Gould MPX-32^{TM} Utilities

Release 3.0

Reference Manual

July 1987

Publication Order Number: 323-004590-000

GOULD
Electronics

HISTORY

The Gould MPX-32 Utilities Release 3.0 Reference Manual, Publication Order Number **323-004590-000**, was printed July, 1987.

This manual contains the following pages:

Title page
Copyright page
iii/iv through xxi/xxii

OVERVIEW

1-1 and 1-2
2-1 through 2-3/2-4

CATALOGER

Title page
iii through v/vi
1-1 through 1-3/1-4
2-1 through 2-25/2-26
3-1 through 3-22
4-1 through 4-4
5-1 through 5-5/5-6

DATAPOOL EDITOR

Title page
iii and iv
1-1 and 1-2
2-1 through 2-8
3-1 through 3-3/3-4
4-1 and 4-2
5-1/5-2

FILE MANAGER

Title page
iii/iv
1-1 through 1-3/1-4
2-1 through 2-8
3-1 through 3-10
4-1 and 4-2
5-1/5-2

MACRO ASSEMBLER

Title page
iii through v/vi
1-1 through 1-3/1-4
2-1 through 2-16
3-1 through 3-29/3-30
4-1 through 4-33/4-34
5-1 through 5-6
6-1 through 6-12
A-1 through A-3/A-4
B-1/B-2
C-1 and C-2
D-1 and D-2

MACRO LIBRARY EDITOR

Title page
iii/iv
1-1 and 1-2
2-1 through 2-4
3-1 through 3-4
4-1 and 4-2
5-1 and 5-2

MEDIA CONVERSION

Title page
iii/iv
1-1 and 1-2
2-1 through 2-3/2-4
3-1 through 3-10
4-1 and 4-2
5-1 through 5-4

Continued

SOURCE UPDATE

Title page
iii and iv
1-1 and 1-2
2-1 through 2-8
3-1 through 3-9/3-10
4-1 through 4-3/4-4
5-1 through 5-4

SUBROUTINE LIBRARY EDITOR

Title page
iii/iv
1-1/1-2
2-1 through 2-5/2-6
3-1/3-2
4-1/4-2
5-1 and 5-2

SYMBOLIC DEBUGGER

Title page
iii through v/vi
1-1 through 1-6
2-1 through 2-26
3-1 through 3-29/3-30
4-1 through 4-10
5-1 through 5-12

TEXT EDITOR

Title page
iii and iv
1-1 through 1-3/1-4
2-1 through 2-6
3-1 through 3-25/3-26
4-1 through 4-4

CONTENTS

GOULD MPX-32 UTILITIES OVERVIEW

CATALOGER (CATALOG)

Section

Section

FIGURES

TABLES

DATAPOOL EDITOR (DPEDIT)

Section

Section

Section

APPENDICES

FIGURES

TABLES

MACRO LIBRARY EDITOR (MACLIBR)

Section

Section

FIGURES

TABLES

SUBROUTINE LIBRARY EDITOR (LIBED)

SYMBOLIC DEBUGGER (SYMDB)

Section

3--DIRECTIVES

Section

GOULD MPX-32 UTILITIES OVERVIEW

SECTION 1 - USING THE MPX-32 UTILITIES MANUAL

The Gould MPX-32 utility package is a collection of the following utilities:

- Cataloger (CATALOG)
- Datapool Editor (DATAPOOL)
- File Manager (FILEMGR)
- Macro Assembler (ASSEMBLE)
- Macro Library Editor (MACLIBR)
- Media Conversion (MEDIA)
- Source Update (UPDATE)
- Subroutine Library Editor (LIBED)
- Symbolic Debugger (SYMDB)
- Text Editor (EDIT)

Because the utilities can be installed on a system individually, each utility description in this manual can be used as a stand-alone manual. To tailor this manual to a particular system, remove the descriptions of the utilities which are not installed on the system.

Each utility description begins with a tabbed page and generally has the following format:

- Overview
- Usage
- Directives
- Errors and Aborts
- Examples

The Overview section describes the utility's function and summarizes its directives.

The Usage section describes how to access, exit, and use the utility. Related information, such as logical file code assignments and utility options, are also described in this section.

The Directives section describes the function and syntax of each directive in alphabetical order.

The Errors and Aborts section describes possible errors, aborts, and their messages. Explanations of the error and abort code numbers are included.

The Examples section contains sample input and/or output illustrating the usage of the utility.

Most utilities provide the capability to use previously created files for input sources and/or output destinations. Valid characters for file names, directories, and other referenced names are A-Z, 0-9, period, and underscore. Although other characters are generally accepted, their use is not recommended.

If a complete pathname is specified, any valid file name can be used. If only a file name is specified, the file name cannot begin with a period or a string of digits (0-9) followed by a period.

For file names containing special characters, enclose the name in single quotes. Use this feature only to gain access to files with names containing unrecommended characters. After gaining access, save or store the file using a file name of recommended characters.

Files assigned to logical file codes (LFC's) will be forced to the appropriate format - blocked or unblocked unless otherwise noted in the LFC description.

Input records to the utilities must be in 80-byte card image format.

When a utility is activated, a copyright statement is issued. If the utility is accessed in the batch mode, the copyright is printed on the listed output. In the interactive mode, the copyright is displayed on the user terminal. The copyright statement has the following format:

```
MPX-32 UTILITIES RELEASE x.x (utility Rx.x.x)
(C) COPYRIGHT 1983, GOULD INC., COMPUTER SYSTEMS DIVISION, ALL RIGHTS RESERVED
```

RELEASE x.x is the release number of the MPX-32 Utilities and utility Rx.x.x is the name of the specific utility and its internal version number.

SECTION 2 - DOCUMENTATION CONVENTIONS

Notation conventions used in directive syntax, messages, and examples throughout this manual are described below.

lower case letters

In directive syntax, lower case letters identify a generic element that must be replaced with a value. For example:

```
!ACTIVATE taskname
```

means replace taskname with the name of a task. For example:

```
!ACTIVATE DOCCONV
```

In messages, lower case letters identify a variable element. For example:

```
**BREAK** ON:taskname
```

means a break occurred on the specified task.

UPPER CASE LETTERS

In directive syntax, upper case letters specify a keyword must be entered as shown for input, and are printed as shown in output. For example:

```
SAVE filename
```

means enter SAVE followed by the name of a file. For example:

```
SAVE DOCCONV
```

In messages, upper case letters specify status or information. For example:

```
taskname,taskno ABORTED

*YOUR TASK IS IN HOLD. ENTER CONTINUE TO RESUME IT.
```

Braces { }

Elements placed one under the other inside braces specify a required choice. You must enter one of the arguments from the specified group. For example:

$$\begin{Bmatrix} \text{counter} \\ \text{startbyte} \end{Bmatrix}$$

means enter the value for either counter or startbyte.

Brackets [ ]

An element inside brackets is optional. For example:

```
[CURR]
```

means the term CURR is optional.

Items placed one under the other within brackets specify you may optionally enter one of the group of options or none at all. For example:

```
┌base name┐
└progname ┘
```

means enter the base name or the program name or neither.

Items in brackets within encompassing brackets specify one item is required only when the other item is used. For example:

```
TRACE [lower address  [upper address]]
```

means both the lower address and the upper address are optional, and the lower address may be used alone. However, if the upper address is used, the lower address must also be used.

Commas between multiple brackets within an encompassing set of brackets are not required unless subsequent elements are selected. For example:

```
M.DFCB fcb,lfc [, [a] , [b] , [c] , [d] , [e]]
```

could be coded as:

```
M.DFCB FCB12,IN

     or

M.DFCB FCB12,IN,,ERRAD

     or

M.DFCB FCB13,OUT,,ERRAD,,PCK
```

Horizontal Ellipsis ...

The horizontal ellipsis indicates the previous element can be repeated. For example:

```
name [,name]...
```

means one or more values separated by commas can be entered.

Vertical Ellipsis

```
.
.
.
```

The vertical ellipsis used in examples indicates that directives, parameters, or instructions have been omitted. For example:

```
COLLECT 1
  .
  .
  .
LIST
```

means one or more directives have been omitted between COLLECT and LIST.

Numbers and Special Characters

In a syntax statement, any number, symbol, or special character must be entered as shown. For example:

```
(value)
```

means enter the proper value enclosed in parentheses, i.e., (234).

Underscore

In syntax statements, underscoring specifies the letters, numbers, or characters that may be used as an abbreviation. For example:

```
LIST  filename
‾‾‾
```

means spell out the directive LIST or abbreviate it to either LIS or L.

Bold

In examples, all terminal input is printed in bold; terminal output is not. For example:

TSM > **EDIT**

means TSM > was written to the terminal and EDIT was typed by the user.

CNTRL Key

CNTRL indicates the terminal Control key. For example:

```
CNTRL I
```

means to simultaneously press the Control and I keys.

Cataloger (CATALOG)

MPX-32 Utilities

CONTENTS

Section

FIGURES

TABLES

CATALOGER (CATALOG)

SECTION 1 - OVERVIEW

1.1 General Description

The Cataloger (CATALOG) utility processes standard, nonbase mode object code to produce load modules that are ready to activate in one of three task environments: real-time, interactive, or batch.

A load module is created using job control language and CATALOG directives. The load module resides in a permanent file specified in an LMPATH directive. If LMPATH is not used, the file name is taken from the load module name on the first BUILD or CATALOG directive.

CATALOG recognizes 1 to 16 character file names. Unless specified, files assigned to logical file codes are forced to the appropriate format--blocked or unblocked.

1.2 Directive Summary

Below is a list of CATALOG directives in alphabetical order. Underlining indicates accepted directive abbreviations. Each directive is explained in more detail in Section 3.

Directive	Function
*(in Column 1)	Indicates a comment line
ABSOLUTE	Specifies an absolute origin for the task data section (DSECT) in the task being cataloged
ALLOCATE	Allocates additional memory for a main load module in the task being cataloged
ASSIGN	Equates system files, pathnames, RIDs, temporary files, devices, and LFCs with an LFC in the task being cataloged
ASSIGN1	Equates a permanent disc file with an LFC in the task being cataloged. This directive is for compatibility only; its use is not recommended.
ASSIGN2	Equates system files SBO, SLO, SYC, or SGO with an LFC in the task being cataloged. This directive is for compatibility only; its use is not recommended.
ASSIGN3	Equates a device with an LFC in the task being cataloged. Also assigns a temporary disc file. This directive is for compatibility only; its use is not recommended.
ASSIGN4	Equates an LFC in the task being cataloged with an existing LFC. This directive is for compatibility only; its use is not recommended.
BUFFERS	Establishes the number of blocking buffers required for dynamic assignments in nonshared tasks. Establishes the total number of blocking buffers required for shared tasks.
BUILD	Identifies and describes a load module to be cataloged in the current working directory or in the pathname of a previous LMPATH directive

Directive	Function
CATALOG	Identifies and describes a load module to be cataloged in the system directory or in the pathname of a previous LMPATH directive
CONNECT	Assigns a Datapool dictionary to a specified Datapool (DPOOL00 through DPOOL99 or DATAPOOL) partition
ENVIRONMENT	Describes the memory class, residency, map size, and sharing or multicopying requirements of a task
EXCLUDE	Specifies referenced global symbol names in library object modules to be excluded from the load module
EXIT	Terminates CATALOG directive input
EXTDMPX	Positions the extended portion of MPX-32 in the logical address space of the task being cataloged (if the expanded space option of MPX-32 is used).
FILES	Establishes the number of dynamic disc file and device assignments in nonshared tasks. Establishes the total disc file and device assignments in shared tasks.
INCLUDE	Specifies unreferenced global symbol names in library object modules to be included in the load module
LINKBACK	Specifies the overlay load modules at lower levels to link to the current overlay load module
LMPATH	Specifies the pathname of the file where the load modules are to be written
LORIGIN	Establishes a new overlay level and origin for an overlay load module
MOUNT	Indicates a nonpublic volume that is required by the task being cataloged
OPTION	Sets execution options for the task being cataloged
ORIGIN	Establishes a new origin (at current level) for an overlay load module
PROGRAM	Specifies an object module by program name from SGO to include in a load module
PROGRAMX	Specifies no object modules from SGO should be included in a load module
RECATALOG	Indicates that one or more overlay segments of a single file load module are being updated. Optionally supplies the name of the file.
SEGFILES	Specifies the number of noncontiguous disc files to be accessed by the task being cataloged
SPACE	Allows the potential maximum task size to be increased above the default 2MB size.
SYMTAB	Specifies that symbol table references saved previously with a CATALOG SYM option are to be used when an overlay load module for a task is cataloged separately or recataloged

Directive	Function
VOLUMES	Specifies the number of nonpublic volumes that can be dynamically mounted at any one time by the task being cataloged

SECTION 2 - USAGE

2.1 Accessing CATALOG

CATALOG can be accessed in the batch or interactive modes in one of three ways:

- $CATALOG
- $RUN CATALOG (valid only from the system directory)
- $EXECUTE CATALOG

When accessing CATALOG interactively, the CAT> prompt is displayed:

```
TSM>$CATALOG
CAT>
```

2.2 Logical File Code Assignments

The following logical file codes are used by CATALOG:

Logical File Code	Description
SYC	CATALOG directive input
SGO	Object modules from compilation or assembly
LIS	Library assignment for object modules from the system subroutine library (default @SYSTEM(SYSTEM)MPXLIB)
LID	Directory assignment for object modules from the system subroutine directory (default @SYSTEM(SYSTEM) MPXDIR)
LIB or Lnn	Library assignment for object modules from user libraries (nn = 00 through 99)
DIR or Dnn	Directory assignment corresponding to assigned library (nn = 00 through 99)
DPD	Dictionary assignment for DATAPOOL variables used in object modules
SLO	Listed output
SYI	Symbol table as input
SYM	Symbol table as output
P00 - P99	Dictionary assignments for DPOOL00 - DPOOL99 variables used in object modules

The following sections describe and Table 2-1 summarizes the default and optional LFC assignments.

2.2.1 Source Input (SYC)

Source input is a file of CATALOG directives that is assigned to logical file code SYC.

SYC Default and Optional Assignments

The default assignment for SYC is to the System Control file (SYC):

```
$ASSIGN SYC TO SYC
```

There are two optional assignments for SYC:

```
$ASSIGN SYC TO  { pathname     }
                { DEV=devmnc   }
```

pathname	is the pathname of a file containing CATALOG source input
devmnc	is the device mnemonic of a device containing CATALOG source input

2.2.2 Object Modules from Compilation or Assembly (SGO)

The file of object modules from compilation or assembly is assigned to logical file code SGO.

SGO Default and Optional Assignments

The default assignment for SGO is to the System General Object file (SGO):

```
$ASSIGN SGO TO SGO
```

There are two optional assignments for SGO:

```
$ASSIGN SGO TO    pathname
                  DEV=devmnc
```

pathname	is the pathname of a file containing object modules from compilation or assembly
devmnc	is the device mnemonic of a device containing object modules from compilation or assembly

2.2.3 Object Modules from Subroutine Libraries (LIS, LIB, and Lnn)

CATALOG links object modules from subroutine libraries assigned to logical file codes LIS, LIB, and Lnn. LIS is used (by default) to access the system subroutine library (MPXLIB). LIB or Lnn should be assigned to access user subroutine libraries.

CATALOG searches the library assigned to logical file code LIS by default and any user-specified libraries assigned to logical file codes LIB and Lnn. The libraries are searched in the following order: LIB, L00 through L99, and LIS. The number of libraries searched is limited only by the number of ASSIGNs which may be processed by TSM plus any statically assigned user libraries added to the CATALOG load module. These LFCs are forced unblocked by CATALOG.

LIS, LIB, and Lnn Default and Optional Assignments

The default assignment for LIS is to the system subroutine library:

```
$ASSIGN LIS TO @SYSTEM (SYSTEM) MPXLIB
```

There is one optional assignment for LIS:

```
$ASSIGN LIS TO pathname
```

pathname	is the pathname of a file containing object modules in Subroutine Library Editor (LIBED) format

There are no default assignments for LIB and Lnn. To access user subroutine libraries, the optional assignments for LIB or Lnn should be specified as follows:

```
$ASSIGN LIB TO pathname

$ASSIGN Lnn TO pathname
```

pathname	is the pathname of a file containing object modules in Subroutine Library Editor (LIBED) format
Lnn	is a user-defined LFC in the range L00 to L99 representing a user subroutine library

2.2.4 Subroutine Library Directories (LID, DIR, and Dnn)

The directory for a subroutine library is assigned to logical file codes LID, DIR, and Dnn. These LFCs are forced unblocked by CATALOG.

The LID assignment is to the directory that corresponds to the LIS system subroutine library assignment. If user subroutine libraries are assigned to LIB or Lnn, the corresponding DIR or Dnn assignments must be made for the related directories.

LID, DIR and Dnn Default and Optional Assignments

The default assignment for LID is to the system subroutine library directory:

```
$ASSIGN LID TO @SYSTEM (SYSTEM) MPXDIR
```

There is one optional assignment for LID:

```
$ASSIGN LID TO pathname
```

pathname	is the pathname of a file containing the subroutine library directory

There are two optional assignments for assigning user subroutine library directories:

```
$ASSIGN DIR TO pathname

$ASSIGN Dnn TO pathname
```

pathname	is the pathname of a file containing the subroutine library directory
Dnn	is a user-defined LFC representing the directory for a user subroutine library. Directory LFCs are D00 to D99, corresponding to the user subroutine libraries L00 to L99.

Note: There are no default assignments for DIR or Dnn.

2.2.5 DATAPOOL Variables Dictionary (DPD)

DATAPOOL variables referenced in object modules are defined in a Datapool dictionary. Datapool dictionaries are built using the Datapool Editor (DPEDIT) utility. The DATAPOOL dictionary for use by CATALOG is assigned to logical file code DPD. This LFC is forced unblocked by CATALOG.

DPD Default and Optional Assignments

There is no default assignment for DPD.

There is one optional assignment for DATAPOOL variables:

```
$ASSIGN DPD TO pathname
```

pathname	is the pathname of a file containing the DATAPOOL dictionary

Note: The DATAPOOL dictionary can optionally be assigned using the CONNECT directive. If this is done, LFC DPD must not be user assigned.

2.2.6 DPOOL00 - DPOOL99 Variables Dictionaries (P00 - P99)

DPOOL00 through DPOOL99 variables referenced in object modules are defined in Datapool dictionaries. Datapool dictionaries are built using the Datapool Editor (DPEDIT) utility. The Datapool dictionaries used by CATALOG are assigned by the CONNECT directive to logical file codes P00 through P99.

P00 - P99 Default and Optional Assignments

There are no default or optional assignments for P00 through P99; the user must not assign these LFCs.

2.2.7 System Listed Output (SLO)

The system listed output file contains the output of the cataloging session. The output includes a directive log, a load map, and any error messages generated. The system listed output file is assigned to logical file code SLO.

SLO Default and Optional Assignments

The default assignment for SLO is to the System Listed Output device (SLO):

```
$ASSIGN SLO TO SLO
```

There are three optional assignments for SLO:

```
$ASSIGN SLO TO  [pathname   ]
                [DEV=devmnc ]
                [LFC=UT     ]
```

pathname	is the pathname of a file to contain listed output
devmnc	is the device mnemonic of a device to which the listed output will be directed
LFC=UT	assigns output to the user terminal

Note: If the origin of CATALOG is interactive, any error messages generated are written to both UT and SLO automatically. If the user wants the load map to appear on the terminal, SLO must be assigned to UT.

2.2.8 Symbol Table Output (SYM)

A symbol table is the mechanism for resolving external references when cataloging a task with overlays in separate CATALOG runs. If a symbol table is desired for later use with logical file code SYI, the symbol table option must be set and a file or device for symbol table output must be assigned. The file or device to contain the symbol table output is assigned to logical file code SYM.

SYM Default and Optional Assignments

There is no default assignment for SYM.

There are two optional assignments for SYM:

```
$ASSIGN SYM TO  {pathname   }
                {DEV=devmnc }
```

pathname	is the pathname of a file to contain the symbol table output
devmnc	is the device mnemonic of a device where the symbol table output is directed

2.2.9 Symbol Table as Input (SYI)

Instead of regenerating the symbol table when recataloging a load module, the symbol table which was assigned to logical file code SYM generated by the previous cataloging of the load module can be used as input. The file or device containing the symbol table is assigned to logical file code SYI.

SYI Default and Optional Assignments

There is no default assignment for SYI.

There are two optional assignments for SYI:

```
$ASSIGN SYI TO  { pathname    }
                { DEV=devmnc  }
```

pathname	is the pathname of a file containing the symbol table
devmnc	is the device mnemonic of a device containing the symbol table

2.2.10 LFC Summary

The following is a table of LFCs used by CATALOG and their default and optional assignments:

Table 2-1
CATALOG LFC Summary

LFC	Default Assignment	Optional Assignment
SYC	SYC	pathname DEV = devmnc
SGO	SGO	pathname DEV = devmnc
LIS	@SYSTEM (SYSTEM)MPXLIB	pathname
LIB	none	pathname
Lnn	none	pathname
LID	@SYSTEM (SYSTEM)MPXDIR	pathname
DIR	none	pathname
Dnn	none	pathname
DPD	none	pathname

SLO	SLO	pathname
		DEV = devmnc
		LFC = UT

SYM	none	pathname DEV = devmnc
SYI	none	pathname DEV = devmnc
P00 - P99	none	Do not assign

Figure 2-1 illustrates the CATALOG process and the LFCs used by CATALOG.

