Honeywell

PROGRAM PREPARATION AND CHECKOUT

SERIES 60 (LEVEL 6)

GCOS 6/MDT

SUBJECT:

Detailed Description of Program Preparation and Checkout for Series 60 (Level 6) GCOS Multi-Dimensional Tasking (GCOS 6/MDT)

SOFTWARE SUPPORTED:

This publication supports Release 0101 of Series 60 (Level 6) GCOS Multi-Dimensional Tasking (GCOS 6/MDT) software. When a later release of the system occurs, see the Subject Directory of the latest Series 60 (Level 6) GCOS 6/MDT Overview and User's Guide (Order No. AX11), to ascertain whether this revision of this manual supports that release.

DATE:

March 1977

ORDER NUMBER:

AX08, Rev. 0

PREFACE

This manual describes program preparation and checkout for Series 60 (Level 6) GCOS Multi-Dimensional Tasking (GCOS 6/MDT). Unless stated otherwise herein, the term MDT is used to refer to the GCOS 6/MDT software; the term Level 6 indicates the specific models of Series 60 (Level 6) on which the described software executes.

File No.: 1S23

AX08

GCOS 6/MDT Subject Directory

This subject directory lists topics in alphabetical order. Each topic is accompanied by the order number of each manual in which the topic is described. Following the Subject Directory is a list, by order number, of all GCOS 6/MDT manuals.

The following publications constitute the GCOS 6/MDT manual set. The Subject Directory in the latest Series 60 (Level 6) GCOS 6/MDT Software Overview and User's Guide lists the current revision number and addenda (if any) for each manual in the set.

Order No.	*Manual Title*
AX07	*Series 60 (Level 6) GCOS 6/MDT System Control*
AX08	*Series 60 (Level 6) GCOS 6/MDT Program Preparation and Checkout*
AX09	*Series 60 (Level 6) GCOS 6/MDT Data File Organization and Format*
AX10	*Series 60 (Level 6) GCOS 6/MDT Monitor and I/O Service Calls*
AX11	*Series 60 (Level 6) GCOS 6/MDT Overview and User's Guide*
AX12	*Series 60 (Level 6) GCOS 6/MDT Assembly Language Reference Manual*
AX13	*Series 60 (Level 6) GCOS 6/MDT COBOL Reference Manual*
AX14	*Series 60 (Level 6) GCOS 6/MDT FORTRAN Reference Manual*
AX15	*Series 60 (Level 6) GCOS 6/MDT Sort Manual*
AX16	*Series 60 (Level 6) GCOS 6/MDT RPG Reference Manual*

In addition to the GCOS 6/MDT manual set, the following documents provide GCOS 6/MDT users with a general hardware reference:

Order No.	*Document Title*
AS22	*Honeywell Level 6 Minicomputer Handbook*
AT04	*Level 6 System and Peripherals Operation Manual*
AU22	*GCOS/BES Programmer's Reference Card*

The following manual provides detailed information regarding programming for the Multiline Communications Processor:

AT97	*Series 60 (Level 6) MLCP Programmer's Reference Manual*

CONTENTS

ILLUSTRATIONS

TABLES

SECTION 1

OVERVIEW

Program preparation and checkout is a series of procedures that permit you to create a source unit, convert it into an executable format, detect and correct errors, and apply patches. These procedures are performed using the Honeywell-supplied tasks described in the subsequent sections of this manual.

Program preparation and checkout can be performed on a Honeywell-supplied preconfigured system, or on a more specialized system that you configure. If you are using the preconfigured system, before performing program preparation and checkout you must perform the initial system startup procedure. If you are configuring your own system, you must first perform both the initial system startup procedure and a specialized system startup procedure. The initial and specialized startup procedures are described in the "System Startup and Configuration" section of the System Control manual. The equipment required for program preparation is described in the "Equipment Requirements" section of the Overview and User's Guide.

Program preparation and checkout is described below; Figure 1-1 illustrates program preparation.

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. The Cross-Reference Program can be run 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. 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] The root remains in memory as long as there are tasks executing on its behalf, unless the LDBU configuration directive was specified; if LDBU was specified, the root remains in memory until the system is reinitialized. An overlay is load into memory whenever it is required.

You can control execution of a program and make desired changes while the program is executing by using Debug. Breakpoints can be set to determine which code is executing, and specified registers and memory locations can be displayed and, if desired, changed. If there is not enough memory for Debug, you can perform debugging by using Patch to append monitor points. Patch permits you to add patches to and/or delete patches from object units and bound units.

There are three methods of obtaining memory dumps. While a program is executing, you can obtain a memory dump by using either Debug or the Dump Edit utility program; dumps produced by Dump Edit are in edited format and are much easier to interpret. If an executing program encounters a problem and it aborts or a halt occurs, to obtain a memory dump you may use just Dump Edit or you may first dump memory to a disk file by using the MDUMP utility program and then print the memory dump by using Dump Edit. To dump the contents of all or part of the Multiline Communications Processor memory, you can use the DUMCP dump routine.

[1] The root is loaded when an –EFN argument is specified in an ECL create group or spawn group command, or an LDBU configuration directive is specified (see the System Control manual).

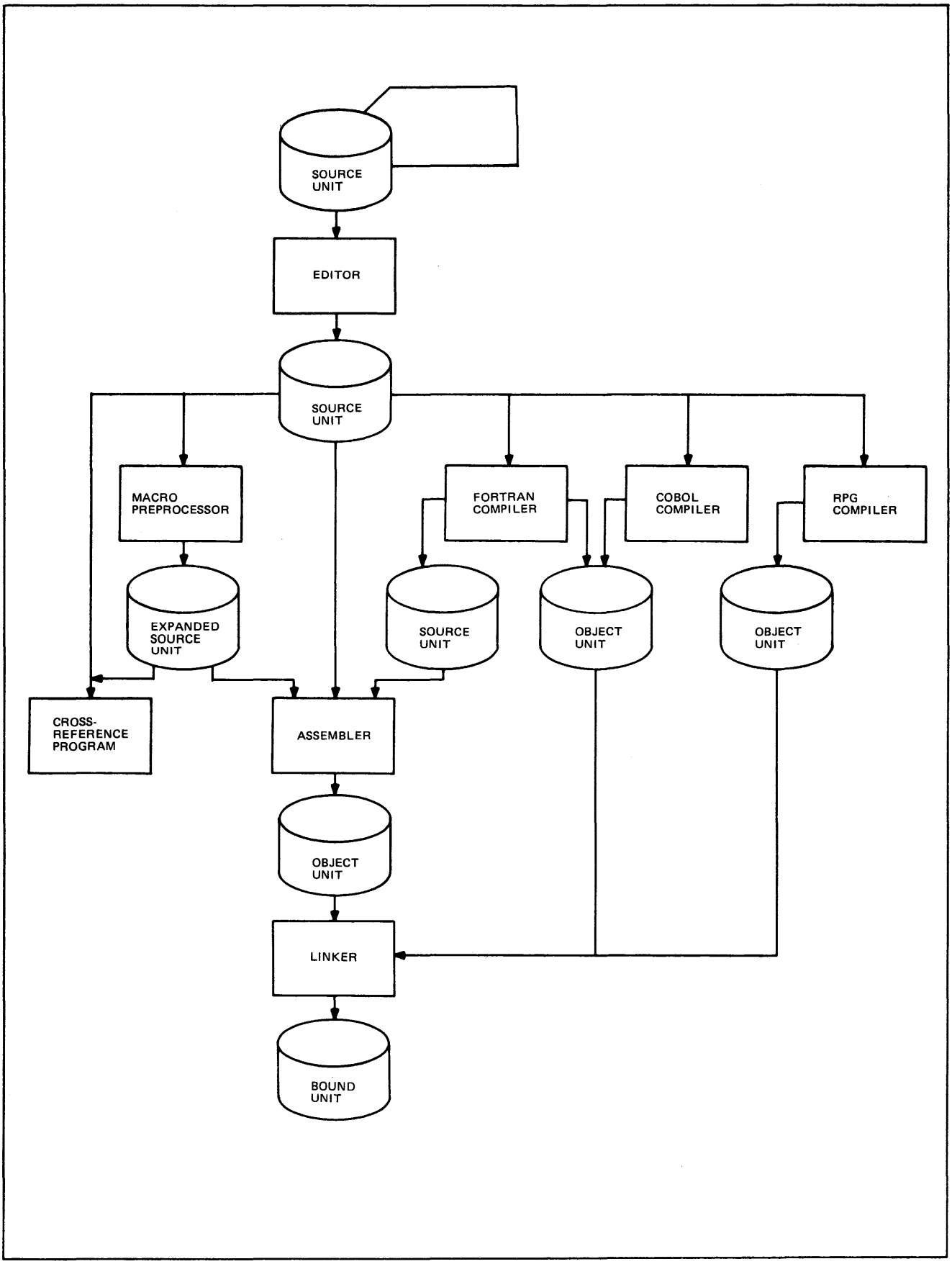
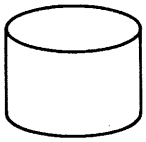


Figure 1-1. Program Preparation Procedure

NOTES: 1. If you are going to perform program preparation and checkout while simultaneously executing other tasks in the foreground, you must be familiar with the Overview and User's Guide.
2. Throughout this manual there are references to the ECL create group, enter group request, spawn group, and enter batch request commands; these commands are described in the "Execution Control Language" section of the System Control manual.

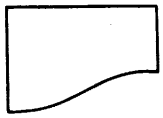
SYMBOLS USED IN THIS MANUAL

Processing; indicates any kind of processing function.

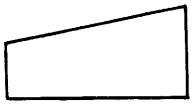


Online storage of information; e.g., diskette or cartridge disk.

Input from card reader.



Document; e.g., printer output.



Manual input; i.e., operator's 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.

HOW TO USE THIS MANUAL

The remainder of 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, Cross-Reference Program, Assembler, FORTRAN Compiler, COBOL Compiler, and RPG Compiler. The Cross-Reference Program is described in detail in this section. 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, COBOL Reference Manual, and RPG Reference Manual, respectively.

Section 4 describes how to load the Linker, Linker functions, and directives that control execution of the Linker.

Section 5 describes how to debug programs using Debug and other methods.

Section 6 describes how to load and use the MDUMP and Dump Edit utility programs.

Section 7 describes how to load Patch, and includes detailed descriptions of Patch directives.

Section 8 describes, in detail, DUMCP, the Multiline Communications Processor dump routine.

Appendix A describes, in detail, how to interpret memory dumps. This appendix includes procedures for determining where a trap occurred, finding the location in memory of your code, and determining where execution of your code terminated.

FILE SYSTEM PATHNAMES

The file system is represented by a tree-structured hierarchy. The basic elements of this structure are known as files. Some of the files are of a special type and are known as directories; the remainder of the files comprise aggregates of data.

Files

A file is defined as any unit of storage, external to the central processor, which is capable of supplying data to, and/or receiving data from a task. A file can be simply a peripheral device such as a printer, card reader, or terminal device; or it can be an aggregate of data stored within a directory structure on a magnetic 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.

Directories

A directory is a file that contains information about other files, such as the physical and logical attributes of the files and the attributes of the peripheral devices upon which they reside. The files whose attributes are described in the directory are said to be immediately contained in, or subordinate to the directory. They may themselves be directories, or they may be data files. At the base of each tree structure is a directory known as the root directory, or simply the root. The root directory name is the same as the disk volume identifier or the tape volume label of the volume on which it resides.

Naming Conventions

Each directory or file name in the file system can consist of ASCII characters from the following sets:

- Uppercase alphabetic (A through Z)
- Numeric (0 through 9)
- Underscore (_)
- Period (.)
- Dollar sign ($)

The first character of any name must be either an alphabetic or the dollar sign ($). The underscore character can be used to join two or more words which are to be interpreted as a single name (e.g., DATE_TIME). The use of the period character followed by one or more alphabetic or numeric characters is normally interpreted as a suffix appended to a file name.

The name of a root directory or a volume identifier can consist of at least one and no more than six characters. The names of other directories, and those of files, can comprise from 1 to 12 characters. The length of a file name must be such that any potential system-supplied suffix does not result in a name of more than 12 characters.

Pathname Construction

The access path to any file system entity (directory or file) begins with a root directory name and proceeds through zero or more subdirectory levels to the desired entity. The series of directory names (and a single file name if a file is the target entity) is known as the entity's *pathname*.

In constructing a pathname, certain symbols are used to indicate the hierarchical relationship between the pathname's elements. These symbols and their meanings are shown below.

- The circumflex (^)—Used exclusively to identify the name of the root directory. It precedes the root directory name, thus: ^VOL01.

- The greater than (>)—Used to connect two or more directory names or a directory name and a file name. Each occurrence of the symbol denotes a change in the directory level; the name to the right of the symbol is immediately subordinate to the name on the left. Reading a pathname from left to right thus indicates movement through the tree structure in a direction *away from* the root If the root directory ^VOL01 contains a directory name DIR1, then the pathname of DIR1 is

  ^VOL01>DIR1

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

  ^VOL01>DIR1>FILEA

- The less than (<)—Used in certain cases to indicate movement through the tree structure in a direction *toward* the root. This symbol can precede only the first element of a relative pathname, and represents a change of one level in a direction toward the root. Consecutive symbols can be used to indicate changes of more than one level; each occurrence represents a one-level change.

The last element in a pathname is the name of the entity upon which subsequent action is to be taken. This element can be either a directory name or a file name, depending on the function to be performed.

