Honeywell

LEVEL 6

SOFTWARE

GCOS 6 PROGRAM PREPARATION

SERIES 60 (LEVEL 6)

GCOS 6 PROGRAM PREPARATION

SUBJECT

Detailed Description of Series 60 (Level 6) GCOS 6 Program Preparation Procedures

SPECIAL INSTRUCTIONS

This manual supersedes CB01, Rev. 0, dated January 1978. Change bars indicate new and changed information; asterisks denote deletions.

SOFTWARE SUPPORTED

This publication supports Release 0110 of the Series 60 (Level 6) GCOS 6 MOD 400 Operating System; see the Manual Directory of the latest *GCOS 6 MOD 400 System Concepts* manual (Order No. CB20) for information as to later releases supported by this manual.

ORDER NUMBER

CB01, Rev. 1

June 1978

Honeywell

Preface

This manual describes program preparation for Series 60 (Level 6) GCOS. Unless stated otherwise herein, the term GCOS refers to the GCOS 6 software; the term Level 6 refers to the Series 60 (Level 6) hardware on which the software executes.

Section 1 provides an overview of the program preparation sequence. This section summarizes how to access files via pathnames, and describes in detail the suffixes that are appended to file names. It is important that you understand these concepts before proceeding with the manual.

Section 2 describes how to load the Editor, and includes detailed descriptions of directives that control execution of the Editor.

Section 3 describes how to load the Macro Preprocessor, Assembler, FORTRAN Compiler, COBOL Compilers, and RPG Compiler. The Macro Preprocessor and Assembler are described in the *Assembly Language Reference* manual; the FORTRAN, COBOL, and RPG Compilers, and their respective languages are described in the *FORTRAN Reference* manual, *Intermediate COBOL Reference* manual, *Entry-Level COBOL Reference* manual, and *RPG Reference* manual, respectively.

Appendix A describes assembly language program independence.

MANUAL DIRECTORY

The following publications comprise the GCOS 6 manual set. The Manual Directory in the latest *GCOS 6 MOD 400 Systems Concepts* manual (Order No. CB20) lists the current revision number and addenda (if any) for each manual in the set.

Order No.	*Manual Title*
CB01	*GCOS 6 Program Preparation*
CB02	*GCOS 6 Commands*
CB03	*GCOS 6 Communications Processing*
CB04	*GCOS 6 Sort/Merge*
CB05	*GCOS 6 Data File Organizations and Formats*
CB06	*GCOS 6 System Messages*
CB07	*GCOS 6 Assembly Language Reference*
CB08	*GCOS 6 System Service Macro Calls*
CB09	*GCOS 6 RPG Reference*
CB10	*GCOS 6 Intermediate COBOL Reference*
CB20	*GCOS 6 MOD 400 System Concepts*
CB21	*GCOS 6 MOD 400 Program Execution and Checkout*
CB22	*GCOS 6 MOD 400 Programmer's Guide*
CB23	*GCOS 6 MOD 400 System Building*
CB24	*GCOS 6 MOD 400 Operator's Guide*
CB25	*GCOS 6 MOD 400 FORTRAN Reference*
CB26	*GCOS 6 MOD 400 Entry-Level COBOL Reference*
CB27	*GCOS 6 MOD 400 Programmer's Pocket Guide*
CB28	*GCOS 6 MOD 400 Master Index*
CB30	*Remote Batch Facility User's Guide*
CB31	*Data Entry Facility User's Guide*
CB32	*Data Entry Facility Operator's Quick Reference Guide*
CB33	*Level 6/Level 6 File Transmission Facility User's Guide*
CB34	*Level 6/Level 62 File Transmission Facility User's Guide*
CB35	*Level 6/Level 64 (Native) File Transmission Facility User's Guide*
CB36	*Level 6/Level 66 File Transmission Facility User's Guide*
CB37	*Level 6/Series 200/2000 File Transmission Facility User's Guide*
CB38	*Level 6/BSC 2780/3780 File Transmission Facility User's Guide*
CB39	*Level 6/Level 64 (Emulator) File Transmission Facility User's Guide*
CB40	*IBM 2780/3780 Workstation Facility User's Guide*
CB41	*HASP Workstation Facility User's Guide*
CB42	*Level 66 Host Resident Facility User's Guide*
CB43	*Terminal Concentration Facility User's Guide*

In addition, the following documents provide general hardware information:

Order No.	*Manual Title*
AS22	*Honeywell Level 6 Minicomputer Handbook*
AT04	*Level 6 System and Peripherals Operation Manual*
AT97	*MLCP Programmer's Reference Manual*
FQ41	*Writable Control Store User's Guide*

Contents

Section 1. Overview

Section 2. Editor

Section 3. Language Processors

Appendix A. Assembly Language Program Independence

Figures

Tables

Section 1
Overview

System-supplied software contains the procedures that are necessary to create a source unit, assemble or compile the source unit to form an object unit, and to convert it into a executable format (including error detection and correction) or to apply a patch. These tasks are described in subsequent sections of this manual and in the *Program Execution and Checkout* manual.

Program preparation can be performed after a system is built as described in the *System Building* manual. The equipment required for program preparation is described in the "Equipment Requirements" section of the *Systems Concepts* manual.

The program preparation process is described below and illustrated in Figure 1-1.

Source units can be created via punched cards or the Editor. A source unit comprises source statements written in assembly language, FORTRAN, COBOL, or RPG. If desired, source units can be altered by the Editor. Source units are converted to object units by a language processor (e.g., the Assembler, FORTRAN Compiler, COBOL Compiler, or RPG Compiler). If assembly language source statements contain one or more macro calls, the source text must be processed by the Macro Preprocessor before it can be processed by the Assembler. The Macro Preprocessor replaces each macro call with a sequence of statements known as a macro routine. Macro Preprocessor output is called an expanded source unit. To obtain a list of all symbolic names in an assembly language source unit, and to determine whether any of the symbols are undefined, multiply defined, or defined and not used, the -CROSS_REF argument (short form is -XREF) of the ASSEM command is used. If necessary, corrections can be made by using the Editor. Separately assembled and/or compiled object units must be linked by the Linker to form a bound unit. A bound unit comprises a root, or a root and one or more overlays. A root is the portion of a bound unit that is loaded into memory when the loader is requested to load a bound unit.[1] An overlay is loaded into memory whenever it is required. Details of using the Linker are given in the *Program Execution and Checkout* manual.

For detailed information about program execution, including execution control and program modification during execution, program debugging, patches and dumping memory, see the *Program Execution and Checkout* manual.

Notes:

1. If you are going to perform program preparation and checkout while simultaneously executing other online tasks, you must be familiar with the *System Concepts* manual.
2. Throughout this manual there are references to the create group, enter group request, spawn group, and enter batch request commands; these commands are described in the *Commands* manual.

[1] The root is loaded when an -EFN argument is specified in a create group or spawn group command or the root pathname is supplied to the command processor (see the *Commands* manual).

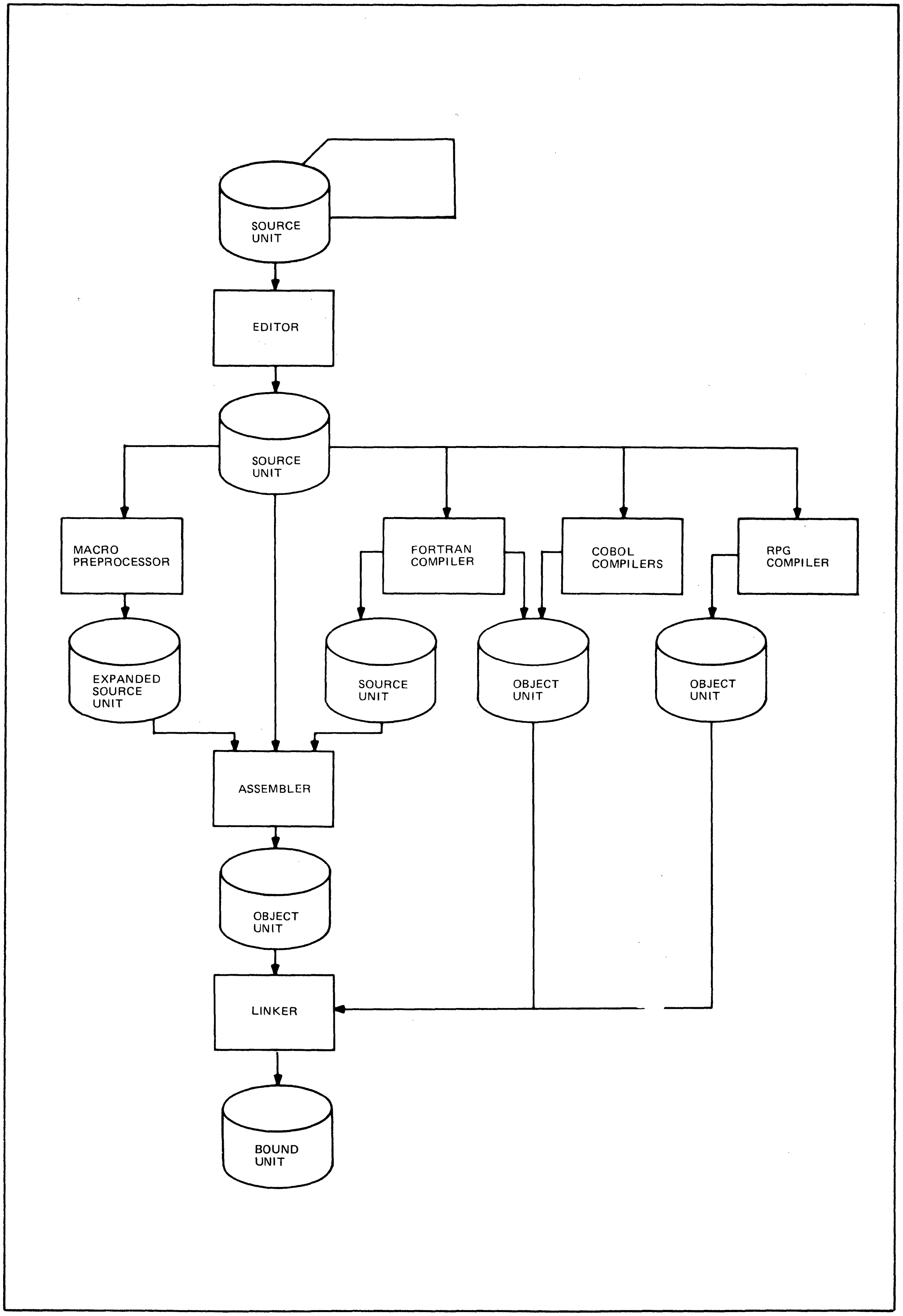
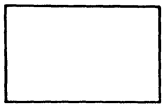
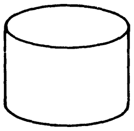


Figure 1-1. Program Preparation Procedure

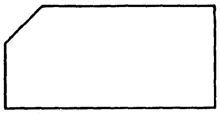
SYMBOLS USED IN THIS MANUAL



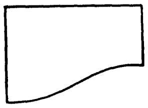
Processing; indicates any kind of processing function.



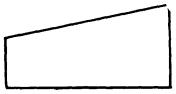
Online storage of information; e.g., diskette, cartridge disk, or storage module.



Input from card reader.



Document; e.g., printer output.



Manual input; i.e., operator terminal or another terminal.

Mandatory; indicates that the designated flow of information, type of processing, input, or output is required.

UPPERCASE CHARACTERS	Reserved words or symbols; must be entered or used exactly as shown.
lowercase characters	Symbolic name or value; you must supply the exact value.
brackets [ ]	Optional information.
braces { }	An enclosed entry must be selected.
ellipses ...	There may be multiple entries of the immediately preceding type of information.

FILE SYSTEM PATHNAMES

The file system is a tree-structured hierarchy through which each volume of storage is identified to the system. The basic element of this structure is the *file.* A special file called a *directory* contains information about other files.

DEFINITION OF A FILE

A file is any unit of storage outside the central processor, which can supply data to or receive data from a task. A file can be a peripheral device such as a printer, card reader, or terminal; or it can be a collection of data stored within a directory structure on a magnetic (tape or disk) storage device. A source unit, object unit, listing, or bound unit is stored as a source unit file, object unit file, list file, or bound unit file, respectively.

DEFINITION OF A DIRECTORY

A directory is a file that contains information about other "subordinate" storage system entries, which in turn may represent other directories or data files. An entry named in a directory is subordinate to that directory, and is "contained" within it. The information in the containing directory describes physical and logical attributes of the subordinate files.

The directory at the base of a tree structure is the *root directory*. Its name is the same as the name (volume id) of the volume where it resides.

When first created, a volume has only a root directory. Later, names and attributes of subordinate directories can be created within this directory.

All references to directories and files begin either explicitly or implicitly with a root directory name.

DIRECTORY OR FILE NAME CONSTRUCTION

A directory or file name can consist of the following ASCII characters:

- Letters (A through Z)
- Decimal digits (0 through 9)
- Underscore character (_)
- Period (.)
- Dollar sign ($)

Each name must begin with a letter or the dollar sign ($). Lowercase letters are equivalent to the corresponding uppercase letters. The underscore is used to join two or more words that the system is to interpret as one name; e.g., DATE_TIME. The period separates a name from its alphabetic or numeric suffix characters. For example, in the name of a COBOL source file called COBPROG.C, COBPROG is any user-specified name, and C is the required suffix, indicating to the system that this is a COBOL source file.

The length of a root directory name or volume identifier can be one (nonblank) to six characters. A directory (other than root) or file name can have one (nonblank) to twelve characters. A specified file name must provide for any possible suffix that might be appended by the system so that its resultant overall length does not exceed 12 characters.