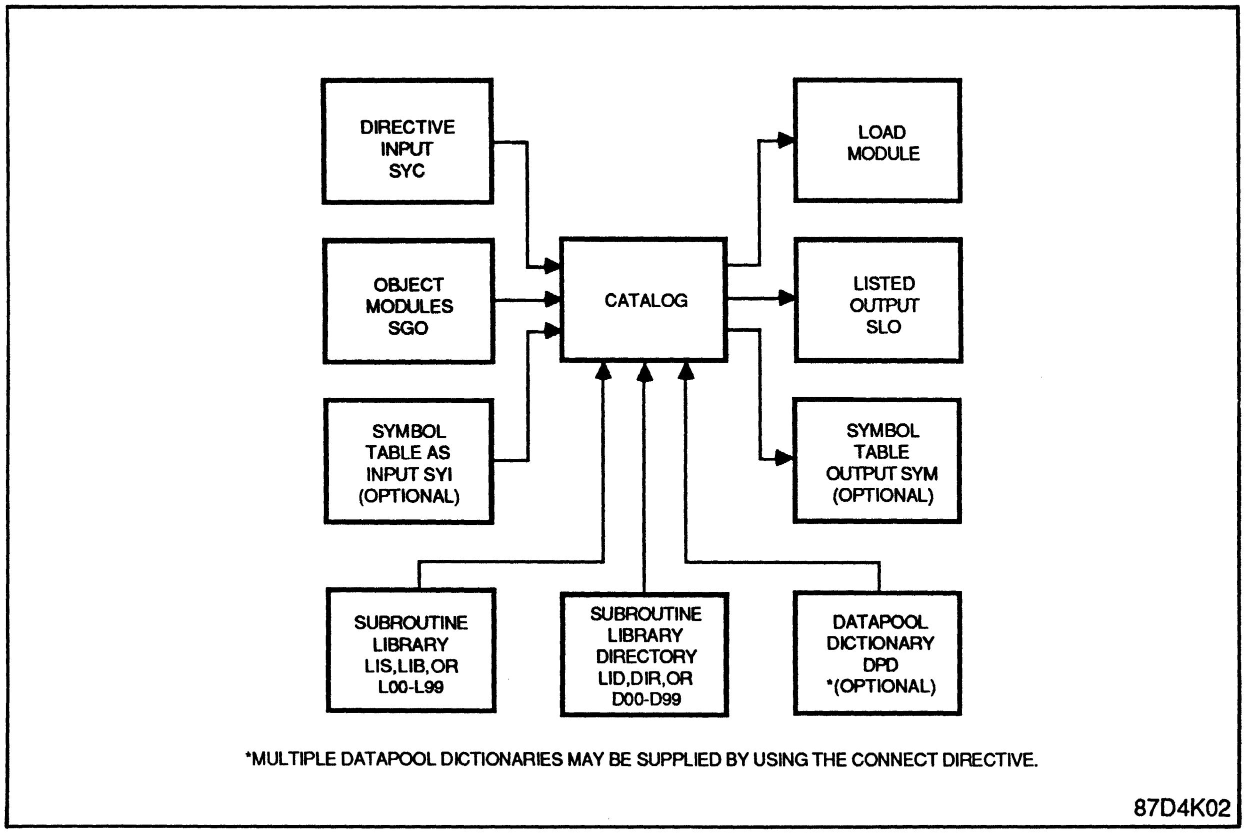

DIRECTIVE INPUT SYC
OBJECT MODULES SGO
SYMBOL TABLE AS INPUT SYI (OPTIONAL)
CATALOG
LOAD MODULE
LISTED OUTPUT SLO
SYMBOL TABLE OUTPUT SYM (OPTIONAL)
SUBROUTINE LIBRARY LIS,LIB,OR L00-L99
SUBROUTINE LIBRARY DIRECTORY LID,DIR,OR D00-D99
DATAPOOL DICTIONARY DPD *(OPTIONAL)
*MULTIPLE DATAPOOL DICTIONARIES MAY BE SUPPLIED BY USING THE CONNECT DIRECTIVE.
87D4K02

Figure 2-1. CATALOG I/O Overview

2.3 Options

Options used by CATALOG control various processing options. Options are specified by number in a $OPTION job control language statement. The $OPTION statement must appear before the $EXECUTE CATALOG statement in a jobstream.

Option	Description
1	Suppress subroutine library search - Suppresses automatic search of system and user subroutine libraries to resolve external references. All object modules to be linked must be specified in INCLUDE directives, or be contained on SGO.
2	Multiple disc files - Produces multiple disc files when cataloging overlay tasks.
3	Branch references - Enforces strict on-branch linkages for local common and global symbols. For more information, refer to Section 2.12.

15	Time, date, and program identification - Include the time and date the object code was assembled or compiled and/or program identification information as part of the load module if present in the object code. This information is included in the object code by setting the appropriate Macro Assembler or compiler options during assembly or compilation. Option 15 is not supported for overlay modules.
18	Inhibit load module generation if errors - Certain error conditions cause CATALOG to take corrective or alternate actions. There is, however, doubt as to the correctness and/or completeness of the load module. This option inhibits writing the load module in these cases. Note: The production of possibly incomplete load modules is provided as an aid to the code development cycle; the programmer can decide to use the load module or not. Production environment jobstreams should always set option 18. See Section 4 (Errors and Aborts) for a description of the conditions that cause incomplete modules.
19	Include symbolic debug information - Includes symbolic debug information which is placed at the end of the load module. Setting option 19 does not affect memory requirements but does increase disc usage. Option 19 is not supported for overlay modules.
20	Inhibits memory resident directory searches. By default, the contents of all assigned library directories are loaded into a dynamically allocated memory buffer. This buffer is expanded automatically as needed and is limited only by available physical memory and the size of the logical address space (as defined by $SPACE). Option 20 forces all directories to be searched on disc and limits CATALOG's dynamic memory buffer to approximately 32KB. Setting option 20 significantly increases CATALOG execution times.
TEXT(23)	Causes CATALOG directives read from system file SYC or a directive file to be echoed to the terminal. Directives are also written to LFC SLO.

2.4 Exiting CATALOG

To exit CATALOG in the batch or interactive modes, specify the EXIT directive.

2.5 Object Modules and Load Modules

A load module is composed of one or more object modules cataloged into executable format. A source module is the source code that produced the object module. After source code is assembled or compiled, the object modules are normally written to the System General Object (SGO) file for use by CATALOG. Object modules can also be stored in a file or incorporated into a library (by LIBED) for subsequent CATALOG access.

A nonbase mode object module produced by an assembly or compilation is identical in format to any other nonbase mode object module; therefore, source modules written in different languages may be linked into a single load module if the source languages support a compatible call/return interface.

2.5.1 Load Modules

CATALOG combines the object code from the various object and library input files into one or more load modules. These load modules are written to one or more permanent disc files. In combining the input object, CATALOG resolves global symbol references and converts the object format data into a relocatable memory image ready for loading. CATALOG also produces the runtime resource requirement summary and optionally, the program element information and the global and local debug symbol tables.

2.5.2 Absolute Load Modules

CATALOG can build an absolute load module. An absolute load module requires no relocation by the loader and reduces the task activation time.

The ABSOLUTE directive resolves all relocatable addresses relative to the base address supplied in the directive. The user is responsible for selecting a base address large enough to be beyond the task's TSA. The TSA is allocated after the end of MPX-32 and varies in size based on the number of files and buffers required in the task.

Tasks that are cataloged as absolute may require recataloging if the size of MPX-32 changes. If there is an overlap between MPX-32 or the task's TSA and the absolute task itself, the task aborts during the loading phase.

2.6 The Cataloging Process

CATALOG makes two passes over the file or device assigned to logical file code SGO and the libraries to resolve external symbolic references and include the proper object modules in the load module.

On the first pass, CATALOG searches the file or device assigned to logical file code SGO for global symbol references and definitions in the object modules. CATALOG builds a table of all references and definitions it finds.

If CATALOG cannot find a definition to match a reference in the modules in the file assigned to logical file code SGO, it searches the assigned user libraries, followed by the file assigned to logical file code LIS. Any definitions in the library that resolve references are added to the symbol table for the load module. Any new references in the library are also added.

After the first pass the symbol table contains the names of all definitions, references, and program names in the order they were found on: SGO, user libraries and the system subroutine library.

On the second pass, CATALOG retrieves an object module for the occurrence of each global symbol definition and matches the definition to its corresponding references. Object modules are retrieved from SGO and the libraries in the order of the symbol table. If CATALOG finds more than one definition with the same name, it uses the first object module that contains the definition. Duplicate definitions and unresolved references are indicated on the listed output.

The symbol table provides the communication medium between the different object modules in the load module. It is also used to resolve references when overlay load modules are cataloged in separate runs.

2.6.1 Selective Retrieval of Object Modules

When object modules are retrieved during CATALOG's first pass to resolve external references and definitions, the order of search is the SGO file, user-assigned libraries, and the file assigned to logical file code LIS. Four directives are used to manipulate the object modules retrieved: PROGRAM, PROGRAMX, INCLUDE, and EXCLUDE.

The PROGRAM directive specifies particular object modules, by program name, contained in the SGO file to be added to the load module. The PROGRAMX directive suppresses all object modules in the SGO file from the load module. If neither directive is used, all object modules from SGO are added to the load module.

Object modules in libraries that are not referenced are included in the load module by specifying them in an INCLUDE directive. The supplied name must be a global symbol defined in the object module.

Object modules in libraries can be excluded from the load module even though they are referenced by specifying them in an EXCLUDE directive. The parameter of the EXCLUDE directive is a global symbol. All global symbols defined in an object module must be explicitly excluded to assure that the object module is not added to the load module.

PROGRAM and PROGRAMX directives relate to object modules on SGO. The parameter on the PROGRAM directive line is a program element (program, subroutine, function, etc.) name.

INCLUDE and EXCLUDE directives relate to object modules in the libraries LIB, L00 - L99, and LIS. The parameter on the INCLUDE and EXCLUDE directive lines is a global symbol.

2.6.2 Allocation and Use of Global Common and Datapool Partitions

Global Common and Datapool are memory partitions defined at system generation (SYSGEN) or by the Volume Manager (VOLMGR).

Labeled common blocks are identified as Global Common by the name GLOBALnn, where nn specifies two decimal digits from 00 to 99. When CATALOG encounters a common block named GLOBALnn, space is not allocated for it in the module's area. Instead, all references to the common block are resolved using the memory partition of the same name. Therefore, the global common memory partition must be created before a program referencing it can be cataloged. If the definition of the partition changes, the programs referencing the partition must be recataloged.

Datapools are structured and resolved according to the Datapool dictionaries created with the Datapool Editor (DPEDIT) utility. Datapools are identified by the name DATAPOOL or DPOOLnn, where nn specifies two decimal digits from 00 to 99. Datapool references in an object module are resolved to locations in the specified Datapool memory partition according to the corresponding user-supplied Datapool dictionary.

There are two mechanisms available to access DATAPOOL. If the corresponding dictionary is assigned to LFC DPD, then the memory partition must be created before the task can be cataloged. If the CONNECT directive is used and the optional starting address and size parameters are specified, then the memory partition is not accessed.

The CONNECT directive allows a load module cataloged on one system (host) to be executed on another system (target). Any datapools referenced are allocated during execution and must reside on the target system.

When a global common or Datapool memory partition must be accessed, CATALOG searches for the definition in directories. The order the directories are searched is:

- With LMPATH and either BUILD or CATALOG:
  - LMPATH target volume/directory
  - Current working volume/directory
  - @ SYSTEM (SYSTEM)
- Without LMPATH and with BUILD:
  - Current working volume/directory
  - @ SYSTEM (SYSTEM)
- Without LMPATH and with CATALOG:
  - @ SYSTEM (SYSTEM)

The memory allocation unit for memory partitions is one map block (2KW). If the partitions are created by VOLMGR (dynamic), they must be allocated in map block increments. In SYSGEN created partitions (static), protection granule allocation allows multiple partitions within a map block. The allocation unit for the task remains one map block. If multiple static partitions are defined within a map block, only one partition can be included in the task's logical address space at a given time. The unused partitions in a map block are write protected.

Static partitions are defined in @ SYSTEM (SYSTEM) by MPX-32 and are automatically included in the referencing task's logical address space. Dynamic partitions must be explicitly included in the logical address space at execution time. The user must be sure that the partition included at run time matches the starting address and size values used at CATALOG time. Also, some run time included services may require that a specific volume and/or directory contain the partition definition.

2.6.3 Allocation of Local Commons

Common blocks with names other than GLOBALnn, DPOOLnn, or DATAPOOL (including BLANK) are called local common. CATALOG allocates space for local common within the load module according to references to the common contained in the object code being linked.

When the object code contains initialization data for the common block (such as a block data subprogram), storage for that common is allocated immediately before the program element containing the data. The amount of memory allocated is established as the largest size of the common block as defined in any referencing program element. If another program declares a larger size, a warning message is issued and the extra size is initialized to binary zeros.

When the object code contains no initialization data, CATALOG allocates storage immediately before the first program element that defines this common. The size of the area allocated is that of the largest definition contained in any referencing program element.

Uninitialized common that is allocated before the first program element of a load module is treated differently than commons allocated in the body of the load module. CATALOG does not allocate either memory or load module file space for these common blocks. Instead, a loading offset is supplied to the task loader and the required memory is allocated (with unpredictable contents) at task loading time. Common blocks that are allocated within the load module body allocate both memory and load module file space as required. These areas are set to binary zeros by CATALOG.

Allocating uninitialized commons in the first program element can be utilized to reduce CATALOG memory requirements and load module file size and to provide faster task activation. For more information, refer to the Local Common Allocation and Global Symbol Resolution in the Segmented Tasks section.

2.7 Load Module Information

The ENVIRONMENT and BUILD/CATALOG directives establish the following special characteristics for a task:

. Residency - A task defined as resident remains memory resident until it exits or aborts. It is not a candidate for swap to disc. The default is nonresident.

. Memory class - A task may need to execute in a special class of physical memory. E executes in class E memory, H executes in class H or E, and S executes in any class of memory available. The default is class S.

. Sharing:

  . Multicopying - A task can be active concurrently in several logical address spaces. The entire task is copied to physical memory each time it is activated.

  . Sectioned sharing - A task can be active concurrently in several logical address spaces. The CSECT area of the task is copied into physical memory once. A new DSECT area is established in physical memory each time the task is activated. DSECT areas are deallocated as sharers exit. CSECT remains allocated until all sharers exit.

  . No sharing (unique) - Only one copy of the load module can be active in one logical address space at a time. The default for a task is unique.

- Privilege - A task that accesses a privileged system service must be cataloged as privileged. A privileged task can write into any area of memory in its logical address space, including the system area, and execute the privileged instruction set. The default is unprivileged.

- Base priority - The priority the task executes at if activated as an independent task (by the TSM or OPCOM ACTIVATE, OPCOM ESTABLISH directive, another task, a timer, or an interrupt). Base priorities are in the range 1 to 64. The default is 60. If activated from TSM or in a batch stream, this priority is overridden by the SYSGEN-defined terminal or batch priority.

- Debugging - A task may prohibit attaching the debugger to it. The default is to allow debugger attachment.

Unless otherwise defined by the ENVIRONMENT directive, a task:

- is nonresident

- is unique

- is executable in any memory class available (S, H, or E)

- allows debugger attachment

Unless otherwise specified by the BUILD/CATALOG directive:

- The base priority of a task is 60

- The status of a task is unprivileged

This information is written at the beginning of the main load module by CATALOG so that it is available for the MPX-32 allocator and execution scheduler when the task is activated.

2.8 Resource Requirements

The resource requirements for a task include all files and devices used by the task:

- default assignments
- run-time assignments that override the defaults
- run-time assignments for required or optional files or devices that do not have default assignments
- dynamic assignments

A task's default resource requirements, if any, are established by CATALOG ASSIGN directives when the main load module is cataloged. Required, optional, or overriding run-time resources are established by TSM $ASSIGN directives when the task is activated.

Dynamic assignment of files or devices is made by the task through MPX-32 service calls, the FORTRAN OPEN statement, or subroutine calls.

A prerequisite for blocked I/O used by a task is a blocking buffer, which the allocator establishes in the Task Service Area (TSA). This can be controlled with the BUFFERS directive. Files on disc and magnetic tape assume the system default for blocking unless otherwise specified by an ASSIGN directive or a dynamic service call. Files also require FPT/FAT table entries in the TSA. This can be controlled with the FILES and SEGFILES directives.

CATALOG preserves resource information on the default files and devices used by a task, including the number of blocking buffers and table entries required. At activation, run-time assigned files and devices are allocated as specified and override default file and device assignments. The appropriate memory is then allocated for table space and buffers. However, if files and/or devices are allocated dynamically by the task, the number of additional file table entries and buffers required must be indicated.

Cataloger FILES, SEGFILES, and BUFFERS directives account for dynamic assignments. The FILES directive specifies the number of files and devices allocated dynamically (and thus the number of table entries to leave room for). The SEGFILES directive specifies the number of noncontiguous disc files allocated dynamically. The BUFFERS directive specifies the number of blocking buffers required for blocked files or devices accessed dynamically.

Resource requirements for shared tasks require special treatment because several concurrent sharers of the task can use different run-time assignments that require different allocation of blocking buffers and file space. FILES, SEGFILES, and BUFFERS directives for cataloging shared tasks must reflect the maximum number of files and devices that can be assigned: default (or override), required, optional, and dynamic. This information is required by CATALOG to ensure that the TSA for each sharer is the same size and that the DSECT section of the shared task begins at the same location in each sharer's logical address space.

2.9 Sectioned and Nonsectioned Tasks

CATALOG supports both sectioned and nonsectioned tasks.

Nonsectioned tasks are allocated in a logically contiguous area immediately above the TSA. In effect, they are structured as one large DSECT. Nonsectioned tasks can be cataloged as multicopied or unique. Multicopied tasks are copied into physical memory to support multiple concurrent activations. A nonsectioned task that is cataloged as unique allows only one activation at a time. If not specified, a nonsectioned task defaults to unique.

Sectioned tasks are created when CSECT/DSECT definitions are contained in the object code. CSECT defines a pure code and constant data section of a task; DSECT defines an impure, user-dependent, variable data section. CATALOG merges all CSECTs into a write protected allocation in upper memory and all DSECTs in lower memory just above the task's TSA. Sectioned tasks can take advantage of CSECT/DSECT sectioning to write protect pure code and data, but the primary purpose of CSECT/DSECT is to support sharing.

A sectioned task can be cataloged as shared, multicopied, or unique. If a sectioned task is cataloged as shared, the CSECT of the task is copied into memory once and only the DSECT is recopied with subsequent activations.

The minimum allocation for a CSECT area is a map block (2KW); DSECT is allocated in a separate map block along with the TSA. The minimum space used for the task's DSECT is one map block, including the TSA size. If a task is less than a map block, multicopying and nonsectioning may allow more efficient use of memory than using sectioning.

2.10 Segmented and Nonsegmented Tasks

Two types of load modules can be part of one task: one main load module and one or more overlay modules required to satisfy references for the task. A task that contains a main load module and one or more overlays is segmented. A task that contains only a main load module is nonsegmented.

Each load module is constructed by a separate BUILD/CATALOG directive. The main and overlay modules can reside on the same disc file or on multiple disc files. Overlay load modules are loaded and/or executed by system service calls within the programs.

A nonsegmented task can reference overlays built in separate cataloging sessions. When a nonsegmented task references such overlays, the main module and all overlay modules are in memory when the task is executing.

Overlays provide a way to segment tasks for more efficient memory utilization. When it is impractical to have a large task in its entirety in memory, it can be divided into a main load module and one or more overlay load modules. A segmented task is activated by using the name of the file containing the main load module.

In a segmented task, only the main module and modules concurrently referenced in the task are in memory at the same time. When modules other than the main module are no longer needed by the task, they are replaced, or overlaid, by other referenced modules.

CATALOG supports two types of overlay load modules and several overlaying strategies. The user may choose the type and strategy that best suits the requirements of a particular application. The two types of overlay load modules are characterized by the method of accessing the overlay. Overlay load modules that contain a transfer address may be loaded and executed by a single service call. Upon completion, control is returned to the calling load module. This overlay is referred to as a single point of call overlay and is used when a particular portion of the application can be achieved by one or more program elements executing off a single call. This type of structure contains no cross module subroutine references and is more flexible with regard to cataloging in stages or recataloging. A drawback is that the passing of parameters must be explicitly handled by the programs.

The second overlay structure is constructed by grouping related subroutines in an overlay load module. The load module is invoked by making the service call to load that overlay. The caller can then reference the various subroutines directly and independently. This type of structure is referred to as the independent subroutine type of load module and is less flexible with regard to cataloging in stages or recataloging, but allows the user to utilize any mechanism for parameter passing defined in the implementation language.

2.11 Overlay Load Modules

The following sections describe the use and structure of overlay load modules.

2.11.1 Single and Multiple Disc File Modes

CATALOG produces overlays in two modes: single disc files and multiple disc files.

In single disc file mode, the root and overlay load modules are produced in a single disc file. Single disc file mode supports a maximum of 75 overlays.

Individual overlays in a single disc file can be cataloged in stages or without recataloging the entire task by using the RECATALOG directive.

In multiple disc file mode, CATALOG produces separate files for the main load module and each overlay. This mode is indicated by setting option two. The overlay load modules in multiple disc file mode can be built in any directory but can only be executed from the system directory.

In multiple file mode, individual overlay load modules can be built in stages or recataloged by providing only the directives for the overlays involved to CATALOG. In this mode, the LMPATH directive may not supply the filename.

Multiple disc file mode supports more than 75 overlays; for less than 75 overlays, it is recommended that single disc file mode be used.

Symbolic debugger information is not available for overlays even if option 19 is set at catalog time. Time, date, and program identification information is not available for overlays even if option 15 is set at catalog time.

2.11.2 Overlay Levels

Single point of call and independent subroutine overlay load modules can be organized into levels. An overlay level consists of one or more overlay load modules that do not reference each other internally and can be loaded into the same logical memory locations within the task.

Low level overlays usually represent the overlays a main load module calls in after it is loaded. Higher level overlays which follow are associated with the root and/or one or more of the lower level overlays.

The simplest overlay structure consists of a single overlay level as illustrated in Figure 2-2. As each overlay is accessed by a system service call, it replaces the previous overlay in memory.

Figure 2-3 illustrates the logical structure of a task with a number of overlays and overlay levels. This task consists of a main load module and seven overlay load modules. The overlay load modules are grouped into levels A and B. Level A overlays are low level; level B overlays are higher level.

A maximum of 255 overlay levels are supported. The root is always level 0. A maximum of 32,768 overlays are supported at each level above level 0.