Absolute Pathnames

An absolute pathname is one which begins with a directory name preceded by a circumflex (^) or a greater-than symbol (>). When it begins with a circumflex, it is called a full pathname.

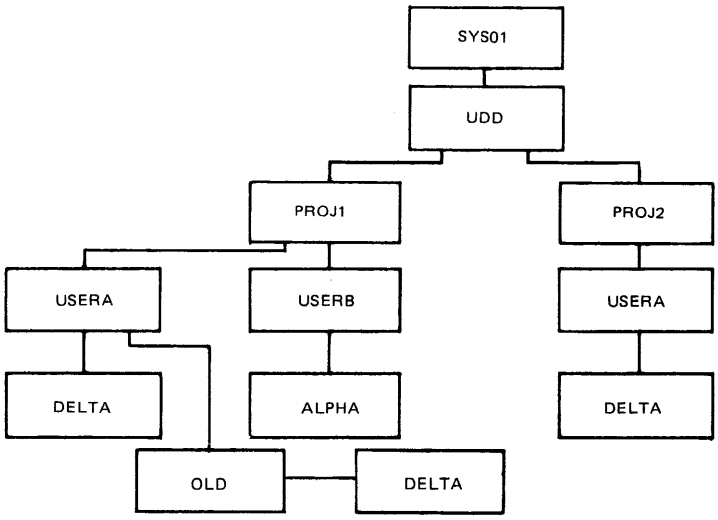
Relative Pathnames and the Current Working Directory

A relative pathname is a pathname that does *not* begin with the circumflex or greater-than symbol. The first (or only) element of this form of pathname identifies a directory or file which is immediately subordinate to a directory known as the *current working directory*. A simple name is a special case of the relative pathname. It consists of only one element, the name of the desired entity in the current working directory.

The following examples and diagram show some relative pathnames and the full pathnames they represent when the current working directory pathname is

SYS01>UDD>PROJ1>USERA

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



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 (except for the Cross-Reference Program) 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 Compiler, RPG Compiler, and Linker, as described below.

NOTE: In the following descriptions there are references to specific ECL commands. In each case, the referenced ECL command is the command that loads the task being described. The ECL LINKER command is described in Section 4. The other referenced ECL commands are described in Section 3.

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 Cross-Reference Program requires that when you specify in the ECL XREF command the name of its input file, you *do* include a suffix: .A indicates that Assembler input is going to be cross-referenced, and .P indicates that Macro Preprocessor input is going to be cross-referenced. If a list file is designated (i.e., the -COUT argument is specified in the ECL XREF command), the Cross-Reference Program does *not* append a suffix to the specified name; otherwise, the Cross-Reference Program forms the name of its list file by appending .L to the specified base name.

The Macro Preprocessor requires that the name of its input file contain a .P suffix. When you specify in the ECL MACROP command the name of the input file, omit the .P suffix. If there is an "include" file, that file name must contain an .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 ECL 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. If a list file is designated (i.e., the -COUT argument is specified in the ECL ASSEM command), the Assembler does *not* append a suffix to the specified name; otherwise, the Assembler forms the name of its list file by appending .L to the specified base name.

The FORTRAN Compiler requires that the name of its input file contain a .F suffix. When you specify in the ECL 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. If a list file is designated (i.e., the -COUT argument is specified in the ECL 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 Compiler requires that the name of its input file contain a .C suffix. When you specify in the ECL COBOL 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 ECL COBOL 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 ECL RPG command the name of the input file, omit the .R 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 ECL 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 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. If you specify in the ECL LINKER command the name of the file that will contain the bound unit, omit the suffix; 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 ECL 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 or default base name.

It is important to note that only the Macro Preprocessor, Assembler, FORTRAN Compiler, COBOL Compiler, 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 that includes one of the following suffixes, if needed: .A, .F, .C, .R, .P, or .IN.A.	Specify file name that includes one of the following suffixes, if needed: .A, .F, .C, .R, .P, or .IN.A.
Cross-Reference Program	Specify file name that includes one of the following suffixes: .A or .P	Omit suffix. Cross-Reference Program appends .L to specified input file name to form the name of the list file if the -COUT argument was not specified in the ECL XREF command. The Cross-Reference Program does not append a suffix to the name designated in the -COUT argument.
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 ECL 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, .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 ECL FORTRAN command.[a]
COBOL Compiler	Omit suffix. COBOL Compiler appends .C to specified file name.	Omit suffix. COBOL Compiler appends .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 ECL COBOL command.[a]
RPG Compiler	Omit suffix. RPG Compiler appends .R to specified file name.	Omit suffix. RPG Compiler appends .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 ECL RPG command.[a]
Linker	Omit suffix. Linker appends .O to each specified file name.	Omit suffixes. 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 ECL 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.

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 ECL "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 can 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.

NOTE: 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).

Editor directives 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.

"Advanced Usage of the Editor" describes advanced Editor directives that you may want to use after you are familiar with the Editor. These directives permit you to execute a task other than the Editor without exiting from the Editor, obtain a printout of the line number(s) or line number(s) and contents of specified line(s) in the current buffer, designate that the Editor search specified lines only if they do or do not contain a specified character string, and perform the following functions using additional buffers (called auxiliary buffers): move lines from the current buffer to an auxiliary buffer (the lines in the current buffer are deleted), copy lines in the current buffer to an auxiliary buffer (the lines in the current buffer are *not* deleted), request the status of auxiliary buffers, designate an auxiliary buffer as the current buffer, and designate (during input mode) that subsequent text be accepted from a specified auxiliary buffer. Auxiliary buffers are most commonly used for moving or copying information from one location to another within a file.

NOTE: 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:

$$\begin{Bmatrix}\text{INPUT}\\ \text{EDIT}\end{Bmatrix}\text{MODE}$$

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 preceded by one or two addresses and followed by text and termination

escape characters (!F) that designate the end of the directive. 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 5) or beginning on the next line (see format 6).

FORMAT 1:

dirname

FORMAT 2:

adr_1 dirname

FORMAT 3:

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

FORMAT 4:

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

FORMAT 5:

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

FORMAT 6:

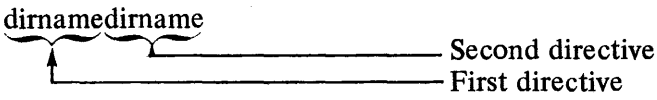
$[adr_1]\left[\left\{\begin{matrix};\\,\end{matrix}\right\} adr_2\right]$dirname
[text]
.
.
.
!F

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).

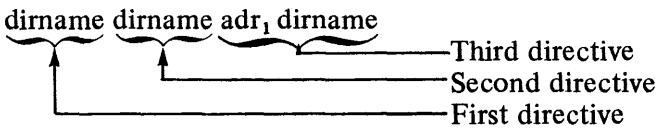
When a single address is specified, the Editor references the specified line in the current buffer. When two addresses are specified within a single directive, the Editor references a specified series of lines in the current buffer; the lines that are referenced depends on whether the addresses are separated 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 parameter 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:



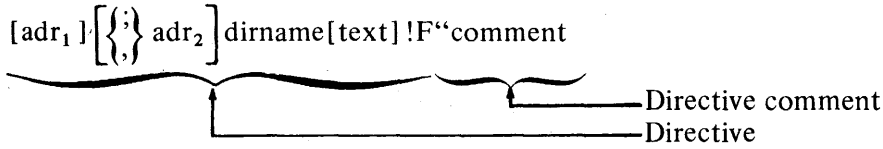
Directives separated by delimiters:



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:

adr_1 dirname dirname"comment

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



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

Methods of Specifying Addresses

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

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

[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.)

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

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

The specified directive will cause the Editor to reference line number 4, since this is the first line after the current line that contains the expression BBB.

[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.

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

Character	*Description*
*	Requests the first expression that contains any number (or none) of the immediately preceding character(s).
^	When designated as the *first* character of an expression, requests the first line that *begins with* the specified expression (excluding the character ^).
$	When specified as the *last* character of an expression, requests the first line that *ends 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).

NOTES: 1. The special meanings of the above characters, / (which delimits an expression), 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.

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.

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 regular 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*. The 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 ECL ED command, which is described below.

After the Editor is loaded, there is a typeout to the error output 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's terminal or another terminal, card reader, or disk. Error messages are written to the error output file. Editor error messages are described in the "Error Messages" section of the System Control manual.

Default: Device specified in the in_path argument of the ECL "enter batch request" or "enter group request" command.

{-LINE_LEN n / -LL n}

Maximum number of characters that can be on each directive line or data line. Must be from 20 through 255. Additional characters are truncated. Default: 80 characters.

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 "Execution Control Language" section in the System Control 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 "Execution Control Language" section in the System Control 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:

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

FORMAT 2:

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

If directives are being entered through the operator's 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

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

-LINE_LEN n argument of the ECL ED command. The last line of text must be followed by the escape sequence !F[4] 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.

NOTE: 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.

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
```

PARAMETER 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
```

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

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, if any, 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; if no text was inserted, the current line is the line immediately preceding the first line deleted. The inserted line(s), if any, are given line numbers and subsequent lines, if any, are renumbered.

FORMAT 1:

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

[text]
.
.
.
!F

FORMAT 2:

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

PARAMETER DESCRIPTIONS:

adr_1

Address of the *first or only* line to be deleted and optionally replaced.

Default: Current line

adr_2

Address of the *last* line to be deleted and optionally replaced.

Default: Only the line identified by adr_1 is deleted or changed.

NOTES: 1. If both adr_1 and adr_2 are omitted, only the current line is deleted and optionally replaced.

2. If no text is included (i.e., C is immediately followed by !F), the addressed line(s) are deleted and not replaced. If no addresses or text are specified, the current line is deleted and not 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
```

PARAMETER 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* contents of source unit from specified file into the current buffer (read directive)
- *Delete* specified line(s) from the current buffer (delete directive)
- *Print* on the user output 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_1 dirname["comment]

FORMAT 3:

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

FORMAT 4:

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.

FORMAT:

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

PARAMETER 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 throuth 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 (the default for adr_1) 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 lines(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 output file; i.e., the file designated in the -OUT out_path parameter of the ECL "enter batch request" or "enter group request" command, unless that file was reassigned in the Editor execute directive (the execute directive is described under "Advanced Usage of the Editor" later in this section). If the typeout occurs on the operator's terminal or another 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:

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

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:

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

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's terminal or another terminal, the quit directive must be immediately followed by a carriage return.

NOTE: 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.

FORMAT:

```
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

Passes control from the Editor to the task that is loaded next.

Read Directive

The read directive reads a source unit 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]

PARAMETER DESCRIPTIONS:

adr

Address of a line in the current buffer; the contents of the specified file will be inserted after this line.

Default: Last line in the buffer; if the buffer is empty, the file is inserted 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. 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 source unit 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 source unit 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:

$$[adr_1]\left[\left\{\begin{matrix};\\,\end{matrix}\right\} adr_2\right] S/regexp/string/$$

PARAMETER 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: Only the line identified by 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 parameter "string."

NOTE: If string contains the character "&" in any position, each occurrence of regexp that will 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.

string
String of characters that will replace each occurrence of regexp.

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 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)

Write Directive

The write directive causes a single source unit line or a series of source unit 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 source unit line(s) written to the file via the write directive replace the existing source unit 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:

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

PARAMETER 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. 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 source unit 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 (current line)

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. Described below are more advanced Editor directives, usage of auxiliary buffers, and how to change the origin of text during input mode.

The advanced Editor directives have the following capabilities:

- Permit execution of a task other than the Editor 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)

These directives are described, alphabetically, below.

Execute Directive

The execute directive permits you to perform a task other than editing without exiting from the Editor; i.e., you can enter any ECL 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 operator's terminal, which is the original user output file. If the user output file is a line printer and a quit directive is entered to exit from the Editor, the user output 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

PARAMETER DESCRIPTION:

command

Any ECL command (see the "Execution Control Language" section of the System Control manual).

Example:

```
E FO >SPD>LPT00
```

This execute directive includes an ECL file out (FO) command, which sets the user output 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:

$$[adr_1\ ]\ \left[\left\{\begin{matrix};\\,\end{matrix}\right\}\ adr_2\right]\ Gx/regexp/$$

PARAMETER 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.

L

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:

1,3GL/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)

Print With Line Number Directive

The print with line number directive 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 output file; i.e., the file designated in the -OUT out_path parameter of the ECL "enter batch request" or "enter group request" command, unless that file was reassigned in the Editor execute directive. If the typeout occurs on the operator's 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:

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

PARAMETER 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,$L

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:

L

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

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:

$$[adr_1]\left[\left\{\begin{matrix};\\,\end{matrix}\right\} adr_2\right] Vx/regexp/$$

PARAMETER 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.

L

VL 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:

1,3VL/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)

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 output file; i.e., the file designated in the -OUT out_path parameter of "enter batch request" or "enter group request" command, unless that file was reassigned in the execute directive.

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

FORMAT:

[adr]=

PARAMETER 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:

=

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

Typeout:

1

Current line: 1

Auxiliary Buffers

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 or copy 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 "Changing 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.

Change Buffer Directive

The change buffer directive designates that a specified auxiliary buffer is the current buffer. The previously designated current buffer becomes an auxiliary buffer.

The change buffer directive must be the last directive on the directive line.

After this directive is executed, lines can be written from the new current buffer to a file.

FORMAT:

Bx

PARAMETER 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.

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.

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.

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:

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

PARAMETER 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.

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. There is no current line in the auxiliary buffer.