PATHNAME CONSTRUCTION

A pathname is a string comprising one or more directory names and possibly one file name. All subordinate names of directories and files within a directory must be unique. The pathname describes the access path to the entity to be acted on. A pathname begins with a root directory name, followed by zero, one or more directory names, and possibly a file name, in order of their hierarchy.

The progressive relationship among pathname elements in the hierarchy is indicated by the following symbols:

- Circumflex (^) — Denotes a *root* directory only, and must precede the root directory name, with *no* intervening space (e.g., ^ VOL011).
- Greater-than symbol (>) — Indicates movement in the hierarchy away from the root; connects two directory names or a directory name and a file name. It can also be the first character in a pathname, in which case it is immediately subordinate to the root directory of the system volume. Each successvie symbol in the string indicates a change of one directory level; the name immediately following the symbol is at the next subordinate level to the name immediately preceding it. Reading a pathname from left to right shows the access through the tree structure, away from the root, to the last element in the pathname. For example, if the root directory VOL011 contains the directory name DIR1, the pathname for DIR1 is ^ VOL011>DIR1. However, if directory DIR1 in turn contains the file FILEA, then the pathname for FILEA is ^ VOL011>DIR1>FILEA. The > symbol is never followed by a space, nor preceded by a space *except* as the first character in a pathname.
- The less-than symbol (<) — Indicates movement in the hierarchy *toward* the root, and a change of one level in that direction. Additional < symbols show successive level changes.

The last element in a pathname is the name of the entity that is to be acted on, and may denote either a directory name or file name, according to the action to be taken.

Total length of any pathname, including all hierarchical symbols, cannot exceed 59 characters, except that a working directory pathname cannot exceed 44 characters.

ABSOLUTE PATHNAME

An *absolute pathname* begins with a directory name preceded by a circumflex (^) or a greater-than symbol (>). With a circumflex, this pathname is a *full pathname;* with a greater-than symbol, the first element is immediately subordinate to the root directory of the system volume.

RELATIVE PATHNAME AND WORKING DIRECTORY

A *relative pathname* is one that does *not* begin with the circumflex or greater-than symbol. For a relative pathname that does not begin with a less-than symbol, the first (or only) name in the pathname identifies a directory or file immediately subordinate to a directory known as the *working directory.* The working directory is your current position in the file system hierarchy.

The simplest form of a relative pathname has only one element, the name of the desired entry in the working directory.

Figure 1-2 contains examples of relative pathnames and the full pathnames they represent. The working directory pathname is

>UDD>PROJ1>USERA

and the system was initialized from the volume SYS01. Below the pathname is the corresponding tree structure.

Relative Pathname	*Full Pathname*
DELTA	^SYS01>UDD>PROJ1>USERA>DELTA
OLD>DELTA	^SYS01>UDD>PROJ1>USERA>OLD>DELTA
<USERB>ALPHA	^SYS01>UDD>PROJ1>USERB>ALPHA
< <PROJ2> USERA> DELTA	^SYS01>UDD> PROJ2>USERA>DELTA
<	^SYS01>UDD>PROJ1

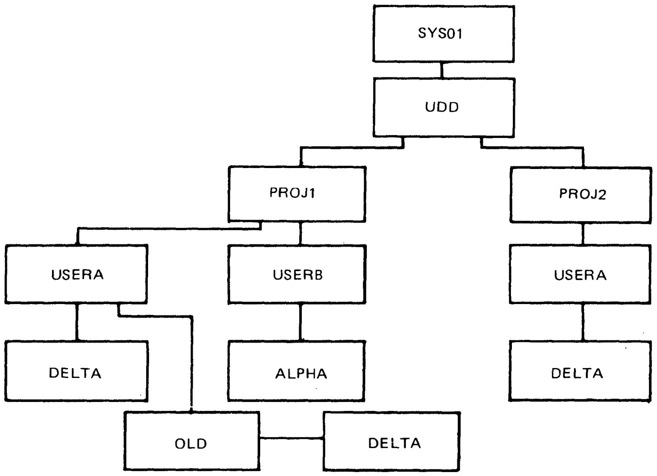


Figure 1-2. Tree Structure and Pathname Examples

DEVICE PATHNAMES

Reference to any device is through the symbolic peripheral device (SPD) directory, which is subordinate to the system root.

DEVICE FILES (OTHER THAN DISK AND TAPE)

The general form of a device file pathname is:

>SPD>dev_name

where dev_name is the symbolic name defined for the card reader, punch, printer, or terminal device during system building.

Device files are always reserved for exclusive use (i.e., the reserving task group has read and write access, but other users are not allowed to share the file).

Tape Files

The general form of a tape file (device) pathname is:

>SPD>dev_name [>volid[>filename]]

where dev_name is the symbolic name defined for the tape device during system building, volid 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, but other users are not allowed to share the file).

Disk Device Files

The general form of a disk device-level access pathname is:

>SPD>dev_name[>volid]

where dev_name is the symbolic name defined for the disk device during system building and volid is the name of the disk volume.

This pathname format is used only when access to the entire volume is required (such as during a volume copy or a volume dump).

If the volid is not supplied, reservation of the disk is exclusive (i.e., the reserving task group has read and write access, but other users are not allowed to share the file). This pathname form is used when creating a new volume.

If the volid is specified, reservation is read/share (i.e., the reserving task group has read access only; other users may read and write). This pathname is used when dumping select portions of a volume without regard to the hierarchical file system tree structure.

The following are examples of device pathnames.

Peripheral Device	*Pathname*
Exclusive line printer	>SPD>LPT01
Exclusive tape volume	>SPD>MT902>VOL3
File on an exclusive tape volume	>SPD>MT902>VOL3>FILEA
Exclusive diskette	>SPD>DSK02
Nonexclusive cartridge disk volume	>SPD>RCD01>V23X

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 must 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; i.e., .A, assembly language; .C, COBOL; .F, FORTRAN; .R, RPG; .P, Macro Preprocessor input.

When you specify a file name in a command to load a program preparation task or in a directive to a task (except for the Editor), do *not* include a suffix in the file name. Suffixes are appended to the specified base name by the Macro Preprocessor, Assembler, FORTRAN Compiler, COBOL Compilers, RPG Compiler, and Linker, as described below.

Note:

> In the following descriptions there are references to specific commands. In each case, the referenced command is the command that loads the task being described. The LINKER command is described in the *Program Execution and Checkout* manual. The other referenced commands are described in Section 3 of this manual.

The Editor requires that when you specify in Editor directives the file names of Editor input and output files, you specify the complete file name, *including* the suffix that denotes the contents of the file; i.e., .A, assembly language; .C, COBOL: .F, FORTRAN; .R, RPG; .P, Macro Preprocessor input, and .IN.A, Macro Preprocessor "include" file. The Editor does not append a suffix to its input or output file names.

The Macro Preprocessor requires that the name of its input file contain a .P suffix. When you specify in the MACROP command the name of the input file, omit the .P suffix. If there is an "include" file, that file name must contain a .IN.A suffix. The Macro Preprocessor forms the name of its output file by appending .A to the specified base name.

The Assembler requires that the name of its input file contain a .A suffix. When you specify in the ASSEM command the name of the input file, omit the .A suffix. The Assembler forms the name of its object unit file by appending .O to the specified base name. The Assembler forms the name of its list file by appending .L to the specified base name. If a list file is designated (i.e., the -COUT argument is specified in the ASSEM command), the Assembler does *not* append a suffix to the specified name.

The FORTRAN Compiler requires that the name of its input file contain a .F suffix. When you specify in the FORTRAN command the name of the input file, omit the .F suffix. The compiler forms the name of its object unit or assembly output file by appending .O or .A, respectively, to the specified base name.

Note:

> This procedure applies in the absence of a SUBROUTINE, FUNCTION or PROGRAM source statement. If one of these statements has been used, the name specified in the statement (i.e., not the file name) is used as the name of the object unit.

If a list file is designated (i.e., the -COUT argument is specified in the FORTRAN command), the compiler does *not* append a suffix to the specified name; otherwise, the compiler forms the name of its list file by appending .L to the specified base name.

The COBOL Compilers require that the name of the input file contain a .C suffix. When you specify in the COBOL or COBOLI command the name of the input file, omit the .C suffix. The compiler forms the name of its object unit output file by appending .O to the specified base name. If a list file is designated (i.e., the -COUT argument is specified in the COBOL/COBOLI command), the compiler does *not* append a suffix to the specified name; otherwise, the compiler forms the name of its list file by appending .L to the specified base name.

The RPG Compiler requires that the name of its input file contain a .R suffix. When you specify in the RPG command the name of the input file, omit the .R suffix. The compiler forms the name of each object unit file by appending .O to fixed compiler-generated base names. In addition, the compiler generates a Linker command file to link these object files. The compiler forms the Linker command file name by appending .Q to the specified input file name. If a list file is designated (i.e., the -COUT argument is specified in the RPG command), the compiler does not append a suffix to the specified name; otherwise, the compiler forms the name of its list file by appending .L to the specified base name.

The Linker (see *Program Execution and Checkout* manual) requires that each of its input file names contain a .O suffix. When you specify a file name in a link directive, omit the .O suffix. In the LINKER command you specify the name of the file that will contain the bound unit; the Linker will *not* append a suffix to the bound unit name. If a list file is designated (i.e., the -COUT argument is specified in the LINKER command), the Linker does *not* append a suffix to the specified name; otherwise, the Linker forms the name of its list file (Linker maps) by appending .M to the specified bound unit name.

It is important to note that only the Macro Preprocessor, Assembler, FORTRAN Compiler, COBOL Compilers, RPG Compiler, and Linker append suffixes to specified file names.

Table 1-1 summarizes how file names are designated.

TABLE 1-1. DESIGNATING FILE NAMES

Program Preparation Task	Input File(s)	Output File(s)
Editor	Specify file name, including any required suffix.	Specify file name, including any required suffix.
Macro Preprocessor	Omit suffix. Macro Preprocessor appends .P to specified file name. If there is an "include" file, the Macro Preprocessor appends .IN.A to the specified file name.	Omit suffix. Macro Preprocessor appends .A to specified input file name(s).
Assembler	Omit suffix. Assembler appends .A to specified file name.	Omit suffix. Assembler appends .O to specified input file name to form the name of the object unit file, and .L to specified input file name to form the name of the list file if the -COUT argument was not specified in the ASSEM command.[a]
FORTRAN Compiler	Omit suffix. FORTRAN Compiler appends .F to specified file name.	FORTRAN Compiler appends .O to specified object unit file name[b], .A to specified file name of assembly language file, and .L to specified input file name to form the name of the list file if the -COUT argument was not specified in the FORTRAN Command.[a]
COBOL Compilers	Omit suffix. COBOL Compilers append .C to specified file name.	Omit suffix. COBOL Compilers append .O to specified object unit file name and .L to specified input file name to form the name of the list file if the -COUT argument was not specified in the COBOL/COBOLI command.[a]
RPG Compiler	Omit suffix. RPG Compiler appends .R to specified file name.	Omit suffix. The RPG compiler generates multiple object unit output files. The suffix .O is appended to each compiler-generated base name. The compiler appends .Q to the input file name to form the name of the Linker command file. The compiler appends .L to specified input file name to form the name of the list file if the -COUT argument was not specified in the RPG command.[a]
Linker	Omit suffix. Linker appends .O to each specified file name.	Omit suffix. The Linker appends .M to specified bound unit file name to form the name of the list file if the -COUT argument was not specified in the LINKER command. The Linker does not append a suffix to the name designated in the -COUT argument.

[a] The language processor does not append a suffix to the name designated in the -COUT argument.

[b] Except when a SUBROUTINE, FUNCTION or PROGRAM statement appears in the source file.

Section 2
Editor

The 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, COBOL, RPG, or assembly language. Throughout this section it is assumed that source unit files are being edited.

Editing is controlled by directives entered to the Editor through the device specified in the in_path argument of the enter batch request or enter group request command. This device can be reassigned in the command that loads the Editor.

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

During a single execution of the Editor, the Editor can operate in input and/or edit mode. During input mode, you an create a source unit and/or add one or more specified lines to an existing source unit. During 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 Editor.

Notes:

1. During a single execution of the 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 Editor you can request a typeout that will indicate whether input or edit mode is in effect. Each time !? is entered, the following typeout is issued:

   $\left\{\begin{array}{l}\text{INPUT}\\ \text{EDIT}\end{array}\right\}$ MODE

Editor processing can be interrupted by:

- Pressing the QUIT, INTERRUPT, or BREAK key on the user terminal, or
- Entering ΔCΔBgroup-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 the user's terminal when the system interrupts the Editor. If the PI (program interrupt) 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 Editor directive's name and function is listed in Table 2-1, later in this section. They are described in detail in "Input Mode Description and Directives," "Edit Mode Description and Directives," and "Advanced Usage of the Editor," later in this section. Directives described in the input and edit mode subsections operate within the current buffer.

CONVENTIONS USED IN EDITOR DIRECTIVE FORMATS

Most 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 Editor to switch from input mode to edit mode. These formats are illustrated below. 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_1 dirname

FORMAT 3:

$$adr_1 \begin{Bmatrix} ; \\ , \end{Bmatrix} adr_2 \text{ dirname}$$

FORMAT 4:

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

FORMAT 5:

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

[text]

.

.

.

!F

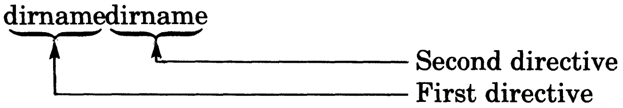
Notes:

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" later in this section).