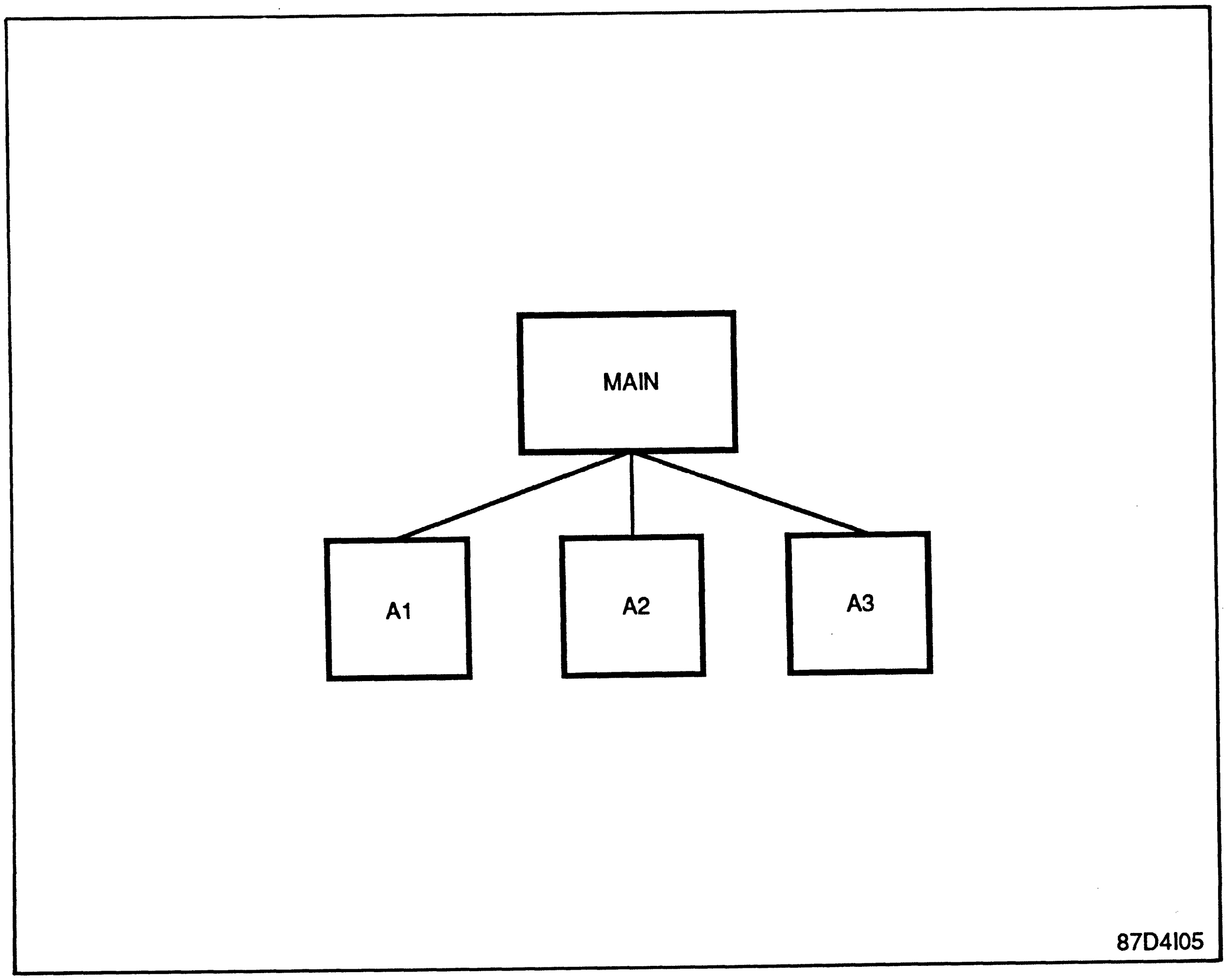


Figure 2-2. Single Overlay Structure

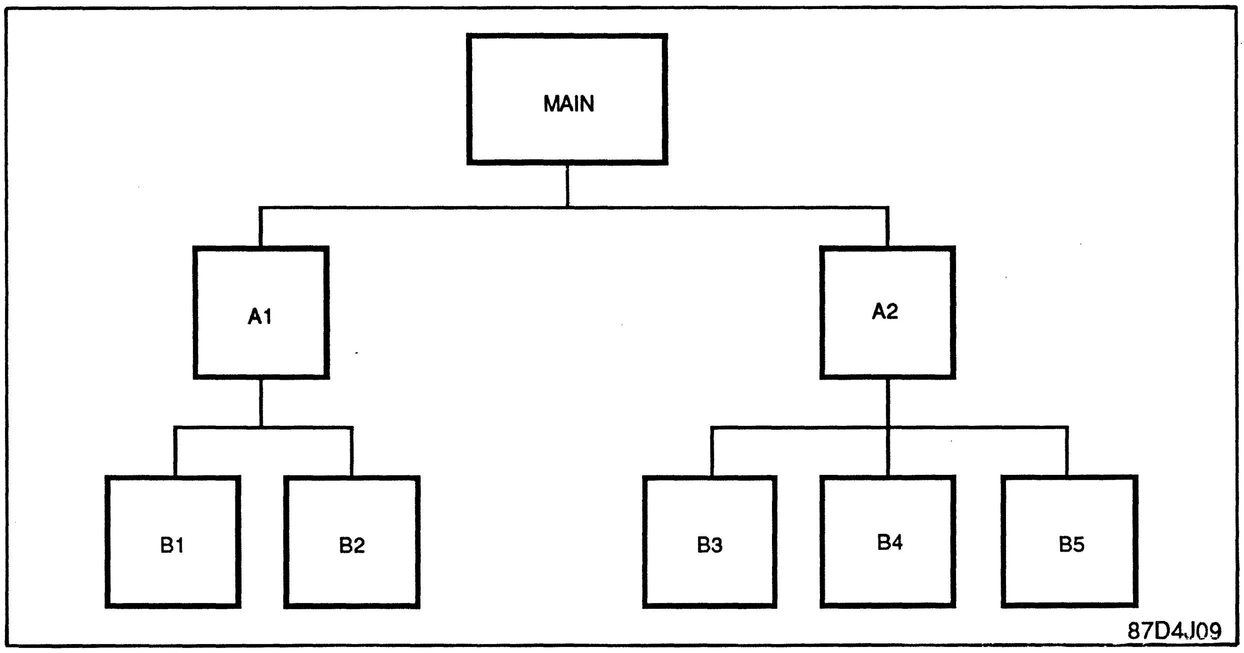


Figure 2-3. Multilevel Overlay Structure

Figure 2-4 illustrates the default memory allocation for the main and overlay load modules shown in Figure 2-3. Example 3 in Section 5 shows directives that would achieve this structure.

Level one is automatically established by the processing of the second BUILD/CATALOG directive. All subsequent load modules are at level one until an LORIGIN directive is processed. Each time an LORIGIN directive is processed, the level is increased by one.

The allocation of memory (overlays above the root) depicted in Figure 2-4 is valid only if the TRA= parameter of the BUILD/CATALOG directive has not been specified. The TRA parameter causes CATALOG to allocate the overlay transient area below the root. This is useful when the application performs dynamic memory allocation during execution.

Using this default memory allocation, any second level overlay (B) can be in memory with any first level overlay. The second level overlay can operate on behalf of the root or any first level overlay at any time. With independent subroutine load modules, the calling program must ensure that all overlays at any level that contains the definitions of any global symbol referenced, are actually in memory when that symbol is referenced.

2.11.3 Modifying Overlay Origins

The ORIGIN or LORIGIN directives modify the memory allocation for the overlay structure. For example, a different origin can be set for higher level overlays associated with A2 (B3, B4, and B5) so that space not being used when A2 is in memory can be used. The total program memory requirements are reduced. Figure 2-5 illustrates how the overlay transient area is modified. Example 8 in Section 5 demonstrates these directives.

Overriding the default memory allocation means that B1 and B2 may be loaded with either A1 or A2, but B3, B4 and B5 may be loaded only with A2 (see Figure 2-5).

If the higher level overlays are intended to operate on behalf of a particular lower level overlay, the user's code must ensure that the correct lower level overlays are loaded.

If the higher level overlays are intended to operate on behalf of the root, any overlay may be loaded at any level without concern for other levels. However, if B3 through B5 are loaded with A1 in memory, A1 must be reloaded before it can be used.

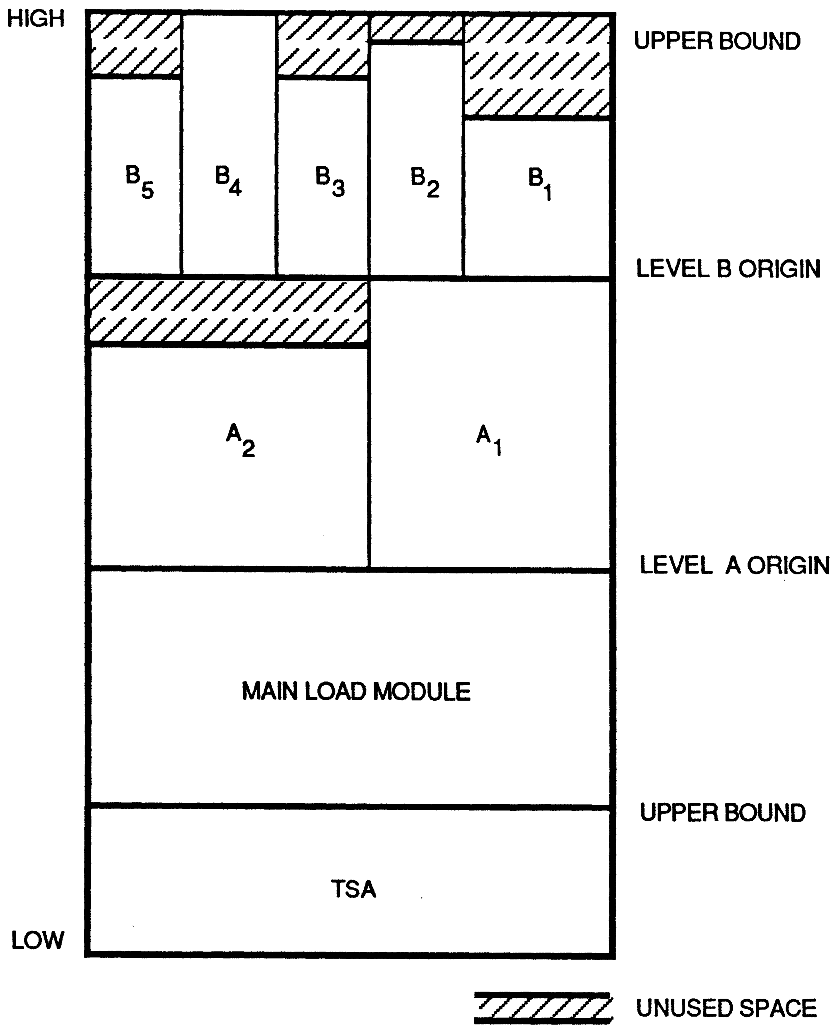


Figure 2-4. Default Memory Allocation for Overlays

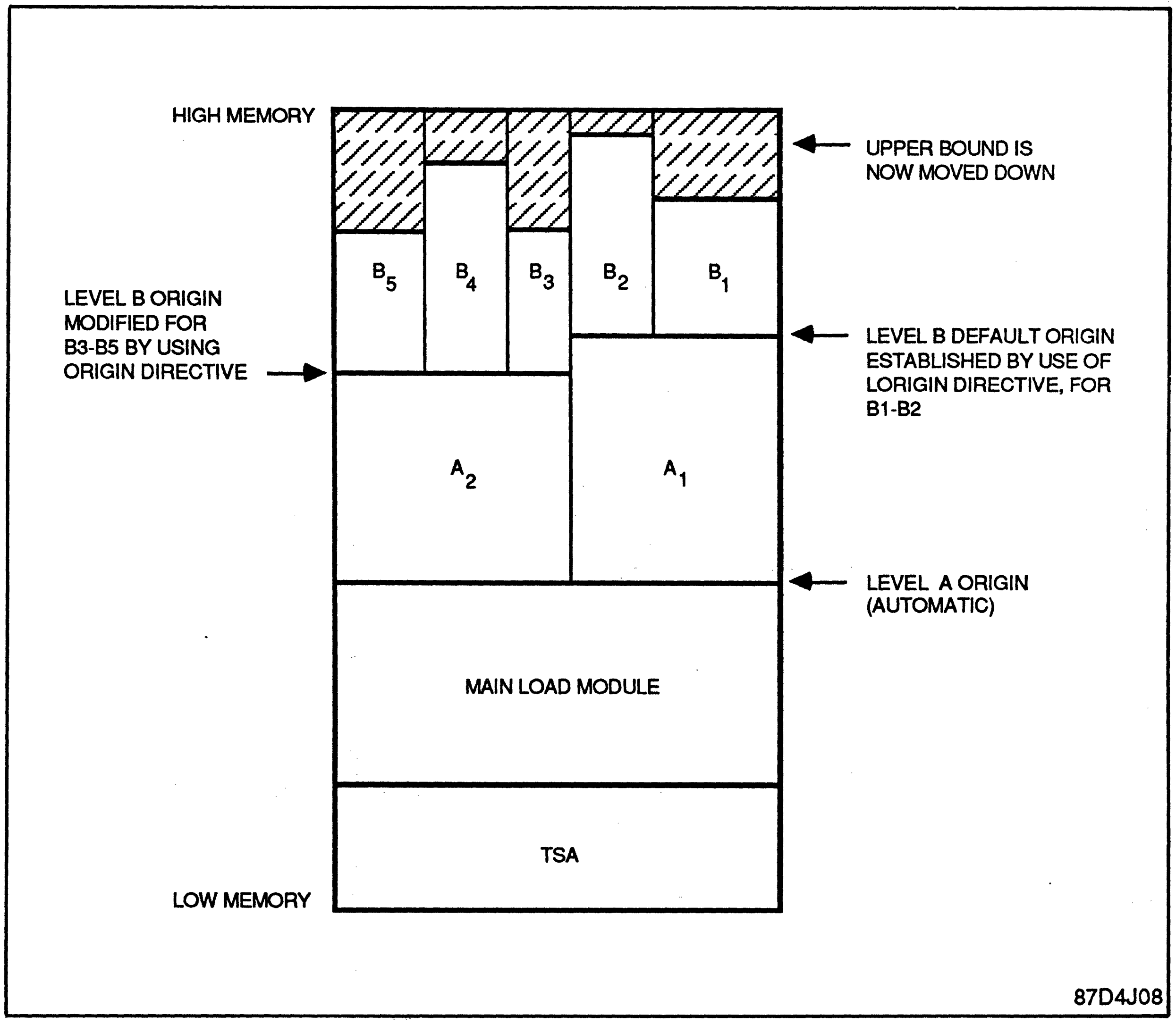

HIGH MEMORY
UPPER BOUND IS NOW MOVED DOWN
B5
B4
B3
B2
B1
LEVEL B ORIGIN MODIFIED FOR B3-B5 BY USING ORIGIN DIRECTIVE
LEVEL B DEFAULT ORIGIN ESTABLISHED BY USE OF LORIGIN DIRECTIVE, FOR B1-B2
A2
A1
LEVEL A ORIGIN (AUTOMATIC)
MAIN LOAD MODULE
TSA
LOW MEMORY
87D4J08

Figure 2-5. Modified Memory Allocation for Overlays

2.11.4 The Overlay Transient Area

By default, CATALOG establishes an overlay transient area above the root (logically higher addresses) that is of a sufficient size. In applications, where dynamic memory allocation above the root is required, the overlays can be directed to load in low memory below the root. This is accomplished by specifying a transient area using the TRA= parameter on the BUILD/CATALOG directive for the root segment. This relocates the root higher in memory by the amount specified. It is the user's responsibility to supply a value large enough to accommodate the overlays.

2.12 Local Common Allocation and Global Symbol Resolution in Segmented Tasks

In segmented tasks comprised of several load modules grouped into several levels, the resolution of common and global symbol references is complicated and can lead to unpredictable results and/or unresolvable situations unless given due consideration.

CATALOG provides options and directives to control the resolution of these references. The user can select an overlay strategy that best suits the requirements of the application.

2.12.1 Local Common Allocation

An overlay load module is essentially, the same as a non-segmented load module. Therefore, the rules in the Allocation of Local Commons section apply to all intra-load module commons. The following discussion applies to inter-load module commons.

A common is said to be "defined" in any program element that references a datum declared in that common. When CATALOG allocates the memory that holds the data declared in a common within a load module, the common is said to be "allocated" in that load module. All definitions are "linked" to the allocated location.

CATALOG Option 3 and the LINKBACK directive allow the user to control the allocation of and references to local common. The use and effects of Option 3 and the LINKBACK directive are described below.

Local commons defined in the root segment are allocated in the root segment. All definitions in high level overlays are linked to the root segment allocation. This ensures that all higher level overlays can communicate through root allocated commons regardless of the transient area contents.

When a local common definition occurs only in higher level overlays, it is allocated in the first, lowest level, overlay that defines it. When a common is defined in more than one load module at the same level, it will be allocated in each defining module.

Usually, this means that the data declared in such a common will be "common" only to the program elements of each load module. (The area cannot be used to communicate between load modules at the same level). However, if all the load modules have the same origin, the common is allocated in the first program element of each overlay and this common is not initialized in any of the overlays, the data contained in the common remains intact from one overlay to the next. This is because uninitialized common at the beginning of a load module contains no space in the load module file and remains unchanged by the loader. This form of cross module common allows the common to be used for inter-module communication.

When a local common is allocated in several overlay load modules at a given level, definitions occurring in higher level overlays are linked to the low level allocations in several different ways depending on Option 3 and the LINKBACK directive.

With Option 3 Reset (the default), local common definitions occurring in higher level overlays are linked to the allocation in the last lower level overlay processed by CATALOG. This occurs regardless of the LINKBACK directive. If the conditions described above for cross module commons are met, then the lower level allocation is at the same place in memory for all lower level modules. It is unaffected by loading activity and can be successfully used in any higher level overlay. In all other cases, Option 3 must be used.

When Option 3 is set, local commons are reallocated in higher level defining overlays unless the common is already allocated in a lower level overlay to which the current load module is linked with the LINKBACK directive. Initialized commons follow the same rules as uninitialized commons with the following additional requirements:

- Cross module common at the same level is unavailable to initialized common. Each load module that initializes a common area resets the area to its initial values as it is loaded.

- The program element that contains the initialization code must be part of the lowest level defining overlay whenever multi-level linkages occur.

2.12.2 Global Symbol Resolution

The following describes the rules for subroutine linkage in overlay environments.

Each overlay load module is built as a complete unit. This means that all external symbol references are resolved by including program elements which contain satisfying definitions found in SGO or any available library in the load module.

To build an overlay structure, it is necessary to indicate to CATALOG that specific references should remain unsatisfied (temporarily) in a load module. This can be accomplished in several ways. By default, all object modules on SGO are processed at the first BUILD/CATALOG directive. By using the PROGRAM directive only specifically named programs are processed from SGO for any particular BUILD/CATALOG. The PROGRAMX directive inhibits all processing of SGO. Further, as programs are processed, all references to external symbols are retained and all assigned libraries are searched for matching definitions. By supplying the global symbol name in an EXCLUDE directive, CATALOG will not load a program containing a matching definition. Alternatively, Option 1 can be set and all global symbol definitions required are then indicated on INCLUDE directives.

Similarly, programs which contain global symbol definitions that are not otherwise referenced can be forced into any particular load module by specifying the symbol name in an INCLUDE directive.

The user explicitly indicates the contents of each load module by using the following:

- The PROGRAM and PROGRAMX directives to control the processing of SGO.

- OPTION 1 and the INCLUDE directive or the INCLUDE/EXCLUDE directives to control processing of the libraries (in conjunction with each BUILD/CATALOG directive).

Once the contents of each overlay is established, CATALOG resolves cross module linkages of global symbols (if any exist) according to the following rules. Option 3 and the LINKBACK directive control the resolution.

Symbols excluded from a particular load module are assumed to be defined in a higher level overlay. CATALOG provides automatic forward linkage to higher level overlays in two ways depending on Option 3. However, to satisfy a symbol reference to lower level overlays, the load module must be explicitly linked to the lower level using the LINKBACK directive.

Global symbols defined in the root segment are available to all higher level overlays and are used first to satisfy references in any higher level (i.e there is an implicit linkback to the root provided to all higher level overlays).

References in modules at levels above the root are satisfied first by definitions in the root. If the symbol is not defined in the root, the first definition in lower level overlays to which the current module is linked, in the order of the LINKBACK directives, is used. If the symbol is not defined in any linked lower level, higher levels are used.

When Option 3 is reset (the default) a definition in any higher level module will be used. The search is performed in the order of the CATALOG/BUILD directives, with the first definition found being used. When Option 3 is set, only higher level which are linked (with the LINKBACK directive) to the module containing the reference are used. The first definition found is linked.

In all cases it is the responsibility of the calling module to ensure that the correct overlay is actually in memory.

2.13 Cataloging a Segmented Task in Stages

A segmented task may be cataloged in one operation or in stages. The main load module can be cataloged in one session, with or without overlay load modules. Overlay modules can be cataloged in subsequent sessions. If the transient area size option (TRA=) is not declared for the main load module in the BUILD or CATALOG directive, CATALOG reserves a transient area large enough to accommodate any overlay modules cataloged in the same run as the main load module. If overlay modules cataloged separately from the main load module require more space, an adequate transient area size must be specified when the main load module is cataloged.

When cataloging in stages, the main load module can be cataloged without its overlays only when the single point of call (load and execute) methodology is used. If the main load module contains references to external symbols that are defined in the overlays, these overlays must be cataloged in the same run as the main.

The symbol table (SYMTAB) resolves external references when load modules are cataloged in separate stages. The SYMTAB contains the definitions of all common blocks and all DEFs from the previous cataloging session. All references must be resolved when the SYMTAB is built.

The SYMTAB is saved by assigning a file or device to logical file code SYM and specifying the SYM option on the BUILD/CATALOG directive for the main load module. SYMTABS are restored by assigning the same file or device (used with SYM) to logical file code SYI and using the SYMTAB directive before the first BUILD/CATALOG directive of a subsequent run.

Common blocks defined in cataloged load modules are not reallocated when new load modules are cataloged. Common block sizes are not expanded as a result of definitions contained in new load modules being cataloged.

References to global common and Datapool are not affected because these areas are allocated in a separate area of memory from the task.

2.13.1 Recataloging a Load Module

When operating in single file mode (option 2 reset), the RECATALOG directive must be used to specify the recataloging of one or more of the overlay load modules contained in the file.

When a load module is recataloged, the new version is written over the existing version. The disc file is automatically expanded, if needed, to accommodate the new version. Other load modules in the file are copied to the new file.

2.13.2 Limitations on Cataloging in Stages

Care is required in recataloging some load modules. Load modules whose sizes increase may result in allocations that overlap the address spaces of load modules that are not being recataloged. In addition, resolution of references to external symbols and common blocks within the task can be affected.

Overlap can be detected by examining the addresses of each load module, which are printed in the module's map. Overlap is indicated when an overlay's end address is greater than the beginning address of a higher level overlay, or is greater than the beginning address of the main load module (with TRA parameter).

Changing the size of the transient area with the TRA parameter changes the location of the main module in relation to the overlay modules. If the size of the transient area is changed, all previously cataloged overlay modules that reference the main load module must be recataloged.

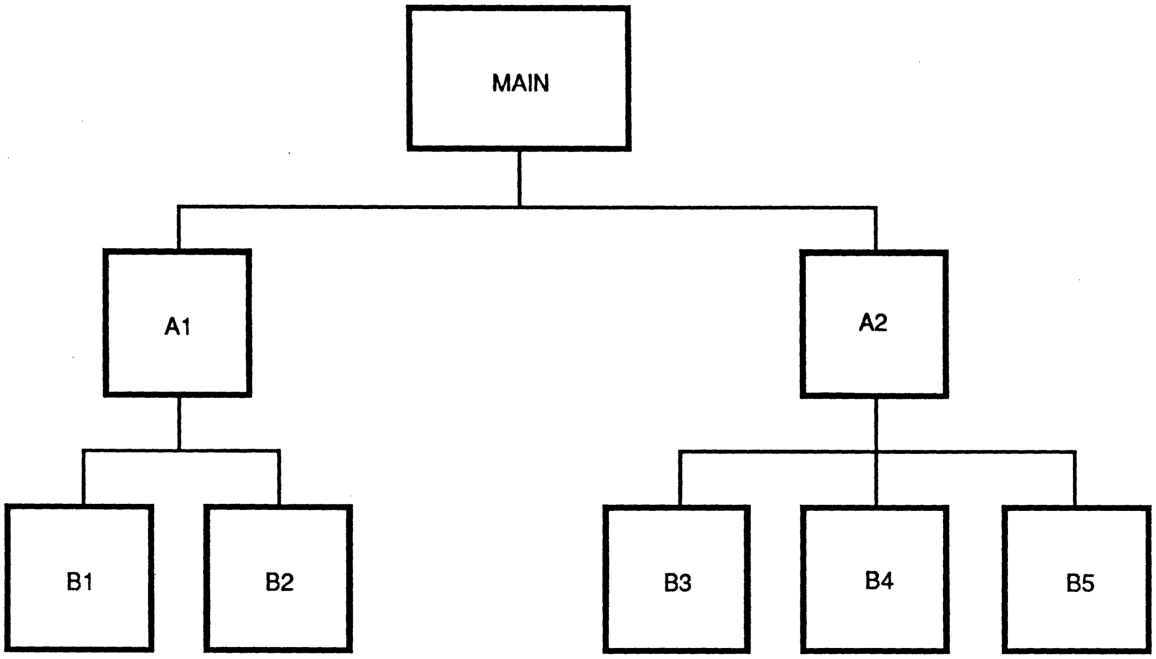
When a load module is recataloged, the resolution of addresses for global symbols and common blocks defined within the task may also change. As a result, references to the global symbols or common blocks by other load modules are incorrect unless they are recataloged. Assume intermodule referencing for the task as illustrated in Figure 2-6.