FORMAT:

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

PARAMETER 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
```

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:

- o Lines from an existing file have been read into the current buffer via a read directive.
- o The contents of the buffer were modified via one or more Editor directives.
- o 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)
```

Changing Origin of Text During Input Mode

The escape sequence !B causes the Editor to accept subsequent text from a specified auxiliary buffer; !B is applicable only during input mode.

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 while accepting text from the specified buffer, the newly encountered escape sequence also will be replaced with the contents of the named buffer.

The last escape sequence !B must be terminated by !F.

FORMAT:

```
!Bx[!Bx]...!F
```

PARAMETER 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:

In this example, the contents of the current buffer and the auxiliary buffer named TEST are:

Current buffer:

```
(1) A
(2) B
(3) C
(4) D
(5) E
```

Auxiliary buffer:

```
(1) X
(2) Y
(3) Z
/D/I
!B(TEST)!F
```

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

PROGRAMMING CONSIDERATIONS

1. Two hexadecimal characters can be interpreted as one ASCII byte by preceding the characters by !H. For example, !H00 produces one 8-bit zero.
2. A tab feature exists within program preparation. Tabbing causes imbedded tab characters to be replaced with the appropriate number of spaces so that printed output on a printer, operator's 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.
3. 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.
4. A quit directive must be entered so that the Editor will close and release temporary work files created in the working directory.
5. 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.

SECTION 3

LANGUAGE PROCESSORS

This section describes how to load each language processor. A detailed description of the Cross-Reference Program and an illustration of a cross-reference listing also are included in this section.

LOADING AND EXECUTING THE MACRO PREPROCESSOR

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

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

MACROP nnnn

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:

{-INCLUDE_CONTROLS
{-IC

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.

{-MACRO_CALLS
{-MC

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.

{-SIZE nn
{-SZ nn

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: Seven-eighths of available memory in the task group's memory pool or available memory in the task group's memory pool minus 400_{10} words, whichever is smaller.

NOTE: The Macro Preprocessor always issues a typeout, of the number of errors found, to the error output file.

CROSS-REFERENCE PROGRAM

The Cross-Reference Program produces an alphabetical list of all symbolic names (i.e., labels and identifiers) in an assembly language source unit. Next to each label is the number of the line in which the label is defined and the number of each line that contains a reference to that label. Beside each identifier is an asterisk and the number of each line that contains a reference to that identifier. Labels are flagged if they are undefined, multiply defined, or defined but not referenced within a source unit. At the end of the cross-reference listing there is summary information indicating the total number of labels, references, records, and flags.

The Cross-Reference Program permits you to locate improperly defined labels before you attempt to assemble the source units in which they are located. Corrections can be made by using the Editor.

Loading and Executing the Cross-Reference Program

To load and execute the Cross-Reference Program, enter the ECL XREF command, which is described below.

After the Cross-Reference Program is loaded, there is a typeout to the error output file of the revision number, in the following format:

XREF nnnn

FORMAT:

XREF path {.A / .P} [ctl_arg]

ARGUMENT DESCRIPTIONS:

path {.A / .P}

Pathname of the source unit file to be cross-referenced. The last two characters must be a suffix; .A indicates that Assembler input is going to be cross-referenced, and .P indicates that Macro Preprocessor input is going to be cross-referenced.

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.

Default: path.L in the working directory.

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

{-SIZE nn / -SZ nn}

nn designates the maximum number of 1024-word blocks of memory that the Cross-Reference Program may use. If insufficient memory is requested, a warning message is issued, and the symbols and references already in memory are sorted and listed.

Default: 1K words of memory.

Sample Cross-Reference Listing

Figure 3-1 illustrates the source listing of a sample source unit. Figure 3-2 illustrates the cross-reference listing of that source unit.

```
 1                TITLE     SAMPLE,'761215'
 2  *
 3  *
 4                XDEF      SAMPLE
 5  *
 6  *             CONVERT HEX VALUE IN R6 TO ASCII DECIMAL (5 BYTES)
 7  *             STARTING AT R2-LEFT BYTE WITH LEADING ZEROS
 8  *             SUPPRESSED. HEX ZERO EQUALS 0. USES 35 LOCATIONS.
 9  SAMPLE       LDV       $R1,0                 R1 EQUALS ZERO
10                LDV       $R7,'0'               R7 EQUALS ASCII ZERO
11  NEXT         STH       $R7,$R2.+$R1          STORE ZERO IN BYTE OF RESULT
12                CMV       $R1,5                 INDEX EQUAL TO 5?
13                BNE       >NEXT1                NO
14                LDV       $R1,0                 RESET R1
15                LAR       $R3,TABLE             R3 EQUAL TO TABLE OF CONSTANTS
16  LOOP         LDR       $R7,=$R6              MOVE R6 TO R7
17                CL        =$R6                  CLEAR R6
18                DIV       $R7,+$R3
19                ADV       $R7,'0'               CONVERT TO ASCII
20                STH       $R7,$R2.+$R1
21                BEZ       $R6,>SUPRES           BRANCH IF NO REMAINDER
22                CMV       $R6,9                 IS THIS UNITS DIGIT?
23                BG        >LOOP                 NO
24                ADV       $R6,'0'               YES
25                STH       $R6,$R2.2             STORE UNITS IN 5TH DIGIT
26  SUPRES       LDV       $R1,0                 RESET R1
27                LDV       $R6,' '               R6 EQUALS ASCII SPACE
28  MORE         LLH       $R7,$R2.$R1           PICK UP ONE BYTE
29                CMV       $R7,'0'               ZERO?
30                BNE       >DONE                 NO
31                STH       $R6,$R2.+$R1          YES - SUPPRESS
32                CMV       $R1,4                 TENS DIGIT?
33                BNE       >MORE                 NO - CONTINUE
34  DONE         JMP       $R5
35  TABL         RESV      0
36  TABLE        DC        0
37  TABLE        DC        =Z'2710'
38                DC        =Z'03E8'
39                DC        =Z'0064'
40                DC        =Z'000A'
41  *
42                END       SAMPLE,SAMPLE
```

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

```
          TITLE    SAMPLE,'761215'LISTING EXAMPLE[a]
  $B2      ****    11    20    25    28    31
  $B3      ****    15    18
  $B5      ****    34
  $R1      ****     9    11    12    14    20    26    28    31    32
  $R6      ****    16    17    21    22    24    25    27    31
  $R7      ****    10    11    16    18    19    20    28    29
  DONE       34    30
  LOOP       16    23
  MORE       28    33
N NEXT       11
U NEXT1    ****    13
  SAMPLE      9     4    42
  SUPRES     26    21
N TABL       35
N TABLE      36
M TABLE      37    15
I  II       III                          IV

    9 LABELS
   41 REFERENCES
   42 RECORDS
    1 U FLAGS
    1 M FLAGS
    3 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.

[a]The contents of the assembly program TITLE statement become the heading for the cross-reference listing.

Figure 3-2. Sample Cross-Reference Listing

LOADING AND EXECUTING THE ASSEMBLER

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

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

ASSEM nnnn

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.

{-LAF / -SAF}

Addressing mode in which source unit will be assembled. -LAF designates long-address form; -SAF designates short-address form.

Default: Source unit is assembled in the same addressing mode that the Assembler is executing in.

{-LIST_ERRS / -LE}

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, followed by a listing of the error lines and codes.

{-NO_LIST / -NL}

Suppresses source listing.

Default: Source listing produced.

{-NO_OBJ / -NO}

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.

{-SIZE nn / -SZ nn}

nn designates the maximum number of 1024-word memory blocks that may be used for the Assembler's symbol table. nn must be numeric and be from 01 through 64.

Default: Available memory in the task group's memory pool minus 300_{10} words.

NOTE: The Assembler always issues a typeout, of the number of errors found, to the error output file.

LOADING AND EXECUTING THE FORTRAN COMPILER

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

After the FORTRAN Compiler is loaded, there is a typeout to the error output file of the revision number, in the following format:

FORTRAN nnnn

FORMAT:

FORTRAN path [ctl_arg]

ARGUMENT DESCRIPTIONS:

path

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

ctl_arg

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

-AS

Output is in assembly language; the assembly language file can be used as input to the Assembler.

-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 FORTRAN Compiler.

-HS

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

{-LIST_ERRS
-LE }

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, followed by a listing of the error lines and codes.

{-LIST_OBJ
-LO }

List object output. Object text listings will be interspersed with source text listings.

{-NO_LIST
-NL }

Suppress all listings.

Default: Listing of the source unit and error diagnostics.

{-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 FORTRAN Compiler.

-SI

Short integer and logical variables; each integer and logical variable is one word.

Default: Two words.

{-SIZE nn
-SZ nn }

nn designates the maximum number of 1024-word blocks of memory that the compiler can use for tables. nn must be from 02 through 20. 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. There must be at least 1K words available.

-UC

Suppress generation of embedded links to any subroutines referenced by a CALL statement.

-UZ

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

-WRK n

Object-time workspace for FORTRAN main programs. n must be a 1- to 4-digit decimal number from 1 to 9999.

Default: 325 words.

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 output file.

LOADING AND EXECUTING THE COBOL COMPILER

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

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

GCOS6 COBOL VERSION nnnn

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.

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.

-LD

List data map, source text, and errors.

{-LIST_ERRS / -LE}

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, followed by a listing of the error lines and codes.

{-NO_LIST / -NL}

Suppress all listings.

{-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 04 to 53. 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. At least 3,000 words must be available; otherwise, the compiler cannot be loaded and executed. If this argument is not specified, the compiler has approximately 3,000 words of memory for table space.

NOTES:
1. Only one of the following listing arguments can be used: LE, LD, LO, or NL. If no listing argument is specified, the compiler produces a listing of the source text and errors.
2. The COBOL Compiler always issues a typeout, of the number of errors found, to the error output file.

LOADING AND EXECUTING THE RPG COMPILER

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

After the RPG Compiler is loaded, there is a typeout to the error output 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. 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 the working directory.

NOTE: Path is the simple pathname, excluding the suffix appended by the RPG Compiler.

{-LIST_OBJ / -LO}

List object text, data map, source text, and diagnostics.

{-NO_LIST / -NL}

Suppress all listings.

{-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 RPG Compiler.

{-SIZE nn / -SZ nn}

nn designates the maximum number of 1024-word blocks of additional 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 and diagnostics.
2. The RPG Compiler always issues a typeout, of the number of errors found, to the error output file.

SECTION 4

LINKER

The Linker combines separately assembled and/or compiled object units and produces a bound unit. If only one object unit is to constitute a bound unit, that object unit still must be linked (converted) by the Linker.

Object units may contain external references to symbols.[1] While linking object units, the Linker resolves external references to symbols by referring to and updating a Linker-created symbol table. A link map of defined and/or undefined symbols can be produced. If desired, you can designate that the Linker link object units and produce only a link map; no bound unit is created.

To load the Linker into memory, enter the ECL LINKER command (see "Loading the Linker" later in this section).

Linking is controlled by directives entered to the Linker through the directive input device. The directive input device is the device specified in the in_path argument of the ECL "enter batch request" or "enter group request" command. This device can be reassigned in the command that loads the Linker.

Each object unit to be processed during a single execution of the Linker must be a variable sequential file. The input files may reside in the same directory or in different directories. Unless specified otherwise, all of the object units are in the working directory (see "Specifying Location(s) of Object Unit(s) to be Linked" later in this section).

During a single execution of the Linker, you can create a single bound unit; a bound unit comprises only a root, or a root and one or more overlays. The root and each overlay may be up to 64K words (128K bytes). The root and each overlay is called a load unit; a load unit is loaded into memory by the Loader. When you request in an ECL create group or spawn group command, or an LDBU configuration directive that a bound unit be loaded, the root is the portion of the bound unit that is loaded by the Loader. The root remains in memory as long as there are tasks executing on its behalf, unless LDBU was specified; if LDBU was specified, the root remains in memory until the system is reinitialized. An overlay is loaded into memory whenever it is required.

Each bound unit has an attribute table associated with it; an attribute table contains information about the bound unit's characteristics and symbol definitions. The attribute table is loaded into memory immediately preceding the root.

FUNCTIONS OF THE LINKER

Creating a Bound Unit

If the name argument is specified in the ECL LINKER command (i.e., the command that loads the Linker), the Linker produces a bound unit file whose pathname is specified in the name argument.

The bound unit comprises only a root unless an OVLY or FLOVLY directive is entered. Each time an OVLY or FLOVLY directive is entered, the Linker initiates creation of a nonfloatable or floatable overlay, respectively. A nonfloatable overlay is loaded by the Loader into the same memory location (relative to the root) each time it is requested. A floatable overlay is linked at relative 0 (see "BASE Directive" later in this section), and can be loaded by the Loader into any available memory location. A floatable overlay must have the following characteristics:

1. External location definitions in the overlay are not referenced by the root or any other overlay.
2. The overlay does not contain references with offsets to symbols defined in the root or any other overlay.
3. The overlay does not contain external references that are not resolved by the Linker.
4. The overlay must be linked after all desired nonfloatable overlays have been linked.

[1] An external reference is a reference to a symbol defined in another object unit as an external symbol.

Resolving External References

The Linker resolves the addresses or values of external symbol references it finds in object units being linked. The references can be between object units comprising the root, between object units comprising an overlay, between overlays, or between the root and an overlay. A symbol can be defined in one bound unit and referenced in another bound unit.

Creating a Symbol Table

A symbol table is a data structure created by the Linker for resolving external references. When the Linker encounters external references to symbols or definitions of symbols, it creates an entry in the symbol table. When that entry is defined, the Linker updates the symbol's entry in the symbol table and all references to the symbol. A symbol can be defined within an object unit, or by an LDEF or VDEF directive. (LDEF and VDEF are explained later in this section under "Linker Directive Descriptions.") A list of defined and/or undefined symbols can be obtained by producing a link map.

Producing a Link Map

A link map is a listing of information in the symbol table. A symbol can be defined in an object unit as a value or location, in the LDEF directive as a location, or in the VDEF directive as a value. If a symbol is defined as a location, the map contains the symbol name and its relative address. If a symbol is defined as a value, the map contains the symbol name and its value. The map also lists the name of each undefined symbol and the relative address of the latest reference to it.

A link map can be produced at any time during the linking process by specifying the MAP or MAPU directive. It is written to the file name.M in the working directory, unless the -COUT argument was specified in the ECL LINKER command. (-COUT permits you to assign the list file to disk, a printer, the operator's terminal, or another terminal.) If maps are assigned to disk, a disk file with variable length records is created; the first character of each record is a print control character.

FUNCTIONAL GROUPS OF LINKER DIRECTIVES

The general functions of Linker directives are listed and described below. For more detailed information, see "Linker Directive Descriptions" later in this section.

- o Specify object unit(s) to be linked
- o Specify location(s) of object unit(s) to be linked
- o Create root and optional overlay(s)
- o Produce link map(s)
- o Define external symbols
- o Protect or purge symbols
- o Designate that the last Linker directive has been entered

Specifying Object Unit(s) to be Linked

Directives:

LINK
LINKN

LINK and LINKN designate that one or more specified object units be linked. Object units specified in LINK directives are not linked immediately; their names are put into a link request list, and they are linked each time a directive other than LINK or START is encountered. Specified object units in the primary directory are linked before specified object units in the secondary directory; within each directory, the object units are linked in the order in which they were requested.

LINKN causes the Linker to link object units already named in the link request list, and then to link object units specified in the LINKN directive, in the order in which they were requested.

The order in which object units are linked may be important if overlays exist.

NOTE: The Linker appends the suffix .O to each specified object unit name; when the Linker searches for an object unit name, it searches for the name including the suffix.

Specifying Location(s) of Object Unit(s) to be Linked

Directives:

IN
LIB

Object units to be linked must be in the primary and/or secondary directory. The primary directory is the first directory searched by the Linker; the secondary directory, if there is one, is the second (last) directory searched. When the Linker is loaded into memory, the primary directory is the working directory, and there is no secondary directory.

IN permits you to designate a different directory as the primary directory.

LIB designates a directory as the secondary directory.

IN and LIB may be specified any number of times.

Creating a Root and Optional Overlay(s)

Directives:

BASE
START
IST
SHARE
SYS
LINK
LINKN
OVLY
FLOVLY
CC
QUIT

The BASE directive defines, for subsequent object units to be linked, the relative load address within the bound unit.

START specifies the relative address at which the root or overlay will begin executing when it is loaded into memory by the Loader.

IST identifies the beginning of initialization code in the root.

SHARE designates that the bound unit is shareable.

SYS designates that the bound unit can be loaded into the system area as part of the system.

LINK and LINKN specify which object units will be linked. The order in which specified object units are linked, and when they are linked, is determined by which link directive is specified.

OVLY names and assigns a number to the next (or only) nonfloatable overlay that follows, and designates the end of the preceding root or overlay.

FLOVLY names and assigns a number to the next (or only) floatable overlay that follows, and designates the end of the preceding root or overlay.

Call-cancel (CC) permits a COBOL program that used CALL and CANCEL statements to call overlays by their names.

QUIT designates that the last Linker directive has been entered. Execution of the Linker terminates after the bound unit has been created.

Producing Link Map(s)

Directives:

MAP
MAPU

A link map is written to the list file by specifying the MAP or MAPU directive. MAP creates a map that lists both defined and undefined symbols, whereas MAPU lists undefined symbols only.

Defining External Symbol(s)

Directives:

LDEF
VDEF
EDEF

A symbol can be defined as a relative location or value by specifying the LDEF or VDEF directive, respectively. The symbol's definition is then put into the symbol table by the Linker.

The EDEF directive permits definitions in the Linker symbol table to be made part of the bound unit so they are available to the Loader at execution time.

Protecting or Purging Symbol(s)

Directives:

PROT
PURGE

The above directives protect or remove symbols and object unit names from the symbol table.

The protect (PROT) directive prevents certain symbols and/or object unit names from being removed from the symbol table. Symbols are protected if they identify a specified address or an address within a specified range; object unit names are protected if they are equated to a specified address or an address within a specified range.

The PURGE directive removes from the symbol table unprotected symbols that define a specified address or an address within a specified range, and/or object unit names equated to a specified address or an address within a specified range.

Designating That the Last Linker Directive Has Been Entered

Directive:

QUIT

QUIT must be the last Linker directive entered.

If a bound unit is being created, execution of the Linker terminates after the bound unit has been created.

If no bound unit is being created, QUIT terminates execution of the Linker.

LOADING THE LINKER

To load the Linker, enter the ECL LINKER command, which is described below.

After the Linker is loaded, there is a typeout to the error output file of the revision number in the following format:

LINKER nnnn

FORMAT:

LINKER [name] [ctl_arg]

ARGUMENT DESCRIPTIONS:

name

Pathname of the relative disk bound unit file. The pathname can be simple, relative, or absolute. If the specified file already exists, the existing information in the file is deleted and replaced with the new bound unit.

Default: No bound unit is created. If this argument is omitted, only a list file is created.

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 Linker directives will be read; can be disk, card reader, operator's terminal, or another terminal.

Error messages are written to the error output file. Linker error messages are described in the "Error Messages" section of the System Control manual.

Default: Device specified in the in_path argument of the ECL "enter batch request" or "enter group request" command.

-COUT out_path

Listing will be written to the file out_path. The list file can be a disk, the operator's terminal, another terminal, or a printer.

Default: name.M in the working directory.

{-LAF / -SAF}

Addressing mode; -LAF designates long address form (two-word addresses), and -SAF designates short address form (one-word addresses).

Default: Bound unit executed in SAF (short address form) mode.

{-SIZE nn / -SZ nn}

nn designates the maximum number of 1024-word blocks of memory available for the Linker symbol table. nn must be a minimum of 01.

Default: Available memory in the task group's memory pool.

Example:

```
LINKER MYPROG -IN >SPD>CONSOLE -COUT >SPD>LPT00 -SIZE 06
```

This LINKER command loads the Linker and designates the following:

- Bound unit will be a relative file named MYPROG in the working directory.
- Linker directives will be entered through the terminal called CONSOLE.
- Link maps, if any, will go to a line printer (configured as LPT00), rather than to a variable sequential file named MYPROG.M in the working directory.
- The symbol table will be a maximum of 6K words of memory.

NOTE: CONSOLE and LPT00 must have been previously defined in the DEVICE configuration directive, which is described in the "System Startup and Configuration" section of the System Control manual.

ENTERING LINKER DIRECTIVES

Linker directives are entered through the directive input device, except for the following directives which may be embedded in assembly language CTRL statements: LINK, LINKN, SHARE, EDEF, and SYS.

Linker directives comprise only a directive name or a directive name followed by one or more parameters. Each directive name *may be preceded by* 0, 1, or more blank spaces. If one or more parameters are to be specified in a Linker directive, the directive name *must be* immediately *followed by* one or more blank spaces.

Multiple directives can be entered on a line by specifying a semicolon (;) after each directive, except for the last directive on the line.

The last (or only) directive on a line can be followed by a comment; to include a comment, specify a slash (/) after the last (or only) parameter and then enter the comment.

If the directive input device is the operator's terminal or another terminal, press RETURN at the end of each line (i.e., at the end of the comment, or at the end of the last directive if there is no comment).

If an error occurs when entering a directive, an error message is written to the error output file. Linker error messages are described in the "Error Messages" section of the System Control manual. Determine what caused the error, and then reenter the directive correctly. If multiple directives are entered on a line and an error occurs, the error does not affect the execution of previously designated directives. The directive that caused the error and subsequent directives on that line are not executed.

PROCEDURE FOR CREATING ONLY A ROOT

To link object units and create only a root, load the Linker and then enter the following directives:

{LINK | LINKN}[2]	Links object units.
QUIT	Designates that the last Linker directive has been entered. After the bound unit has been created, execution of the Linker terminates.

All other directives are optional.

PROCEDURE FOR CREATING A ROOT AND ONE OR MORE OVERLAYS

When creating a root and overlays, the following rules must be followed:

- o The root must be created before its overlays.
- o A root and all of its overlays must be created during the same execution of the Linker.
- o Nonfloatable overlays must be created before floatable overlays.
- o Overlays may contain references to symbols defined in the root or other overlays.
- o A root or overlay can be up to 64K words of memory.

To link object units and create a root and one or more overlays, load the Linker and then enter the following required directives:

{LINK | LINKN}[3]	Links object units that will constitute the root.
{OVLY | FLOVLY}	Designates end of the root, and names and numbers the overlay that immediately follows.
{LINK | LINKN}	Links object units that will constitute an overlay.

NOTE: An OVLY or FLOVLY directive and at least one link directive must be specified for each overlay associated with the root.

QUIT	Designates that the last Linker directive has been entered. After the bound unit has been created, execution of the Linker terminates.

All other directives are optional.

NOTE: It is advisable to specify a MAP directive before each FLOVLY directive. The base address of a floatable overlay is relative 0, so all unprotected symbols that define locations will be purged from the symbol table.

[2],[3] Multiple LINK and/or LINKN directives may be entered.

OBTAINING SUMMARY INFORMATION OF A LINKER SESSION

The Linker designates on the list file summary information regarding the bound unit created during the current execution of the Linker.

The list file includes the name and revision number of each object unit linked, the Assembler or compiler error count, and the sections described below:

ROOT	Name of the root.
HIGHEST OVLY	Number of the last overlay;[4] if there are no overlays HIGHEST OVLY is followed by a blank.
/NUM OF SYMS	Number of symbols specified in EDEF directives.
{SAF} {LAF}	Type of addressing form used in the bound unit; SAF is short-address form, and LAF is long-address form.
{ROOT} {OVLY}	Name of the root or overlay.
BASE	Base address of the root or overlay.
ST	Start address of the root or overlay.

sfri

Specifies characteristics of the bound unit. An X appears in the appropriate column if the condition is met; otherwise, the column contains a period (.).

s

Shareable bound unit.

f

Floatable overlay(s) included.

r

One or both of the following conditions exists:

There are undefined or forward references between the root and overlays or between overlays.

There are resolved forward, external, references.

i

IMA addresses are present.

LINK DONE

Designates that execution of the Linker has been successful.

The format for this information is illustrated below:

ROOT rootname

HIGHEST OVLY number/NUM OF SYMS number

****

{SAF} {LAF}

****

{ROOT} {OVLY} dirname BASE address ST address - {X} {.} {X} {.} {X} {.} {X} {.} [5]

LINK DONE

[4] The Linker assigns numbers to overlays. The first overlay is 00; subsequent overlays are numbered sequentially in ascending order.

[5] This line is repeated for each overlay.

LINKER DIRECTIVE DESCRIPTIONS

Linker directives are described below, alphabetically. Some examples are provided to illustrate directive usage.

BASE Directive

The BASE directive defines, for subsequent object units to be linked, the relative link address within the bound unit. At load time, all addresses are relative to the beginning of available memory (relative 0) in the memory pool of the task group. When a task group is created, you specify the memory pool into which its bound units are to be loaded.

Unless BASE directives specify otherwise, the root will be linked, by default, at relative 0, and subsequent object units are linked at successive relative addresses. A BASE directive can be used at any point during linking to change the relative locations of the root, overlays, or individual object units. A floatable overlay always begins at relative 0; therefore, in a floatable overlay, BASE can be specified only *after* the first (or only) LINK or LINKN directive. A BASE parameter can specify a previously used or defined location, or an address relative to the beginning of the available memory.

If unprotected symbols define locations that are equal to or greater than the location designated in the BASE directive, those symbols are removed from the symbol table.

FORMAT:

BASE { $ | % | X'address' | =object-unit-name | xdef [{ + | - } X'offset'] }

PARAMETER DESCRIPTIONS:

$

Next location after the highest address of the linked root or previously linked nonfloatable overlay.

%

Highest address+1 ever used in the linked root or *any* previously linked nonfloatable overlay.

address

Hexadecimal address comprising one to four integers enclosed in apostrophes and preceded by X. The specified address is relative to the beginning of available memory (relative 0) in the memory pool at load time.

=object-unit-name

Specified object unit's base address; the subsequent root, overlay, or object unit will be linked at the same relative address as the specified object unit.

xdef [{ + | - } X'offset']

Address of any previously defined external symbol. If an offset is specified, it must be a hexadecimal integer with an absolute value less than 7FFF (32768 decimal).

Default:

Root–0

Nonfloatable overlay–Next location after the highest address of the preceding root or nonfloatable overlay

Floatable overlay–0

Example:

This example illustrates usage of BASE directives in a bound unit that comprises a root and overlays. The root will be loaded into memory wherever there is room. However, in this example, assume that the bound unit being created is going to be executed as part of task group A1, and memory pool AA is to be used by this task group. Figure 4-1 illustrates memory pool AA's location in memory relative to the system pool and another pool, and the locations within that memory pool that each object unit specified in the following directives will be loaded.

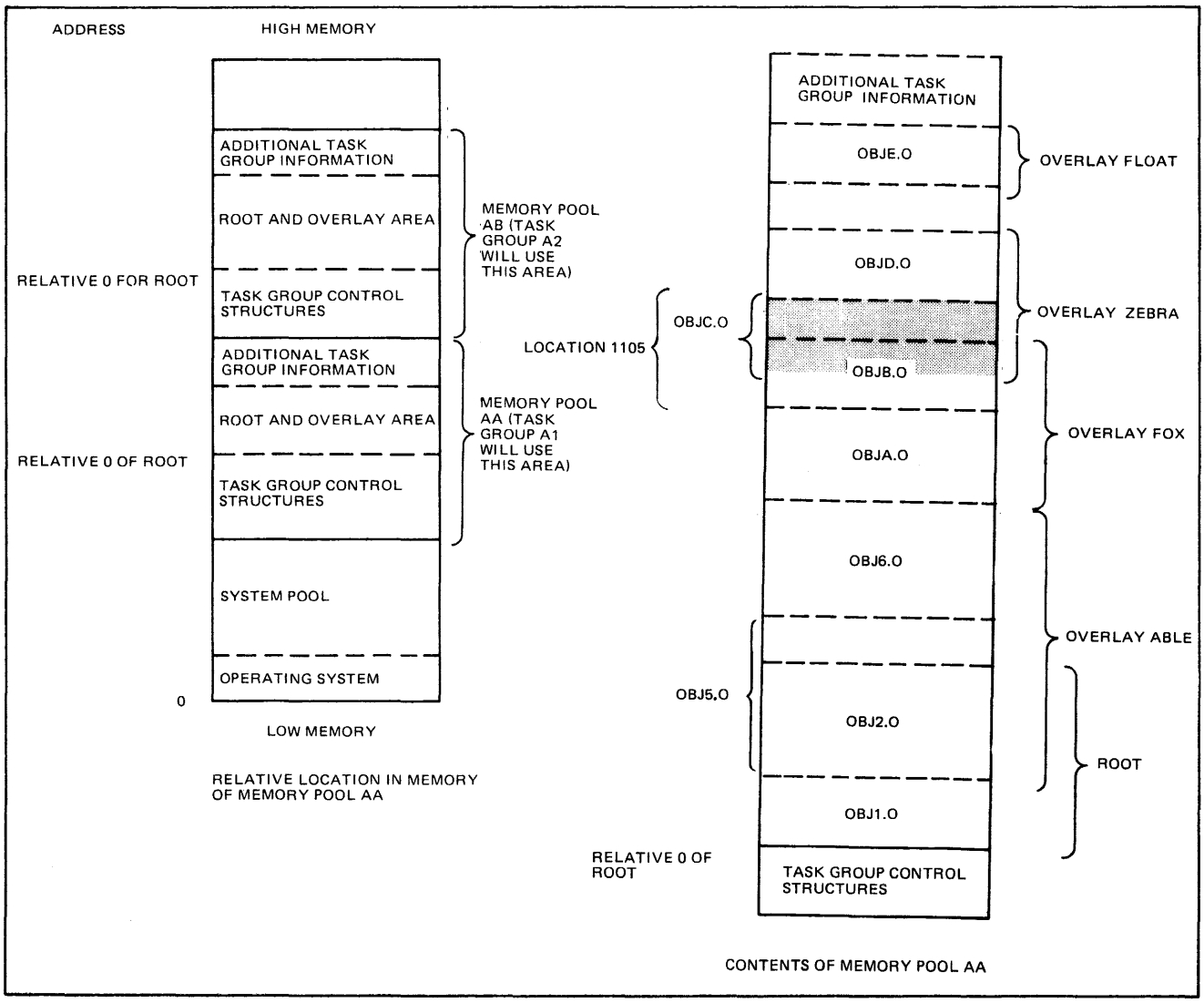


Figure 4-1. Schematic of Previous Example Illustrating Usage of BASE Directives