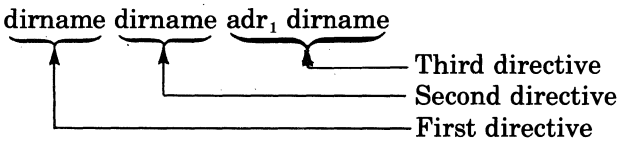
When a single address is specified, the Editor references the specified line in the current buffer. When two addresses are specified within a single directive, the Editor references a specified series of lines in the current buffer; the lines that are referenced depends on whether the addresses are separated by a comma or a semicolon (see "Referencing a Series of Lines" later in this section). If an 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 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 below:

$$\text{adr}_1\text{ dirname dirname"comment}$$

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

$$\underbrace{\left[\text{adr}_1\left[\begin{Bmatrix};\\,\end{Bmatrix}\text{ adr}_2\right]\right]\text{dirname[text]!F}}_{\text{Directive}}\underbrace{\text{"comment}}_{\text{Directive comment}}$$

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 the line.

DESIGNATING A LINE NUMBER AS AN ADDRESS

Each line in the current buffer can be referenced by a decimal number that indicates the current position of the line within the buffer.[1] 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 above expressions request line number 10. The last line can be referenced by its line number or by the character $.

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.

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

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

Most Editor directives affect either the current line or a line a designated number of positions from the current line. If the last 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 Editor (regardless of whether the condition specified in the directive was met); if the last 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 referenced by specifying a period (.).

Note:

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

You can reference lines relative to the current line by specifying an address that consists of a

[1] 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 Editor," later in this section.)

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 Editor reference 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 Editor will search the lines in the current buffer until it finds the *first* occurrence of the specified expression; unless specified otherwise,[2] the expression can be in any position within the line. The 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 has not been found, the Editor then searches line 1 to the current line.

Example:

/BBB/dirname

In the above directive format, the address is the expression BBB. The specified directive name will cause the Editor to search as may 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 will cause the Editor to reference line number 4, since this is the first line after the current line that contains the expression BBB.

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

Character	*Description*
*	Requests expressions that contains 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 string of characters preceding or 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),

[2] 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. Usage of these special characters is described below.

and !? (which causes a typeout 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 a typeout of the mode that is in effect.

2. The characters . and $ can be specified as line numbers or as special characters in expressions; the Editor can interpret their meaning from the way they are used.
3. For the 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 considered 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 above characters, it is preferable to execute a change directive, reentering the line using hexadecimal and ASCII characters for the entire line.

Examples:

Following are some examples of expressions specified as addresses in Editor directives. Following each expression is a description of the line/character(s) in the current buffer for which the Editor will search. In each case, the 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.
/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 Editor remembers the last expression designated as an address. That expression can be reinvoked in a subsequent Editor directive by specifying a null expression (e.g., //).

Example:

/ABC/dirname — Expression ABC is specified as an address.
2dirname — Second line in buffer is specified as address.

//dirname — Specifies ABC as an address, since ABC was the last *expression* designated as an address.

An address can be specified as an expression followed by one or more signed decimal integers.

Example:

Each of the following three expressions requests the second line after the line that contains ABC.

```
/ABC/2
/ABC/+2
/ABC/+5−3
```

COMPOUND ADDRESSES

An address can be formed by combining the methods described above. 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 Editor will search for the designated expression. If the first element is a line number, the Editor searches for the expression starting with the line that immediately follows the specified line number. (Ordinarily, the Editor searches starting with the line that immediately follows the current line.)

Example 1:

10/ABC/

This address causes the Editor to search the lines in the current buffer, starting with line 11, for the characters ABC.

Example 2:

.−8/ABC/

This address causes the Editor to search the lines in the current buffer, starting with eight lines before the current line, for the characters ABC.

Example 3:

/ABC//DEF/

This address causes the Editor to search for the first line containing DEF that occurs *after* a line containing ABC.

Each expression in a compound address can be followed by a signed decimal integer.

Example:

/ABC/−10/DEF/5

This address causes the Editor to search 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

An Editor directive that permits *two* addresses to be specified causes the Editor to reference 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 referenced by the Editor; otherwise, only the addresses are relevant.

If the addresses are separated by a *comma,* the Editor references the line at the first address through the line at the second address, inclusive. The current line remains unchanged until after 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 referenced by the first address becomes the current line and then 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 listed below. The numbers within parentheses represent line numbers.

(1) ABC
(2) DEF (current line)
(3) GHI
(4) ABC
(5) XYZ
(6) ABC

The above addresses specify the line immediately following the current line through the second line after the current line. The Editor will reference lines 3 and 4. Line 4 will then become the current *line*.

Example 5:

.1;.2dirname

These addresses are the same as those in Example 4, but in this example 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 Editor to reference *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 Editor reference 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 are listed below. 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 Editor reference the first line that contains ABC (i.e., line 1) through the third line after that line (i.e., lines 2, 3, and 4). Line 4 will then become the current line.

The second series of addresses specifies that the Editor reference the first line that contains ABC (i.e., line 1), make that line the current line, and then reference three lines from the "new" current line (i.e., lines 2, 3, and 4). Line 4 will then become the current line.

LOADING THE EDITOR

To load the Editor, enter the ED command, which is described below.

After the Editor is loaded, there is a typeout to the error-out file of the revision number, in the format: ED nnnn

FORMAT:

```
ED  [ctl_arg]
```

ARGUMENT DESCRIPTIONS:

ctl_arg

Control arguments; none or any number of the following control arguments may be entered, in any order:

-IN path

Pathname of the device through which Editor directives will be entered; can be the operator terminal or another terminal, card reader, or disk. Error messages are written to the error-out file. Editor 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.

$\left\{\begin{array}{l}\text{-LINE\_LEN n}\\\text{-LL n}\end{array}\right\}$

Maximum number of characters that can be on each directive line or data line. Must be from 20 through 256, decimal. Additional charcters are truncated.

Default: 80 characters.

$\left\{\begin{array}{l}\text{-PROMPT}\\\text{-PT}\end{array}\right\}$

If the input device is a terminal, there is a typeout of E? whenever the Editor is ready to accept another directive.

SUMMARY OF EDITOR DIRECTIVES AND ESCAPE SEQUENCES

Table 2-1 lists each 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," "Edit Mode Description and Directives," and the following subsections of "Advanced Usage of the Editor": "General Advanced Editor Directives," "Auxiliary Buffer Directives and Escape Sequences," "Editor Debugging Directives," and "Editor Programming Directives."

TABLE 2-1. SUMMARY OF 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 Editor without exiting from Editor.	Execute directive (advanced usage — general)
G	Search for specified line(s) that contain specified character string.	Global directive (advanced usage — general)
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)
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 Editor.	Quit directive (edit mode)
R	Read text from file to current buffer.	Read directive (edit mode)
S	Substitute character stirng with another character string.	Substitute directive (edit mode)
T	Display a line of text on user-out file subsequent input/output will be on the next line.	Type directive (advanced usage — programming)
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)
!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.	

TABLE 2-1 (CONT). SUMMARY OF EDITOR DIRECTIVES AND ESCAPE SEQUENCES

Directive Name/Escape Sequence	Function	Topic Under Which Described
!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)
!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 Editor.	Quite directive (edit mode)
!R	Accept single line from operator terminal.	Accept single line from operator terminal directive (advanced usage — auxiliary buffers)
!T	Display a line of text on user-out file; subsequent input/output will be on the same line.	Type directive (advanced usage — programming)
!?	Cause typeout indicating whether input or edit mode is in effect.	
#	If current buffer contains data, execute specified directive(s).	If empty directive (advanced usage — programming)
address#	If current line is specified line, execute specified directive(s).	If line directive (advanced usage — programming)
addresses#	If current line is within specified lines, execute specified directive(s).	If range directive (advanced usage — programming)
^ #	If current buffer does *not* contain data execute specified directive(s).	If not 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 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)
>	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)

CREATING A SOURCE UNIT

To create a source unit, take the steps listed below. Input mode directives are described under "Input Mode Description and Directives" later in this section. Each of the directives referenced below 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 command (see the *Commands* manual).
2. Load the Editor, if it is not already loaded. (See "Loading the Editor" earlier in this section.)
3. If there already are lines in the current buffer, delete unwanted lines by specifying the delete directive.
4. Enter the appropriate input directive and text to be input.
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. (Optional) Exit from the Editor by entering the quit directive.

CHANGING AN EXISTING SOURCE UNIT

To change an existing source unit, take the steps listed below. Input mode directives are described under "Input Mode Description and Directives" later in this section. Each of the directives referenced below 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 command (see the *Commands* manual).
2. Load the Editor, if it is not already loaded. (See "Loading the 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. (Optional) Exit from the Editor by entering the quit directive.

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:

$$\left[\text{adr}_1\left[\begin{Bmatrix} ; \\ , \end{Bmatrix}\ \text{adr}_2\right]\right]\text{dirname}$$

```
[text]
  .
  .
  .
!F[3]["comment]
```

FORMAT 2:

$\left[adr_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} adr_2 \right] \right]$dirname[text]!F[3]["comment]

If directives are being entered through the operator terminal or another terminal, the directive name may be immediately followed by a carriage return, which in turn is followed by the text (i.e., the lines to be included in the source unit), or the first line of text can be on the same line as the directive name, and additional lines (if any) can be on the 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[3] to terminate input mode; otherwise, the next 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.

APPEND DIRECTIVE

The append directive puts 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
```

[3] When entering directives from a card reader, the punch for an exclamation point is 12-8-7.

ARGUMENT DESCRIPTION:

adr

Identifies the 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 DIRECTIVE

The change directive deletes a single line or a series of lines in the current buffer and then inserts 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:

$$\left[\text{adr}_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} \text{adr}_2 \right] \right] \text{C}$$

text
.
.
.
!F

FORMAT 2:

$$\left[\text{adr}_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} \text{adr}_2 \right] \right] \text{Ctext!F}$$

ARGUMENT DESCRIPTIONS:

adr_1

Address of the *first or only* line to be deleted and replaced.
Default: Current line.

adr_2

Address of the *last* line to be deleted and replaced.
Default: Only the line identified by adr_1 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 above change directives 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 DIRECTIVE

The insert directive inserts 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 DESCRIPTION:

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:

- *Substitute* a designated string of characters in specified line(s) with another specified string of characters (substitute directive).
- *Read* text from specified file into the current buffer (read directive).
- *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).
- *Write* specified line(s) from the current buffer to specified file (write directive).
- *Terminate* execution of the Editor (quit directive).

Notes:

1. To edit an existing source unit, the read directive must be previously specified.
2. Until you are familiar with the Editor, it is recommended that you 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 Editor.

Most edit mode directives have one of the following formats:

FORMAT 1:

```
dirname["comment]
```

FORMAT 2:

adr_1dirname["comment]

FORMAT 3:

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

The delete directive deletes 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:

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

ARGUMENT DESCRIPTIONS:

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_1 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_1 is CCC and adr_2 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 DIRECTIVE

The print directive causes a printout of 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 or enter group request command, unless that file was reassigned in the file out (FO) command. If the typeout 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:

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

ARGUMENT DESCRIPTIONS:

adr_1

Address of the first or only line to be printed.

Default: Current line.

adr_2

Address of the *last* line to be printed.
Default: Only the line identified by adr_1 is printed.

Note:

If both adr_1 and adr_2 are omitted and P is specified, only the current line is printed.

FORMAT 2:

Format *excluding* directive name P:

$$\text{adr}_1\left[\begin{Bmatrix};\\,\end{Bmatrix}\ \text{adr}_2\right]$$

ARGUMENT DESCRIPTIONS:

adr_1

If adr_2 is not specified, adr_1 designates the address of the only line to be printed.

adr_2

Address of only line to be printed.

Note:

If both adr_1 and adr_2 are specified, only adr_2 is 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 typeout 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 typeout 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 typeout 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 typeout 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 typeout 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 typeout 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 typeout of line number 1, and that line will be the current line.

QUIT DIRECTIVE

The quit directive is used to exit from the 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 the operator terminal or another 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, below), 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 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 Editor directives are not saved. In an unconditional quit request (Format 2, below), modified buffers are *not* checked before quit is executed.

FORMAT 1:

```
Q
```

FORMAT 2:

```
!Q
```

Example:

A

Append directive puts specified lines into current buffer.

AAABBB
CCCDDD
EEEFFF

Lines that will be put into current buffer.

!F

Designates the end of the insertion.

2D

Deletes the second line of text (e.g., CCCDDD).

W FIRST

Writes all lines in buffer to file named FIRST.

Q

Returns control from the Editor to the command processor.

READ DIRECTIVE

The read directive reads text from a specified ASCII variable sequential 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]
```

ARGUMENT DESCRIPTIONS:

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 1.) The pathname may be preceded by any number of blank spaces.

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 Editor" later in this section. If the path parameter is not specified and a pathname was not previously specified, an error message is issued.

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.

```
A      Causes subsequent lines to be put 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:

AAA
BBB
CCC
AAA
BBB
CCC (current line)

SUBSTITUTE DIRECTIVE

The substitute directive replaces 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 referenced by the Editor.

FORMAT:

$$\left[\mathrm{adr}_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} \mathrm{adr}_2 \right] \right] \mathrm{S/regexp/string/}$$

ARGUMENT DESCRIPTIONS:

adr$_1$

Address of the first line to be searched for the specified string of characters.

Default: Current line.

adr$_2$

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 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 "Editor Debugging Directives," later in this section.

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 character; i.e., point in the resulting line at which the line is to be split into two lines.

   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//
```

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 1:

```
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 2:

```
.,4S-CCC-UUU-
```

This substitute 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 3:

```
-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 4:

/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 DIRECTIVE

The write directive causes a single line or a series of lines in the current buffer to be written 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 Editor directives, you must specify the write directive before you terminate execution of the 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:

$\left[\text{adr}_1\left[\begin{Bmatrix};\\,\end{Bmatrix}\text{adr}_2\right]\right]$W[path]

ARGUMENT DESCRIPTIONS:

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 1.) 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:

```
1,3W
```

This write directive writes lines 1 through 3 to the file specified in the last read or write directive.