In the table at the bottom of Figure 2-6, if any load module(s) are recataloged, all other load modules which correspond to Xs in the vertical column beneath the load module must also be recataloged. For example, if the main load module is recataloged, A1 and A2 must be recataloged. If A1 and A2 are recataloged, all load modules must be recataloged.

As a general rule, partial catalogs (with option 2 or RECATALOG in single file mode) are only practical when the load modules are executed in the single point of call load and execute mode. When the overlays consist of collections of independently called subroutines, a change in size of any subroutine will invalidate all linkages to all subroutines above the one changed in the load module.

2.14 Cataloging a Nonsegmented Task

Cataloging a nonsegmented task is similar to cataloging the main load module of a segmented task.



LOAD MODULE REFERENCED

	MAIN	A1	A2	B1	B2	B3	B4	B5
MAIN		X	X					
A1	X			X	X			
A2	X					X	X	X
B1		X						
B2		X						
B3			X					
B4			X					
B5			X					

87D4I09

Figure 2-6. Recataloging Illustration

SECTION 3 - DIRECTIVES

3.1 Introduction

CATALOG directives are summarized in the Overview section and described in detail in this section.

All CATALOG directives begin in column one. Most directives can be abbreviated to four characters. Valid abbreviations are indicated by underlining.

Legal delimiters between directive parameters are commas or blanks. Commas need to be used only where shown.

3.2 Directive Order Requirements

The following directives can appear as needed in any order after the $EXECUTE CATALOG directive and before the first BUILD/CATALOG directive. They cannot be used after the first BUILD/CATALOG directive.

ABSOLUTE
ALLOCATE
ASSIGN
ASSIGN1
ASSIGN2
ASSIGN3
ASSIGN4
BUFFERS
CONNECT
ENVIRONMENT
EXTDMPX
FILES
LMPATH
MOUNT
OPTION
RECATALOG
SEGFILES
SPACE
SYMTAB
VOLUMES

Note: When CONNECT directives require location of a Datapool partition definition, the LMPATH target volume/directory will be searched only if the LMPATH directive precedes the CONNECT directives.

These directives supply parameter values and static resource requirements to the task being cataloged. Many of these directives are similar in syntax and function to TSM directives. Directives such as ASSIGN, ALLOCATE, OPTION, etc. entered before the $EXECUTE CATALOG directive affect the execution of the CATALOG task. Directives entered after the $EXECUTE CATALOG directive affect the user task being built.

The following directives appear as required in the order shown after the BUILD/CATALOG directives:

EXCLUDE
INCLUDE
$\left\{\begin{array}{l}\text{PROGRAM}\\\text{PROGRAMX}\end{array}\right\}$

When cataloging overlay load modules, the following directives appear as required in the order shown for each overlay and following the directives for the root:

```
{LORIGIN }
{ORIGIN  }
 BUILD/CATALOG load module 0
 LINKBACK
 EXCLUDE
 INCLUDE
{PROGRAM }
{PROGRAMX}
```

The directive stream is terminated by:

```
EXIT
```

A directive line which contains an asterisk (*) in column one is treated by CATALOG as a comment. Comment lines may appear anywhere between the $EXECUTE CATALOG directive and the EXIT directive. See Section 5 for examples.

3.3 ABSOLUTE Directive

The ABSOLUTE directive builds an absolute load module. An absolute load module is one that requires no relocation by MPX-32 at load time. The base address specified must be higher than MPX-32 and the TSA. If the base address creates an overlap between the task and MPX-32 or the task's TSA, the task will not load. When the task is loaded at the specified address, memory between the end of the TSA and the start of the task is allocated to the task and is available for use by the task.

The CSECT origin is not affected by this directive. The transient area option on the CATALOG and BUILD directives (TRA=) has no effect when the ABSOLUTE directive is used. Multiple ABSOLUTE directives are not allowed.

Syntax:

```
ABSOLUTE [base]
```

base is a hexadecimal logical address specifying the base address of the task. This address is rounded up to the nearest 512 word boundary. If no base is supplied, the default is a value of $40000_{(16)}$.

3.4 ALLOCATE Directive

The ALLOCATE directive increases the memory allocation for a task at execution time.

If the ALLOCATE directive is used when cataloging a task, additional static memory is allocated every time the task is run. The allocation cannot be reduced at run time or by dynamic service calls.

Syntax:

```
     ALLOCATE bytes
```

bytes specifies the hexadecimal number of additional bytes to allocate to the task

3.5 ASSIGN Directive

The ASSIGN directive supplies default assignments for logical file codes used by the task being cataloged. Assignments for a task must be cataloged with the main load module.

Syntax:

```
                       ( SBO                    )
                       ( SLO                    )
                       ( SYC                    )
                       ( SGO                    )
                       { @ANSITAPE(lvid)file    }
                       { pathname               }
                       ( RID=resid              )
     ASSIGN lfc TO     ( TEMP[= (volname)]      )   [FORMAT=format] [SIZE=blocks]
                       ( DEV=devmnc             )
                       ( LFC=lfc                )

   [SHARED= bool] [GENERATION=gennum] [GENVERSION=genvum] [BSIZE=bsize]
   [RECLENGTH=recsize] [ACCESS=( [READ] [WRITE] [MODIFY] [UPDATE] [APPEND] ) ]
   [BLOCKED= bool]
                                                  [           ( N    ) ]
                                                  [           ( P    ) ]
   [ EXPIRE =  { date  } ]   [ PRINT ]            [           { G    } ]  [              { 0    } ]
   [           { +days } ]   [ PUNCH ]            [ DENSITY=  { 800  } ]  [ PROTECT =    { A...Z} ]
                                                  [           ( 1600 ) ]
                                                  [           ( 6250 ) ]

   [MULTIVOL=number] [ID=id] [BBUF=buffers]
```

SBO	treat resource as System Binary Output
SLO	treat resource as System Listed Output
SYC	treat resource as a System Control file
SGO	treat resource as a System General Object file
@ANSITAPE	treat resource as an ANSI labeled tape
lvid	is the one- to six-character logical volume identifier previously mounted by the ANSI labeled tape AMOUNT utility
file	is a one- to seventeen-character file identifier
pathname	is the pathname to be associated with the resource
resid	is a unique resource identifier (including the volume name, creation date, creation time, resource descriptor block, resource type, and code) returned by the system when a resource is created
volname	is the volume name on which temporary space is to be allocated. If not specified, the default is any volume.

devmnc	is the device mnemonic of a configured peripheral device. See Appendix A.
lfc	is a one- to three-character logical file code used in the task. For an ANSI labeled tape, only one LFC can be assigned to an lvid. Before further assignments can be made, the M.DASN service must be used.
format	is the ANSI labeled tape record format. If not specified, the default for write access is D. For read access, the format is read from the tape. The formats are:

Format	Description
F	Fixed length
D	Variable length
S	Spanned

blocks	specifies the initial size, not greater than 65,535 blocks, of a file in logical blocks. If not specified, the default is 16 blocks. If EOM is encountered, the file extends automatically. This option is only valid when used with the TEMP parameter.
<u>SHAR</u>ED	if yes (Y) is specified, the resource is explicitly shared. If no (N) is specified, the resource is exclusive. If not specified, the default is implicitly shared. This option is only valid when used with the pathname, RID, TEMP, and DEV parameters.
gennum	is the one- to four-decimal digit ANSI labeled tape file generation number. On input (read access), this number must match the generation number of the ANSI tape file that is being assigned. On output (write, update, or append access), this value becomes the generation number of the new ANSI tape file. If not specified, the default is one on output; no check on input.
genvum	is the one- or two-decimal digit ANSI labeled tape file generation version number. On input (read access), this value must match that of the ANSI tape file. On output (write, update or append access), this value becomes the generation version number of the new ANSI tape file. If not specified, the default is zero on output; no check on input.
bsize	is read from the ANSI labeled tape on read access. For other types of access, the value specifies the byte size of each data block including the padding on an ANSI labeled tape. A maximum bsize of 2048 provides sufficient space for ANSI tape-switch label information after the physical end-of-tape marker. If not specified, the default is 2048 bytes.
recsize	is read from the ANSI labeled tape header on read access. For other types of access, this value specifies the record size for fixed length records or the maximum record size for spanned and variable length record formats. The maximum size for recsize is bsize. If not specified, the default is 80.
<u>ACCE</u>SS	specifies the type of access for resource. This must be a subset of access allowed at resource creation. If not specified, the default is the access specified at resource creation. This option is only valid when used with the @ANSITAPE, pathname, RID, TEMP, and DEV parameters.

For ANSI tapes, only read, write, update and append can be specified. The ANSI default is read. ACCESS for ANSI labeled tapes, is as follows:

Value	Description
R	Read existing file
W	Create file at first unexpired file on tape
A	Create file at end of tape
U	Overwrite existing file with a new file of the same name

BLOCKED	if yes (Y) is specified, the resource is explicitly blocked. If no (N) is specified, the resource is explicitly unblocked. If not specified, the default is blocked. This option is only valid when used with the @ANSITAPE, pathname, RID, TEMP, and DEV parameters.
EXPIRE	specifies the termination date of an ANSI labeled tape file. If the file has a termination date that is later than the file that physically precedes it, the termination date is identical to the termination date of the preceding file. If a file has a termination date that is earlier than the file that physically precedes it, the files will expire on the earlier termination date. If not specified, the default is +30 days from creation.
date	specifies the date after which an ANSI labeled tape file can be overwritten. The date is given in ASCII format--YYDDD where YY is the year and DDD is the day number within the year (January 1 is 001). If the date is 00000, or a date prior to the current date, the file has been terminated and is no longer accessible.
+days	specifies the number of days after the creation date that an ANSI tape file can be overwritten. This number must be preceded with a plus (+) when entered. If not specified, default is +30 days. WARNING: If the number of days is not preceded by a plus (+), the number entered can be read as the date.
PRINT	indicates the file is to be printed after deassignment. This option is only valid when used with the pathname, RID, and TEMP parameters.
PUNCH	indicates the file is to be punched after deassignment. This option is only valid when used with the pathname, RID, and TEMP parameters.
DENSITY	specifies density of high speed XIO tape. If not specified, the default is 6250 BPI. This option is only valid when used with the DEV parameter.
PROTECT	specifies protection for new ANSI labeled tape files. Zero specifies owner only access. A...Z are reserved by the ANSI specification for installation-specific protection. MPX-32 treats A..Z as owner-only protection. If the correct protection value is not specified when using an ANSI labeled tape, an I/O error occurs. If a user signs on as 'system', any protection value or owner name written by J.LABEL can be overridden. If not specified, the default is no protection.

MULTIVOL	is a volume number for a multivolume tape. If not specified, the default is zero (not multivolume). This option is only valid when used with the DEV parameter.
ID	is an identifier for an unformatted medium. If not specified, the default is SCRA (scratch). This option is only valid when used with the DEV parameter.
buffers	is the number of 192W blocking buffers if using a large blocking buffer. If not specified, the default is one. This applies only to permanent disk files.

Usage:

```
ASSIGN SYM TO DEV=M9 DENSITY=800 BLOC=Y

ASSIGN SGO TO OUTFILE

ASSIGN IN TO MYFILE BBUF=10
```

Notes:

1. To continue parameters over more than one input line, a hyphen (-) must terminate the current input line. A blank space is required before the hyphen as shown in the following example:

   ```
   ASSIGN ABC TO DEV=M9 DENSITY=800 -
   BLOCKED=Y
   ```

2. An individual parameter cannot be split between input lines.

3.6 ASSIGN1 Directive

The ASSIGN1 directive supplies default file assignments for logical file codes used by the task being cataloged. This directive is for compatibility with MPX-32 1.x. Its use is not recommended.

Syntax:

$$\underline{\text{A}}\text{SSIGN}\underline{\text{1}}\ \text{lfc=filename}\ \left[\begin{array}{ll}\text{,password} & \\ \text{,password,U} & [\text{lfc=...}] \\ \text{,,U} & \end{array}\right]$$

lfc	is a logical file code used in the task to denote a generic input or output source
filename	is the name of a permanent disc file to assign to the LFC Any one of the optional parameters following the file name may be entered in the order shown in the syntax statement. Commas separate options. If an option is omitted, the comma must be supplied: filename,,U
password	is ignored
U	indicates the file is unblocked. If not specified, the default is blocked.

Usage:

```
ASSIGN1 LIB=LIBRARY,,U DIR=DIRECTORY,,U

ASSIGN1 OT=OUTFILE IN=INFILE,MYPASS
```

3.7 ASSIGN2 Directive

The ASSIGN2 directive supplies default system file assignments for logical file codes. This directive is for compatibility with MPX-32 1.x. Its use is not recommended. At run time, an LFC assignment to a system file results in the creation of one of the following types of files for use by the task:

SBO	System Binary Output - A temporary file for buffering output to the device defined at SYSGEN or by the OPCOM SYSASSIGN directive as POD (Punched Output Device).
SLO	System Listed Output - A temporary file for buffering output to the device defined at SYSGEN or by the OPCOM SYSASSIGN directive as LOD (Listed Output Device).
SYC	System Control - A temporary system file associated only with jobs processed in the batch mode (one SYC per job). SYC is used for buffering input from the device defined at SYSGEN or by the OPCOM SYSASSIGN directive as SID (System Input Device). Tasks not designed to run only in the batchstream should not make assignments to SYC. Batch tasks can use SYC to input data records.
SGO	System General Object - A system file associated only with jobs processed in the batch mode. SGO is a permanent file used to accumulate object code. The SGO file is deleted after the job is complete.

Syntax:

```
ASSIGN2 lfc=  { SBO,cards       }    [lfc=...]
              { SLO,printlines  }
              { SYC             }
              { SGO             }
```

lfc	is a logical file code used in the task to denote a generic input or output source
SBO	is the System Binary Output file
cards	is the number of cards expected as object deck output. This number determines the size of the SBO temporary file.
SLO	is the System Listed Output file
printlines	specifies the number of print lines required for listed output. This number determines the size of the SLO temporary file.
SYC	is the System Control file. Use only if the task runs solely in the batch mode.
SGO	is the System General Object file

Usage:

```
A2 INN=SYC
A2 OT=SLO,100 OT2=SBO,50
```

3.8 ASSIGN3 Directive

The ASSIGN3 directive supplies default device assignments for logical file codes used by the task being cataloged. It also assigns a temporary disc file. This directive is for compatibility with MPX 1.x. Its use is not recommended.

Syntax:

```
ASSIGN3 lfc=devmnc, [blocks       ] [,U] [lfc=...]
                   , [reel ,[vol] ]
```

lfc	is a logical file code used in the task to denote a generic input or output source
devmnc	is the device mnemonic of a configured peripheral device
blocks	specifies the number of disc blocks (192 words) to allocate for the file
reel	specifies a one to four character identifier for the reel. If not specified, the default is SCRA (scratch).
vol	indicates the volume number for a multivolume tape. If not specified, the default is 0 (not multivolume).
U	indicates that the tape or disc is unblocked. If not specified, the default is blocked.

Usage:

Tape: A3 IN=M91000,SRCE,,U OT=PT

Disc: A3 IN=DC,20

3.9 ASSIGN4 Directive

The ASSIGN4 directive associates one or more logical file codes used by the task being cataloged with an existing LFC assignment. This assignment remains for the associated file or device even if the original assignment is deallocated. This directive is for compatibility with MPX 1.x. Its use is not recommended.

Syntax:

```
     ASSIGN4 lfc=lfc  [lfc=lfc]
```

lfc=lfc	is a pair of logical file codes. The first LFC is the new assignment and the second is the LFC already associated with a file or device in any previous ASSIGN directive, including ASSIGN4. Any number of LFC to LFC associations can be established.

Usage:

```
A2 6=SYC
A3 IN=M91000,REEL
A4 OUT=IN
A4 IN2=6
```

3.10 BUFFERS Directive

The BUFFERS directive specifies the number of blocking buffers required to support dynamically assigned blocked files in the task being cataloged.

Syntax:

```
     BUFFERS buffers
```

buffers	is the number of 768-byte blocking buffers required. The range is 0-255. If not specified, the default is three.

If option 19 is set, the number of buffers specified is added to the three buffers required by the Debugger. If option 19 is not set, the number of buffers specified is the number of buffers reserved.

NOTES:

For shared tasks, BUFFERS supplies the total blocking buffer allocation for both static and dynamic file allocations.

The total buffer count from all sources (static, dynamic, and run time) is limited to 254 buffers at execution time.

3.11 CATALOG and BUILD Directives

The CATALOG and BUILD directives supply the load module name plus other control information for the task being built. CATALOG creates a file whose name is equal to the load module name in directory @SYSTEM (SYSTEM). BUILD creates a file whose name is equal to the load module name in the default working volume and directory. To create a load module file with a different file name, directory name, or volume name, use the LMPATH directive. See the LMPATH directive and Table 3-1 for a summary.

When cataloging the main module of a task, CATALOG and BUILD specify the task's privilege, priority, and overlay transient area. The optional parameters can be specified in any order.

BUILD and CATALOG cannot be used in the same CATALOG job.

Syntax:

```
{CATALOG}  loadmod   [P] [ TRA=size][ priority][ NOM][ NOP][ SYM]
{BUILD  }            [U]
                     [O]
```

loadmod	is the name of the load module being built and, if not supplied by LMPATH, the name of the file which contains the load module. The name can be a maximum of eight characters. File names that begin with the letters SYSG are loaded with a TSA address of X'60000'. This facilitates SYSGEN's remapping between host and target systems.
P,U,O	for the main module only, specify P for a privileged task or U for an unprivileged task (default). For an overlay module, specify O. Overlays assume the privileged or unprivileged status of the main load module.
TRA=size	is used with the main load module to specify the hexadecimal number of bytes to allocate for the overlay transient area below the main load module. The default is an area above the main load module which is large enough to accommodate all overlay load modules cataloged in the same run as the main load module.
priority	for main load module only, specifies a base priority in the range 1 to 64. If not specified, the default is 60. Overlay load modules assume the priority of the related main load module. If the BUILD or CATALOG directive pertains to an overlay module, do not specify priority.
	The priority at which a task executes depends on how the task is activated (on-line, batch, or real time). In real time, the task maintains its cataloged priority. If activated in TSM or in the batchstream, its priority changes to the SYSGEN-defined priorities of either interactive or batch.
NOM	inhibits printing a main or overlay load module map
NOP	inhibits output of a main or overlay load module to the file specified as the load module file
SYM	saves the symbol table for the main load module on a device or file. This option is used when cataloging load modules of a segmented task in different CATALOG runs. If the module is an overlay module, do not specify SYM.

Usage:

```
BUILD     LOAD1  P  TRA=40000  NOM
CATALOG   LOAD2  NOP
```

3.12 CONNECT Directive

The CONNECT directive establishes a connection between a specified Datapool (DATAPOOL or DPOOL00 through DPOOL99) partition and its corresponding Datapool dictionary. The parameters specified with this directive supply CATALOG with information on which dictionary to access when a Datapool variable is referenced in the object code.

Syntax:

```
CONNECT pathname TO partition [ [PROTGRAN=]
        number [FIRSTPAGE=] start]
```

pathname	is the Datapool dictionary pathname
partition	is the Datapool partition (DATAPOOL, DPOOL00 - DPOOL99)
number	is the number of 512-word protection granules included in the partition. If not specified, the partition must be defined prior to catalog time.
start	is the beginning page number of the partition. If not specified, the partition must be defined prior to catalog time.

The dictionary for DATAPOOL may be statically assigned to LFC DPD. This is provided for compatibility; its use is not recommended.

The CONNECT directive may be continued on a subsequent line by entering a hyphen (-) as the last nonblank character on a line. The hyphen must be preceded by a blank.

When CONNECT directives require location of a Datapool partition definition, the LMPATH target volume/directory will be searched only if the LMPATH directive precedes the CONNECT directives.

Usage:

```
CONNECT @VOLUME(SOME_DIR)POOL00.DICT TO DPOOL00 -
PROT=4 FIRST=192
```

3.13 ENVIRONMENT Directive

The ENVIRONMENT directive establishes residency, memory execution class, sharing characteristics, and other environmental parameters for a task. The entries with this directive supply information for the load module information area (preamble) in the main load module.

If the ENVIRONMENT directive is not used, a task is nonresident and executable in any available memory class (S, H, or E), UNIQUE, MAP8192, and DEBUGGABLE.

Syntax:

```
                                [,E]  [,UNIQUE]
ENVIRONMENT   [RESIDENT]        [,H]  [,SHARED]  [,MAP2048]
                                [,S]  [,MULTI ]  [,MAP8192]  [,NODEBUG]
```

RESIDENT	specifies the task is resident in memory and cannot be swapped
E	executes in class E memory only. If class E is unavailable, delay execution until class E is available.
H	executes in class H or faster memory. If both class H and E memory are unavailable, delay execution until either one is available. If the requested class of memory is not installed on the system, the first lower speed memory available is allocated to the task.
S	executes in any class of memory available (H, S, or E). Class S is the default if no memory class is specified.
UNIQUE	specifies the task is unique and not available for multiple concurrent activations. Only one copy of the load module can be active in the system at one time. This is the default and can be used with sectioned or nonsectioned tasks.
SHARED	copies the CSECT area of a sectioned task into physical memory once and copies DSECT as needed for sharing. Use only with a sectioned task.
MULTI	multicopies the entire load module into physical memory as needed for concurrent activations. Can be used with a sectioned or nonsectioned task.
MAP2048	indicates the map size of the target system is 2KW. This establishes the memory allocation and bounding requirement for the CSECT in sectioned tasks to be 2KW. This is the default if a map size is not specified.
MAP8192	indicates the map size of the target system is 8KW. This establishes the memory allocation and bounding requirement for the CSECT in sectioned tasks to be 8KW.
NODEBUG	indicates the Debugger cannot be attached to the load module. If not specified, the Debugger can be attached.

Usage:

```
ENVIRONMENT RESI,H,MULTI,MAP2048
```

3.14 EXCLUDE Directive

The EXCLUDE directive excludes object modules in the system or user libraries from the load module being cataloged, even though the modules contain definitions for referenced global symbols.

Object modules INCLUDEd from a library during cataloging may also reference the EXCLUDEd object modules. The references are ignored and the specified object modules remain excluded.

Object modules are excluded by specifying the referenced global symbol name. All global symbols defined in an object module must be excluded for the object module to be excluded from the load module.

Syntax:

```
    EXCLUDE name [name] ...
```

name	is the name of a global symbol in the object module

3.15 EXIT Directive

The EXIT directive terminates CATALOG processing. In interactive mode, control returns to TSM. In batch mode, processing continues with the next JCL statement.

Syntax:

```
    EXIT
```

3.16 EXTDMPX Directive

The EXTDMPX directive positions the extended portion of MPX-32 in the logical address space of the task being cataloged. This directive pertains to the expanded execution space option of MPX-32.

