dump resulted from a Multiuser Debugger or a $D DEBUG DP directive.
0024-007F	Trap vectors. Each trap vector is associated with a specific trap condition and points to that trap handler's entry address. The trap vector for trap number 1 is in location 007F (7E/7F). The trap vectors for subsequent trap numbers are in descending, contiguous locations; i.e., the trap vector for trap number 2 is in location 007.
0080-00FF	Pointers to interrupt save areas (ISAs) for levels 0 through 63, respectively. A null value means there is no dedicated task (i.e., driver) or nondedicated task ready to execute on the specified level.

Figure H-2. Data Structure Map

Locations Relative to the System Control Block or Group Control Block

SCB+6/7

Pointer to the group control block (GCB) queue

GCB+0/1

Pointer to next GCB in linked list of GCBs

GCB+2/3

Task group identification ($S is the system group; $B is the batch group). The system will convert your user identification to non-ASCII representation.

GCB+D/E

Pointer to LFN 0 of logical file table (LFT).

GCB+B/C

Pointer to LRN 0 of task group's logical resource table (LRT).

GCB+5/6

Pointer to first task control block (TCB) of the group.

LRT-1

Number of entries in the LRT.

LRT+0/1

Pointer to LRN 0's resource control table (RCT); the RCTs for subsequent LRNs are in contiguous, ascending locations (LRT+1 points to LRN 1's RCT). A null entry indicates that the associated LRN is not used.

NOTE

Within an RCT, location 0 is the channel number of the resource if it is an input/output device.

RCT-2/-1

Pointer to task control block (TCB) for that resource.

Locations Relative to the Task Control Block (TCB) Pointer of the Desired Priority Level

TCB-8

Hardware-assigned priority level of the task.

TCB-1C/-1B

Pointer to current bound unit BUD.

TCB-10/-F

Pointer to top of queue of requests for the task.

TCB-E/-D

Pointer to end of queue of requests for the task (e.g., I/O requests for a driver).

TCB-13/-12

Pointer to the group control block (GCB) for the group to which this task belongs.

TCB-D -15/-14

Pointer to next TCB in this group.

TCB-A/-9

Pointer to last TCB on this priority level.

TCB-C/-B

Link to other task control blocks (TCBs) of the same or different task groups assigned to the same level.

TCB-2/-1

Pointer to the queue of trap save areas (TSAs) for the task. (Trap save areas are described in detail in the System Programmer's Guide.) If a TSA is present, the task is executing system code or a user trap; if no TSA is present, check the program counter in the interrupt save area (ISA) portion of the TCB to determine the task's progress.

TCB+0

Device word, including channel number and level number. This entry is null if the task does not drive a device.

TCB+1

Hardware interrupt save area.

INTERPRETING THE CONTENTS OF A DPEDIT LOGICAL DUMP

This subsection describes memory dump interpretation when the DPEDIT logical dump format is used.

Finding the Location in Memory of Your Code

Locate your group-id and the TCB for your bound unit (BU). The first six characters of the BU filename are printed beside each TCB of the group in a logical dump.

The address at TCB-1A/-19 is the address of the bound unit (BU) description. The load address of the bound unit is found at this address +A. Calculate relative zero of the BU by subtracting the relative start address on its link map from this address.

Determining the State of Execution of Your Code at the Time of the Dump

Dump analysis begins with gathering all relevant information: the dump itself, the console hard-copy (if any) of the activity of a particular group (or groups), copies of the CLM_USER and >START_UP.EC files, plus any link maps.

These materials are required to understand the environment of the system represented in the dump.

Three conditions are discussed below:

1. Halt at level 2
2. User level active at the time of dump
3. No level active at the time of dump, except level 63.

HALT AT LEVEL 2

Examination of the level activity indicators at locations 20-23 confirms that level 2 is active. The system will force this condition to occur if either TSA or IRB resources are exhausted (see CLM SYS directive). Note that once level 2 becomes active, other lesser priority levels may activate but will not receive CPU time.

The D1 register contains an ASCII "IR" (4952) when IRB exhaustion has occurred. Location 10/11 is zero when TSA exhaustion has occurred.

If this symptom persists after augmenting the number of TSA/IRBs available to the system, it is possible that either your code or the system is improperly altering the TSA/IRB chains.