This example illustrates usage of the above directive in a sample 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

Reads 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/

Searches each line in the current buffer and changes each occurrence of AAA to XXX. The buffer will contain:

```
(1) XXX
(2) BBB
(3) CCC
(4) DDD (current line)
```

1,3W

Writes 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

Terminates execution of the Editor.

ADVANCED USAGE OF THE EDITOR

The directives described on the previous pages permit you to create a source unit and perform basic editing. The following subsections describe 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.

GENERAL ADVANCED EDITOR DIRECTIVES

The general advanced Editor directives have the following capabilities:

- Permit execution of a command instead of Editor directives without exiting from the Editor (execute 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).
- Cause another specified directive to act on only those lines that contain a specified character string (global directive).
- Cause another specified directive to act on only those lines that do *not* contain a specified character string (exclude directive).
- Make a different line the current line (new current line directive).

EXCLUDE DIRECTIVE

The exclude directive (V) 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 Editor; i.e., the line specified in adr$_2$ (see below).

FORMAT:

$$\left[\text{adr}_1\left[\begin{Bmatrix};\\,\end{Bmatrix}\text{adr}_2\right]\right]\text{Vx/regexp/}$$

ARGUMENT DESCRIPTIONS:

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 used; 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 Editor will search; only lines that do *not* contain regexp will be acted upon by the Editor during execution of the directive name specified in parameter 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 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 *not* 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 DIRECTIVE

The execute directive permits you to execute a command instead of Editor directives without exiting from the Editor; i.e., you can enter any command and then continue to use the Editor. For example, the execute directive can be used to designate a printer as the Editor output file. Otherwise, if you want a printout of Editor output, the printout is issued to the user 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 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 DESCRIPTION:

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 DIRECTIVE

The global 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 contain a specified character string.

After the global directive is executed, the current line is the last line searched by the Editor.

FORMAT:

$$\left[\text{adr}_1\left[\begin{Bmatrix};\\,\end{Bmatrix}\ \text{adr}_2\right]\right]\text{Gx/regexp/}$$

ARGUMENT DESCRIPTIONS:

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

Deletes all line(s) in the specified range containing regexp.

P

Prints the contents of line(s) containing regexp.

!P

Prints the line number(s) and contents of line(s) containing regexp (see "Print With Line Number Directive" later in this section).

=

Prints 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 Editor will search; only lines that contain regexp will be acted upon by the directive name specified in the parameter 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 Editor to search lines 1 through 3 and print the line number and contents 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)
```

NEW CURRENT LINE DIRECTIVE

The new current line directive (N) causes 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 DESCRIPTION:

adr

Specifies 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 DIRECTIVE

The print line number directive causes a typeout of the *line number* of a specified line in the current buffer.

The typeout 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 that file was reassigned.

After this directive is executed, the current line is the line whose line number was typed.

FORMAT:

[adr]=

ARGUMENT DESCRIPTION:

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 typeout of the line number of the first line that contains CCC.

Typeout:

2

Current line: 2

Example 2:

=

```
= / !P
```

This print line number directive causes a typeout of the line number of the current line.

Typeout:

```
1
```

Current line: 1

PRINT WITH LINE NUMBER DIRECTIVE

The print with line number directive (!P) causes a typeout of the *line number and contents* of a single line or consecutive lines in the current buffer. The typeout 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 that file was reassigned. If the typeout occurs on the operator terminal or another 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:

$\left[\mathrm{adr}_1\left[\begin{Bmatrix};\\,\end{Bmatrix}\ \mathrm{adr}_2\right]\right]$!P

ARGUMENT DESCRIPTIONS:

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 typeout 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 typeout of the line number and contents of each line in the current buffer.

Typeout:

```
1 AAA
2 BBB
3 CCC
4 DDD
```

Current line: 4

Example 2:

```
!P
```

This print with line number directive causes a typeout of the line number and contents of only the current line.

Typeout:

```
2 BBB
```

Current line: 2

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 five auxiliary buffers available for use.

The most common usage of auxiliary buffers is for moving or copying text from one part of a file to another.

To make available an auxiliary buffer and to put lines into it, specify the move, move-append, copy, or copy-append directive, which are described below.

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:

- 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).
- 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)
- Make specified auxiliary buffer the current buffer (change buffer directive).
- Cause Editor to accept subsequent text from a specified auxiliary buffer.
  — During input mode (change origin of text during input mode).
  — During edit mode (change origin of text during edit mode)
- Cause Editor to accept a line from operator terminal (accept single line from operator terminal).
- Determine status of each buffer in use (buffer status directive).

ACCEPT SINGLE LINE FROM OPERATOR TERMINAL

The escape sequence !R permits a single line of directives or text to be entered through the user terminal. !R normally is used when Editor directives are being executed from a buffer. When the 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
```

The above directives are in the buffer that is being executed.

There will be the following typeout 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 DIRECTIVE

The buffer status directive (X) causes a typeout of the status of each buffer currently in use. The current line is not changed.

FORMAT:

```
X
```

The following information is designated:

- Name of each buffer. The *original* current buffer always is 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 typeout includes the pathname specified in the last read or write.

If the contents of the current buffer have been modified (i.e., in the typeout, 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 Editor directives.
- The contents of the buffer have *not* been written to a file.

Each typeout 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

Reads the contents of USE into the current buffer, which is named 0.

1,$S*BBB*XXX*

Searches 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

Moves 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

Requests the status of each buffer currently in use. The following typeout will be issued:

```
2  ->MOD  (0)  USE
2         (2)
```

CHANGE BUFFER DIRECTIVE

The change buffer directive designates 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 DESCRIPTION:

x

Buffer name. The name must be 1 to 16 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 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

The escape sequence !B causes the 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 Editor encounters this sequence in an expression, pathname, or text, the entire escape sequence is removed from the input stream and replaced with 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 Editor requests, literal text or both. If the 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 which 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.

FORMAT:

!Bx

ARGUMENT DESCRIPTION:

x

Name of the buffer that contains subsequent Editor text. The buffer name must be 1 through 16 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.

CHANGE ORIGIN OF TEXT DURING INPUT MODE

The escape sequence !B can appear within text of an input directive, causing the Editor to accept subsequent text from a specified auxiliary buffer.

When the 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 also will be replaced with the contents of the named buffer.

FORMAT:

```
[text]!Bx [[text]!B] . . .
```

ARGUMENT DESCRIPTIONS:

x

Name of the buffer that contains subsequent Editor text. The buffer name must be 1 through 16 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 DIRECTIVE

The copy directive writes into a specified auxiliary buffer a single line or consecutive lines that are in the current buffer. The lines in the current buffer are *not* 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:

$$\left[\mathrm{adr}_1\left[\begin{Bmatrix};\\,\end{Bmatrix}\mathrm{adr}_2\right]\right]\mathrm{Kx}$$

ARGUMENT DESCRIPTIONS:

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 DIRECTIVE

The copy-append directive (!K) writes 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:

$$\left[\text{adr}_1 \left[\begin{Bmatrix} ; \\ , \end{Bmatrix} \text{adr}_2 \right] \right] \text{!Kx}$$

ARGUMENT DESCRIPTIONS:

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)	MMM
(2) BBB	NNN
(3) CCC	
(4) DDD	

They will contain the following after execution:

Current	*ABUF*
(1) AAA	MMM
(2) BBB	NNN
(3) CCC	AAA
(4) DDD (current line)	BBB
	CCC

MOVE DIRECTIVE

The move directive moves 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 overlaid.

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:

$$\left[\text{adr}_1\left[\begin{Bmatrix} ; \\ , \end{Bmatrix}\text{adr}_2\right]\right]\text{Mx}$$

ARGUMENT DESCRIPTIONS:

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 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:

```
1,3M5
```

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 DIRECTIVE

The move-append directive (!M) moves 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:

$$\left[\mathrm{adr}_1\left[\begin{Bmatrix};\\,\end{Bmatrix}\mathrm{adr}_2\right]\right]\mathrm{!Mx}$$

ARGUMENT DESCRIPTIONS:

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 16 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)	AAAAA
(2) SECOND	BBBBB
(3) THIRD	
(4) FOURTH	

The buffers will contain the following after the move:

Current	*SOOZ*
(1) FOURTH (current line)	AAAAA
	BBBBB
	FIRST

SECOND
THIRD

EDITOR DEBUGGING DIRECTIVES

The functions of Editor debugging directives are listed below.

- Print contents of specified line(s) on the user terminal (hexadecimal dump directive).
- Display, on the user-out file, the last specified regular expression (ZREGEXP directive).

HEXADECIMAL DUMP DIRECTIVE

The hexadecimal dump directive (ZDUMP) prints the contents of specified line(s) on the user's 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:

$$\left[\text{adr}_1\left[\begin{Bmatrix};\\,\end{Bmatrix}\text{adr}_2\right]\right]\text{ZDUMP}$$

ARGUMENT DESCRIPTION:

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 user's 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_{16}); 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_{16}). 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 DIRECTIVE

The ZREGEXP directive displays, on the user-out file, the last specified expression. The current line is not changed.

FORMAT:

```
ZREGEXP
```

Example:

```
S/ABC/DEF/
ZREGEXP
```

This ZREGEXP directive displays the last specified expression; i.e., /ABC/.

EDITOR PROGRAMMING DIRECTIVES

Editor programming directives cause conditional execution of subsequent directives, change the location of subsequent 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 Editor directives. The directives must be on a single line. If the specified condition exists, the subsequent imbedded 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 not empty 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 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 DIRECTIVE

If a specified line exists in the current buffer, directives in the address prefix directive line are executed; otherwise, they are not.

FORMAT:

?adr $\left\{\begin{matrix};\\,\end{matrix}\right\}$ directive $\left[\text{directive}\right]$ ...

ARGUMENT DESCRIPTIONS:

adr

Address of the line for which the 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 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 DIRECTIVE

The go to directive changes the location from which the Editor accepts subsequent directives.

If the go to directive is encountered in the buffer that is currently being executed, the Editor accepts subsequent directives from a specified location in that buffer. The location must have been previously defined in a label directive.

If the go to directive is entered interactively, only directives in the current directive line are accessed.

FORMAT:

```
>label
```

ARGUMENT DESCRIPTION:

label

Location to which control is transferred; the 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

Assigns label REPEAT to print directive line.

1,$S/MASS$/MASSACHUSETTS/P

Substitutes 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 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

Substitutes 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 the above directive is entered interactively, the actions listed below take place. The information to the right of each action indicates how the action is requested in the above 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 EMPTY DIRECTIVE

If the current buffer is empty, the directive(s) in the if empty directive line are executed; otherwise, they are not.

FORMAT:

^ #directive [directive] ...

ARGUMENT DESCRIPTION:

directive

Any Editor directive(s); they are executed only if the current buffer does not contain data.

IF NOT EMPTY DIRECTIVE

If the current buffer contains data, the directive(s) in the if not empty directive line are executed; otherwise, they are not.

FORMAT:

#directive [directive] ...

ARGUMENT DESCRIPTION:

directive

Any Editor directive(s); they are executed only if the current buffer contains data.

IF LINE DIRECTIVE

If the current line *is* the specified line, the if line directive(s) are executed; otherwise, they are not.

FORMAT:

adr#directive [directive] ...

ARGUMENT DESCRIPTIONS:

adr

Address of the line being checked to see if it is the current line.

directive

Any Editor directive(s); they are executed only if the specified line is the current line.

IF NOT LINE DIRECTIVE

If the current line is *not* the specified line, the if not line directive(s) are executed; otherwise, they are not.

FORMAT:

adr ^ #directive [directive] ...

ARGUMENT DESCRIPTIONS:

adr

Address of the line being checked to see if it is the current line.

directive

Any Editor directive(s); they are executed only if the specified line is *not* the current line.

IF RANGE DIRECTIVE

If the current line is within specified lines, directive(s) on the if range directive line are executed; otherwise, they are not.

FORMAT:

$$\text{adr}_1 \begin{Bmatrix} ; \\ , \end{Bmatrix} \text{adr}_2\text{\#directive} \begin{bmatrix}\text{directive}\end{bmatrix} \ldots$$

ARGUMENT DESCRIPTIONS:

adr_1

Address of the first line to be searched.

adr_2

Address of the last line to be searched.

directive

Any Editor directive(s); they are executed only if the current line is within addresses adr_1 through adr_2.

IF NOT RANGE DIRECTIVE

If the current line is *not* within specified lines, directive(s) on the if not range directive line are executed; otherwise, they are not.

FORMAT:

$$\text{adr}_1 \begin{Bmatrix} ; \\ , \end{Bmatrix} \text{adr}_2\,\text{^\#directive} \begin{bmatrix}\text{directive}\end{bmatrix} \ldots$$

ARGUMENT DESCRIPTIONS:

adr_1

Address of the first line to be searched.

adr_2

Address of the last line to be searched.

directive

Any Editor directive(s); they are executed only if the current line is *not* within addresses adr_1 through adr_2.

Example:

```
1,10 ^ #S/yes/no/
```

This if not range directive specifies that if the current line is not within lines one through ten, in the current line substitute each occurrence of "yes" with "no."

SEARCH DIRECTIVE

If a specified expression is within specified lines, directive(s) on the search directive line are executed; otherwise, they are not.