Syntax:

$$\text{EXTDMPX}\left\{\begin{array}{l}\text{MINADDR}\\\text{MAXADDR}\\\text{[MBLK]} = \text{mapblock}\end{array}\right\}$$

MINADDR	locates the extended portion of MPX-32 at the top of the task service area, below the DSECT
MAXADDR	locates the extended portion of MPX-32 at the top of the task's extended data space
mapblock	is a 1 to 4 digit decimal value between 64 and 2047 that specifies a particular map block in the task's logical address space where the extended portion of MPX-32 is to be located

At run time, values for mapblock below 64 (other than MINADDR) or above MAXADDR cause an abnormal termination in task activation.

For shared tasks, the cataloged value of EXTDMPX cannot be overridden by the EXTDMPX TSM directive.

This directive has no effect if the expanded execution space option is not in use.

3.17 FILES Directive

The FILES directive specifies the number of resources (files or devices) required for dynamic assignments in a task.

Syntax:

```
    FILES number
```

number	is the number from 0 to 255 of dynamic resource assignments required for the task. If not specified, the default is five.

If option 19 is set, the number specified is added to the five files required by the Debugger. If option 19 is not set, the number of files specified is the number of files reserved.

Notes: For shared tasks, this specifies the total number of resources from all sources (run time, static, and dynamic) that may be allocated.

The total file count at run time may not exceed 248.

3.18 INCLUDE Directive

The INCLUDE directive includes object modules from the system or user library in the load module being cataloged, even though the modules are not referenced. If the PROGRAMX directive is used to suppress SGO as an input source, INCLUDE must be used to retrieve object modules from a library.

Syntax:

INCLUDE name [name] ...

name	is the name of a global symbol in the object module

3.19 LINKBACK Directive

The LINKBACK directive specifies overlay load modules at lower levels for backward links when cataloging an overlay load module. Forward links from lower to higher level overlay load modules are established automatically by CATALOG. LINKBACK allows resolution of global symbol references in the current load module to definitions in the specified lower level overlays. In addition, if Option 3 is set, references to local commons in the current load module are resolved by corresponding local commons in the specified lower level overlays.

Syntax:

LINKBACK loadmod [loadmod] ...

loadmod	is the name of an overlay load module at a lower level. More than one name can be supplied.

3.20 LMPATH Directive

The LMPATH directive specifies the pathname (including the file name) of a resource in which to store the load module(s). LMPATH is optional. If not supplied, the file name is the load module name taken from the first BUILD/CATALOG directive. Volume and directory are the current default or @SYSTEM(SYSTEM) as appropriate. See the CATALOG/BUILD directive and Table 3-1 for further information.

Syntax:

LMPATH pathname

pathname	is the pathname of a file in which the load module is cataloged

When CONNECT directives require location of a Datapool partition definition, the LMPATH target volume/directory is searched only if the LMPATH directive precedes the CONNECT directives.

3.21 LORIGIN Directive

The LORIGIN directive establishes a new overlay level and origin. The default origin (no parameter specified) is above the largest overlay load module at the preceding level. LORIGIN does not have to be used for the lowest level of overlays, but must be used for all higher levels.

If the second or higher level overlay is being replaced when recataloging an overlay load module, the load module specified in the LORIGIN directive must have been previously cataloged by a BUILD or CATALOG directive within the same CATALOG run.

Syntax:

$$\underline{\text{LORI}}\text{GIN} \quad \begin{bmatrix} \text{X bytes} \\ \text{loadmod} \end{bmatrix}$$

X bytes	is the hexadecimal number of bytes to offset this level from the beginning of the overlay transient area. The value is specified by X, one or more blanks, and the number of bytes in hexadecimal.
loadmod	specifies the override origin at the end of a specific overlay load module at the previous level. This overlay does not have to be the largest overlay at that level.

3.22 MOUNT Directive

The MOUNT directive specifies non-public volume requirements for the task being cataloged.

Syntax:

$$\underline{\text{MOUNT}}\text{ volname ON devmnc [SYSID=id] [OPTIONS=([PUBLIC] [,NOMSG])]}$$

$$\left[\text{SHARED} = \begin{Bmatrix} \text{Y} \\ \text{N} \end{Bmatrix}\right]$$

volname	is the name of the volume to be mounted
devmnc	is the device mnemonic of a configured peripheral device
id	specifies the port identifier required for multiport volumes only. Must be MPx where x is a single hexadecimal digit.
OPTIONS	specifies options for the mounted volume. If PUBLIC is specified, the volume is to be mounted for public use (valid only if task has System Administrator attribute). If not specified, the default is nonpublic. If NOMSG is specified, a mount message is not displayed on the operator's console. If not specified, a mount message is displayed.
SHARED	specifies sharing attributes for the volume. If yes (Y) is specified, the resource is explicitly shared. If no (N) is specified, the resource is for exclusive use. If not specified, the resource is implicitly shared.

Usage:

```
MOUNT DIR1 ON DM0202 SHARED=Y
```

3.23 OPTION Directive

The OPTION directive specifies up to 32 options that become permanent attributes of the load module being cataloged. Options 1 to 32 set bits in the option word in the task's TSA. The bit set is determined by subtracting the option number from 32.

When activated, the task can use the M.PGOW service to return the contents of the TSA option word, check the bit settings, and take action as required.

Options can also be specified before a task is run in the interactive or batch mode. Options supplied at run time may override cataloged options or may be added to (ORed with) cataloged options. Options 1 to 20 are task-dependent. Options 21 to 32 are system-defined and available to all tasks. Refer to the MPX-32 Reference Manual for more information.

Syntax:

```
     OPTION n [n] ...
```

n is a number from 1 to 32 which sets the corresponding bit in the TSA status word. CATALOG options are described in Section 1. System options for the load module can be specified by name or number:

Option Number	Option Name	Description
21	PROMPT	displays the first three characters of the task name (load module name) before reading from the terminal when the task is run in the interactive mode
22	LOWER	inhibits converting lower case to upper case. This option is only valid if the task is run in the interactive mode.
23	TEXT	echoes text to the user terminal (interactive) or SLO file or device (batch) as it is read from the SYC file
24	DUMP	specifies that if the task aborts a dump of the task's area of memory will be generated
25	CPUONLY	executes the task on the CPU only
26	IPUBIAS	executes the task on the IPU if the task is IPU-compatible

3.24 ORIGIN Directive

The ORIGIN directive establishes a new origin (level unchanged) for subsequent overlay load modules. It can be used to override the default origin for a set of overlays at a particular level. The default origin (no parameter specified) is above the largest overlay load module at the preceding level.

Syntax:

```
    ORIGIN [X bytes ]
           [loadmod ]
```

X bytes	is the hexadecimal number of bytes to offset this level from the beginning of the overlay transient area. The value is specified by X, one or more blanks, and the number of bytes in hexadecimal.
loadmod	specifies the new origin to be at the end of a specific overlay load module at the previous level. The specified overlay does not have to be the largest overlay at that level. If replacing the second or higher level overlay when recataloging an overlay load module, the loadmod name cannot be used unless the referenced load module has been previously cataloged by a BUILD or CATALOG directive within the same CATALOG run.

3.25 PASSWORD Directive

The PASSWORD directive is included for compatibility and is ignored by CATALOG. Items following this directive on the same line are ignored.

Syntax:

```
    PASSWORD
```

3.26 PROGRAM Directive

The PROGRAM directive specifies object modules to include from SGO in a main or overlay load module. If omitted, all object modules on the file or device assigned to SGO are included.

Syntax:

```
    PROGRAM objmod [objmod] ...
```

objmod	is the name of the object module (such as, program/subroutine name) to include. More than one name can be specified.

3.27 PROGRAMX Directive

The PROGRAMX directive excludes all object modules from SGO when cataloging a load module. An INCLUDE directive is required to get object modules from a library if the PROGRAMX directive is used.

Syntax:

```
    PROGRAMX
```

3.28 RECATALOG Directive

The RECATALOG directive is used when cataloging a segmented task in phases or when recataloging one or more overlays of a segmented task. RECATALOG can only be used with single file load modules. The load module file must exist if RECATALOG is used.

Syntax:

RECATALOG [loadmod]

loadmod	is the one- to eight-character name of the permanent disc file containing the load modules. If LMPATH supplies a file name, loadmod is ignored.

3.29 SEGFILES Directive

The SEGFILES directive specifies the number of noncontiguous disc files required for use by the task. If this directive is not used, the default is the number of files specified in the FILES directive. If neither the SEGFILES or FILES directives are specified, the default is five.

Syntax:

SEGFILES number

number	is the number of noncontiguous disc files required by the task. This number must not be greater than the number specified in the FILES directive.

If option 19 is set, the number specified is added to the five files required by the Debugger. If option 19 is not set, the number of files specified is the number of files reserved.

3.30 SPACE Directive

The SPACE directive allows the potential maximum task size to be increased above the default 2MB size.

Syntax:

$$\text{SPACE} \quad \begin{Bmatrix} \text{increment} \\ \text{MBLK = mapblock} \end{Bmatrix}$$

increment	is a 1 to 2 digit number that specifies the maximum task size in one megabyte increments. The range is from 3 to 16MB.
mapblock	is a 1 to 4 digit decimal number that specifies the maximum task size in map blocks. The range is from 256 to 2048.

The SPACE directive establishes the maximum size to which a task can grow. No memory is actually allocated to the task.

For shared tasks, the cataloged value can not be overridden with the TSM SPACE command.

The SPACE directive has no effect on tasks executed on a CONCEPT 32/27, or CONCEPT 32/87 system.

3.31 SYMTAB Directive

The SYMTAB directive loads the symbol table containing the names of all common blocks, definitions, and references from a previous CATALOG session. The symbol table is used when cataloging a segmented task in phases or when recataloging a segmented task. If the SYMTAB directive is used, the SYMTAB file or device must be assigned to logical file code SYI prior to executing CATALOG.

Syntax:

```
    SYMTAB
```

3.32 VOLUMES Directive

The VOLUMES directive specifies the number of nonpublic volumes that can be dynamically mounted by the task at one time.

Syntax:

```
    VOLUMES number
```

number	is the number of entries to be reserved. This number is in addition to the current working volume plus any MOUNT directives processed. If not specified, the default is zero.

Table 3-1.

LMPATH/BUILD/CATALOG Interaction

	Name and Location of Load Module File		
LMPATH Condition	BUILD X	CATALOG Y	Execution Directives
NO LMPATH	@working(working) X		$@working(working) X (or)
		@SYSTEM(SYSTEM) Y	$@SYSTEM(SYSTEM) Y
LMPATH is: @VOL(DIR)	@VOL(DIR)X		$@VOL(DIR) X (or)
		@VOL(DIR)Y	$@VOL(DIR) Y
LMPATH is: @VOL(DIR) FILENAME	@VOL(DIR)FILENAME	@VOL(DIR)FILENAME	$@VOL(DIR)FILENAME

Notes:

- X and Y are limited to eight characters and may contain any printable characters if LMPATH has supplied the file name. If this field will be used as the file name, then normal MPX-32 rules for file names apply. This field may optionally be enclosed in single quotes (not counted in the eight characters).
- FILENAME may be up to 16 characters long and adheres to normal MPX-32 rules for file names.
- Load modules are placed in execution by referencing the file that contains them.
- By default, the execution time task name is the name of the file that contains the load module (truncated to eight characters).

SECTION 4 - ERRORS AND ABORTS

4.1 Error Overview

CATALOG reports error conditions as (WARNING) or <<FATAL>> depending on severity. Fatal type errors may cause immediate termination of processing or may allow processing to continue. In either case, any fatal error will inhibit the generation of a load module file, and will set the task abort flag. Warning type errors never cause process termination and inhibit the production of a load module file only in certain cases and when option 18 is set.

These cases are conditions that CATALOG has resolved, but there is doubt as to the correctness and/or completeness of the load module. In such cases, the process abort flag is set; if option 18 is set, the load module file is not updated. By examining the flagged conditions, the usability of the load module can be determined. If the abort flag is set, CATALOG always indicates, at termination whether the load module file has been updated or not.

4.1.1 Phase One Errors

During phase one (the linking phase) CATALOG processes the directives and performs the first pass over the object code. Every effort is made to complete phase one and report as many problems as possible.

Directive errors are reported by a message of the form:

```
    *ERROR IN FIELD n: description*
```

which is displayed immediately under the incorrect directive. The following example demonstrates how field n is assigned:

```
Directive:          BUILD        TESTMOD      P       TRA=1000     NOM
Field : n=            1             2         3        4     5      6
```

If appropriate, a second line is displayed which provides more information on the error and/or possible corrective actions.

Errors in the object code are of two origins:

- physically corrupted records which fail the tests for record type, checksum, or sequence
- logically incorrect operations which, in the context of this run, direct CATALOG to perform an inconsistent operation

Object code errors are reported in the following form:

```
    LFC: lfc
    MODULE: module
    PROGRAM: program, OBJECT REC X'nnn' - description
```

lfc	is the logical file code presenting the origin of the record (SGO or a library)
module	is the load module currently being linked
program	is the name of the program element currently being processed
nnn	is the program's logical object record number
description	indicates the nature of the error

A blank program name field indicates that the error occurred while processing the **first** record of a new program element.

All directive and object code errors detected in phase one are fatal. CATALOG issues a fatal message and terminates at the end of phase one without updating the load module file.

4.1.2 Phase Two Errors

During phase two (the building phase) a fatal error generally causes immediate termination of CATALOG. Warning errors are reported and always result in an updated load module file, unless option 18 has been set.

4.1.3 Errors from MPX-32 (Phase One and Two)

When input/output operation errors occur, the File Control Block (FCB) status word, logical file code, and other pertinent information are displayed. Such errors usually indicate a hardware failure in the I/O device involved. See the MPX-32 Reference Manual Volume I for a description of the FCB status word (word 3) and its interpretation for different devices.

CATALOG also reports errors returned from MPX-32 services. The MPX-32 error/abort code is contained in a message that supplies pertinent information. To interpret the error/abort code, use the TSM $ERR directive, or consult Appendix C of the MPX-32 Reference Manual.

4.1.4 Conditions that Cause Incomplete Load Modules

The following paragraphs describe conditions that CATALOG has resolved, but there is doubt as to the correctness and/or completeness of the load module.

Allowing incomplete load modules to be built is a feature that is provided to aid code development. The programmer can assess the problems and decide whether the load module is executable or usable in a debugging session.

The production of an incomplete load module can be inhibited by option 18. If the replacement of an existing load module with a faulty one could cause problems, it is recommended that option 18 be used.

Multiple Transfer Addresses

When the object code linked into a load module contains more than one transfer address, CATALOG selects and uses the first one detected. Subsequent transfer addresses are reported in a warning message.

No Transfer Addresses

When the object code linked into a root load module contains no defined transfer address, CATALOG attempts to select an address. The selection criteria is:

1) The address of the first DEFed symbol of the first program element of the load module is selected.

2) If the first program element contains no DEFed symbols, the address of the first noncommon word allocated in the first program element of the load module is selected.

The selected symbol or address is displayed in a warning message and the load map header indicates no transfer address.

Providing a transfer address allows the load module to be loaded. Even if the selected location is incorrect, the load module can be loaded with the debugger, and execution starts at the correct location using the appropriate DEBUG directive.

The following three conditions result from memory reference instructions for which CATALOG cannot provide a valid address. The action taken by CATALOG is to replace the instruction with a call to DEBUG. The effect of this is to cause the debugger to be automatically loaded if the task is executed and the faulty instruction is encountered. If the debugger is already loaded and a faulty instruction is executed, DEBUG signals a BREAK occurrence.

1. Unresolved External References

   When a program element references an external symbol which is not defined, CATALOG issues a warning message that displays the symbol name and the location of the reference (both program and load module-relative addresses are displayed).

   When an external symbol is referenced several times in one program element, the references are linked together in the object code. The warning message issued by CATALOG provides the address of all the instructions in the list.

2. Unresolved Datapool References

   When a program element references a Datapool variable and that variable is not defined in any connected dictionary, CATALOG issues a warning message that displays the symbol name and the program and load module relative addresses of the reference.

3. Out of Range Datapool References

   When a program element references a Datapool variable whose dictionary definition causes the generated address to be beyond the bounds of the partition definition, CATALOG issues a warning message that displays the symbol name and the location of the reference (both program and load module-relative addresses are displayed).

In the following three conditions, the executable portion of the load module is not affected, but the requested information is missing.

1. DEBUG Symbol Data Processing Errors

   Any errors detected while DEBUG symbol data is being processed cause CATALOG to reset option 19. Processing then continues, and a warning message is displayed. The load module preamble will indicate that symbolic DEBUG data is not present.

2. Program Information Data Processing Errors

   Any errors detected while program identification or time/date records are being processed cause CATALOG to reset option 15. Processing continues, and a warning message is displayed. The load module preamble will indicate that program information data is not present.

3. SYMTAB Save File Generation Errors

   Any errors detected while producing the symbol table save file cause an appropriate warning message to be issued and the operation terminated. CATALOG processing continues.

4.2 Abort Codes

CT04 UNRECOVERABLE I/O ERROR ON FILE OR DEVICE ASSIGNED TO LFC: SLO,

ST=RMXX IF ERROR ON OPEN; IOXX IF ERROR ON WRITE

Abort status includes Resource Manager (RM) status if the error occurred on open, or IOCS (IO) status if the error occurred on a write operation.

CT06 CATALOG EXECUTION ERRORS AS DESCRIBED ON LFC: SLO AND/OR UT

All <<FATAL>> errors indicate a CT06 abort status. The specific reason for the abort displays in the listed output stream and on the terminal.

SECTION 5 - EXAMPLES

This section provides sample programming sequences illustrating the use of CATALOG.

Example 1 - Catalog Load Module in User Directory

The following example catalogs a load module named X.TST1 in the user's current working directory:

```
TSM >$ASSIGN SGO TO O.TST1            (Object file to be cataloged)
TSM >$OPTION 19                       (Option to include debugger symbols)
TSM >$CATALOG
MPX-32 UTILITIES RELEASE x.x (CATALOG Rx.x.x)
(C) COPYRIGHT 1983 GOULD INC., CSD, ALL RIGHTS RESERVED
CAT > BUILD X.TST1                    (Load module is built in current directory)
CAT > EXIT
TSM >
```

Example 2 - Catalog Load Module in User Directory

The following example catalogs a load module named X.TST2 in the user's current working directory in file LONGFILENAME, and satisfies external references from a user object code library:

```
TSM >$ASSIGN SGO TO O.TST.2
TSM >$ASSIGN LIB TO ULIB.L            (Subroutine library file)
TSM >$ASSIGN DIR TO ULIB.D            (Subroutine directory file)
TSM >$OPTION 15 19
TSM >$CATALOG
MPX-32 UTILITIES RELEASE x.x (CATALOG Rx.x.x)
(C) COPYRIGHT 1983 GOULD INC., CSD, ALL RIGHTS RESERVED
CAT> LMPATH LONGFILENAME
CAT > BUILD X.TST2
CAT > EXIT
TSM >
```

Example 3 - Catalog Segmented Task

The following example catalogs, with selective use of SGO, a segmented task with the overlay structure illustrated in Figure 2-3, with default origins as illustrated in Figure 2-4:

```
TSM >$ASSIGN SGO TO O.TST3
TSM >$ASSIGN SYM TO SYMFILE           (File for symbol table output)
TSM >$CATALOG
MPX-32 UTILITIES RELEASE x.x (CATALOG Rx.x.x)
(C) COPYRIGHT 1983 GOULD INC., CSD, ALL RIGHTS RESERVED
CAT > BUILD X.TST3 SYM                (Catalog root of program and output
                                       symbol table to SYM)
```

```
CAT > PROGRAM MAIN
CAT > BUILD A1 O                     (Catalog overlay A1)
CAT > PROGRAM A1
CAT > BUILD A2 O                     (Catalog overlay A2)
CAT > PROGRAM A2
CAT > LORIGIN
CAT > BUILD B1 O                     (Catalog overlay B1)
CAT > PROGRAM B1
CAT > BUILD B2 0                     (Catalog overlay B2)
CAT > PROGRAM B2
CAT > BUILD B3 0                     (Catalog overlay B3)
CAT > PROGRAM B3
CAT > BUILD B4 0                     (Catalog overlay B4)
CAT > PROGRAM B4
CAT > EXIT
TSM >
```

Example 4 - Catalog Segmented Task

The following example catalogs a segmented task with the overlay structure illustrated in Figure 2-2:

```
TSM>$ASSIGN SGO TO O.TST3
TSM >$ASSIGN SYM TO SYMFILE          (File for symbol table output)
TSM >$CATALOG
MPX-32 UTILITIES RELEASE x.x (CATALOG Rx.x.x)
(C) COPYRIGHT 1983 GOULD INC., CSD, ALL RIGHTS RESERVED
CAT > BUILD X.TST3 SYM               (Catalog root of program and output symbol
                                     table to SYM)
CAT > PROGRAM MAIN
CAT > BUILD A1 O                     (Catalog overlay A1)
CAT > PROGRAM A1
CAT > BUILD A2 O                     (Catalog overlay A2)
CAT > PROGRAM A2
CAT > BUILD A3 O                     (Catalog overlay A3)
CAT > PROGRAM A3
CAT > EXIT
TSM >
```

Example 5 - Replace Overlay

The following example replaces an overlay in the load module created in example 4:

```
TSM >$ASSIGN SGO TO O.TST4
TSM >$ASSIGN SYI TO SYMFILE          (File for symbol table input)
TSM >$CATALOG
MPX-32 UTILITIES RELEASE x.x (CATALOG Rx.x.x)
(C) COPYRIGHT 1983 GOULD INC., CSD, ALL RIGHTS RESERVED
CAT > SYMTAB                         (Restore symbol table from SYI)
CAT > LMPATH @VOLUME(DIRECTORY)      (Set directory for RECATALOG)
CAT > RECATALOG X.TST3
CAT > BUILD A1 O                    (Replace overlay A1)
CAT > PROGRAM A1
CAT > EXIT
TSM >
```

Example 6 - Catalog Overlays as Separate Files

The following example catalogs three overlays as separate files by setting the option for multiple disc file mode:

```
TSM >$ASSIGN SGO TO O.TST5
TSM >$ASSIGN LIB TO ULIB.L
TSM >$ASSIGN DIR TO ULIB.D
TSM >$ASSIGN SYM TO SYMFILE
TSM>$OPTION  2                      (Option to create separate files for each
                                    overlay)
TSM >$CATALOG
MPX-32 UTILITIES RELEASE x.x (CATALOG Rx.x.x)
(C) COPYRIGHT 1983 GOULD INC., CSD, ALL RIGHTS RESERVED
CAT > CATALOG X.TST5 SYM
CAT > PROGRAM MAIN
CAT > CATALOG A1 O
CAT > PROGRAM A1
CAT > CATALOG A2 O
CAT > PROGRAM A2
CAT > CATALOG A3 O
CAT > PROGRAM A3
CAT > EXIT
TSM >
```

Example 7 - INCLUDE and EXCLUDE Directive Usage

The following example illustrates the use of INCLUDE and EXCLUDE directives:

```
TSM >$ASSIGN LIB TO ULIB.L
TSM >$ASSIGN DIR TO ULIB.D
TSM >$CATALOG
MPX-32 UTILITIES RELEASE x.x (CATALOG Rx.x.x)
(C) COPYRIGHT 1983 GOULD INC., CSD, ALL RIGHTS RESERVED
CAT > BUILD X.TST6
CAT > EXCLUDE A1,A2,A3         (These subroutines will be in overlays)
CAT > INCLUDE  MAIN            (Get global symbol MAIN from the subroutine
                               library)
CAT > PROGRAMX                 (No object code is read from SGO)
CAT > BUILD A1 O
CAT > INCLUDE A1               (Get global symbol A1 from the subroutine library)
CAT > PROGRAMX                 (No object code is read from SGO)
CAT > BUILD A2 O
CAT > INCLUDE A2               (Get global symbol A2 from the subroutine library)
CAT > PROGRAMX                 (No object code is read from SGO)
CAT > BUILD A3 O
CAT > INCLUDE A3               (Get global symbol A3 from the subroutine library)
CAT > PROGRAMX                 (No object code is read from SGO)
CAT > EXIT
TSM >
```

Example 8 - Catalog Load Module

The following example catalogs the load module illustrated in Figure 2-3 with memory allocated as in Figure 2-5.

```
TSM >$ASSIGN SGO TO O.TST7
TSM >$CATALOG
MPX-32 UTILITIES RELEASE x.x (CATALOG Rx.x.x)
(C) COPYRIGHT 1983 GOULD INC., CSD, ALL RIGHTS RESERVED
CAT > CATALOG MAIN
CAT > PROGRAM MAIN
CAT > CATALOG A1 O
CAT > PROGRAM A1
CAT > CATALOG A2 O
CAT > PROGRAM A2
CAT > LORIGIN A1               (Start new overlay level, origin at end of A1)
CAT > CATALOG B1 O
CAT > LINKBACK A1              (Links overlay through A1)
CAT > PROGRAM B1
CAT > CATALOG B2 O
CAT > LINKBACK A1              (Links overlay through A1)
CAT > PROGRAM B2
CAT > ORIGIN A2                (Change origin to end of A2, remain at same overlay
                               level)
CAT > CATALOG B3 O
CAT > LINKBACK A2              (Links overlay through A2)
CAT > PROGRAM B3
CAT > CATALOG B4 O
CAT > LINKBACK A2             (Links overlay through A2)
CAT > PROGRAM B4
CAT > CATALOG B5 O
CAT > LINKBACK A2             (Links overlay through A2)
CAT > PROGRAM B5
CAT > EXIT
TSM >
```

Datapool Editor (DPEDIT)

MPX-32 Utilities

CONTENTS

FIGURES

TABLES

DATAPOOL EDITOR (DPEDIT)

SECTION 1 - OVERVIEW

Datapools are memory partitions defined either at system generation (SYSGEN) or with the Volume Manager (VOLMGR). Datapool partitions (DATAPOOL, DPOOL00 - DPOOL99) contain data structured by Datapool dictionaries that are built and maintained by the Datapool Editor (DPEDIT) utility. DPEDIT can add, change, delete, and equate variables in an existing dictionary, or build a new dictionary.