To verify this, take a memory dump immediately after system startup. This allows easy location of the TSA chains from location 10/11 and the IRB chains from the first location of the SCB. Compare this dump to one taken after all TSA/IRBs are supposedly exhausted to verify that they really are. If the system is suspect, supply both dumps to Honeywell. TSAs can also be exhausted by a recursive trap. A recursive trap uses up all available TSAs. Adding TSAs simply allows for greater recursion. In this instance, the system is suspect and dumps should be supplied to Honeywell.

The optionally configured defective-memory trap handler may also force a level 2 halt if a defective memory trap indicates the operating system's trap save area is exhausted. In this case, $R1 will contain X'DEFA'; $B1, the physical address of the defective memory; and $B2, the logical address of the defective memory.

USER LEVEL ACTIVE AT THE TIME OF DUMP

This often indicates a halt or software loop condition on the active level. When a level is active, the pointer to the TCB associated with the code running is in the interrupt vector for that level. Match the TCB pointer with the TCBs listed for the groups present in the system. When a level is active, use the P-counter in the ISA portion of the TCB to locate the software running at the last time this level's context was saved. Since the system clock is active on level 4, the P-counter in the ISA for this level is usually helpful. It is also helpful to record the contents of R/B registers and E0 when entering STEP mode at the control panel prior to taking the dump.

NO LEVEL ACTIVE AT THE TIME OF DUMP

This condition usually indicates a system failure in that all tasks have been suspended and none are being reactivated. In this situation it is helpful to determine the conditions existing at this time. To do this, examine all TCBs in groups other than the $S group. If the TCB under examination has not experienced a default trap condition, it may or may not have an associated TSA. If a TSA is shown, DPEDIT will display the monitor call function code if the trapped instruction is 0001 (monitor call generic).

When the system is called for a monitor function, only those registers that must be preserved by the system are saved in the TSA workspace. The saved registers are: B7, B6, B5, B1, R5, R4, M1, beginning at TSA location E/F.

Determining Where a Trap Processed by the System Default Handler Occurred in Your Code

If a trap message occurs on the operator terminal from the system default trap handler, i.e., (id) BUname (0303zz) level, the TCB of the referenced task group may be located using the bound unit name (BUname). In this situation, unless the TCB is subsequently requested, the last two areas associated with the TCB are related to the system handling of the trap. The first TSA following the TCB was used by the system to forcibly terminate the task request in progress when the trap occurred. Your information is found in the next TSA associated with the TCB. It contains the hardware information described in the previous section of this appendix, followed by a complete set of registers current when the trap occurred. The order of the registers, beginning at location E/F of the TSA, is: B7, B6, B5, B4, B2, B1, I, R7, R6, R5, R4, R2, R1, M1 (B3, R3, I are already in the TSA). When the TCB has been rerequested, only this second TSA remains attached to the TCB.

FINDING THE LOCATION IN MEMORY OF YOUR CODE

The three activities above may be performed from the DPEDIT physical dump presentation. The examination of TCB contents is the same once the TCB is located. Use the following procedure to find the TCBs for your group.

1. Go to location 0018/19; this location contains a pointer to the system control block (SCB).

2. Go to location SCB+6/7; this location contains a pointer to the group control block (GCB) queue; GCB+0/1 links to the next GCB in the queue. Determine the group id at GCB+2/3 is your group id.

3. Go to location GCB+4 (+5/+6) to determine the location of the task control block (TCB) queue of the task group.

4. Go to location TCB-1C/-1B to determine the location of your current bound unit descriptor (BUD).

5. Go to location BUD+A/B). This location is the relocation factor of the bound unit; your code should start at this location.

6. Go to location BUD+8/9; this location points to the location of the bound unit attribute section (BAS).

7. Go to location BAS+0 to determine the bound unit's root name; this name should be the same as the bound unit's file name.

8. If you did not find the root name for which you were looking, go to location TCB-15/-14; this location points to the next TCB of the task group. Follow through the chain of TCBs until you find your task's task control block.

PRINTING AN INCOMPLETE MEMORY DUMP

By specifying the DPEDIT command with the -FORCE argument an incomplete memory dump may be printed. See the DPEDIT command definition earlier for information on requesting the incomplete memory dump.

INDEX

PLEASE FOLD AND TAPE—
NOTE U S Postal Service will not deliver stapled forms

FOLD ALONG LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO 39531 WALTHAM MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN PUBLICATIONS, MS486

FOLD ALONG LINE

Honeywell

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO 39531 WALTHAM MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

FOLD ALONG LINE

FOLD ALONG LINE

Honeywell