```
LINKER TEXT -COUT >SPD>LPT00
START TEXTEN
```

Designates address at which execution will begin when the root is loaded.

```
IST INIT
```

Defines INIT as the beginning of initialization code.

```
LINK OBJ1,OBJ2
```

Request that OBJ1.O and OBJ2.O be linked.

```
MAP
```

Causes OBJ1.O and OBJ2.O to be linked, and produces a link map.

OVLY ABLE	Designates end of the root, and that a nonfloatable overlay named ABLE immediately follows. The Linker assigns the number 00 to this overlay.
BASE =OBJ2	Subsequent object unit(s) constituting overlay ABLE will be linked starting at the base address of the object unit OBJ2.O; this address can be determined from the map. Unprotected symbols that define locations equal to or greater than the address of OBJ2 are removed from the symbol table.
LINK OBJ5	Requests that OBJ5.O be linked.
MAP	
LINK OBJ6	Requests that OBJ6.O be linked.
OVLY FOX	Designates the end of the above overlay, and specifies that a nonfloatable overlay named FOX immediately follows. The Linker assigns the number 01 to this overlay.
BASE $	Subsequent object unit(s) constituting the overlay named FOX will be linked starting at one location higher than the ending address of OBJ6.O. This is the default BASE address, so BASE $ need not be specified.
LINK OBJA	Requests that OBJA.O be linked.
LINK OBJB	Requests that OBJB.O be linked.
MAP	
OVLY ZEBRA	Designates end of above overlay 01 and names subsequent nonfloatable overlay. The Linker assigns the number 02 to this overlay.
BASE X'1105'	Designates that subsequent object units constituting overlay ZEBRA will be linked starting at relative location 1105.
LINK OBJC	Object unit OBJC.O will be linked starting at relative location 1105.
LINK OBJD	Requests that OBJD.O be linked.
MAP	
FLOVLY FLOAT	Designates end of above overlay, and that a floatable overlay named FLOAT immediately follows. The Linker assigns the number 03 to this overlay. This overlay will be linked starting at the default base address of 0.
LINK OBJE	Requests that OBJE.O be linked.
MAP	
QUIT	

Call-Cancel Directive (CC)

The call-cancel directive (CC) must be used when linking COBOL programs that contain CALL/CANCEL statements that reference overlays. The Linker will place each overlay name and its associated Linker-generated overlay number into the bound unit attribute table so that the COBOL program can call/cancel overlays by name.

To support the CALL/CANCEL facility, the object unit ZCCEC is required. ZCCEC will be automatically linked into the root by COBOL; it requires no link directive.

The CC directive must be specified before the first LINK or LINKN directive in the root.

FORMAT:

CC

EDEF Directive

The EDEF directive causes the definition of the specified symbol to be placed in the attribute table attached to the bound unit being created. The bound unit attribute table is part of the bound unit.

Secondary entry points of bound units, whose code is to execute under control of a task, must be defined in an EDEF directive. This includes secondary entry points of overlays and the root entry point when it will be explicitly used in an ECL create group command. The start address of the root and each overlay is placed by the Linker in the bound unit attribute table and does not need an EDEF definition.

If a bound unit is memory resident, global symbols (entry points and references) can be EDEFed so that they can be referenced by any bound unit loaded by the system. At system configuration time, when the resident bound units are loaded using the LDBU system configuration directive, these symbols are placed in the system symbol table. When the Loader loads other bound units that contain unresolved references, it tries to resolve them with the list of symbols defined for resident bound units.

If the bound unit is transient (shareable or not shareable), the symbols in the attribute table of the bound unit are meaningful only as definitions of secondary entry points. Although shared bound units can be in the address space of more than one task group, the bound unit attribute table is available to the Loader only when the bound unit is being loaded. Unresolved references in any bound unit will be resolved only to symbols defined in attribute tables of resident bound units.

An EDEF directive can only specify a symbol that has been defined using XDEF, LDEF, or VDEF. When EDEF is specified, the symbol's definition must already be in the symbol table.

The EDEF directive can be embedded in assembly language CTRL statements.

FORMAT:

$$\left\{\begin{matrix}\text{EDEF}\\ \text{EF}\end{matrix}\right\} \text{ symbol}$$

PARAMETER DESCRIPTION:

symbol

Any external definition comprising one to six characters. The symbol must have been defined. If the symbol was multiply defined, the first definition is used.

Example:

This example illustrates usage of EDEF directives in bound units.

LINKER MYPROG	Loads the Linker. The bound unit named MYPROG will be created on the working directory.
LINK A	
LINKN B	
MAP	
EDEF B	
LDEF SYM,X'1234'	Assigns relative location 1234 to external symbol named SYM.
OVLY FIRST	Designates end of root, and names nonfloatable overlay that immediately follows.

LINK X,Y EDEF SYM QUIT	Designates that the last Linker directive has been entered. Execution of the Linker terminates after the bound unit has been created.
LINKER PROG2 -COUT >SPD> LPT00 -SIZE 02	Loads the Linker; the bound unit to be created is named PROG2. The list file is the printer. The symbol table is a maximum of 2K words of memory.
BASE X'2222'	Subsequent object units will be loaded into memory starting at the relative address 2222.
LINKN W	Requests that object unit W.O be linked.
MAP	Produces a link map; in this map, it is determined that object unit W.O contains an unresolved reference to the symbol SYM, which was defined in the root of the bound unit MYPROG.
QUIT	

If MYPROG is loaded into memory via an LDBU configuration directive, when the Loader loads PROG2 the Loader will resolve the unresolved reference in PROG2 to the symbol SYM, which was defined in the root of MYPROG.

FLOVLY Directive

The FLOVLY directive assigns the specified name and a number to the *floatable* overlay that immediately follows, and designates the end of the preceding root or overlay. The characteristics of floatable overlays are described earlier in this section under "Creating a Bound Unit."

FLOVLY must be specified as the first directive of each floatable overlay. Floatable overlays must be linked after all desired nonfloatable overlays have been linked.

The Linker assigns a two-digit number to each overlay. Overlays are numbered sequentially, in ascending order; the first overlay is 00.

FORMAT:

FLOVLY name

PARAMETER DESCRIPTION:

name

Name of the floatable overlay that immediately follows. The overlay name must comprise one to six alphanumeric characters; the first character must be alphabetic.

Example:

LINKER BU	Loads the Linker and designates BU as the bound unit name.
LINK A LINK B MAP	Produces a link map. The link map should be referenced to determine if there are any unprotected symbols that define locations. These symbols, if any, will be removed from the symbol table since the floatable overlay that immediately follows does not contain a base address, and the default base address is 0.

FLOVLY GR	Designates the end of the root (which comprises object units A.O and B.O), and specifies that the next overlay is a floatable overlay named GR. The Linker assigns the number 00 to this overlay.
LINK X LINK Y MAP	
FLOVLY BR	Designates the end of floatable overlay GR, and designates that the floatable overlay that immediately follows is named BR. The Linker assigns the number 01 to this overlay.
LINK R6 MAP QUIT	

IN Directive

The IN directive designates a different directory as the primary directory.[6] This directive permits the linking of object units that are in directories other than the current primary directory or secondary directory (if any). If the IN directive is not specified, the working directory is the primary directory. (The secondary directory is designated in the LIB directive.)

The IN directive must be specified before the first LINK or LINKN directive that requests the linking of an object unit that is in the specified directory.

The specified directory remains the primary directory until another IN directive is entered. If the primary directory is changed via an IN directive and at a later time you want the task group's working directory to be the primary directory, you must enter the IN directive and specify in that directive the working directory's pathname.

FORMAT:

IN path

PARAMETER DESCRIPTION:

path

Pathname of the directory being designated as the primary directory. The pathname may comprise a maximum of 64 characters. A simple, relative, or absolute pathname may be specified (Methods of designating pathnames are described in Section 1.)

Example 1:

IN ^DIR>PRIM

This directive designates that ^DIR>PRIM is the primary directory.

Example 2:

This example illustrates usage of the IN directive in conjunction with directives that request the linking of object units. The primary directory is the working directory, whose pathname is WORK> CURR; object units X.O,Y.O, and Z.O are in the working directory.

---

[6]The primary directory is the first directory that the Linker searches for the specified object unit(s) to be linked.

LINKER OUTPUT	Loads the Linker; a bound unit named OUTPUT will be created.
LINK X	Requests the linking of object unit X.O; X.O is in the working directory.
IN ^NEW>PRIM	Designates that ^NEW>PRIM now is the primary directory.
LINK A	Requests the linking of object unit A.O, which is in the primary directory. DIR>PRIM>A.O is the pathname of A.O, as appended by the Linker.
LINK C	Requests the linking of object unit C.O, which is in the primary directory. DIR>PRIM>C.O is the pathname of C.O, as expanded by the Linker.
IN WORK>CURR	Designates that the primary directory now is the working directory.
LINKN Y	Requests the linking of object unit Y.O, which is in the working directory. WORK>CURR>Y.O is the pathname of Y.O, as expanded by the Linker.

MAP
QUIT

IST Directive

The IST directive identifies the beginning of initialization code in the root. Initialization code is code that you want to execute only once immediately after the root is loaded. After initialization code is executed, the space is made available for overlays.

The external symbol must be specified in an EDEF directive.

FORMAT:

$$\left\{\begin{matrix}\text{IST}\\\text{IT}\end{matrix}\right\} \text{ external symbol}$$

PARAMETER DESCRIPTION:

external symbol

Symbol defined within the *root* as an external location.

LDEF Directive

LDEF assigns a relative *location* to an external symbol. A symbol should be defined only once, either as a location or as a value. When a symbol is defined, its definition is put into the Linker symbol table so that it can be used to resolve references to the symbol during linking. When a symbol defined as a location is no longer referenced, its symbol table entry can be cleared by specifying the PURGE directive. PURGE has no effect if a protect (PROT) directive was previously specified.

FORMAT:

$$\left\{\begin{matrix}\text{LDEF}\\\text{LF}\end{matrix}\right\} \text{ symbol,} \left\{\begin{matrix}\$\\\%\\\text{X'address'}\\\text{=object-unit-name}\\\text{xdef}\left[\left\{\begin{matrix}+\\-\end{matrix}\right\}\text{X'offset'}\right]\end{matrix}\right\}$$

PARAMETER DESCRIPTIONS:

$

Next location after the highest address of the linked root or *previously* linked nonfloatable overlay.

%

Highest address+1 ever used in the linked root or *any* previously linked nonfloatable overlay.

address

Hexadecimal address comprising one to four integers enclosed in apostrophes and preceded by X. The specified address is relative to the beginning of available memory (relative 0) in the memory pool.

=object-unit-name

Specified object unit's base address.

xdef$\left[\left\{\begin{matrix}+\\-\end{matrix}\right\}\right.$ X'offset'$\left.\right]$

Address of any previously defined external symbol. If an offset is specified, it must be a hexadecimal integer with an absolute value less than 7FFF (32768 decimal).

Example:

This example illustrates usage of each format of the LDEF directive.

LINKER BOUND	Loads the Linker and designates BOUND as the bound unit name.
LINK A	
LINK B,C	
MAP	
LDEF SYM,X'1234'	SYM assigned relative location 1234
OVLY FIRST	Designates end of root and names first nonfloatable overlay
LINK R	
MAP	
LDEF QUIZ,=C	QUIZ assigned base location of the previously linked object unit named C.O .
OVLY SECOND	
LINKN D	
LINK F	
MAP	
LDEF NEW,SYM	NEW assigned same location as the symbol SYM, which was defined in the root; i.e., NEW is assigned relative location 1234.
OVLY NEXT	
BASE X'1300'	
LINK W,X	
MAP	
LDEF ANY,$	ANY assigned next location after highest address of the previously linked nonfloatable overlay.
OVLY THIRD	
LINK Z	
LINK Q	
MAP	

LDEF FIND,%	FIND assigned next location after highest address of the root or *any* previously linked nonfloatable overlay. (A previous nonfloatable overlay was named SECOND; if it ended at location 1566 and this is the highest address ever reached during the linking of object units constituting this bound unit, FIND would be assigned location 1567.)
QUIT	

LIB Directive

The LIB directive designates a directory as the secondary directory. This directory permits the linking of object units that are in a directory other than the primary directory. If the Linker cannot find in the primary directory an object unit specified in the LINK or LINKN directive, the Linker then searches the secondary directory.

If LIB is not specified, there is no secondary directory; the Linker searches only the primary directory.

The LIB directive must be specified before the first LINK or LINKN directive that requests the linking of an object unit that is not in the primary directory.

The specified secondary directory remains in effect until the LIB directive is respecified with a different directory name.

FORMAT:

LIB path

PARAMETER DESCRIPTION:

path

Pathname of the directory being designated as the secondary directory. A simple, relative, or absolute pathname may be specified. (Methods of specifying pathnames are described in Section 1.)

Example 1:

LIB DIR>SECND

This directive designates that DIR > SECND is the relative pathname of the secondary directory.

Example 2:

This example illustrates usage of a secondary directory, which contains object units W.O, Y.O, and Z.O.

LIB DIR>SECND	Designates that DIR>SECND is the relative pathname of the secondary directory.
LINK B	Requests the linking of object unit B.O; B.O resides in the primary directory.
LINK A	Requests the linking of object unit A.O; A.O resides in the primary directory.
LINK W	Requests the linking of object unit W.O; W.O resides in the secondary directory. DIR>SECND>W.O is the full pathname of W.O, as expanded by the Linker.

All specified object units in the primary directory are linked first; then all specified object units in the secondary directory are linked. To cause object units to be linked in a specific order, the LINKN directive must be used.

LINK Directive

The LINK directive specifies that the Linker link one or more specified object units. Each specified object unit name is put into the link request list. The object units are linked each time a directive other than LINK or START is encountered. When this occurs, the Linker searches the primary directory and links the specified object units in the order in which they were requested. If all of the object units are not found and there is a secondary directory, the Linker searches the secondary directory and links specified object units, in the order in which they were requested. If there is a copy of an object unit in both the primary and secondary directory, the copy in the primary directory is linked.

The order in which object units are linked is important for the following reasons: (1) it determines which object units will be in memory simultaneously and which object units will overlay other object units and (2) within the root and each overlay, the first start address encountered by the Linker (either in an END statement or a START directive) is used as the start address for that root or overlay.

During each execution of the Linker, at least one LINK or LINKN directive must be entered. Multiple LINK directives can be specified within a single root or overlay. If LINK and/or LINKN directives request that the same object unit be linked more than once within a single bound unit, only the first request is honored.

LINK directives can be embedded in assembly language CTRL statements; the specified object unit(s) are added to the link request list immediately following the object unit in which they were embedded. See "LINKN Directive" for the order in which object units are linked if there are embedded LINK directives and/or LINKN directives.

FORMAT:

LINK obj-unit$_1$ [,obj-unit$_2$] ...

PARAMETER DESCRIPTION:

object-unit$_n$

Name of an object unit to be linked. An object unit name must be one to six alphanumeric characters and not include a suffix; the first character must be a letter or a dollar sign ($). The Linker will search for the specified object unit name, with a .O suffix.

Example 1:

```
LINK FIRST
```

This directive causes the Linker to link the object unit named FIRST.O . The primary directory is searched first; if FIRST.O is not found, the secondary directory, if any, is searched.

Example 2:

```
LIB SECOND>FILE
LINK R
LINK T
```

The above LIB directive designates that SECOND>FILE is the pathname of the secondary directory. In this example, object unit R.O is in the secondary directory, and object unit T.O is in the primary directory.

The above LINK directives will link T.O before R.O, since T.O is in the primary directory.

Example 3:

```
LINK A,B,C,D
```

This directive causes the Linker to link the object units named A.O, B.O, C.O, and D.O. If the primary directory contains B.O, and the secondary directory contains A.O, C.O, and D.O, the object units are linked in the following order:

B.O
A.O
C.O
D.O

LINKN Directive

The LINKN directive causes object units to be linked in the following order:

1. Object units previously specified in LINK directives, and any object units requested in embedded LINK directives. The object units are linked in the order in which they are found by the Linker.
2. First (or only) object unit specified in the LINKN directive.
3. Object units specified in LINK and/or LINKN directives that are embedded in the object unit linked as a result of step 2 above.
4. Additional object units, if any, specified in the LINKN directive; the object units are linked in the order in which they were specified in LINKN, regardless of whether they are in the primary or secondary directory. If an object unit contains an embedded directive to link another object unit, the object unit designated in the embedded directive is linked after the object unit that contains the embedded directive.

If directives designate that an object unit be linked more than once within a single bound unit, only the first request is honored.

During each execution of the Linker, at least one LINKN or LINK directive must be specified.

Multiple LINKN directives can be specified within a single root or overlay.

LINKN directives can be embedded in assembly language CTRL statements; the specified object unit(s) are added to the link request list immediately following the object unit in which they were embedded.

FORMAT:

$$\begin{Bmatrix}\text{LINKN}\\\text{LN}\end{Bmatrix}\ \text{obj-unit}_1\ [,\text{obj-unit}_2\ ]\ldots$$

PARAMETER DESCRIPTION:

obj-unit$_n$

Name of an object unit to be linked. An object unit name must be one to six alphanumeric characters and not include a suffix; the first character must be a letter or dollar sign ($). The Linker appends the suffix .O to each object unit name, and searches for the specified object unit name, including the suffix.

Example 1:

```
LINKN X,W
```

This directive designates that the Linker link the object unit named X.O and then link the object unit named W.O.

Example 2:

This example illustrates the order in which object units are linked if LINKN directives are used in conjunction with LINK directives, and there are embedded LINKN directives.

LINK A	Requests the linking of object unit A.O; this name is put into the link request list.
LINK B	Requests the linking of object unit B.O; this name is put into the link request list. In this example, B.O contains an embedded LINK directive to link object unit C.O.
LINKN D,G	Requests the linking of object units D.O and G.O

In this example, all of the specified object units are in the primary directory, and D.O contains an embedded LINK directive to link object unit E.O.

When the LINKN directive is executed, the Linker will link the object units in the following order:

A.O (requested in first LINK directive)
B.O (requested in second LINK directive)
C.O (requested in a LINK directive embedded in object unit B.O)
D.O (first object unit requested in LINKN)
E.O (requested in an embedded directive in object unit D.O)
G.O (second object unit requested in LINKN)

MAP and MAPU Directives

MAP and MAPU directives cause a link map of defined symbols that were not purged and/or undefined symbols to be written to a disk file for deferred printing (i.e., it is written to the disk file name.M in the working directory, unless the -COUT argument was specified in the ECL LINKER command) or to be printed directly on the operator's terminal, another terminal, or a printer.

If MAP is specified, each defined and undefined symbol generated by the linking of object units is listed in the map and preceded by the name of the object unit in which it is located. A map also includes the names of object units that were linked because of embedded Linker directives, and the symbols contained in those object units.

If MAPU is specified, the map contains each undefined symbol and the object unit in which it is located.

MAP and MAPU directives can be interspersed among other Linker directives. When these directives are encountered, all object units named in the link request list are linked before a map is produced. Maps are useful for determining whether all required object units have been linked, and whether all symbols referenced in those object units have been defined.

FORMAT:

$$\begin{Bmatrix} \text{MAP} \\ \text{MP} \end{Bmatrix} \quad \begin{Bmatrix} \text{MAPU} \\ \text{MU} \end{Bmatrix}$$

Default: No map produced.

A full link map (a map generated by the MAP directive) comprises the following sections:

START	Address at which execution of the root or overlay will begin; specified in the START directive or in a linked object unit.
LOW	Lowest memory address at which the current root or overlay was based.
HIGH	Next location after the highest address of the current root or overlay.

$COMM	Address assigned to unlabeled COMMON for the bound unit.
CURRENT	Next location after the current address of the root or overlay (when the map was created).
EXT DEFS	All external symbols currently defined in the symbol table.[7]
UNDEF	If an object unit contains no references to undefined symbols, the object unit name is listed and no symbol names are specified. If object units contain references to undefined symbols, the map indicates, for the root and each overlay, the first object unit[8] in which each symbol was referenced and the relative address of the last reference to each symbol; i.e., if an undefined symbol is referenced in the root and an overlay or in two or more overlays, the symbol will appear more than once in the map. The last reference need not be in the same object unit. If there are external references in both P-relative and immediate memory address forms to an undefined symbol, the symbol is listed twice under UNDEF.

Figure 4-2 illustrates the formats of maps generated by the MAP and MAPU directives. In a single-word (SAF) system, each address or value is specified in four hexadecimal digits; in a double-word (LAF) system, each address or value is specified in eight hexadecimal digits.

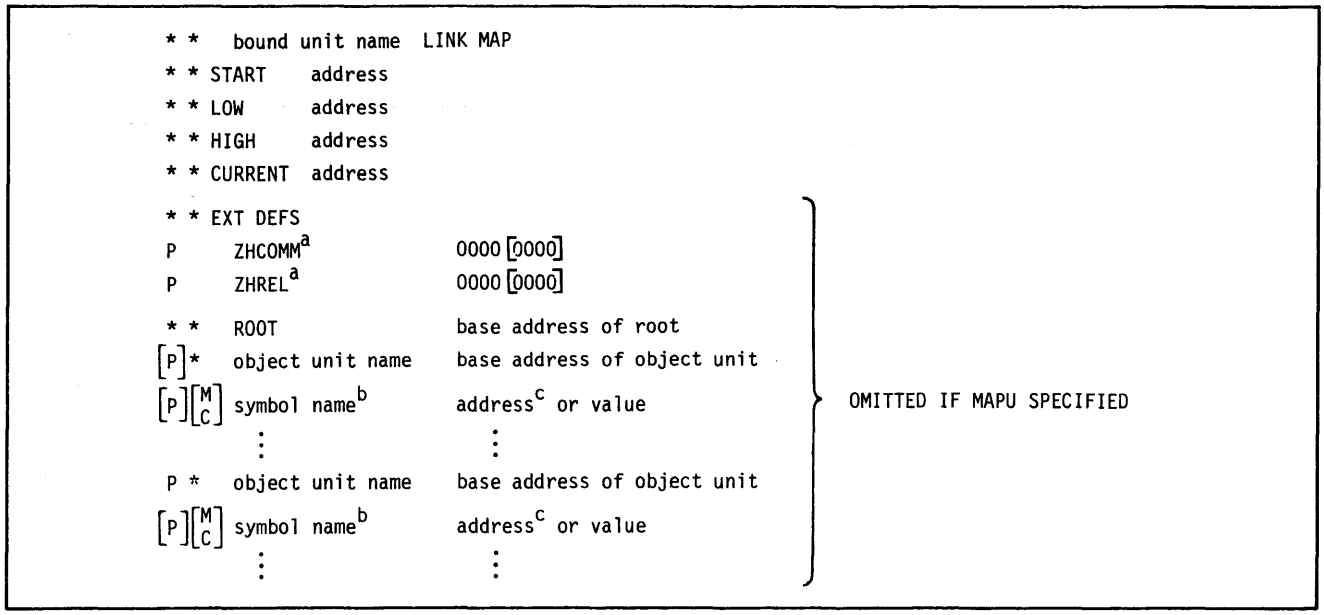


Figure 4-2. Link Map Formats

[7]Unprotected symbols defined in the root or a previously linked overlay will appear in the map unless the symbols are purged via a PURGE or BASE directive. Symbols erroneously defined as both a value and a location will appear twice under EXT DEFS.

[8]The first reference may occur in the root or a previously linked overlay.