In order to build and maintain Datapool dictionaries on one (host) system and then run the task using the same dictionary on another (target) system, the Datapool partitions DPOOL00 through DPOOL99 need not be present on the host system when using DPEDIT. DPEDIT does not check if each entry into the dictionary is within the range of the partition.

The word Datapool or Datapool partition used in the following sections refers to the partitions named DATAPOOL and DPOOL00 through DPOOL99.

DPEDIT recognizes file names of 1 to 16 characters. Unless otherwise specified, files assigned to logical file codes are forced to the appropriate format - blocked or unblocked.

1.1 General Description

A task using a location in a common partition other than Datapool must define all of the common partition's locations. Whenever a common partition is changed, the source for each task accessing the partition must be modified to reflect the change.

Datapool and Datapool dictionaries allow tasks to reference memory locations symbolically and define only the locations to be accessed.

Each task structures and shares a given Datapool partition through a Datapool dictionary. Different tasks can access the same Datapool variables by assigning the same dictionary.

The Datapool partition can be defined by a single dictionary or multiple dictionaries. A change in a variable is reflected by a change in a dictionary. All tasks referencing the partition are then recataloged with the modified dictionary. If multiple dictionaries are used, their modification depends on whether they reference Datapool locations whose offset would be affected by the change. Variables can be grouped into different offsets from the beginning of the Datapool partition. Therefore, unrelated tasks need not be concerned with a redefined location.

1.1.1 Datapool Dictionaries

A Datapool dictionary file must exist before DPEDIT can be used. A Datapool dictionary is a permanent file created by the Volume Manager (VOLMGR). The VOLMGR parameters EOFM=N and ZERO=Y must be specified when the Datapool dictionary is created. Make the size of the file large enough to hold twice the number of symbols to be defined in the dictionary. Because a block contains eight records, a formula for determining the file size necessary is:

$$\frac{\text{2 x number of symbols}}{8} = \text{necessary file size in blocks}$$

The minimum allowable size for a dictionary is five blocks.

Multiple dictionaries allow tasks to communicate with each other. For example, assume task A stores a status variable in Datapool that is listed in Datapool dictionary AAA, and task B stores a status variable in Datapool that is listed in Datapool dictionary BBB. Task C can access these status variables through the Datapool dictionaries.

A task cannot modify a location not defined in its dictionary.

1.1.2 Static versus Dynamic Datapool

Datapool can be created statically at system generation (SYSGEN) or dynamically with the Volume Manager (VOLMGR).

SYSGEN permanently allocates memory for a Datapool partition in protection granule increments (512W). SYSGEN marks the allocated protection granules as unavailable for outswap and creates an entry defining the partition in the system directory.

The VOLMGR CREATE COMMON directive creates a Datapool partition when required by a task. These partitions are allocated in map blocks (2KW on a CONCEPT/32 computer).

MPX-32 can dynamically generate multiple Datapool map blocks into more than one logical address space. The physical space for a Datapool partition created with VOLMGR is not permanently allocated, as it is when a Datapool partition is created with SYSGEN.

1.2 Directive Summary

Directive	Function
/DPD	Assigns a new permanent file name to the LFC used for the Datapool dictionary
/ENTER	Indicates the following are data records to be used to change the Datapool
/LOG	Lists the contents of the Datapool dictionary
/REMAP	Reuses the Datapool dictionary by rebuilding from the /SAVE dictionary entries and hashing them into the Datapool dictionary
/SAVE	Preserves the binary contents of each active entry in the Datapool dictionary
/VERIFY	Verifies Datapool elements in the dictionary for proper bounding, duplicate entries, and improper relative addresses

SECTION - 2 USAGE

2.1 Accessing DPEDIT

DPEDIT can be accessed in the batch or interactive modes in one of three ways:

```
$DPEDIT
$RUN DPEDIT
$EXECUTE DPEDIT
```

$RUN DPEDIT is valid only from the system directory.

When accessing DPEDIT interactively, the DPE> prompt is displayed:

```
TSM>$DPEDIT
DPE>
```

2.2 Logical File Code Assignments

There are eight logical file codes (LFCs) associated with DPEDIT: Dictionary (DPD), Input (SYC), Listed Output (LO), Error Listing (ER), Save File (OT), Remap File (IN), and Scratch Files (U1 and XU1). LFC assignment statements must be made before DPEDIT is called.

2.2.1 Dictionary (DPD)

A Datapool dictionary is a permanent file containing symbol definitions. The Datapool dictionary is assigned to logical file code DPD. This LFC is forced unblocked by DPEDIT.

DPD Default and Optional Assignments

There is no default assignment to DPD.

Each time a Datapool dictionary is used, the dictionary must be assigned as follows:

```
$ASSIGN DPD TO pathname
```

pathname	is the pathname of a file containing a Datapool dictionary. The file is forced unblocked by DPEDIT.

2.2.2 Source Input (SYC)

The source input file contains DPEDIT directives and data statements used for structuring the partition. Data statements are described in Section 2.4. The source input file is assigned to logical file code SYC.

SYC Default and Optional Assignments

The default assignment for SYC is to the System Control file (SYC):

```
    $ASSIGN SYC TO SYC
```

There are two optional assignments for SYC:

```
    $ASSIGN SYC TO  { pathname   }
                    { DEV=devmnc }
```

pathname	is the pathname of a file containing input
devmnc	is the device mnemonic of a device containing input

2.2.3 Listed Output and Error Listings (LO and ER)

As DPEDIT processes directives, one line of listed output is generated for each operation performed. Operations that produce errors are written to a separate file or device. The listed output file is assigned to logical file code LO. The error file is assigned to logical file code ER. Listed output and errors can be produced on one file or device by equating the two file codes with $ASSIGN statements.

LO and ER Default and Optional Assignments

The default assignment for LO and ER is to logical file code (UT):

```
    $ASSIGN { LO } TO LFC=UT
            { ER }
```

In the interactive mode, output is generated on the user terminal.

In the batch mode, output is generated on the SLO device.

There are two optional assignments for LO and ER:

```
    $ASSIGN { LO } TO { pathname   }
            { ER }    { DEV=devmnc }
```

pathname	is the pathname of a file to contain listed or error output
devmnc	is the device mnemonic of a device to contain listed or error output

2.2.4 Save and Remap Files (OT and IN)

The /REMAP directive restructures an existing Datapool dictionary that was saved (by the /SAVE directive) during a previous DPEDIT run or in the current DPEDIT run.

The file or device used by the /SAVE directive is assigned to logical file code OT. The file or device to be used by the /REMAP directive is assigned to logical file code IN. The /DPD directive can assign a different file for Datapool dictionary output, or the name of the file can be specified with /REMAP. If the assignment is not changed, the existing dictionary is overwritten.

OT and IN Default and Optional Assignments

There are no default assignments for OT and IN.

There are two optional assignments for OT and IN:

$$\$AS \left\{\begin{matrix}\text{OT}\\ \text{IN}\end{matrix}\right\} \text{ TO } \left\{\begin{matrix}\text{pathname}\\ \text{DEV=devmnc}\end{matrix}\right\}$$

pathname	is the pathname of a file containing the /SAVE or /REMAP file
devmnc	is the device mnemonic of a device containing the /SAVE or /REMAP file

2.2.5 Scratch Files (U1 and XU1)

A sorted alphabetical listing and a sorted address listing are produced by the /LOG directive. The alphabetical listing is assigned to logical file code U1. The address listing is assigned to logical file code XU1. Both U1 and XU1 are forced unblocked by DPEDIT.

U1 and XU1 Default and Optional Assignments

The default assignment for U1 is to a temporary file of 100 blocks:

```
$AS U1 TO TEMP SIZE=100
```

The file size in the SIZE= parameter can be increased if necessary.

The default assignment for XU1 is to logical file code U1:

```
$AS XU1 TO LFC=U1
```

There are no optional assignments for U1 and XU1.

2.2.6 LFC Summary

The following is a table of LFCs used by DPEDIT and their default and optional assignments.

Table 2-1
DPEDIT LFC Summary

LFC	Default Assignment	Optional Assignment
DPD	N/A	pathname
ER	LFC=LO	pathname DEV=devmnc
IN	N/A	pathname DEV=devmnc
LO	LFC=UT	pathname DEV=devmnc
OT	N/A	pathname DEV=devmnc
SYC	SYC	pathname DEV=devmnc
U1	temporary file	N/A
XU1	LFC=U1	N/A

2.3 Exiting DPEDIT

To exit DPEDIT from the interactive mode, enter CNTRL C. In the batch mode, DPEDIT exits when it encounters a job control statement without a / in column one.

2.4 Input Data Format

Datapool dictionaries are structured through data records. These data records are built built in 72-byte card image format and are used to add, delete, or change Datapool symbols.

The structure of a data record is shown in Figure 2-1.

All fields of the data record except the SOURCE and DESCRIPTION fields must be left-justified and may not contain embedded blanks. The VARIABLE SYMBOL field contains the one to eight character name of the symbol to be added, deleted, or changed as specified by the U field.

The U field specifies the add function with a blank, the delete function with a minus sign, and the change function with an asterisk.

To specify the add function, all fields up to and including the BASE SYMBOL field must be used. The remaining fields are optional. A symbol can be added to the dictionary if it has not been previously defined in the dictionary. If the PRECISION field is specified, address bounding is verified before adding the symbol to the dictionary.

The delete function requires only the VARIABLE SYMBOL and U fields. The remaining fields are ignored. A symbol can be deleted only if it is not used as a base. If the symbol to be deleted references a base, the responsibility count for the base symbol is decremented. Responsibility count is the number of times the symbol is used as a base for other symbols.

The change function requires the VARIABLE SYMBOL and U fields. The remaining fields are optional. All fields describing the symbol can be changed if the symbol is not used as a base. If the symbol is used as a base, no changes can be made in the BASE SYMBOL or DISPLACEMENT fields.

Each blank column on the data record causes no change to the corresponding column of the original specification; a column containing a number sign (#) generates blanks in the corresponding column of the original specification; a column containing any other character causes a replacement of the corresponding column of the original specification.

The change function is column oriented. When an entire field is to be replaced, the high-order columns of the field should contain number signs (#) to blank out unwanted characters from the original specification. For example, if the BASE SYMBOL field entry is replaced with an entry of fewer characters, the unused columns in the changed record should contain number signs. The E field, which equates symbols with base symbols, must contain EQU. Any other character string is invalid.

The BASE SYMBOL field is used with the VARIABLE SYMBOL field and the E field. The base symbol referenced must have been previously defined by the Datapool dictionary. The BASE SYMBOL field may optionally contain a dollar sign ($) indicating location 0 of the dictionary.

The DISPLACEMENT field modifies the base symbol location if column 22 contains a plus sign (+). Absence of the plus sign in column 22 causes DPEDIT to ignore the displacement.

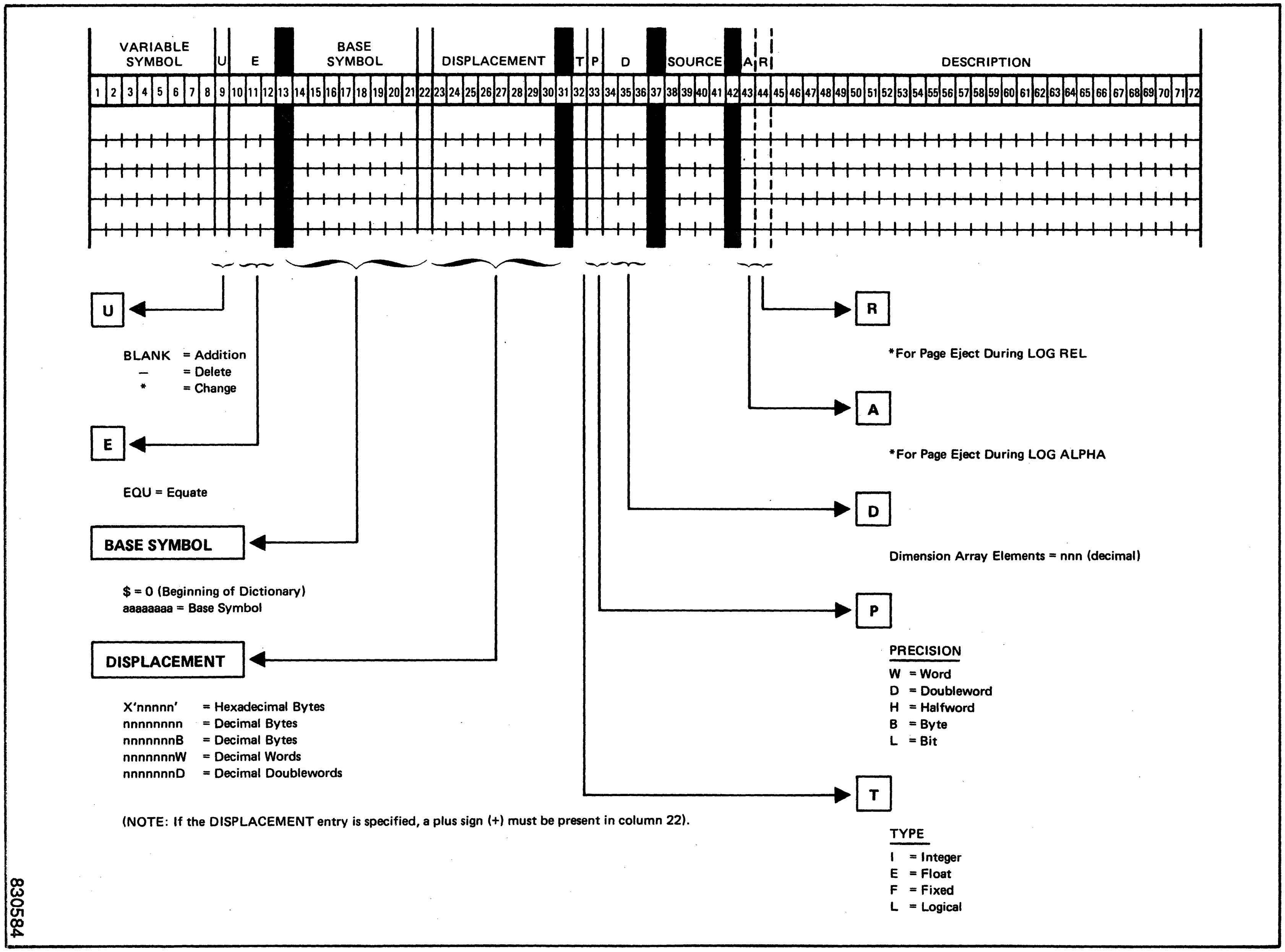


Figure 2-1. DPEDIT Data Record Format

The T field is for symbol type. If used, the T field must contain an E, F, I, or L.

The P field specifies precision. If used, the specified boundary, L, B, H, W, or D, is verified against the actual symbol address to ensure proper bounding.

The D field is for array dimensions. If used, the D field must contain decimal integers.

The SOURCE and DESCRIPTION fields are for user documentation. The SOURCE field provides a User Descriptor Area to identify the originator of the symbol. An asterisk in the first column of the DESCRIPTION field (column 43) causes a page eject during alphabetical logging (LOG ALPHA). An asterisk in the second column of the DESCRIPTION field (column 44) causes a page eject during relative logging (LOG REL). Columns 45 through 72 of the DESCRIPTION field can be used for comments.

2.5 Dictionary Records

Figure 2-2 shows the format for a Datapool dictionary entry built by DPEDIT. The dictionary entry record is a binary record of the entire dictionary entry, including a checksum and a sequence number. When a task is cataloged, and the partition used is named DATAPOOL, then the Datapool dictionary to be used must be assigned to logical file code DPD. If the partition used is named DPOOL00 through DPOOL99, then the Datapool dictionary to be used must be connected to the corresponding partition by the cataloger's CONNECT directive.

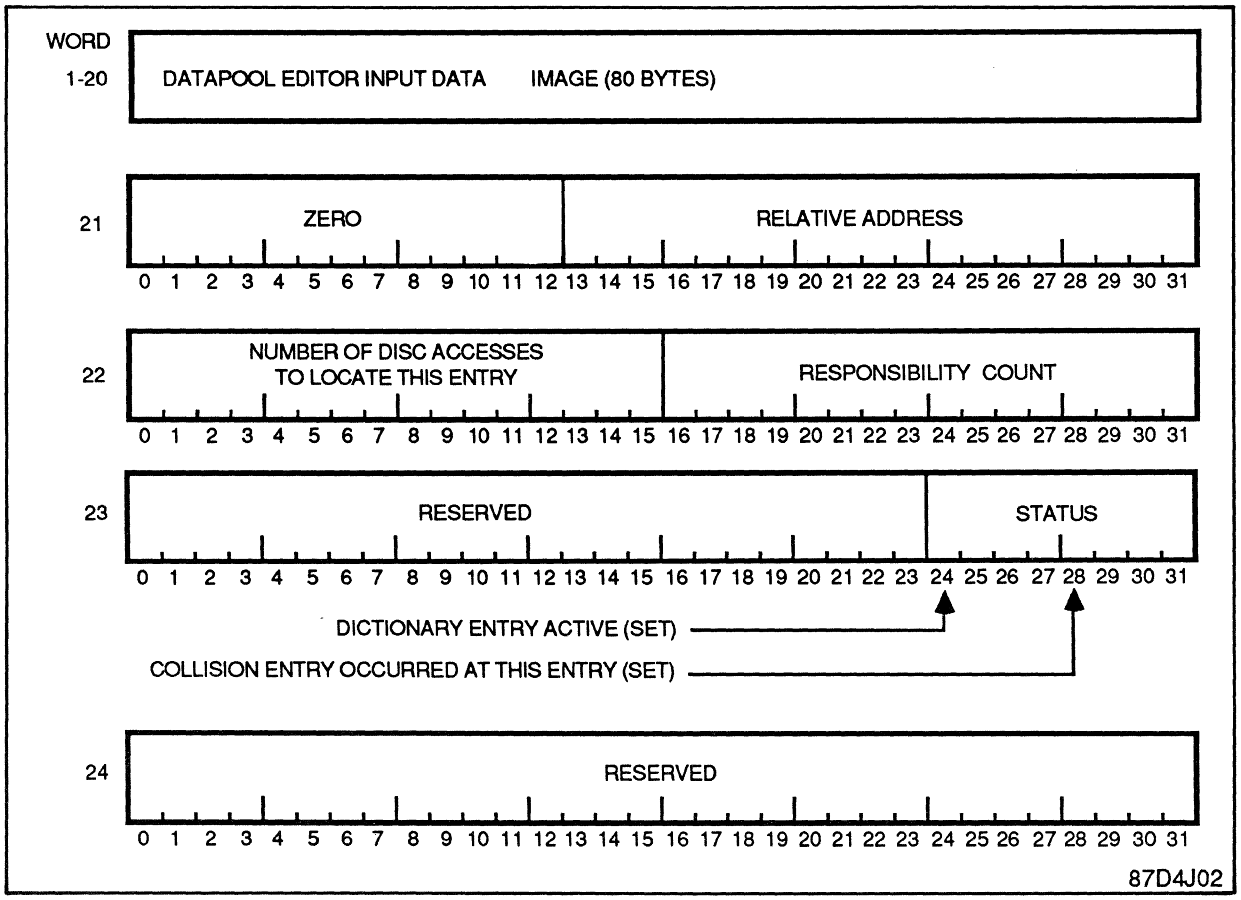


Figure 2-2. Datapool Dictionary Entry Format

2.6 Listings

DPEDIT produces two output files: the listed output file accessed through logical file code LO and the error file accessed through logical file code ER. If either file overflows and is assigned to the System Listed Output (SLO) file, the old SLO is dynamically deallocated (released for system output on job termination) and a new SLO file is allocated with the same size requirements as the original. Figure 2-3 describes listed output format.

Listed output contains a definition of the operations performed, the source records (Figure 2-1), the relative address within the Datapool of the symbol defined by the dictionary entry, the number of additional disc accesses required to locate the entry, the number of times this symbol is used as a base, and when applicable, an error code defining why the requested operation was not performed.

Format

```
CURRENT DATAPOOL FILE: MAIN
 ERROR    FUNCTION  SYMBOL  U E BASE    DISP T P D SC A R  DESCR  RELATIVE RESP  CM
 CODE                            SYMBOL                           ADDRESS  CNT
```

Format Explanation

The CURRENT DATAPOOL FILE is MAIN unless otherwise specified by a /DPD or /REMAP directive.

The ERROR CODE field contains four character codes on the lines where errors occurred.

The FUNCTION field indicates the function in effect (add, delete, log, or change).