FORMAT:

$$\text{adr}_1 \begin{Bmatrix} ; \\ , \end{Bmatrix} \text{adr}_2\text{*/regexp/directive} \begin{bmatrix}\text{directive}\end{bmatrix} \ldots$$

ARGUMENT DESCRIPTIONS:

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 Editor is searching.

directive

Any Editor directive(s); they are executed only if the specified expression is within the specified addresses.

SEARCH NOT DIRECTIVE

If a specified expression is *not* within specified lines, directive(s) on the search not directive line are executed; otherwise, they are not.

FORMAT:

adr$_1$ $\begin{Bmatrix} ; \\ , \end{Bmatrix}$ adr$_2$ ^*/regexp/directive $\begin{bmatrix} \text{directive} \end{bmatrix}$ ...

ARGUMENT DESCRIPTIONS:

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 Editor is searching.

directive

Any Editor directive(s); they are executed only if the specified expression is *not* within the specified addresses.

LABEL DIRECTIVE

The label (:) directive defines a location to which the 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 $\begin{bmatrix} \text{directive} \end{bmatrix}$ ...

ARGUMENT DESCRIPTION:

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 Editor directive(s); they are executed when control passes to the specified label.

TYPE DIRECTIVE (T)

The type directive displays 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 typeout, on the same line; otherwise, the next typeouts are on subsequent lines.

FORMAT:

```
[!]T/text/
```

ARGUMENT DESCRIPTIONS:

/

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 typeout 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. A tab feature exists within program preparation. Tabbing causes embedded tab characters to be replaced with the appropriate number of spaces so that printed output on a printer, operator terminal, or other terminal has "tab stops" at character position 11 and at every subsequent 10 character positions. Tab characters can be entered into 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 Editor uses a minimum of two temporary work files in the working directory. These files are created by the Editor when the Editor is invoked; they exist only during the

current execution of the Editor. A minimum of 20 diskette or 10 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. A quit directive must be entered so that the Editor will close and release temporary work files created in the working directory.
4. 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.
5. If a file manager error (190223, lack of space) or a physical error (190107) is encountered, use the quit (Q) directive to exit from the 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 Editor may terminate processing and a fatal error message is issued.

Section 3

Language Processors

This section describes how to load each language processor.

LOADING AND EXECUTING THE MACRO PREPROCESSOR

To load and execute the Macro Preprocessor, enter the MACROP command, which is described below.

After the Macro Preprocessor is loaded, there is a typeout to the error-out file of the revision number, in the following format:

MACROP-nnnn-mm/dd/hh/mm

where nnnn is a release identification and mm/dd/hhmm is the Macro Preprocessor link date and time (mm-month, dd-day, hh-hour, mm-minutes).

Macro Preprocessor output is generated as the file path.A in the working directory.

Note:

Path is the simple pathname, excluding the suffix appended by the Macro Preprocessor.

FORMAT:

MACROP path [ctl_arg]

ARGUMENT DESCRIPTIONS:

path

Pathname of the unexpanded source unit file to be processed by the Macro Preprocessor. Omit the suffix.

ctl_arg

Control arguments; none or any number of the following control arguments may be entered, in any order:

$\left\{\begin{array}{l}\text{-INCLUDE\_CONTROLS}\\ \text{-IC}\end{array}\right\}$

Instructs the Macro Preprocessor to incorporate as comment statements in the expanded source output all macro control statements and inline macro definitions.
Default: Exclusion of such comments from the expanded source output.

$\left\{\begin{array}{l}\text{-MACRO\_CALLS}\\ \text{-MC}\end{array}\right\}$

Instructs the Macro Preprocessor to incorporate all macro call statements as comment statements in the expanded source output.
Default: Exclusion of such comments from the expanded source output.

$\left\{\begin{array}{l}\text{-SIZE nn}\\ \text{-SZ nn}\end{array}\right\}$

nn designates the maximum number of 1024-word blocks of memory that the Macro Preprocessor may use for work space. nn must be from 01 through 64.
Default: 2K words (2048_{10})

Notes:

1. The Macro Preprocessor always issues a typeout, of the number of errors found, to the error-out file.
2. If an unexpanded source statement contains an error, the Macro Preprocessor flags the statement, converts it to a comment statement, and writes it to the expanded source file.

LOADING AND EXECUTING THE ASSEMBLER

To load and execute the Assembler, enter the ASSEM command which is described below.

After the Assembler is loaded, there is a typeout to the error-out file of the revision number, in the following format:

```
ASSEM-nnnn-mm/dd/hhmm
```

where nnnn is a release identification and mm/dd/hhmm is the Assembler link date and time (mm-month, dd-day, hh-hour, mm-minutes).

FORMAT:

```
ASSEM path [ctl_arg]
```

ARGUMENT DESCRIPTIONS:

path

Pathname of the source unit file to be assembled. Omit the suffix.

ctl_arg

Control arguments; none or any number of the following control arguments may be entered, in any order.

-COUT out_path

Listing will be written to the file out_path; a suffix is *not* appended to the file name. If this argument is omitted, the listing will be written to the file path.L in the working directory.

Note:

Path is the simple pathname, excluding the suffix appended by the Assembler.

$\left\{\begin{array}{l}\text{-LAF}\\ \text{-SAF}\\ \text{-SLIC}\end{array}\right\}$

Addressing mode in which the source unit will be assembled. -LAF designates long-address form; -SAF designates short-address form. -SLIC designates that the source unit will be able to execute in either SAF or LAF.

Default: The mode configuration in which the Assembler is executing (must be SAF or LAF).

$\left\{\begin{array}{l}\text{-LIST\_ERRS}\\ \text{-LE}\end{array}\right\}$

Specifies that only those source lines containing assembly errors, together with their error codes, are to be listed.

Default: If omitted, and -NL is not specified, the complete source program is listed, including error codes, if any.

$\left\{\begin{array}{l}\text{-NO\_LIST}\\ \text{-NL}\end{array}\right\}$

Suppresses source listing.
Default: Source listing produced.

$\left\{\begin{array}{l}\text{-NO\_OBJ}\\ \text{-NO}\end{array}\right\}$

Suppresses object text unit output.
Default: Object text unit is generated as the file path.O in the working directory.

Note:

Path is the simple pathname, excluding the suffix appended by the Assembler.

$\left\{\begin{array}{l}\text{-SIZEnn}\\ \text{-SZ nn}\end{array}\right\}$

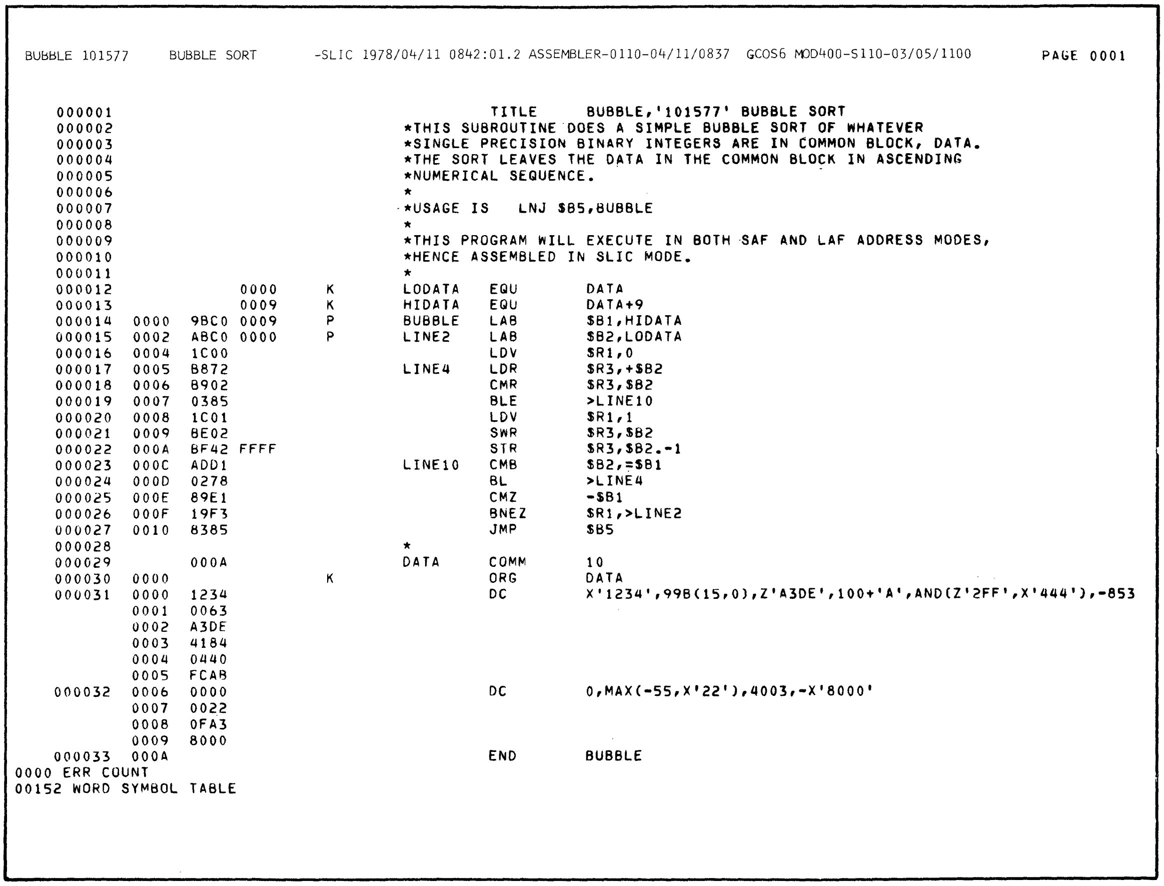
nn designates the maximum number of 1024-word memory blocks that may be used for the Assembler's symbol table and for producing a cross-reference listing, if requested. nn must be numeric and be from 01 through 64.
Default: 1K words (1024_{10})

$\left\{\begin{array}{l}\text{-XREF}\\ \text{-CROSS\_REF}\end{array}\right\}$

Produces a cross-reference listing, even if -NL or -LE is specified. The listing is appended to the source listing. If there is no source listing, the cross-reference list will be produced anyway. Figure 3-1 illustrates the source listing of a sample source unit and Figure 3-2 illustrates the cross-reference listing of that unit.

Notes:

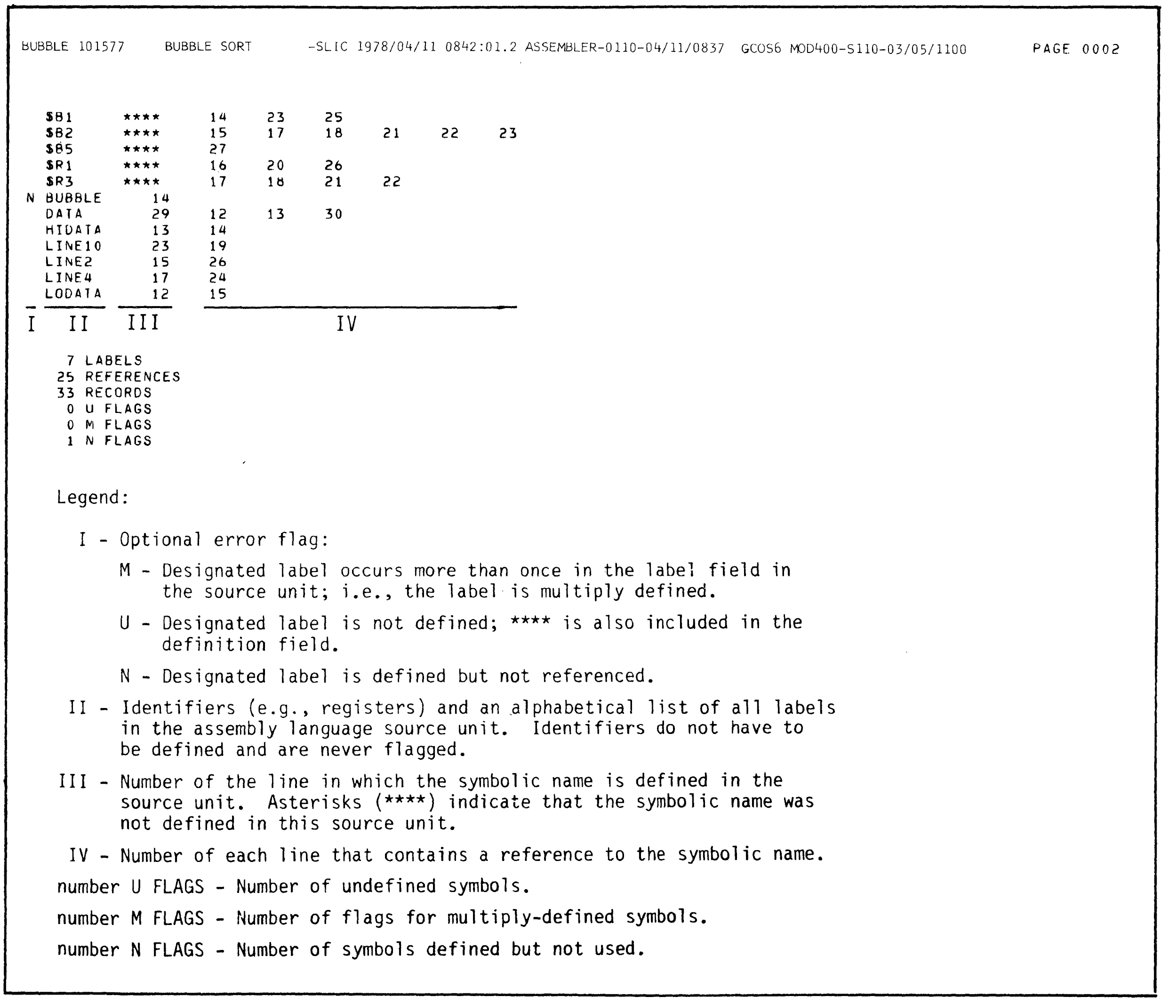
1. Files created and used during current execution of the component (ASSEM with -CROSS_REF or -XREF) are released upon termination of the execution.
2. The Assembler creates a file named path.W in the working directory. This file is deleted by the Assembler when the cross-reference listing is produced.
3. The Assembler always issues a typeout, of the number of errors found, to the error-out file.



```
BUBBLE 101577      BUBBLE SORT       -SLIC 1978/04/11 0842:01.2 ASSEMBLER-0110-04/11/0837  GCOS6 MOD400-S110-03/05/1100        PAGE 0001

    000001                                        TITLE     BUBBLE,'101577' BUBBLE SORT
    000002                               *THIS SUBROUTINE DOES A SIMPLE BUBBLE SORT OF WHATEVER
    000003                               *SINGLE PRECISION BINARY INTEGERS ARE IN COMMON BLOCK, DATA.
    000004                               *THE SORT LEAVES THE DATA IN THE COMMON BLOCK IN ASCENDING
    000005                               *NUMERICAL SEQUENCE.
    000006                               *
    000007                               *USAGE IS   LNJ $B5,BUBBLE
    000008                               *
    000009                               *THIS PROGRAM WILL EXECUTE IN BOTH SAF AND LAF ADDRESS MODES,
    000010                               *HENCE ASSEMBLED IN SLIC MODE.
    000011                               *
    000012             0000      K       LODATA   EQU       DATA
    000013             0009      K       HIDATA   EQU       DATA+9
    000014  0000  9BC0 0009      P       BUBBLE   LAB       $B1,HIDATA
    000015  0002  ABC0 0000      P       LINE2    LAB       $B2,LODATA
    000016  0004  1C00                            LDV       $R1,0
    000017  0005  B872                   LINE4    LDR       $R3,+$B2
    000018  0006  B902                            CMR       $R3,$B2
    000019  0007  0385                            BLE       >LINE10
    000020  0008  1C01                            LDV       $R1,1
    000021  0009  BE02                            SWR       $R3,$B2
    000022  000A  BF42 FFFF                       STR       $R3,$B2.-1
    000023  000C  ADD1                   LINE10   CMB       $B2,=$B1
    000024  000D  0278                            BL        >LINE4
    000025  000E  89E1                            CMZ       -$B1
    000026  000F  19F3                            BNEZ      $R1,>LINE2
    000027  0010  8385                            JMP       $B5
    000028                               *
    000029        000A                   DATA     COMM      10
    000030  0000                 K                ORG       DATA
    000031  0000  1234                            DC        X'1234',99B(15,0),Z'A3DE',100+'A',AND(Z'2FF',X'444'),-853
            0001  0063
            0002  A3DE
            0003  4184
            0004  0440
            0005  FCAB
    000032  0006  0000                            DC        0,MAX(-55,X'22'),4003,-X'8000'
            0007  0022
            0008  0FA3
            0009  8000
    000033  000A                                  END       BUBBLE
0000 ERR COUNT
00152 WORD SYMBOL TABLE
```