```
* *  OVLY                 base address of overlay
[P]*  object unit name     base address of object unit
[P][M C]  symbol name[b]   address[c] or value
        :                      :
[P]*  object unit name     base address of object unit
[P][M C]  symbol name[b]   address[c] or value
        :                      :                    } OMITTED IF MAPU SPECIFIED
* *  UNDEF
P *   object unit name[d]  base address of object unit
     [symbol name[b]       address of most recent reference[e]]
        :                      :
P *   object unit name[d]  base address of object unit
     [symbol name[b]       address of most recent reference[e]]
        :                      :
```

P - Protected symbol

M - Multiply defined symbol

C - Symbol defines labeled or unlabeled common

[a] ZHCOMM and ZHREL are reserved symbol names; they appear on every map as protected symbols. ZHCOMM is located at unrelocatable zero. ZHREL is located at relocatable zero.

[b] The map contains the names of all external symbols currently defined in the symbol table. If there are external references in both P-relative and immediate memory address forms to an undefined symbol, the symbol is listed twice under UNDEF.

[c] To find a location definition, add the relocation factor at load time to the address shown on the map.

[d] All object units linked are listed under UNDEF, even if they contain no unresolved references.

[e] Within the root or a single overlay, the latest reference to an undefined symbol need not be in the object unit that contained the first reference to the symbol. For each undefined symbol, the following information is given under UNDEF: name of the first object unit that contains a reference to the designated symbol, and the relative address of the most recent reference.

Figure 4-2 (cont.) Link Map Formats

OVLY Directive

The OVLY directive assigns the specified name and a number to the *nonfloatable* overlay that immediately follows, and designates the end of the preceding root or overlay.

OVLY must be specified as the first directive of each nonfloatable overlay.

The Linker assigns a two-digit *number* to each overlay. Overlays are numbered sequentially, in ascending order; the first overlay is 00.

FORMAT:

OVLY name

PARAMETER DESCRIPTION:

name

Name of the nonfloatable overlay that immediately follows; the overlay name must comprise one to six alphanumeric characters; the first character must be alphabetic.

Example:

LINKER BU	Loads the Linker and designates BU as the bound unit name.
LINK A	
LINK B	
MAP	
OVLY A2	Designates the end of the root (which comprises object units A.O and B.O) and specifies that the next overlay is a nonfloatable overlay named A2. The Linker assigns the number 00 to this overlay.
LINK X	
LINK Y	
MAP	
QUIT	

Protect Directive

The protect directive prevents certain symbols and/or object unit names from being removed from the symbol table. Symbols that identify addresses from the first operand through the second operand are protected, and object unit names equated to addresses within that range are protected. If a second operand is not specified, the symbol at the address of the first operand and any other symbols or object unit names equated to that address are protected. Once a symbol or object unit name is protected, it cannot later be purged.

FORMAT:

$$\begin{Bmatrix}\text{PROT}\\\text{PT}\end{Bmatrix}\ \begin{Bmatrix}\$\\\%\\\text{X'address'}\\\text{=object-unit-name}\\\text{xdef}\end{Bmatrix}\ \left[\ ,\ \begin{Bmatrix}\$\\\%\\\text{X'address'}\\\text{=object-unit-name}\\\text{xdef}\end{Bmatrix}\ \right]$$

PARAMETER DESCRIPTIONS:

$

Next location after the highest address of the linked root or *previously* linked nonfloatable overlay.

%

Highest address+1 ever used in the linked root or *any* previously linked nonfloatable overlay.

address

Hexadecimal address comprising one to four integers enclosed in apostrophes and preceded by X. The specified address is relative to the beginning of available memory (relative 0) in the memory pool.

=object-unit-name

Specified object unit's base address.

xdef
Address of any previously defined external symbol.

Example 1:

PROT X'1234',X'4565'

This directive protects symbols and object unit names that identify addresses from 1234 through 4565.

Example 2:

PT =FIRST

This directive protects symbols that identify the base address of the object unit FIRST, and all symbols equated to that address. The base address of FIRST is determined by producing a link map (see "MAP and MAPU Directives").

Example 3:

PROT SYM,X'5555'

This directive protects symbols that identify addresses from the address of the previously defined external symbol named SYM through 5555; object unit names equated to those addresses also are protected.

PURGE Directive

The PURGE directive causes the Linker to remove from the symbol table unprotected symbols that define addresses from the first operand through the second operand, and/or object unit names equated to addresses within that range. If a second operand is not specified, the symbol at the address of the first operand and any other symbols or object unit names equated to that address are purged.

An object unit currently being linked may contain definitions used for previously linked object units that won't be used for subsequent object units to be linked. By removing from the symbol table symbols that are no longer required, there is more room for symbols that will be required by subsequently-linked object units.

NOTES:
1. Undefined symbols cannot be purged.
2. Symbols and object unit names that are protected by a protect directive cannot be purged.
3. Only symbol addresses (not values) can be purged.

FORMAT:

$$\begin{Bmatrix}\text{PURGE}\\\text{PE}\end{Bmatrix}\ \begin{Bmatrix}\$\\\%\\\text{X'address'}\\\text{=object-unit-name}\\\text{xdef}\end{Bmatrix}\left[\left[,\begin{Bmatrix}\$\\\%\\\text{X'address'}\\\text{=object-unit-name}\\\text{xdef}\end{Bmatrix}\right]\right]$$

PARAMETER DESCRIPTIONS:

$
Next location after the highest address of the linked root or *previously* linked nonfloatable overlay.

%

Highest address+1 ever used in the linked root or *any* previously linked nonfloatable overlay.

address

Hexadecimal address comprising one to four integers enclosed in apostrophes and preceded by X. The specified address is relative to the beginning of available memory (relative 0) in the memory pool.

=object-unit-name

Specified object unit's base address.

xdef

Address of any previously defined external symbol.

Example 1:

```
PURGE X'1234',X'4565'
```

This directive purges symbols that identify addresses from 1234 through 4565, and object unit names equated to addresses within that range.

Example 2:

```
PE =FIRST
```

This directive purges symbols that identify the base address of the load unit FIRST, and any other symbol names equated to that address. The base address of FIRST is determined by producing a link map (see "MAP and MAPU Directives").

Example 3:

```
PURGE SYM,X'5555'
```

This directive purges symbols that identify addresses from the address of the previously defined external symbol SYM through 5555; object unit names equated to addresses within that range also are purged.

QUIT Directive

The QUIT directive designates that the last Linker directive has been entered. Specify QUIT after the last (or only) overlay, or at the end of the root if there are no overlays.

If a bound unit is being created, execution of the Linker terminates after the bound unit has been created.

If no bound unit is being created, QUIT terminates execution of the Linker.

FORMAT:

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

SHARE Directive

The SHARE directive designates that the bound unit is shareable; i.e., it will be loaded into the system pool and if another task requests that the bound unit be loaded, instead of another copy of the bound unit being loaded, the existing copy in memory is used. The bound unit *must* have reentrant code, but the system does not check to see that it does.

SHARE must be specified in the definition of the root before the first overlay is defined.

SHARE directives can be embedded in assembly language CTRL statements.

FORMAT:

```
{SHARE}
{SE   }
```

START Directive

The START directive designates the relative location within a root or overlay at which execution of the root or overlay will begin once it is loaded into memory by the Loader.

If a linked object unit contains a start address (an Assembler or compiler END statement was specified) and the START directive is specified, the first start address encountered (in either a START directive or an END statement) is used by the Linker for that root or overlay.

FORMAT:

```
{START} symbol
{ST   }
```

PARAMETER DESCRIPTION:

symbol

Name of the external symbol whose address designates the relative address at which the root or overlay will begin executing.

Default: Start address specified in the first linked object unit that has a start address. If the symbol is never defined or a start address is not found, the start address is relocatable 0.

SYS Directive

The SYS directive designates that the bound unit bein created can be used as a system task in the system task group. To use the bound unit in a system task group, it must be loaded during system configuration using the LDBU configuration directive, which is described in the "System Startup and Configuration" section of the System Control manual. If SYS is not specified, the CLM Loader will not load the bound unit. The SYS directive can be embedded in assembly language CTRL statements.

FORMAT:

```
SYS
```

Example:

```
SYS
```

If source units are written to create a function not provided by the MDT operating system and the SYS directive is specified during linking, the bound unit created can be loaded during system configuration and the capability it provides can be used.

VDEF Directive

The VDEF directive assigns a *value* to an external symbol. A symbol should be defined only once, as a value or as a location. When a symbol is defined, its definition is put into the Linker symbol table so that it can be used during linking to resolve external references.

FORMAT:

$$\left\{\begin{matrix}\text{VDEF}\\\text{VF}\end{matrix}\right\}\ \text{symbol,X'value'}$$

PARAMETER DESCRIPTION:

value

Value of the designated symbol; must be a one-word hexadecimal integer enclosed in apostrophes and preceded by X.

Example:

```
VDEF  XMP,X'12'
```

This directive assigns the value 12 to the symbol XMP.

EXAMPLE ILLUSTRATING USAGE OF THE LINKER

LINKER TEST -COUT >SPD>LPT00	The bound unit will be a relative file named TEST created in the working directory. Link maps will be printed on the printer configured as LPT00.
START LOC	
IST INITST	Defines the beginning of initialization code.
LINK OBJ1	Requests that OBJ1.O be linked.
LIB ^DSK03	Names secondary directory.
LINK OBJ2	Requests that OBJ2.O be linked.
OVLY ABLE	Causes OBJ1.O and OBJ2.O to be linked, designates the end of the root, and specifies that a nonfloatable overlay named ABLE immediately follows. The Linker assigns the number 00 to this overlay.
LINKN OBJ3	
LINKN OBJ4	
PROT =OBJ3	Protects the symbol OBJ3. This symbol is protected because a subsequent overlay may be loaded starting at the base address of OBJ3.O .
MAP	Requests a link map.
OVLY BAKER	Designates the beginning of the nonfloatable overlay named BAKER. The Linker assigns the number 01 to this overlay.
LINKN OBJ5	
LINKN OBJ6	
PROT =OBJ5	Protects the symbol OBJ5.
MAP	
OVLY DOG	Designates the beginning of the nonfloatable overlay named DOG. The Linker assigns the number 02 to this overlay.

BASE =OBJ5	The overlay named DOG will be loaded starting at the address where overlay BAKER began.
LINK OBJ7	
MAP	
OVLY FOX	Designates the beginning of the nonfloatable overlay named FOX. The Linker assigns the number 03 to this overlay.
BASE =OBJ3	FOX will be loaded at starting address of overlay ABLE.
IN ^DSK01>MYFILE	Designates that the primary directory now is the directory named ^DSK01>MYFILE.
LIB ^DSK02>MYLIB	Designates that the new secondary directory is named ^DSK02>MYLIB; if necessary, this directory will be searched after the primary directory.
LINK OBJA	
LINK OBJB	
MAP	
OVLY X-RAY	A nonfloatable overlay named X-RAY immediately follows. The Linker assigns the number 04 to this overlay.
BASE =OBJ5	X-RAY will be loaded starting at the beginning address of BAKER.
LINK OBJC	
MAP	
FLOVLY FLOAT	Designates that a floatable overlay named FLOAT immediately follows. The Linker assigns the number 05 to this overlay.
LINK OBJE	
MAP	
QUIT	

PROGRAMMING CONSIDERATIONS

1. While processing object units, the Linker creates a work file LNKWRK.W in the working directory. This file is a variable sequential file. It is initially allocated with 50 control intervals of 256 bytes each, but it can be expanded to the amount of space available in the working directory.
2. If the relative output file is preallocated, it must have the same name as that specified in the name argument of the ECL LINKER command, it must be a fixed, relative file, and it must have a record size of 256 bytes.
3. If multiple object units contain labeled and unlabeled common, the object units will be linked with common blocks appearing in the following order:
   a. Labeled or unlabeled common (defined in first object unit linked)
   b. First object unit (including external references and definitions)
   c. Labeled common (defined in second object unit linked)
   d. Second object unit (including external references and definitions)
   e. Object unit n
4. A root or any overlay may reference any symbol defined in any other root or overlay including "common" symbol definitions. A common area cannot, however, be initialized in any overlay other than the one in which it initially occurs (is made known to the Linker).
5. Relocation can occur during one or both of the following procedures:
   a. Assembly; by specifying an ORG statement, subsequent object text within the object unit is relocated. (See the Assembly Language Reference Manual.)
   b. Linking; by specifying the BASE directive, subsequent object units to be linked within the root or overlay have a specified relative load address. (See "BASE Directive" earlier in this section.)

Example:

Described below are three methods of relocating a unit so that it is executed at relative location 100 within the memory assigned to the bound unit. This unit will constitute a root.

	Assembly	*Linking*
Method I:	ORG X'0100' before the first line of executable code.	Don't specify BASE directive.
Method II:	Don't specify ORG. (Default is 0.)	Specify BASE X'100'
Method III:	ORG X'10'	BASE X'F0'

6. When relocating object units or nonfloatable overlays during the assembly or linking procedure, it is your responsibility to ensure that code is not inadvertently overwritten.
7. If more than six characters are specified for an object unit name or symbol name, or more than eight characters are designated for a bound unit name, subsequent characters are truncated.
8. Forward external references with offsets are not permitted. If the Linker encounters them, an error message is issued and execution of the Linker terminates.
9. Common definitions may appear more than once in object units being linked. Only the first occurrence of either a labeled or unlabeled common definition block is used to reserve the defined amount of memory. Therefore, the largest definition of labeled or unlabeled common should be linked first. Common blocks are allocated space by the Linker by assigning the current location counter (address) to the symbol name, and then incrementing the current location counter by the size of memory specified for the common block.
10. A BASE directive in the root or an overlay cannot specify an address less than the beginning of that root or overlay; i.e., it cannot be less than the first word of the first object unit linked in that root or overlay.
11. If BASE $ or BASE % is specified in the root, it is equivalent to BASE 0.
12. The start address of the root or an overlay must be in the first 32K-1 words.

SECTION 5

DEBUGGING PROGRAMS

While a program is executing, it can be debugged by using Debug. If there is not enough room in memory for Debug, you can debug a program by temporarily leaving space in the program or by using Patch to append monitor points. (See "Debugging Programs Without Using Debug" later in this section.)

DEBUG

Debug provides patching and testing facilities for application programs running under the operating system. Debug runs as its own task group.

Program testing and error correction is performed as an interactive dialogue between the operator and Debug. Execution of Debug is controlled by directives entered to Debug. Addresses used with Debug are system-wide absolute memory addresses; therefore, Debug directives are effective across task and task group boundaries. Debug directives are entered through the device designated during loading of Debug as the directive input device. The directive input device usually is a terminal.

The following functions can be performed using Debug:

- Define, store, and execute (either immediately or after a delay, depending on the directive) a sequence of directives either entered through the input device, or referenced when a breakpoint directive or trace trap (BRK generic instruction) is encountered in the load unit being tested.[1]
- Set, clear, or print breakpoints in task code to monitor task status. (Breakpoints are described in detail later in this section.)
- Display, change, and dump either memory or registers; information may be printed on a line printer, the operator's terminal, or another terminal.
- Evaluate expressions.

Debug File Requirements

If predefined or delayed execution Debug directives are to be used, they are stored in a preallocated, relative disk file DEBUG.WORK (these directives are identified and described in Table 5-2, "Summary of Debug Directives, by Function," later in this section). The file DEBUG.WORK must be in the volume major directory of the disk device referenced in the specify file (SF) directive. (The SF directive is described later in this section.)

Loading the Debug Task Group

During initialization, you must specify in a MEMPOOL configuration directive a memory pool large enough for Debug. (MEMPOOL is described in "System Startup and Initialization" in the System Control manual.) The identification of the pool to be used by Debug must be $D. The pool must comprise a minimum of 3500 words.

[1] Breakpoints and trace traps either cause a specified Debug directive line to be executed, or interrupt execution of the task so that its status can be determined.

Example:

```
MEMPOOL ,$D,3500
```

This MEMPOOL directive creates a nonexclusive memory pool comprising 3500 words that can be specified when the Debug task group is loaded into memory.

To load Debug, enter through the operator's terminal either the spawn group (SG) command or both the create group request and the enter group request commands (see the "Execution Control Language" section of the System Control manual for descriptions of these commands).

NOTE: To load Debug, you must enter command(s) through the operator's terminal. However, you can designate in the spawn group command or enter group request command that Debug directives will be entered through another specified terminal.

The identification for the Debug task group must be $D. To achieve control over the task code being tested, Debug must be given a priority that is higher than that assigned to the task code, but lower than that given to the directive input device and optional printer used by the operator for dialog. DEBUGDB is the name of the bound unit that contains Debug.

The following examples illustrate both of the methods of loading Debug. Example 1 illustrates a spawn group command. Example 2 illustrates a create group request and an enter group request. The following description applies to both examples:

The Debug task group's identification is $D, your identification is GALE.TECH, and the priority level of Debug is 12. Directives to Debug will be entered through the operator's terminal, which is identified by its pathname >SPD>CONSOLE. The bound unit DEBUGDB will be loaded, if necessary, and executed by the task group's lead task.

Example 1:

Loading Debug by a spawn group command:

```
SG $D GALE.TECH 12 >SPD> CONSOLE -EFN DEBUGDB
```

Example 2:

Loading Debug by create group request and enter group request commands:

```
CG $D 12 -EFN DEBUGDB
EGR $D GALE.TECH >SPD>CONSOLE
```

NOTE: The operator's terminal is controlled by a system software component called the operator interface manager (OIM) that provides a standard means by which all tasks can communicate with an operator. OIM keeps track of the messages output to the operator's terminal by providing the task group identification in the prefix to each message. If you are entering Debug directives through the operator's terminal, it is recommended that you designate Debug as the OIM default task group; otherwise, each Debug directive must be preceded by Δ$D Δ. To designate Debug as the OIM default task group, enter the following command at any time prior to entering the first Debug directive:

ΔCΔ:$D:

Debug Directives

Debug directives consist of only a directive name or a directive name and one or more parameters. Within a directive, parameters are separated from each other by one or more spaces. All parameter values are entered using hexadecimal notation.

Multiple Debug directives can be entered on a single line. Each directive, except the last, must be followed by a semicolon (;).

Press RETURN at the end of each line (i.e., immediately after the last or only directive).

Symbols used in Debug directive lines are described in Table 5-1.

TABLE 5-1. SYMBOLS USED IN DEBUG DIRECTIVE LINES

Symbol Type	Meaning
Arithmetic Operators	
plus sign (+)	Performs addition.
minus sign (-)	Performs subtraction.
K	Multiplies a hexadecimal integer by 1024 decimal (400 in hexadecimal) when K is the last character of an integer expression.
Address Operators	
period (.)	Represents the last start address used in a previous memory reference directive (DH,CH,DP).
ampersand (&)	Represents the address of the next location beyond the last one used by a previous memory reference directive (DH,CH,DP).
brackets [ ] [a]	Signifies the contents of the location defined by the expression within the brackets. Three levels of nesting may be used.
Reserved Symbols	
$Bn	Contents of base register n of the active level. The values 1 through 7 can be used for n.
$Rn	Contents of the data register n of the active level. The values 1 through 7 can be used for n.
$P	Contents of the program counter of the active level.
$I	Contents of the indicator register of the active level.
$S	Contents of the system status register (level number and privilege bit only) of the active level.
$SL	Represents the value of the level number of the active level.
G through Z	Twenty single-character symbols having initial values of zero. Values may be assigned using the AS directive.
Notational Symbols	
braces { } [a]	Indicate optional parameters.
ellipses ...	Indicates the ability to repeat parameters within braces.
delta (Δ)	Indicates one or more spaces.
Debug Language	
parentheses ( )	Indicate directive or header information to be stored for later use. Unmatched right parenthesis results in an error. A right parenthesis that is paired with the first left parenthesis terminates the directive definition.
exp	Indicates a valid expression formed using expression elements.
rexp	Consists of exp_1/exp_2, where exp_1 is a hexadecimal number that is a value or a location; exp_2 is an optional hexadecimal repeat factor whose value must be between 1 and 32,767. If exp_2 is omitted, the value of exp_1 is assumed.
;	Separation character between directives on the same line.
*	Signifies "all" in certain print, clear, and list directives.

[a] In this section, brackets and braces have special meanings, as described above. In each other section, they are interpreted differently (see "Symbols Used in This Manual," in the "Overview" section).

Table 5-2 summarizes Debug directives by function. These directives are described in detail alphabetically on the following pages. In each directive's format, it is assumed that Debug was previously designated as the OIM default task group, so Δ$DΔ is not specified before each directive name.

NOTES: 1. Pay careful attention to the format of each directive, because the usage of delimiters, if any, between a directive name and the first (or only) parameter varies according to which directive is being specified.

2. If a directive has a parameter in which you may specify the logical resource number (lrn) of the device on which information will be printed, Debug uses the specified device without first determining whether the device has been reserved for exclusive use by another task; i.e., Debug bypasses the MDT file system.

TABLE 5-2. SUMMARY OF DEBUG DIRECTIVES, BY FUNCTION

Function	Directive Name	Meaning
Directive line definition and handling	Dn	Define directive line n
	En	Execute directive line n
	P*	Print all predefined directive lines
	Pn	Print directive line n
Breakpoint control	C*	Clear all breakpoints
	Cn	Clear breakpoint n
	GO	Proceed form breakpoint
	L*	List all breakpoints
	Ln	List breakpoint n and associated directive line
	Sn	Set breakpoint n
Trace trap control	DT	Define trace directive line
	PT	Print trace directive line
Active level control	SL	Set active level
	TL	Set temporarily active level
Memory and register control	AR	Print contents of all active level registers
	CH	Change memory
	DH	Display memory in hexadecimal
	DP	Dump memory in hexadecimal and ASCII
Symbol control	AS	Assign a hexadecimal value to symbol
	VH	Print value of expression in hexadecimal
General execution	AL	Activate level(s)
	Hn	Print header line
	LL	Specify line length of operator's terminal or another terminal currently in use
	RF	Reset file location
	SF	Specify file location

NOTES: 1. The memory and register control directives (AR, CH, DH, and DP) apply to registers on the active level. To determine which level is the active level and/or to set the active level to a specified value, see "Determining/Setting the Active Level" below.
2. The following directives are predefined or delayed execution directives and are stored in the file DEBUG.WORK: Sn, Hn, Dn, DT

Planning Considerations

Setting Breakpoints

Breakpoints can be set to trap at selected task code locations. At breakpoints, memory and register values can be displayed and changed. In this way, a task can be executed, the values of its variables checked as execution proceeds, code modified, and if necessary, variable values changed in order to test the sequence of code up to the next breakpoint.

Following are guidelines for setting breakpoints:

1. Breakpoints can be set in a task group (or in an overlay in a task group) only when the task group/overlay currently is memory resident and all bound units for the task group have been loaded into memory by the Loader.
2. Breakpoints may *not* be set in code that will be executed at the inhibit level.
3. If sharable code contains breakpoints, each task that uses the code encounters the breakpoint, regardless of which task group the task is in.

Breakpoints are set by specifying the set breakpoint directive (Sn); the detailed description of Sn includes additional rules for specifying breakpoints.

Determining/Setting the Active Level

The active level is the priority level currently in effect. Some directives are effective only on the active level. You must initially establish a level as the active level by specifying the set level directive. Thereafter, the active level assumes the value that will most probably be needed, based on the Debug action in progress; i.e., breakpoint, trace trap, or temporary reference to a different level.

If you want to do something on another priority level, you can change the active level by respecifying the set level directive (SL) or *temporarily* designate another level as the active level by specifying the set temporary level directive (TL); in the latter case, the level is considered the temporarily active level. After the desired actions are performed on the temporarily active level, the active level reverts to the level specified in the previous set level directive.

Following are guidelines for determining which level is the active level, and methods of setting the active and temporarily active level.

1. The set level directive (SL) sets (or changes) the active level. The specified level becomes the default level accessible by the operator's terminal or another terminal that is the directive input device.
2. The set temporary level directive (TL) designates a level as the temporarily active level; this permits you to display or alter registers of a level different from the default terminal level without actually changing the terminal level.
3. Whenever input from the operator's terminal or another terminal is processed, the active level is set to the level of that device. After the input is processed, the level reverts to what was specified in the last SL directive.

Deactivating Real-Time Clock

Some applications require that the real-time clock (RTC) be activated at load time. While the clock is turned on, the CPU is difficult to use in single instruction mode because the RTC is continually generating an interrupt at clock level 4. In the early stages of application debugging, it may be useful to turn off the clock to facilitate "stepping" through a code sequence without interference. This is easily done using the capability of executing a single instruction from the D0 register, as described in the following procedure.

To turn off the clock, use the capability of executing one instruction from the instruction register (whose selection code is D0) to execute an RTCF instruction while in single instruction mode. Perform the following steps:

1. Press Stop, and record value in E0
2. Select D0; change to 0005
3. Press Execute (this turns off clock)
4. Select D0; change to 0000
5. Select E0; change to recorded E0 halt address
6. Press Ready and Execute

Maintaining a Trace History

When using Debug with disk-stored directive lines that execute upon encountering a trap or a breakpoint, a trace history may be maintained on a line printer.

Activate Level Directive

The activate level directive (AL) activates a priority level corresponding to each specified expression

FORMAT:

ALΔexp{ΔexpΔ...}

PARAMETER DESCRIPTION:

exp

Priority level to be activated.

Example:

```
AL  A  A+2
```

This example activates priority levels 10 and 12 (decimal)

All Registers Directive

The all registers directive (AR) causes the printing of all registers for the active level.

FORMAT:

```
AR {/lrn}
```

PARAMETER DESCRIPTION:

/lrn

Logical resource number of the device on which the printout will occur.
Default: Operator's terminal

Example:

```
AR/3
```

This example causes the contents of all the registers for the *active* level to be printed on the device referred to as logical resource number 3.

Assign Directive

The assign directive (AS) assigns a specified hexadecimal value to a specified symbol; this directive is used to alter registers of the active level, and reserved symbols.

FORMAT:

```
ASΔsymΔexp{ΔsymΔ exp... }
```

PARAMETER DESCRIPTIONS:

sym

Register or reserved symbols G through Z.

exp

Hexadecimal value that will be assigned to the specified register or symbol.

Example:

```
AS  $R1  -2  X  1408  $B7  X+15
```

This example causes -2 to be assigned to data register 1, 1408 to be assigned to the reserved symbol X, and 141D to be assigned to base register 7.

Clear All Directive

The clear all directive (C*) clears all defined breakpoints.

FORMAT:

```
C*
```