The SYMBOL, U, E, BASE SYMBOL, DISP (displacement), T, P, D, SC (source), A, R, and DESCRIPTION (DESCR) fields are identical to those in Figure 2-1.

The RELATIVE ADDRESS field contains a hexadecimal address assigned to the variable SYMBOL relative to the beginning of the Datapool partition.

The RESPCNT (responsibility count) field contains the decimal number of times SYMBOL is used as a base.

The CM (collision mapping) field contains the decimal number of disc accesses required to locate SYMBOL.

Figure 2-3. DPEDIT Listed Output Format

SECTION 3 - DIRECTIVES

3.1 Introduction

The following sections describe DPEDIT directives. The legal delimiter between parameters is a comma. Only one blank is allowed between a directive and a parameter.

3.2 /DPD Directive

The /DPD directive assigns a different permanent file to logical file code DPD. This directive allows the use of multiple dictionary files during a single edit run.

The specified file is dynamically allocated unblocked by MPX-32 services.

Syntax:

```
/DPD  filename
```

filename	is a permanent file name of a Datapool dictionary

3.3 /ENTER Directive

The /ENTER directive indicates the following are data records to be processed by DPEDIT. DPEDIT processes data records until it encounters a different directive or an end-of-file. More than one /ENTER directive can be used in a DPEDIT directive stream.

Syntax:

```
/ENTER
```

3.4 /LOG Directive

The /LOG directive provides a listed output audit trail of all symbols defined in the datapool dictionary, the total number of entries in the dictionary, and the number of active entries.

Syntax:

$$\text{/LOG}\left[\begin{Bmatrix}\text{ALPHA}\\ \text{REL}\end{Bmatrix}\right]$$

ALPHA	specifies listed output in alphabetical order
REL	specifies listed output in the relative order symbols reside in the Datapool memory partition

If neither is specified, both types of output are generated.

3.5 /REMAP Directive

The /REMAP directive expands or rebuilds a Datapool dictionary without recreating dictionary entries.

/REMAP rebuilds a dictionary from the file assigned to logical file code IN. This file must contain the image of a dictionary specified with a /SAVE directive. Each entry is remapped through the hash coding scheme and written to the dictionary assigned to logical file code DPD. A new assignment to DPD can be made with the /REMAP and /DPD directives or the $ASSIGN job control language statement. If DPD is not reassigned, the file currently assigned to DPD is overwritten.

Syntax:

```
/REMAP [file][,R]
```

file	is the name of a permanent file to assign to logical file code DPD
R	rewinds the file assigned to logical file code IN before the dictionary entry records are processed

3.6 /SAVE Directive

The /SAVE directive preserves the contents of each active entry in the Datapool dictionary in dictionary entry records on the file assigned to logical file code OT. An end-of-file is written to OT when the function is complete.

When a /SAVE directive is specified, logical file codes DPD and OT must not be assigned to the same file.

Syntax:

```
/SAVE
```

3.7 /VERIFY Directive

The /VERIFY directive checks each active entry in the Datapool dictionary for proper placement in the dictionary, for precision to assure proper bounding, and for relative address within the range of the Datapool to ensure the correct computed value at entry time. Any discrepancies detected in the dictionary are noted on a listed output file.

Improperly mapped new entries are corrected and no error flags are generated. Improperly mapped entries whose names are already in the dictionary are deleted and an error flag is generated.

Incorrect relative addresses are corrected and an error flag is generated. Entries with no base symbol in the dictionary, or entries whose data record is invalid, are deleted and error flags are generated.

Range and precision errors generate flags.

Syntax:

```
/VERIFY
```

SECTION 4 - ERRORS AND ABORTS

4.1 DPEDIT Error Codes

When an error occurs, an error code is displayed. The following are DPEDIT error codes and their explanations.

Code	Explanation
EC11	Attempt to delete a symbol not found in the dictionary.
EC12	Attempt to delete a symbol used as a base for another variable.
EC13	Change requested for a symbol used as a base that may result in a change in the relative address.
EC14	The calculated relative address does not fall on the specified boundary (precision).
EC15	The referenced base symbol is not in the Datapool dictionary.
EC16	Attempt to add a symbol that is already defined in the dictionary.
EC19	Invalid specification in directive.
EC20	Log not processed; not enough memory to sort data.
EC21	Log not processed; scratch sort file not large enough to hold the necessary data.
EC22	Log not processed; unrecoverable I/O error on the scratch sort file.
EC23	Attempt to change a symbol not found in the dictionary.
EC24	Computed relative address does not agree with actual address.
EC25	Entries are multiply defined.
ERnn	Error encountered in processing data statement fields. The column number in which the error occurred is specified by "nn".

4.2 DPEDIT Abort Codes

The following are DPEDIT abort codes and their messages.

Code	Message
DP01	UNRECOVERABLE I/O ERROR WHILE READING OR WRITING THE DATAPOOL DICTIONARY.
DP02	DICTIONARY LOGICAL FILE CODE (DPD) UNASSIGNED.
DP03	UNRECOVERABLE ERROR ON ERROR (ER) FILE.
DP04	UNRECOVERABLE ERROR ON LISTED OUTPUT (LO) FILE.

Code	Message
DP05	UNABLE TO ALLOCATE ADDITIONAL SLO SPACE FOR LISTED OUTPUT. INITIAL FILE IS FILLED.
DP06	UNABLE TO ALLOCATE ADDITIONAL SLO SPACE FOR ERROR FILE. INITIAL FILE IS FILLED.
DP07	INVALID DIRECTIVE.
DP09	DICTIONARY OVERFLOW.
DP10	UNABLE TO REASSIGN THE FILE ASSIGNED TO LOGICAL FILE CODE DPD.
DP11	END-OF-TAPE OR ILLEGAL END-OF-FILE ENCOUNTERED ON FILE OR DEVICE ASSIGNED TO LOGICAL FILE CODE IN.
DP12	PHYSICAL END-OF-MEDIA ENCOUNTERED ON FILE OR DEVICE ASSIGNED TO LOGICAL FILE CODE OT.
DP13	UNRECOVERABLE ERROR ON FILE OR DEVICE ASSIGNED TO LOGICAL FILE CODE IN.
DP14	UNRECOVERABLE ERROR ON FILE OR DEVICE ASSIGNED TO LOGICAL FILE CODE OT.
DP15	LOGICAL FILE CODE OT UNASSIGNED AND THE SAVE FUNCTION REQUESTED.
DP16	LOGICAL FILE CODE IN UNASSIGNED AND THE REMAP FUNCTION REQUESTED.
DP17	SEQUENCE ERROR ON DICTIONARY ENTRY RECORD (ACCESSED THROUGH LOGICAL FILE CODE IN).
DP18	CHECKSUM ERROR ON DICTIONARY ENTRY RECORD (ACCESSED THROUGH LOGICAL FILE CODE IN).
DP19	INVALID SPECIFICATION ON /REMAP DIRECTIVE.
DP20	INVALID SPECIFICATION ON /DPD DIRECTIVE.
DP21	UNRECOVERABLE ERROR ON DIRECTIVE INPUT (SYC) FILE.
DP22	DICTIONARY SIZE IS LESS THAN THE REQUIRED MINIMUM (FIVE BLOCKS).

4.3 Console Messages

The following messages are issued by DPEDIT to the operator console if logical file code IN is assigned to a card device. The "devmnc" specification gives the device mnemonic. After the message is issued, DPEDIT enters a program hold. To retry the read, reposition the deck in the reader and enter the OPCOM directive CONTINUE DPEDIT. If no retry is desired, enter the OPCOM directive ABORT DPEDIT.

DPEDIT devmnc CKSM

DPEDIT encountered a checksum error on the input (IN) file to the remap function.

DPEDIT devmnc SQER

DPEDIT encountered a sequence error on the input (IN) file to the remap function.

SECTION 5 - EXAMPLES

The following example saves several dictionaries:

```
$JOB DPEDIT1 OWNER
$ASSIGN DPD TO DPD1 BLOC=N
$ASSIGN OT TO DEV=MT ID=DPDS
$EXECUTE DPEDIT
/SAVE
/DPD DPD2
/SAVE
/DPD DPD3
/SAVE
$EOJ
$$
```

The following example remaps a dictionary:

```
$JOB DPEDIT2 OWNER
$ASSIGN DPD TO DPD1 BLOC=N
$ASSIGN IN TO DEV=MT ID=DPDS
$EXECUTE DPEDIT
/REMAP ,R
/VERIFY
/REMAP DPD2
/REMAP DPD3
/ENTER
A EQU $    (See Figure 2-1 for column placement)
$EOJ
$$
```

The following example expands, saves, and remaps a dictionary:

```
$JOB DPEDIT3 OWNER
$EXECUTE VOLMGR
EXTEND DPD1 EXTS=100
$ASSIGN OT TO DEV=MT ID=DPDT
$ASSIGN IN TO LFC=OT
$ASSIGN DPD TO DPD1 BLOC=N
$EXECUTE DPEDIT
/SAVE
/REMAP ,R
$EOJ
$$
```

The following example saves a dictionary on magnetic tape:

```
$JOB DPEDIT4 OWNER
$ASSIGN DPD TO DPD1 BLOC=N
$ASSIGN OT TO DEV=MT
$EXECUTE DPEDIT
/SAVE
$EOJ
$$
```

File Manager (FILEMGR)

MPX-32 Utilities

CONTENTS

FILE MANAGER (FILEMGR)

SECTION 1 - OVERVIEW

1.1 General Description

The File Manager (FILEMGR) utility provides a compatible mode of operation to facilitate converting files from pre-MPX-32 Release 2.x systems to Release 2.x and 3.x systems. FILEMGR also creates or deletes permanent disc files, global or Datapool partitions, and can be used to make permanent backup copies of system and user files. (Use of these functions is not recommended).

FILEMGR is the predecessor of the MPX-32 Volume Manager (VOLMGR). With the exception of converting pre-MPX-32 Release 2.x files, VOLMGR provides all of the functions of FILEMGR, plus additional capabilities. It is recommended that FILEMGR be used to convert files to MPX-32 Release 2.x and later systems, and VOLMGR be used for all other requirements.

FILEMGR directives are based on the current user volume, system volume, or current working directory. FILEMGR cannot be used to copy a file from one user directory into another; to do this, use the Volume Manager (VOLMGR).

Placement of files into a specific directory is done by changing the user directory in effect. The user name specified at logon establishes the current working directory. This can be changed by the TSM $USERNAME job control language statement or by the File Manager USERNAME directive:

```
TSM> $USERNAME name

FIL> USERNAME name
```

Device mnemonics and passwords are ignored by FILEMGR. Device and password parameters are shown in directive syntax statements for compatibility purposes only.

FILEMGR recognizes file names of 1 to 8 characters. Unless otherwise specified, files assigned to logical file codes will be forced to the appropriate format - blocked or unblocked.

1.2 Directive Summary

Following is a list of FILEMGR directives in alphabetical order. Each directive is explained in more detail in Section 3. A U appended to the directive specifies a user file. System files are assumed by default when a U is not specified.

Directive	Function
BACKFILE	Backspaces the magnetic tape assigned to LFC IN or OUT a specified number of EOF marks.
CREATE or CREATEU	Creates a permanent system or user file on the current working volume.
CREATEM	Defines a dynamic area of memory with a global common variable name (GLOBAL 00-99) or the name DATAPOOL. The defined area can be included in the user's logical address space by the M.INCLUDE system service. Can also define a memory partition in the user's extended address space.
DELETE or DELETEU	Deletes a specified system or user file and deallocates disc space.
DELETEW	Deletes more than one specified system or user file and deallocates disc space.
EXIT	In interactive mode, exits FILEMGR and returns control to TSM. In batch, designates the end of FILEMGR directives in a jobstream.
EXPAND or EXPANDU	Expands disc space of a permanent system or user file in 192-word blocks.
LOG, LOGU, or LOGC	Provides information about all system or user files.
PAGE	Puts a page eject and header on listed output.
RESTORE DEVICE or RESTOREU DEVICE	Restores all permanent system or user files to the current default volume/directory. If a file being restored does not already exist in the directory, it is added. If the file already exists, it is replaced.
RESTORE FILE or RESTOREU FILE	Restores permanent system or user files to disc from the device assigned to LFC IN (usually magnetic tape). (This was the device assigned to LFC OUT when the SAVE command was used.) If the files being restored already exist in the directory, existing contents are replaced by the IN contents. If the file does not exist, it is created.
REWIND	Positions an input file or device (LFC IN) or output file or device (LFC OUT) at its beginning.
SAVE DEVICE or SAVEU DEVICE	Saves all permanent system or user files from the current working volume/directory (except those created by SYSGEN).

Directive	Function
SAVE FILE or SAVEU FILE	Saves specified system or user files on device assigned to LFC OUT.
SAVELOG	Lists the files in the current working directory on LFC IN, beginning at the current location.
SKIPFILE	Advances past a specified number of EOFs on the file or device assigned to LFC IN or OUT.
USERNAME	Associates a new user name (current working directory) with FILEMGR operations.

SECTION 2 - USAGE

2.1 Accessing FILEMGR

FILEMGR can be accessed in the batch or interactive modes in one of three ways:

```
$FILEMGR
$RUN FILEMGR
$EXECUTE FILEMGR
```

$RUN FILEMGR is valid only from the system directory.

When accessing FILEMGR interactively, the FIL prompt is displayed:

```
TSM> $FILEMGR
FIL>
```

2.2 Saving, Restoring, and Creating Files

When files are saved, FILEMGR builds a directory containing directory entries for all files saved in a group. A group is one or more files specified with one SAVE or SAVEU directive. FILEMGR logs the current working and system directories and copies files in the group to an output medium. The data is copied after the directory.

When files are restored, a directory entry is created for each file to be restored. The directory itself must have been previously created. FILEMGR locates the file on the input medium and reads it to temporary space on disc. It matches the name against the user or system directory and deletes the existing file that matches the name. FILEMGR then creates a new permanent file in the directory for the file being restored. The existing user name from the saved version is used as the name.

Reading files into temporary space ensures that an I/O error in the restoration process does not result in the loss of existing disc files. This function can be bypassed by specifying option 2 as described in the Options section.

FILEMGR cannot create, save, restore, or delete temporary files. Temporary files are tracked by the system during task execution. FILEMGR cannot use them because they are not logged in the volume directory. Disc space for temporary files is allocated only for the duration of the task. FILEMGR does not allocate permanent files in space concurrently being used by temporary files.

Global common and Datapool partitions defined by FILEMGR are considered system files.

When saving or restoring files, a question mark (wild card character) can be used in place of any character. Since file names can contain one to eight characters, one to eight question marks can be used in specifying a file name.

For example, using five question marks as a file name saves all files with five or less characters. Using PJ?????? as a file name saves all files beginning with PJ.

If more than one prototype is specified in a directive line and a wild card character is used as part of the second prototype, the following message is displayed when FILEMGR processes the second prototype if the same file matches both prototypes:

```
WARNING PROTOTYPE filename NOT MATCHED BY ANY FILE
```

If a file name specified with a SAVE or RESTORE directive contains any of the following characters, the name must be enclosed in single quotes:

```
:
;
(
)
/
```

For example:

```
SAVEU FILE='EM:02'
```

Files with special characters can only be accessed by interfaces that accept special characters.

2.3 Computing the Size of a File

When a file is created by the CREATE or CREATEU directives, the initial file size is defined in blocks. A block is 192 words (768 bytes). In unblocked files, records are stored one per block.

The maximum record size for a blocked file is 254 bytes. A guide for approximating the space required for a blocked file is:

- Records between 4 and 254 bytes long (1 and 53 words) are packed together up to a block boundary.
- Records cannot span block boundaries.
- A new file always begins on a block boundary.
- Two header and two trailer bytes are automatically inserted on each packed record for identification and tracking.

In computing the space needed for blocked files, allow four extra bytes in each packed record. Fixed length records less than 254 bytes take up blocks on the file as follows:

$$\frac{\text{768 (bytes per block)}}{\text{Record Length (bytes)} + 4} = \text{Number of Records per Block}$$

$$\frac{\text{Number of Records}}{\text{Number of Records per Block}} = \text{Number of Blocks}$$

For example, if each record is 80 bytes long, each block holds nine records (768/84). To hold 2000 records, a file must be 223 blocks long (2000/9, rounded).

The number of blocks required to accommodate variable length records can be estimated by figuring an average byte/record value and using that value as the record length in the formula. For a file with approximately 150 variable-length records, none exceeding 254 bytes, and the average about 50 bytes long, the computation would be:

$$\frac{768}{50+4} = 14 \qquad \frac{150}{14} = \text{11 blocks (rounded)}$$

Output to all disc files is assumed to be in blocked format unless otherwise specified when assigning or allocating a file.

2.4 The System Master Directory (SMD)

The presence of a System Master Directory (SMD) is emulated to allow conversion of pre-MPX-32 Release 2 files to MPX-32 Release 2.x and 3.x. The volume directory on each disc contains entries for all permanent files and global common/Datapool partitions located on that disc. Each entry is reconstructed, or defaults are used, to construct a compatible SMD entry that shows:

. File name/partition name (limited to eight characters)

. User name, if any (limited to eight characters, defaults to current working directory)

. Beginning address for the file (block number)

. Password (is ignored or set to zero)

. Number of 192-word blocks in the file or protection granules in the partition (this is the sum of all segments in the file)

. Access speed (FAST/SLOW), disc type, channel, subaddress, and other information used by the system when the file or partition is accessed.

2.5 Logical File Code Assignments

Default logical file code (LFC) assignments are provided for all FILEMGR operations except the input and output assignments for SAVE and RESTORE operations. For these operations, the logical file codes IN (for RESTORE input) and OUT (for SAVE output) are provided, but the files or devices assigned to them must be specified.

2.5.1 Source Input (SYC)

The source input file contains FILEMGR directives. The source input file is assigned to logical file code SYC.

SYC Default and Optional Assignments

The default assignment for SYC is to the System Control file (SYC):

```
    $ASSIGN SYC TO SYC
```

In the interactive mode, source is input from the user terminal. In the batch mode, source is input from the SYC file.

There are two optional assignments for SYC:

```
    $ASSIGN SYC TO  { pathname    }
                    { DEV=devmnc  }
```

pathname	is the pathname of the file containing FILEMGR source input
devmnc	is the device mnemonic of a device containing FILEMGR source input

2.5.2 Listed Output (SLO)

The listed output file contains an audit trail of FILEMGR activity. The file or device to be used for listed output is assigned to logical file code SLO.

SLO Default and Optional Assignments

The default assignment for SLO is to logical file code UT:

```
            $ASSIGN SLO TO LFC=UT
```

In the interactive mode, output is generated on the user terminal. In the batch mode, output is generated on the SLO device.

There are two optional assignments for SLO:

```
            $ASSIGN SLO TO   { pathname    }
                             { DEV=devmnc  }
```

pathname	is the pathname of a file to contain the listed output. The file must have been previously created.
devmnc	is the device mnemonic of a device to contain the listed output

2.5.3 Input for Restores (IN)

Logical file code IN is used to specify the input file or device from which to restore files that were previously saved by a SAVE directive. Logical file code IN should be the same file or device defined for logical file code OUT when the files were saved. The IN assignment is usually a tape from which files are to be restored. Logical file code IN is forced unblocked by FILEMGR.

IN Default and Optional Assignments

There is no IN default assignment.

These are two optional assignments for IN:

$$\text{\$ASSIGN IN TO} \begin{Bmatrix} \text{pathname} \\ \text{DEV=devmnc} \end{Bmatrix}$$

pathname	is the pathname of a file containing files to be restored
devmnc	is the device mnemonic of a device containing files to be restored

2.5.4 Output for Saves (OUT)

Logical file code OUT is used to specify the file or device on which to save files. The OUT assignment is usually to a magnetic tape. Logical file code OUT is forced unblocked by FILEMGR.

OUT Default and Optional Assignments

There is no OUT default assignment.

There are two optional assignments for OUT:

$$\text{\$ASSIGN OUT TO} \begin{Bmatrix} \text{pathname} \\ \text{DEV=devmnc} \end{Bmatrix}$$

pathname	is the pathname of a file to contain saved files
devmnc	is the device mnemonic of a device to contain saved files

2.5.5 LFC Summary

The following is a table of the LFCs used by FILEMGR and their default and optional assignments.

Table 2-1
FILEMGR LFC Summary

LFC	Default Assignment	Optional Assignment
IN	N/A	pathname DEV=devmnc
OUT	N/A	pathname DEV=devmnc
SLO	LFC=UT	pathname DEV=devmnc
SYC	SYC	pathname DEV=devmnc

2.6 File-to-Tape Transfers

All SAVEs in a FILEMGR session apply to the magnetic tape or set of tapes assigned to logical file code OUT prior to executing FILEMGR. A set of tapes is implied by indicating multivolume on the device assignment. For example:

```
    $AS OUT TO DEV=MT MULT=1 ID=SAVE
```

When FILEMGR is ready to execute, it issues a mount message on the operator's console prompting the operator to mount the appropriate tape. When the tape is mounted and the operator responds on the console, FILEMGR processing continues.

SAVEs and RESTOREs must be coordinated in the following way:

. A SAVE or SAVEU directive produces a group of one or more files on tape with one EOF mark at the end of the group.

. If a RESTORE directive in a subsequent session selects files or a group of files from a tape that contains several groups, FILEMGR must be informed of the physical location on the tape.

. FILEMGR assumes a sequential restoration in the order that files were saved. If files are restored outside the order in which they were saved, FILEMGR REWIND and/or BACKFILE directives must be used to position the input device.

Figure 2-1 illustrates the physical result of multiple SAVE/RESTORE operations.

The left side of the figure illustrates SAVEs used to output disc files to a magnetic tape. The right side illustrates how RESTOREs could be used to restore the files back to disc.

In the figure, one tape contains five groups of files, each with a separate set of directory entries. All files are saved for User A. One file is saved for User B, followed by files from two other users. USERNAME with no following parameter specifies system files. All system files are saved, then all User C files that begin with CC are saved.

User A's file, ONEFILE, is restored. FILEMGR then goes past the EOF marking the end of User A's files. To restore the system files, the SKIPFILE directive is used to move past two EOFs to the beginning of the system files.

Restoring User C's files requires no special directive because FILEMGR is already positioned at the beginning of that group.