Figure 3-1. Source Listing of Source Unit to be Cross Referenced



```
BUBBLE 101577     BUBBLE SORT      -SLIC 1978/04/11 0842:01.2 ASSEMBLER-0110-04/11/0837  GCOS6 MOD400-S110-03/05/1100      PAGE 0002


   $B1      ****     14    23    25
   $B2      ****     15    17    18    21    22    23
   $B5      ****     27
   $R1      ****     16    20    26
   $R3      ****     17    18    21    22
 N BUBBLE      14
   DATA        29    12    13    30
   HIDATA      13    14
   LINE10      23    19
   LINE2       15    26
   LINE4       17    24
   LODATA      12    15
 _ ______   ______  _____________________________
 I   II      III                IV

     7 LABELS
    25 REFERENCES
    33 RECORDS
     0 U FLAGS
     0 M FLAGS
     1 N FLAGS
```

Legend:

I - Optional error flag:

M - Designated label occurs more than once in the label field in the source unit; i.e., the label is multiply defined.

U - Designated label is not defined; **** is also included in the definition field.

N - Designated label is defined but not referenced.

II - Identifiers (e.g., registers) and an alphabetical list of all labels in the assembly language source unit. Identifiers do not have to be defined and are never flagged.

III - Number of the line in which the symbolic name is defined in the source unit. Asterisks (****) indicate that the symbolic name was not defined in this source unit.

IV - Number of each line that contains a reference to the symbolic name.

number U FLAGS - Number of undefined symbols.

number M FLAGS - Number of flags for multiply-defined symbols.

number N FLAGS - Number of symbols defined but not used.

Figure 3-2. Sample Cross-Reference Listing

FORTRAN

LOADING AND EXECUTING THE FORTRAN COMPILER

To load and execute the FORTRAN Compiler, enter the FORTRAN command, which is described below.

After the FORTRAN Compiler is loaded, it issues a component identification to the error-out file in the following format.

```
FORTRAN nnnn mm/dd/hhmm
```

where nnnn is a release identification and mm/dd/hhmm is the compiler link date and time (mm-month, dd-day, hh-hour, mm-minutes).

FORMAT:

```
FORTRAN path [ctl_arg]
```

ARGUMENT DESCRIPTIONS:

path

Pathname of the source unit file to be compiled. Omit the suffix (.F).

ctl_arg

Control arguments; none or any number of the following control arguments may be entered, in any order:

-AS

Output is assembly language text contained in the file path.A. This file can be used with the -SAF option as input to the Assembler. Modifications may be required to assemble the file with the -LAF option (see the *Assembly Language Reference* manual).

Default: If omitted and -NO is *not* specified, an object text unit is produced as file path.O.

·COUT out_path

Listing will be written to the file out_path; a suffix is *not* appended to the file name. If this argument is omitted, the listing will be written to the file path.L in the working directory.

-FS

The compiler will not define the size of the work area when compiling a subroutine, nor will it cause the implicit initialization of the work area when the subroutine is executed. See note 3, below.

-HS

The source unit comprises Hollerith code or the source unit was created using a Series 200/2000 or Model 716 Central Processor.

$\begin{Bmatrix}\text{-LIST\_ERRS}\\\text{-LE}\end{Bmatrix}$

Specifies that only those source lines containing compilation errors, together with their error codes, are to be listed.

Default: If omitted, and -NL is not specified, the complete source program is listed, including error codes, if any.

$\begin{Bmatrix}\text{-LIST\_OBJ}\\\text{-LO}\end{Bmatrix}$

List object output. Object text listings in assembly language format will be interspersed with source text listings.

Default: Object text is not listed.

Note:

This argument is not meaningful when used with the -AS argument.

$\left\{\begin{array}{l}\text{-NO\_LIST}\\ \text{-NL}\end{array}\right\}$

Suppress all listings.

Default: If omitted and if -LE is not specified, the complete source program is listed, including error codes, if any.

$\left\{\begin{array}{l}\text{-NO\_OBJ}\\ \text{-NO}\end{array}\right\}$

Generation of the object text unit is suppressed. (This option should not be used in conjunction with -AS.)

Default: If omitted and -AS is not specified, an object text unit is produced as file path.O.

-SI

One word is allocated for each integer and logical variable.

Default: Two words are allocated for each integer and logical variable.

Note:

This argument affects space allocation only. The range of values of integer and logical variables is the same regardless of whether the argument is specified.

$\left\{\begin{array}{l}\text{-SIZE nn}\\ \text{-SZ nn}\end{array}\right\}$

nn designates the maximum number of 1024-word blocks of memory that the compiler can use for tables. If the requested amount of memory is not available, the compiler will use the available amount of memory.

Default: Available memory in the task group's memory pool; up to approximately 1700 words is used. There must be at least 1K words available.

-UC

Suppress generation of embedded links to any subroutines referenced by a CALL statement, and to functions other than intrinsic functions.

-UZ

Suppress generation of embedded links to system subroutines (i.e., all subroutines beginning with the letters ZF).

-WRK n

Establishes the size in words of the object time workspace for FORTRAN programs. "n" specifies the number of words and must be a one- to four-digit decimal number from 1 to 9999. See Note 3 below.

Default: 356 words.

$\left\{\begin{array}{l}\text{-LAF}\\ \text{-LA}\end{array}\right\}$

Specifies that long address form (LAF) object text is to be generated.

Default: Short address form (SAF) object text is generated.

Note:

This argument is not meaningful when used with the -AS argument.

Notes:

1. Either LO or NL may be specified, but not both. If neither is specified, the compiler produces a listing of the source text and diagnostics.
2. The FORTRAN Compiler always issues a typeout, of the number of errors found, to the error-out file.
3. Most FORTRAN programs call input/output routines and intrinsic functions, the majority of which utilize a workspace. Prior to the invocation of any one of these modules (routines or functions) the workspace must be initialized. The FORTRAN Compiler automatically generates a workspace declaration and the prologue code for initialization of the workspace in each main program and each subprogram for which the -FS argument is not specified (the -FS argument is ignored if specified for a main program). Use of the -FS argument implies that the subprogram either does not need the workspace or depends upon another program to declare and initialize the workspace.

   To avoid execution time errors involving use of the workspace, the declarations of workspace in modules linked together in a bound unit must be identical. Therefore, if you want to create general purpose subroutines to be used in applications which require workspace areas of various sizes, you should compile the subroutines with the -FS argument.

   In some applications, variations in the workspace size may be necessary to increase of decrease the default input/output buffer space of 128 words. The -WRK arugment is used to make this modification. For details, refer to the *FORTRAN Reference* manual.
4. The compiler is designed for batch compilations. That is, many source modules can be passed to the compiler under one file name and each source module will be compiled separately. The compiler expects an END statement in each source module, followed by either an end-file or a new source module. In addition, the SI, WRK, FS and HS arguments can be passed to the compiler as part of the source module, rather than as arguments of the FORTRAN command. Arguments specified in the FORTRAN command apply to all source modules in the batch. Including these arguments in the source module permits them to be varied on a module-by-module basis. To include these arguments in a source module, enter each as a special comment immediately following the PROGRAM, SUBROUTINE or FUNCTION statement for the module. The general form is:

$$\text{C*OPT=}\left\{\begin{array}{l}\text{SI}\\ \text{WRK=n}\\ \text{FS}\\ \text{HS}\end{array}\right\}$$

For example,

```
C*OPT=WRK=400
```

Note that the work area size argument requires an equals sign when it is specified as part of a source module. More than one argument may be specified, but each requires a separate comment line.

LOADING AND EXECUTING THE ENTRY-LEVEL COBOL COMPILER

To load and execute the Entry-Level COBOL Compiler, enter the COBOL command, which is described below.

After the Entry-Level COBOL Compiler is loaded, there is a typeout to the error-out file of the revision number, in the following format:

COBOL nnnn mm/dd/hhmm

where nn is a release identification, mm/dd/hhmm is the compiler link date and time (mm-month, dd-day, hh-hour, mm-minutes).

FORMAT:

COBOL path [ctl_arg]

ARGUMENT DESCRIPTIONS:

path

Pathname of the source unit file to be compiled. Omit the suffix. The name must be the same as that specified in the PROGRAM-ID clause of the COBOL source program.

ctl_arg

Control arguments; none or any number of the following control arguments may be entered, in any order:

-COUT out_path

Listing will be written to the file out_path; a suffix is *not* appended to the file name. If this argument is omitted, the listing will be written to the file path.L in the working directory. If a file other than the printer is requested, the file must already exist.

Note:

Path is the simple pathname, excluding the suffix appended by the COBOL Compiler.

-DB

Compile debugging lines as comments, ignoring the WITH DEBUGGING MODE clause.

$\left\{\begin{array}{l}\text{-NO\_OBJ}\\ \text{-NO}\end{array}\right\}$

Suppress object unit output.

Default: Object unit output produced as the file path.O in the working directory.

Note:

Path is the simple pathname, excluding the suffix appended by the COBOL Compiler.

$\left\{\begin{array}{l}\text{-SIZE nn}\\ \text{-SZ nn}\end{array}\right\}$

Requests nn additional 1024-word blocks of memory for compiler tables. nn must be from 04 to 64. The additional memory specified in the argument is used instead of the original table size, and permits the COBOL Compiler to improve performance when compiling large programs. If you request more memory than is available, the compiler uses the available amount of memory. If specified, at least 3072 words must be available; otherwise, the compiler will use the default memory size (3000 words). If this argument is not specified, the compiler has approximately 3000 words of memory for table space.

Note:

The following control arguments are listing options. Only one listing option may be specified at a time. Further, if no listing option is chosen and -NL is not specified, the complete source program (along with any error codes) is listed. This is the default for all listing options shown here.

-LD

List data map, source text, errors, and file map.

{-LIST_ERRS
-LE}

* Specifies that only the error list is to be printed.

{-LIST_OBJ
-LO}

List source text, data map, errors, file map, and object code.

{-NO_LIST
-NL}

Suppress all listings.

-XREF

Specifies that a cross-reference listing is to be produced. A listing option other than -NL must be specified.

Note:

The Entry-Level COBOL Compiler always issues a typeout, of the number of errors found, to the error-out file.

LOADING AND EXECUTING THE INTERMEDIATE COBOL COMPILER (COBOLI)

To load and execute the Intermediate COBOL Compiler, enter the COBOLI command, which is described below.

After the Intermediate COBOL Compiler is loaded, there is a typeout to the error-out file of the revision number, in the following format:

```
COBOLI nnnn mm/dd/hhmm
```

where nnnn is a release identification, mm/dd/hhmm is the compiler link date and time (mm-month, dd-day, hh-hour, mm-minutes).

FORMAT:

```
COBOLI path [ctl_arg]
```

ARGUMENT DESCRIPTIONS:

path

Pathname of the source unit file to be compiled. Omit the suffix. The name must be the same as that specified in the PROGRAM-ID clause of the COBOL source program.

ctl_arg

Control arguments; none or any number of the following control arguments may be entered, in any order:

-COUT out_path

Listing will be written to the file out_path; a suffix is *not* appended to the file name. If this argument is omitted, the listing will be written to the file path.L in the working directory. If a file other than the printer is requested, the file must already exist.

Note:

Path is the simple pathname, excluding the suffix appended by the Intermediate COBOL Compiler.

-DB

Compile debugging lines as comments, ignoring the WITH DEBUGGING MODE clause. *

{-NO_OBJ
-NO}

Suppress object unit output.

Default: Object unit output produced as the file path.O in the working directory.

Note:

Path is the simple pathname, excluding the suffix appended by the COBOL Compiler.

{-SIZE nn
-SZ nn}

Requests nn additional 1024-word blocks of memory for compiler tables. nn must be from 15 to 64. The additional memory specified in this argument is used instead of the original table size, and permits the COBOL Compiler to improve performance when compiling large programs. If you request more memory than is available, the compiler uses the available amount of memory. If specified, at least 15,000 words must be available; otherwise, the compiler will use the default memory. If this argument is not specified, the compiler has approximately 14,000 words of memory for table space.

Note:

The following control arguments are listing options. Only one listing option may be specified at a time. Further, if no listing option is chosen and -NL is not specified, the complete source program (along with any error codes) is listed. This is the default for all listing options shown here.

-LD

List data map, source text, errors and file map.

{-LIST_ERRS
-LE}

Specifies that only the error list is to be printed. *

{-LIST_OBJ
-LO}

List source text, data map, errors, file map, and object code.

-XREF

Specifies that a cross-reference listing is to be produced. A listing option other than -NL must be specified.

{-NO_LIST}
{-NL}

Suppress all listings.

Note:

The Intermediate COBOL Compiler always issues a typeout, of the number of errors found, to the error-out file.

COBOL COPY FILES

An Intermediate COBOL source program may contain statements in the form: COPY text-name. The COBOL Compiler treats the text-name as a file name and appends the suffix ".IN.C". This restricts the user-created portion of the text name to seven characters (i.e., overall file name length cannot exceed twelve characters).

When a file containing text is requested, the compiler searches up to three directories for text-name .IN.C in the following order:

- the current working directory
- >UDD>account>INCLUDE, where "account" would be the user identification account field
- >LDD>INCLUDE

If the file is not found in any of these directories, an error message is produced and the COPY statement is bypassed by the compiler.

LOADING AND EXECUTING THE RPG COMPILER

To load and execute the RPG Compiler, enter the RPG command, which is described below.

After the RPG Compiler is loaded, there is a typeout to the error-out file of the revision number, in the following format:

```
RPG nnnn
```

FORMAT:

```
RPG path [ctl_arg]
```

ARGUMENT DESCRIPTIONS:

path

Pathname of the source unit file to be compiled, without the suffix ".R". The suffix .R is automatically appended before the search for the source unit file.

ctl_arg

Control arguments; none or any number of the following control arguments may be entered, in any order:

{-COUT} out_path
{-C}

Listing will be written to the file out_path; out_path may specify the line printer. A suffix is *not* automatically appended to the out_path. If this argument is omitted, the listing will be written to the file source.L in the working directory, where "source" is the simple name of the source unit file to be compiled. In either case, the file can later be written to a line printer by using the print utility command.

{-LIST_OBJ}
{-LO}

List data map and object text in addition to the source text, diagnostics, and Linker commands.

{-NO_LIST}
{-NL}

Suppress all listings.

{-NO_OBJ}
{-NO}

Suppress object unit output.
Default: Object units are produced in the working directory as a number of files, each with a ".O" suffix and a compiler-generated base name. If a particular .O file already exists in the working directory, it is overlaid by the output of the current compilation.

{-SIZE nn}
{-SZ nn}

nn designates the maximum number of 1024-word blocks of memory that the RPG Compiler may use for tables; nn must be from 04 to 28. *
Default: 03

Notes:

1. Either LO or NL may be specified, but not both. If neither is specified, the compiler produces a listing of the source text, diagnostics and the Linker command file.
2. The RPG Compiler always writes the number of diagnostics produced to the error-out file.

Appendix A

Assembly Language Program Independence

ASSEMBLY LANGUAGE PROGRAM HARDWARE INDEPENDENCE

If an assembly language program written for a model 6/30 is to be used on a model 6/40 or 6/50, the program must be written to be program independent of the hardware model. The additional features in the larger model 6/40 and model 6/50 that must be considered are instruction prefetching that affects self-modifying procedures and long address form (LAF). The GCOS 6 MOD 400 Linker produces SAF, LAF, or SAF-LAF Independent Code (SLIC) bound units.

SELF-MODIFYING PROCEDURES

Use of a self-modifying procedure should be carefully considered for two reasons: (1) a self-modifying procedure cannot be made reentrant, and (2) the instruction, as modified, might not be executed because of the instruction prefetching feature of the models 6/40 and 6/50. With instruction prefetching, an arbitrary number of words are prefetched in parallel with the execution of the current instruction. The prefetch buffer is emptied only when a transfer of control occurs. If an instruction is stored in a word that previously was prefetched, the prefetch buffer is *not* cleared and the prefetched instruction will be executed as it was prior to modification.

However, if a self-modifying procedure must be used, the program must contain code to remove the prefetched instruction after modification is complete but before the modified code is executed. This can be done by executing an unconditional branch of the form:

```
B   $+2   FLUSH THE PREFETCH
```

WRITING SOURCE PROGRAMS THAT CAN BE EXECUTED IN BOTH SAF AND LAF CONFIGURATIONS

There are two methods for writing a source program so that it can be executed in both SAF and LAF configurations: SAF/LAF independence by assembly, which produces a program that is assembled differently for each type of configuration, and SAF/LAF independence by loading, which produces a program that is assembled and linked in the same way but is loaded differently. For the second method, SAF/LAF Independent Code (SLIC) is used to create the source program.

A SLIC program consists entirely of Assembler control statements, assembly instructions, and macro calls, all of which are described in the *Assembly Language Reference* manual. These items must be selected and combined according to the rules and restrictions described in the following text. SLIC is the code that results from procedure.

As shown by Figure A-1, a program can run on a SAF and LAF configuration, if all the compilation units are SLIC compilation units and linking is done by a GCOS 6 MOD 400 Linker. When requested, the Assembler produces SLIC compilation units. However, the Assembler does not check that the code conform to the SLIC rules and restrictions. If the code does not conform, the results of the program are unpredictable.

The valid ways in which SAF and SLIC compilation units and LAF and SLIC compilation units can be linked into bound units are shown in Figure A-2.

The following system service macro calls should not be used in a program written in SAF/LAF independent code (SLIC):

$CRB
$CRBD
$IORB
$IORBD
$PRBLK
$RBD
$SRB
$SRBD
$TRB
$TRBD
$WAITL
$WLIST
$MGCRB
$MGIRB
$MGRRB
$MGCRT
$MGIRT
$MGRRT

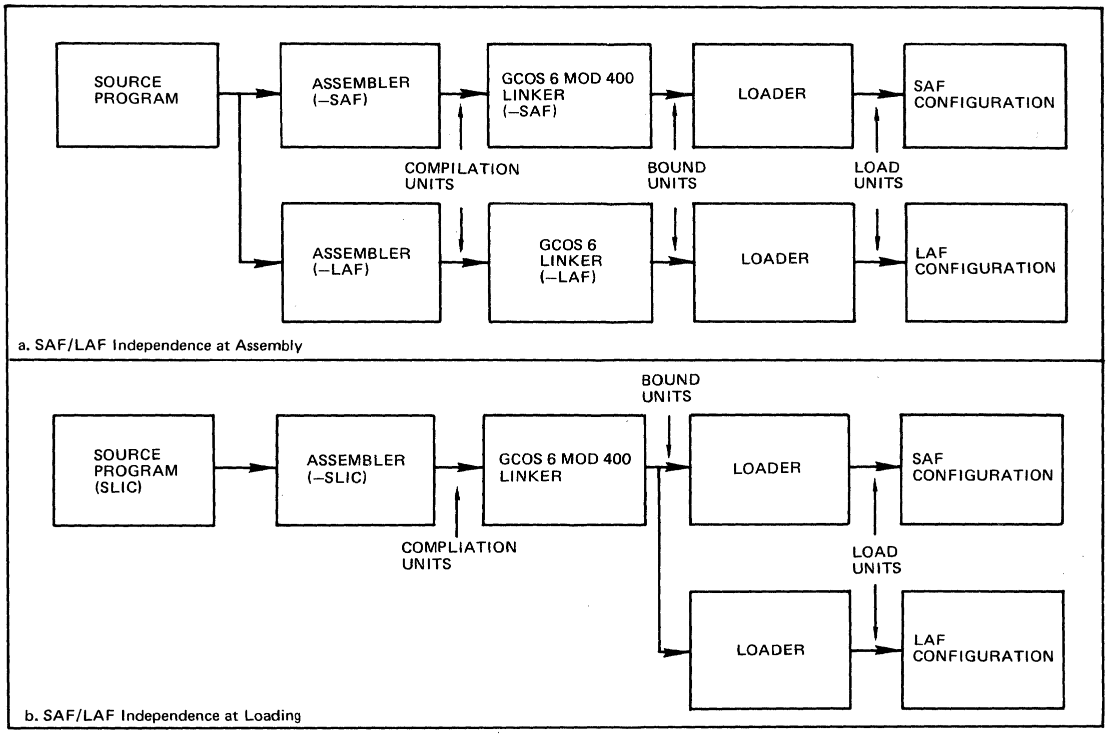


Figure A-1. Methods of Achieving SAF/LAF Independence

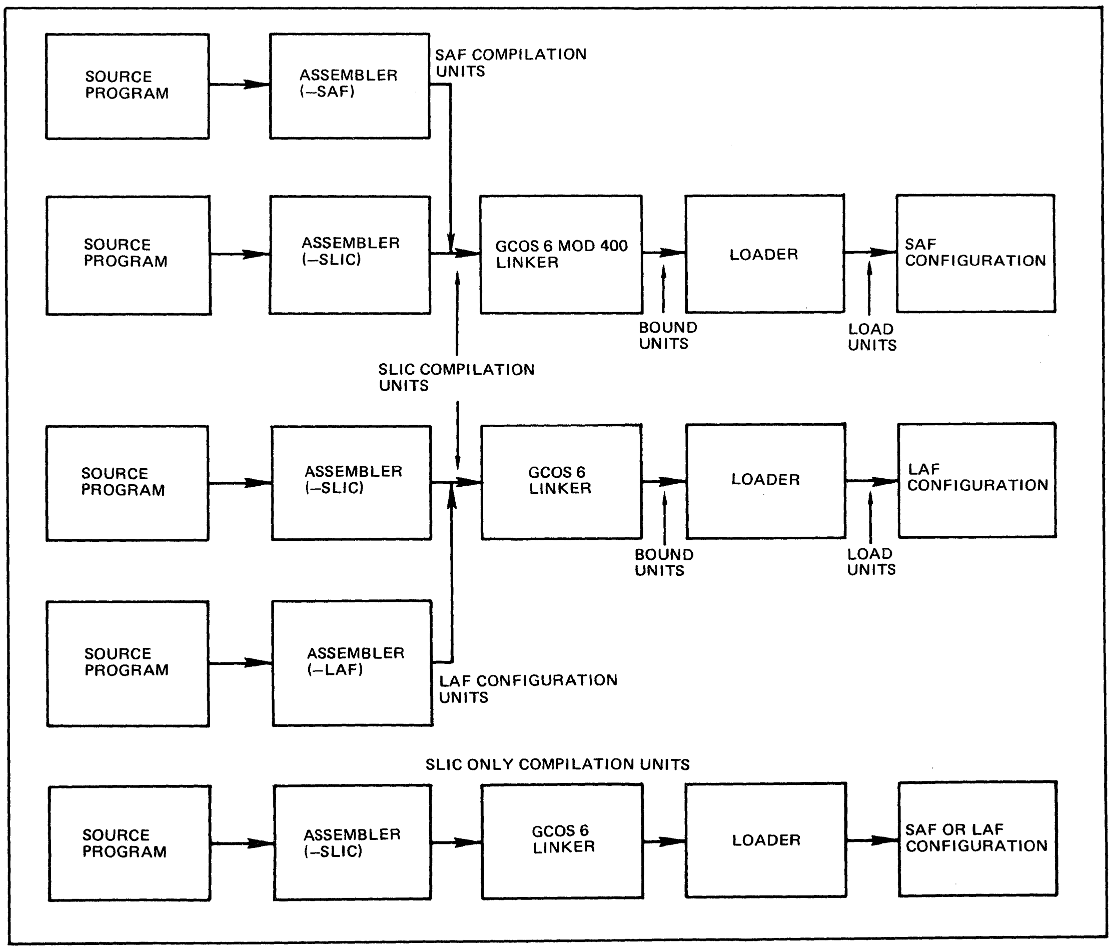


Figure A-2. Valid Combinations of Compilation Units for Linking

SAF/LAF INDEPENDENCE BY ASSEMBLY

An assembly language program assembled to execute under a SAF system can be converted to execute under a LAF system by simply reassembling the program for execution on the LAF system. Reassembly is usually possible *provided that the following rules are observed when the program is written.*

1. The program must be written so it will assemble without errors in either configuration; e.g., a short displacement branch must satisfy the conditons $-64 \leq d \leq 1$ or $2 \leq d \leq 63$ words on the LAF configuration as well as on the SAF configuration.
2. All memory locations should be referenced by their symbolic names. The assembly language label $AF can also be used in expressions to correctly reference the desired memory location; however, the $AF reference should be used with care since its use requires a good understanding of how the hardware operates.
3. Offsets to elements of a data structure containing pointers must be defined symbolically. When the data structure actually exists in another program, the assembly language label $AF can be used in an equate statement to provide the proper template.
4. All constants used in index computation to reference arrays of structures containing pointers must be symbolically defined.

   For example: if the span of an array element is "a" words plus "b" addresses, then the constant should be defined by the expression a+b*$AF. This constant can then be used to compute an index register value which is in turn used in a LAB instruction to set a base register to the beginning of the desired occurrence of the array element.
5. All fields that are to contain pointers must be defined as address constants or a reserve of $AF words. Such fields must be referenced by their symbolic names.
6. All external procedure calling sequences that modify their argument list must be designed to operate correctly, through the use of $AF, whether assembled for a SAF or LAF configuration.
7. The size of a common block that contains pointers must be specified by an expression involving the label $AF to give the correct size, whether the program is assembled for a SAF or LAF configuration.
8. All address manipulation must be performed using base registers (B1—B7). The LAB instruction with base plus displacement or base plus index addressing is useful for address manipulation.

SAF/LAF INDEPENDENCE BY LOADING

This section contains rules for writing assembly language programs that can be executed (without reassembly or relinking, but with suitable modifications by the loader) in either a SAF or LAF configuration. That is, the source language program can be assembled and linked into a bound unit. This bound unit can then be loaded and executed on either a SAF or LAF configuration.

DIFFERENCES BETWEEN SAF AND LAF

Memory is allocated and most memory addresses are determined by the Assembler or a compiler. SAF and LAF differ in their definition and use of memory addresses. This difference affects the following items:

1. Instructions or data whose size (space allocated) depends on the addressing mode; that is,
   a. Instructions that use IMA operands (and certain instructions whose operands are base registers and that use IMO operands).
   b. Declarations of memory addresses as data; that is, address constants or address variables.
2. Data whose location in memory depends on the addressing mode; in particular, data structures whose address or format is determined by hardware specifications, such as interrupt and trap vector and save areas (IV, TV, ISA, TSA).

3. References to such instructions or data. The significant instances of this are:
   a. References to (sequences of) instructions using IMA operands.
   b. References to data strucqs containing pointers.
   c. References to data structures defined by hardware.

   Such references are resolved by (1) the Assembler or compiler (for most internal or common references), (2) the Linker (for some internal references, some common references, and external references), or (3) the loader (for some external references and for relocation).
4. Memory addresses, whether in instructions or in data declarations, that contain values prior to the start of execution. The significant instances of this are:
   a. IMA operands and instructions whose operands are base registers and that use IMO operands.
   b. Declarations of pointers with initial values; that is, address constants (DC <locationexpression).

   These memory addresses must be examined because the value of the memory address must be resolved in a single word for SAF and in two words for LAF.
5. Addressing formats and instructions whose execution is different in the two addressing modes. Specifically, the addressing formats for indexing with or without pre-decrement or post-increment (the .$R, .+$R, .−$R types) and for push and pop (+$B and −$B) operate differently when used with the five base register instructions:

   LDB, STB, CMB, SWB, CMN

GENERAL RULES FOR WRITING SLIC PROGRAMS

1. Allocate two words for all memory addresses, whether they are instruction operands or data declarations. That is, generate or assemble essentially in LAF. This ensures that sufficient space is allocated to execute in LAF. (The Assembler will set $AF equal to 2 when invoked with the -SLIC control argument.)
2. When loading a SLIC program for execution in SAF, the loader will:
   a. Replace a sequence of (two word) pointers in an argument list or a pointer array by a sequence of one-word pointers followed by an equal number of one word NOPs. That is, the sequence is compressed into consecutive words. Adjustment of references to such argument lists and pointer arrays is *not* performed. In the case of an argument list, the control word is also adjusted appropriately.
   b. Replace an individual (two word) memory address, whether an instruction operand or a data item, by a single-word memory address followed by a one-word NOP. That is, the value is moved into the first of the two words. References to the leftmost of the two words work for both SAF and LAF execution.

PROCEDURES FOR WRITING SPECIFIC PARTS OF A SLIC PROGRAM

The following procedures for writing specific parts of a SLIC program are derived from the general rules described previously. Methods for handling data structures, pointers, argument lists, and other commonly used items are described.

ADDRESSING MODE

Invoke the language processor with the -SLIC argument. For the Assembler, this sets $AF equal to 2. Assembly language programs should use the ARGLST and PTRAY Assembler control statements to define argument lists and pointer arrays, respectively. The CALL statement will also generate any appropriately identified argument list. Individual pointers should be defined as address constants or by a RESV statement with the reserved label $AF.

DATA STRUCTURES CONTAINING POINTERS

The techniques used for declaring, allocating space for, and referencing data structures containing pointers differ somewhat, depending on the kind of data structure. The most commonly used data structures containing pointers are classified as follows:

- Data management structures (FIBs).
- Argument lists (in calls) and pointer arrays.
- Request blocks (RBs).
- Individual pointers.
- Hardware defined structures.

For FIBs, two words are allocated for each pointer whether execution is to be in SAF or LAF. For argument lists, pointer arrays, and request blocks, one word is allocated for each pointer when execution is to be in SAF or two words are allocated when execution is to be in LAF.

For argument lists and pointer arrays in a SLIC program, the loader compresses the sequence of two-word pointers into consecutive single-word pointers for execution in SAF. For request blocks, the loader does not compress the structure.

With this approach, software — including the Monitor — has to deal with only one form for a given system data structure. For FIBs (and individual pointers), there is only one form, regardless of the addressing mode in which the program is executing. For argument lists, pointer arrays, request blocks, and hardware defined structures, a program executing in a given addressing mode receives only the form corresponding to that address mode.

An individual pointer must always be addressed by its first (or only) word. This is how the hardware works, and is why the loader moves the value into the first word when loading for execution in SAF. (Elements of an argument list or a pointer array, other than the first, cannot be referenced symbolically, as noted later.)

References to a pointer should be with instructions that explicitly operate on addresses; e.g., LDB. Other instructions, such as those that always operate upon two-word items, should be used carefully in a SLIC program. For example, arithmetic operations cannot be performed because when they are executed in SAF, the value of a pointer appears in the high-order position (first of the two words), not in the low-order position (second of the two words) appropriate for arithmetic.

DATA MANAGEMENT STRUCTURES (FIBS)

Data management structures (FIBs) must be allocated with two words for each pointer in them. A FIB is not compressed when loaded for execution in SAF; but the loader does move the value of the pointer from the second word into the first.

This kind of structure can be declared symbolically. Honeywell supplies the declaration as a macro for use in assembly language programs.

Data items, including pointers, can be referred to symbolically via the declaration. Refer to a data item by the label assigned to it or by an expression not using $AF.

When referring to pointers with base register instructions, do *not* use the indexed, push, or pop addressing formats. These addressing formats will not work with this kind of structure, because the formats index, increment, or decrement by *one* word units when they are executed in SAF.

An initial value can be declared for any data item, including pointers.

ARGUMENT LISTS AND POINTER ARRAYS

When argument lists and pointer arrays are used as system data structures (e.g., in inter-program communication), a standard form is required. Argument lists and pointer arrays use one-word (consecutive) pointers when they are executed in SAF. They must be allocated with two-word pointers in a SLIC program, so that it can be executed in LAF. However, they are compressed by the loader when they are loaded for execution in SAF. This permits them to be declared with initial values — in particular, it minimizes the need to assign values to arguments at execution time.

Thus, when a SLIC program executes in SAF, the pointers in an argument list or a pointer array occupy consecutive words, and the remainder of the structure is initialized to a sequence of one-word NOPs.

Although this kind of structure can be declared with initial values (because it will be altered appropriately by the loader for execution in SAF), it should be referenced only by base register instructions because the addresses of the pointers in it depend on the addressing mode used at execution time.

Assembly language programs should define argument lists via the CALL statement or through use of the ARGLST Assembler control statement. Pointer arrays should be defined by the PTRAY Assembler control statement.

The first pointer of an argument list or pointer array and the control word of an argument list can be referred to symbolically. When a SLIC program is loaded for execution in SAF, the pointers are compressed from a sequence of two-word values into a sequence of one-word (consecutive) values. As a result, references to other data items in this kind of structure must be computed at execution time.

Refer to a pointer in this kind of structure only with base register instructions with indexed, push, or pop addressing formats. These addressing formats will work because they index, increment, or decrement by one-word units when executing in SAF and two-word units when executing in LAF. For example, suppose there are n elements (arguments or elements of a pointer array), and the location named N contains the desired element number in the range 1 to n. Let register B7 point either to the argument list's control word or directly to the first word of the pointer array. Then, a convenient way of referring to the desired element is:

```
For argument lists          For arrays

LAB   $B1, $B7.1            LAB   $B1, $B7
LDR   $R1,N                 LDR   $R1, N
LDB   $B2, $B1.-$R1         LDB   $B2, $B1.-$R1
```

If the element number is known at assembly time, rather than being a variable as assumed in the code sequences above, then the references to N can be replaced by an immediate memory operand (=N) or the LDR may be replaced by an LDV if N≤127. Do *not* use the base register plus displacement addressing format (as in LDB $B2,$B1.N-1), because that addressing format does not adjust for addressing mode.

REQUEST BLOCKS (RBS)

A request block must be allocated with two words for each pointer in it when it is executed in LAF, but only one word for each pointer when it is executed in SAF. In a SLIC program, the two-word allocation is *not* compressed by the loader for execution in SAF.

A request block *cannot* be declared symbolically in a SLIC program. Since it has one-word pointers when it is executed in SAF and two-word pointers when it is executed in LAF, the same declaration cannot be used for execution in both addressing modes. This kind of structure must be constructed (have values placed in it) at execution time.

Data items (including pointers) in request blocks *cannot,* in general, be referred to symbolically. Since pointers occupy a different number of words in the two addressing modes, addresses within the structure are not known at assembly time. References to data items in a request block must be computed at execution time.

A convenient technique for constructing a request block is to step through it item by item, using the automatic incrementation addressing formats. When pointers are referenced, base register instructions can be used with the indexed, push, or pop addressing formats. These instructions work on either addressing mode because they use one-word units when executed in SAF and two-word units when executed in LAF. Do *not* use the LAB instruction with indexing, incrementation, or decrementation, since the LAB uses one-word units in both addressing modes.

An initial value *cannot* be declared for a data item in a request block.

INDIVIDUAL POINTERS

An individual pointer ("DC <location-expression" in assembly language) can be declared symbolically. Each pointer must be declared as an individual data item.

An individual pointer should be referred to by its label. However, as is the case in SAF/LAF independence by assembly, an individual pointer can also be referred to by a location expression involving the use of $AF.

When referring to an individual pointer (with a base register instruction), do not use the indexed, push, or pop addressing formats.

An initial value can be declared for an individual pointer. Thus, a SLIC program can contain individual address constants (as well as address constants in FIBs, argument lists, and pointer arrays).

HARDWARE-DEFINED STRUCTURES

Certain data structures are defined by the hardware. These structures have one-word pointers when executing in SAF, and two-word pointers when executing in LAF. Structures of this kind are:

- Base register areas used with SAVE and RSTR instructions. The same mask should be used to restore registers as was used to save them, and the save area must have two words reserved for each base register to be saved.
- Trap and interrupt vectors and save areas:

  User programs must reference trap save areas in the same way that request blocks are referenced; i.e., the addresses needed must be computed at execution time. Only the Monitor is allowed to access the trap vectors, interrupt vectors, and interrupt save areas.
- Queue frames and stack headers:

  Queue frames and stack headers are treated the same as request blocks for the purpose of creating a SLIC program.

IMMEDIATE MEMORY ADDRESS OPERANDS

An immediate memory address (IMA) operand ("< location-expression" in assembly language) cannot be followed by other fields of the instruction because the loader would not move those other fields when loading for execution in SAF. The loader places the value of the IMA operand only into the first word, and sets the second word to a NOP.

This constraint applies to the following instructions:

- Input/output instructions — IO, IOH, and IOLD.
- Bit instructions — LB, LBC, LBF, LBS, and LBT;

  these instructions cannot be masked, but can be indexed if they use the IMA operand field.
- SAVE, RSTR, SRM.

Other instructions either do not allow IMA operands or have only one possible address operand and do not have control fields following, so they can be used without restriction.

ABSOLUTE ADDRESSES

Only the Monitor and certain system programs such as Debug need to reference absolute memory locations. If any of these programs are written as SLIC programs, all absolute addresses must be generated at execution time.

INDEX

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

TITLE SERIES 60 (LEVEL 6)
GCOS 6 PROGRAM PREPARATION

ORDER NO. CB01, REV. 1

DATED JUNE 1978

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by appropriate technical personnel and action will be taken as required. If you require a written reply, check here and furnish complete mailing address below. □

FROM: NAME ______________________ DATE ____________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

CUT ALONG LI

PLEASE FOLD AND TAPE –
NOTE: U. S. Postal Service will not deliver stapled forms

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

FOLD ALONG LINE

Honeywell

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

20940, 3.5678, Printed in U.S.A.