The REWIND or BACKFILE directives are used if files are not restored in the same order in which they were saved. <u>Do not use REWIND in the middle of a multivolume restoration.</u>

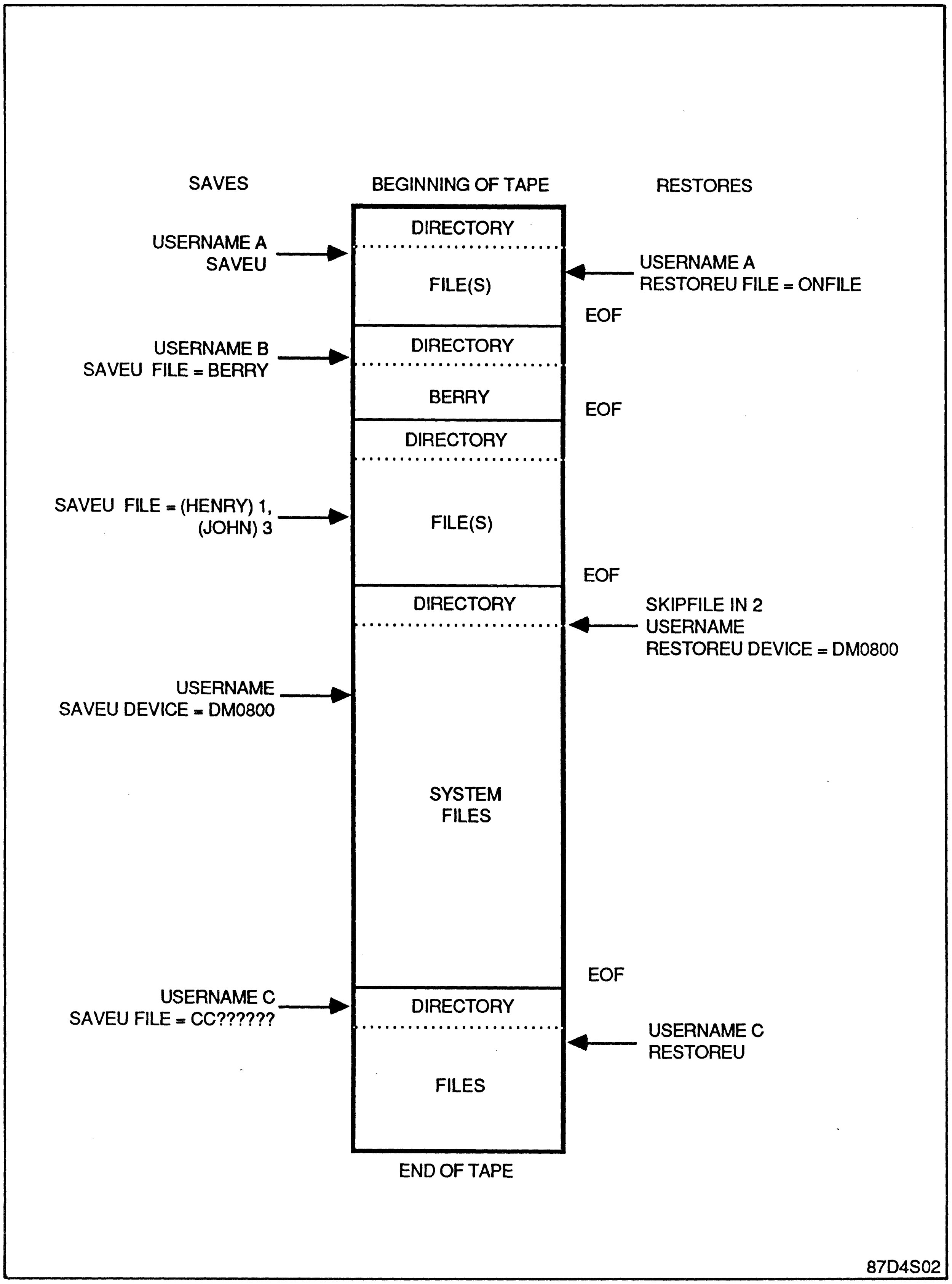


Figure 2-1. File-to-Tape Transfers

2.7 Options

FILEMGR options are specified by number on a TSM $OPTION job control language statement. The $OPTION statement must appear before the $FILEMGR statement in a job stream. FILEMGR options affect processing of SAVE, SAVEU, RESTORE, RESTOREU, and SAVELOG directives.

Option	Description
1	Tape Assigned to LFC IN is Pre-MPX-32 All files restored are assumed to have eight-word, RTM 6.x or later formatted SMD entries. Eight-word MPX-32 formatted entries are written to the directory.
2	Delete Existing File Before Restoring Normally, when restoring files, LFC IN is written first to temporary disc file space. This option causes FILEMGR to delete the existing disc file specified for the restore before copying the saved file back to disc from LFC IN.
3	Save NOSAVE Files Overrides the NOSAVE attribute specified when a file is created and allows NOSAVE files to be saved.
4	Change User Name to Current Working Directory Allows all files on the save tape to be restored to the current working directory in effect. This overrides the user name associated with the file when it was saved on tape.
5	Change User Name to System Allows all files on the save tape to be restored as system files. This overrides the user name associated with the file when it was saved on tape.

2.8 Exiting FILEMGR

To exit FILEMGR from the batch and interactive modes, specify the EXIT directive.

SECTION 3 - DIRECTIVES

3.1 Introduction

FILEMGR directives are summarized in the Overview section and explained in detail in alphabetical order in this section.

FILEMGR directives cannot be abbreviated and must begin in column one. Several directives and their associated parameters can be specified on one line by separating them with commas. Blanks are ignored.

User files are denoted specifically by a U as part of the directive (SAVEU, RESTOREU, CREATEU). If U is specified, FILEMGR first searches the directory of the user name last specified in a TSM $USERNAME job control statement or a FILEMGR USERNAME directive. If neither type of USERNAME statement was specified, the operation defaults to the current working directory.

System files are assumed by default when a directive does not end in a U and if a USERNAME statement is specified without supplying a user name.

3.2 BACKFILE Directive

The BACKFILE directive backspaces the magnetic tape assigned to logical file code IN or OUT a specified number of EOF marks. Use BACKFILE when restoring files out of the order in which they were saved. Because of characteristics of unblocked disc files, BACKFILE is not valid for assignments to disc files.

Syntax:

$$\text{BACKFILE} \begin{Bmatrix} \text{IN} \\ \text{OUT} \end{Bmatrix} [,n]$$

IN	specifies the device assigned to logical file code IN
OUT	specifies the device assigned to logical file code OUT
n	specifies the number of files to backspace. If not specified, the default is one.

3.3 CREATE and CREATEU Directives

The CREATE directive allocates file space for a system file. For CREATEU, the file is created in the current working directory or the directory of the user whose name was most recently specified in a USERNAME statement. If CREATEU is used and no user name is associated for FILEMGR operations, or if a USERNAME statement with no following parameter has been specified, a system file is created.

Syntax:

```
CREATE [U] filename , [devmnc], blocks , [type], ┌ FAST ┐ ,  [NZRO], [NSAV]
                                                  └ SLOW ┘
          ┌   ┌PO┐              ┐
          │ , │  │ , password   │
          └   └RO┘              ┘
```

filename	is a one to eight character file name
devmnc	allocates a disc file. Defaults to DC for Release 2.x compatible mode even if disc device code, channel, and subaddress are specified.
blocks	the initial increment size of the file. Specifies the number of 192-word blocks to allocate for the file.
type	specifies the hexadecimal equivalent of a two-character ASCII code to display or print with the file name. If not specified, the default is 00. Files containing load modules must be type CA.
FAST	is ignored
SLOW	is ignored
NZRO	suppresses initializing the disc file space to zero. If not specified, the default is to initialize the file space to zero.
NSAV	suppresses saving the file when all files on the disc are saved by a SAVE DEVICE directive. If not specified, the default is to save the file.
RO	allows write access by the owner of the file. All other users have read only access.
PO	is ignored
password	is ignored

NOTE: This directive is provided for compatibility. Its use is not recommended; instead use the VOLMGR CREATE FILE directive.

3.4 CREATEM Directive

The CREATEM directive defines a global common partition, a Datapool partition, or a partition in the user's extended address space (above the 128KW logical address space mapped for each task). Memory partitions defined by FILEMGR are dynamically allocated when required by a task. They do not remain allocated in physical memory regardless of use as do SYSGEN-defined partitions. To use a memory partition defined by FILEMGR, tasks must use the M.INCLUDE system service.

A partition defined by FILEMGR is 2KW minimum on a CONCEPT/32 computer or 8KW on a 32/7x series computer. A SYSGEN-defined partition is structured in protection granule increments of 512 words per protection granule.

A partition name can be created only once. If created through SYSGEN, a partition name cannot be created again with FILEMGR. MPX-32 has the ability to multicopy partition space into more than one logical address space.

Syntax:

```
                 (GLOBALnn )                          [[E]  [ (RO)            ]]
      CREATEM   < DATAPOOL  > ,protgran ,firstpage ,[[H]  [,<PO> , password ]]
                 (extname  )                          [[S]  [ (  )            ]]
```

GLOBALnn creates a global common partition (00-99) which can be physically located in any class of memory (E, H, or S) and is mapped into the address space of each task that accesses it through the M.INCLUDE system service

DATAPOOL creates a Datapool partition whose structure is defined by one or more Datapool dictionaries. Like global common, the Datapool area can be physically located in any class of memory (E, H, or S) and is mapped into the logical address space of each task that accesses it. The first task calling M.INCLUDE defines the partition as sharable.

extname is a one to eight character name of a memory partition in a task's extended address space. This partition may be mapped into memory above the first 128KW logical address space available to a task. Since the partition is in extended memory, certain restrictions apply. Refer to the MPX-32 Reference Manual.

Partitions in a task's extended address space can be located in any class of physical memory (E, H, or S).

The name used for a partition that is allocated in extended address space must not be GLOBALnn.

protgran specifies the number of 512-word protection granules to include in the partition. (Four protection granules equal one map block on a CONCEPT/32 computer; sixteen protection granules equal one map block on a Series 32/7x computer.) Unused physical protection granules within the last map block allocated to the partition are write-protected from all sharing tasks. Only one partition may be defined in any one map block.

firstpage	specifies the starting protection granule in the nonextended logical address space (pages 0 to 255) or in the extended address space (pages 256 to 495 on a Series 32/7x or pages 256 to 1019 on a CONCEPT/32) where the partition is to be mapped. Protection granules in the first several map blocks should not be specified because they are used for the MPX-32 operating system. Protection granules for global and datapool partitions are normally allocated from the top down in a task's logical address space, or below any SYSGEN-created common partitions.
E	allocates the partition in the first 128KW. If not available, the partition is queued in the Memory Request Queue (MRQ) until class E becomes available.
H	allocates the partition in class H (high speed) or E memory. If H and E are not available, the partition is queued in the MRQ until class H or E becomes available.
S	allocates the partition in class S (slow), H, or E memory. If no memory is available, the partition is queued in the MRQ until memory becomes available.
RO	allows write access to the owner of the file. All other users have read only access.
PO	is ignored
password	is ignored

NOTE: This directive is provided for compatibility only. Its use is not recommended; instead use the VOLMGR CREATE COMMON directive.

3.5 DELETE and DELETEU Directives

The DELETE and DELETEU directives delete files from disc and free the disc space. When a file is deleted, its directory entry is removed. The DELETE directive deletes system files. The DELETEU directive deletes user files from the current working directory or the directory of the user whose name was specified in a TSM $USERNAME statement or FILEMGR USERNAME directive.

Syntax:

```
DELETE[U] filename [,password]
```

filename	specifies a one to eight character file name. The name cannot contain blanks or wild card characters.
password	is ignored

To use the DELETE command, the user name in effect must have delete access to the file to be deleted and delete entry access to the directory where the file is located.

NOTE: This directive is provided for compatibility only. Its use is not recommended; instead use the VOLMGR DELETE FILE directive.

3.6 DELETEW Directive

The DELETEW directive deletes more than one file per directory from the disc, frees disc space, and removes the directory entry for each deleted file from the current working directory. Up to 20 file prototypes can be specified per directive. A directive can be continued on several lines. Each line must contain a comma as the last nonblank character.

There are no defaults for the DELETEW directive. For each file to be deleted, the word SYSTEM or the user name of the current working directory and the file name must be specified (see the Examples section for sample use of this directive).

Syntax:

```
    DELETEW [FILE=] prototype [,prototype...]
```

prototype identifies a file as follows:

```
    (username [,key] ) ['] filename ['] [;password]
```

username,key	user name and optional key must be the same as the current working directory and must be enclosed in parentheses. The name SYSTEM can be used to specifically indicate system files and must be enclosed in parentheses.
filename	specifies a one to eight character file name. Wild card characters (?) are allowed.
;password	is ignored

NOTE: This directive is provided for compatibility only. Its use is not recommended; instead use the VOLMGR DELETE FILE directive.

3.7 EXIT Directive

The EXIT directive exits FILEMGR and returns control to TSM when running in the interactive mode.

When running in the batch mode, the EXIT directive signifies the end of FILEMGR directives in a jobstream.

Syntax:

```
    EXIT
```

3.8 EXPAND and EXPANDU Directives

The EXPAND and EXPANDU directives increase the size of an existing file. If the file space is increased in size, the additional space is zeroed if the file was not created with the NZRO attribute.

Syntax:

```
    EXPAND[U]  filename, blocks [,password]
```

filename	specifies a one to eight character file name. The file name cannot contain blanks or wild card characters.
blocks	is the new size of the file. Specifies the number of 192-word blocks to allocate for the file.
password	is ignored
NOTE:	This directive is provided for compatibility only. Its use is not recommended; instead use the VOLMGR EXTEND directive.

3.9 LOG, LOGU, and LOGC Directives

The LOG and LOGU directives provide information about all permanent files defined in the system or user directory or a subset of files. Output includes the file name, directory name, device on which the file resides, starting address of the file, file size, and file type. If no parameters are specified, the output contains data on all permanent files in the system directory (for LOG) or current working directory (for LOGU).

Syntax:

$$\text{LOG} \begin{bmatrix} \text{U} \\ \text{C} \end{bmatrix} \text{[[FILE=] prototype] [,prototype ...]}$$

C	provides same results as if no parameters were specified
FILE=	limits the list to a specific file or set of files
prototype	identifies files as described in the DELETEW directive. Up to 20 file prototypes per directive can be specified. A directive can be continued on several lines or cards if each line or card contains a comma as the last nonblank character.

3.10 PAGE Directive

The PAGE directive forces a page eject and prints a header on the listed output. A header is automatically printed on the first page of the listed output without specifying the PAGE directive.

Syntax:

```
    PAGE
```

3.11 RESTORE and RESTOREU Directives

The RESTORE and RESTOREU directives copy files saved by a SAVE or SAVEU directive back to disc. Assign the medium that was assigned to logical file code OUT during the save to logical file code IN to restore the files. The RESTORE and RESTOREU directives can be used to restore:

. All system and/or user files assigned to logical file code IN to the system or a user directory on the current working volume

. A subset of system and/or user files

When specifying a list of prototypes, files from other user names can also be restored.

Syntax:

```
    RESTORE[U] [ DEVICE=devmnc ] , [ FILE= prototype ] [ ,prototype ... ]
```

DEVICE=	restores all files from logical file code IN to the system or user directory on the current working volume
devmnc	is ignored
FILE=	limits the restoration to a specific file or set of files
prototype	identifies files as described in the DELETEW directive. Up to 20 file prototypes per directive can be specified. A directive can be continued on several lines or cards if each line or card contains a comma as the last nonblank character.
	If no parameters are specified, all files from logical file code IN are restored to the system or user directory on the current working volume.

Usage:

FIL> **RESTORE**

This example restores all files from logical file code IN to the system or user directory.

FIL> **RESTORE DEVICE=DM0800**

This example restores all system and user files from logical file code IN to the current system or user directory.

3.12 REWIND Directive

The REWIND directive rewinds a file or device.

FILEMGR does not rewind automatically after saves or restores. If a tape has not been rewound off-line, the REWIND directive should be used.

Syntax:

$$\text{REWIND} \left\{ \begin{array}{l} \text{IN} \\ \text{OUT} \end{array} \right\}$$

IN	specifies the device assigned to logical file code IN
OUT	specifies the device assigned to logical file code OUT

3.13 SAVE and SAVEU Directives

The SAVE and SAVEU directives copy permanent disc files to the medium assigned to logical file code OUT. Normally files are saved on and restored from magnetic tape. SAVE is usually used for creating backup copies.

The SAVE and SAVEU directives can be used to copy:

. All system files and/or all files belonging to a particular user
. A subset of system and/or user files

If no user name is associated with FILEMGR operations, or if a USERNAME statement with no following parameter has been specified, system files are saved.

As files are saved, FILEMGR builds a compatible SMD entry containing the same information as the pre-MPX-32 Release 2 SMD. The SMD entry is output at the beginning of each group of files saved on the medium assigned to logical file code OUT. An error message and a zero-filled block on logical file code OUT indicating an end-of-file (EOF) is produced if a SAVE directive is specified and no files are saved.

An audit trail of all files saved in a particular FILEMGR session is listed automatically on the device assigned to logical file code SLO. The files are listed in the order in which they were saved.

Syntax:

$$\text{SAVE [U]} \left[\left\{ \begin{array}{l} \text{[DEVICE=devmnc]} \\ \text{[FILE=] prototype [,prototype ...]} \end{array} \right\} \right]$$

If no parameters are supplied, all system and user files (SAVE) or all user files in the current working directory (SAVEU) are saved on logical file code OUT.

DEVICE=	saves all files from the system or current working directory
devmnc	is ignored
FILE=	limits the save to a specific file or set of files.
prototype	identifies a file as described in the DELETEW directive. Up to 20 file prototypes per directive can be specified. A directive can be continued on several lines or cards if each line or card contains a comma as the last nonblank character.

Usage:

FIL> **SAVE**

This example saves all system and user files in the system and current working directories.

3.14 SAVELOG Directive

The SAVELOG directive lists the files grouped in the current directory that resides on the tape assigned to logical file code IN. After the files are listed, the tape returns to the beginning of the current directory.

The SAVELOG directive is useful during restoration because it allows matching of RESTORE directives against the actual saved files on a tape. Checking the contents of a tape also ensures that the right tape is mounted.

If a tape contains several directories, the SKIPFILE directive can be used to get to and list the next directory. For example, if a tape has three directories, the following directives:

```
SAVELOG
SKIPFILE IN
SAVELOG
SKIPFILE IN
SAVELOG
REWIND IN
```

output all directory entries to the device assigned to logical file code SLO. If SAVELOG is inserted between the RESTORE directives, each directory list precedes the RESTORE operations shown on the SLO device.

Syntax:

```
SAVELOG
```

3.15 SKIPFILE Directive

The SKIPFILE directive advances past one or more end-of-file (EOF) marks on the file or device assigned to the logical file code IN or OUT.

Syntax:

$$\text{SKIPFILE} \quad \left\{ \begin{matrix} \text{IN} \\ \text{OUT} \end{matrix} \right\} \; [,n]$$

IN	specifies the device assigned to logical file code IN
OUT	specifies the device assigned to logical file code OUT
n	specifies the number of EOFs to skip. If not specified, the default is one.

For sample use of this directive, see Figure 2-1 and the SAVELOG directive description.

3.16 USERNAME Directive

The USERNAME directive names the directory to be used for subsequent FILEMGR operations.

If running from a terminal in TSM, the initial user name defaults to the directory name established at logon. The initial user name can be changed by supplying a different name in a USERNAME directive.

Syntax:

```
USERNAME [username] [,key]
```

username	specifies the name of a directory on the current working volume. If no name is supplied, defaults to system files.
key	is ignored

SECTION 4 - ERRORS AND ABORTS

4.1 Abort Codes

FILEMGR generates the following abort codes and messages:

Code	Message
FM13	UNRECOVERABLE I/O ERROR TO THE DIRECTORY
FM14	UNRECOVERABLE I/O ERROR TO THE SYC FILE
FM15	UNRECOVERABLE I/O ERROR TO THE SLO FILE
FM16	UNRECOVERABLE I/O ERROR TO THE 'IN' FILE
FM17	UNRECOVERABLE I/O ERROR TO THE 'OUT' FILE
FM20	UNRECOVERABLE I/O ERROR ON SAVE, RESTORE, OR EXPAND FILE
FM41	END-OF-MEDIUM ON LFC SLO
FM42	INVALID USER NAME
FM99	ERROR(S) (DESCRIBED ON LFC SLO) DETECTED DURING EXECUTION

4.2 Error Messages

The following are the possible error messages for errors which lead to an FM99 abort code.

INSURE BOUNDING

INVALID COMMAND VERB

REQUIRED ARGUMENTS ARE ABSENT

REQUEST IGNORED - FILE ALREADY EXISTS

INVALID DEVICE SPECIFICATION

INVALID NUMERIC SPECIFICATION

SPECIFIED FILE IS ACCESS PROTECTED

REQUEST TO EXPAND MEMORY PARTITION

REQUEST IGNORED - SPACE UNAVAILABLE

REQUEST IGNORED - COLLISION MAPPING

UNABLE TO DELETE FILE - VMxx

INSUFFICIENT FAT SPACE FOR SMD

INVALID ARGUMENT

```
CANNOT ALLOCATE REQUIRED RESOURCE -  VMxx
                                     RMxx
```

UNEXPECTED EOF ON IN FILE

FILE SPECIFIED NOT FOUND

OVER 20 PROTOTYPES SPECIFIED IN COMMAND

EOF EXPECTED - NOT FOUND

ERROR ENCOUNTERED WHILE MAKING TEMP FILE PERMANENT

UNABLE TO ALLOCATE SCRATCH FILE

INVALID CHARACTERS IN FILENAME

ERROR IN SORTING LOG LISTING

```
WARNING:  FILE (username) filename -- IS "NO SAVE".  OPTION #3
                                      MUST BE SPECIFIED TO   {SAVE   }
                                                             {RESTORE}
```

WILD CARD DELETE (DELETEW) MUST HAVE ARGUMENTS

USER NAME MUST BE EXPLICITLY STATED WITH WILD CARD DELETE (DELETEW)

FILE NAME(S) NOT SPECIFIED

END-OF-MEDIUM ENCOUNTERED ON LOGICAL FILE CODE "lfc"

NAMES "GLOBAL" AND "DATAPOOL" INVALID IN EXTENDED MEMORY

NO DIRECTORY ENTRIES ARE AVAILABLE

UNABLE TO ACCESS RESOURCE IN REQUIRED MODE. ACCESS RIGHTS VIOLATION.

UNABLE TO CREATE MEMORY PARTITION.
ERROR = VMxx REPLACED WITH ASCII NUMBER

INVALID COMMAND - USE VOLUME MANAGER TO CREATE DIRECTORIES

INVALID COMMAND - USE VOLUME MANAGER TO CREATE SDT TAPES

UNABLE TO EXPAND FILE - VMxx

SECTION 5 - EXAMPLES

This section provides sample programming sequences illustrating the use of various FILEMGR directives.

The following example converts 2.x/3.x disc files to 1.x tape files. FILEMGR saves all files in the current working directory to tape.

```
TSM>AS OUT TO DEV=M9
TSM>FILEMGR
FIL>SAVEU
FIL>EXIT
TSM>
```

The following batch example restores 1.x tape files to 2.x/3.x disc files. FILEMGR restores all JJ. files with five or less trailing characters and a username of DIRABC to disc in directory DIRABC.

```
$JOB EXAMPLE2 OWNER
$AS IN TO DEV=M9
$FILEMGR
 USERNAME DIRABC
 RESTORE FILE=(DIRABC)JJ.?????
$EOJ
$$
```

Macro Assembler (ASSEMBLE)

MPX-32 Utilities

CONTENTS

MACRO ASSEMBLER (ASSEMBLE)

SECTION 1 - OVERVIEW

1.1 General Description

The Macro Assembler (ASSEMBLE) utility translates Macro Assembler source code into standard non-base object programs for execution. Assembler source code consists of Assembler language instructions and Macro Assembler directives. The Assembler language instruction set is described in the CPU reference manual that corresponds to the machine type being used.

Macro Assembler directives provide capabilities for program control, symbol and data definition, listed output, and macro support. Macro Assembler source statement format is described in Section 3.

The Assembler recognizes 1 to 16 character file names. Unless specified, files assigned to logical file codes will be forced to the appropriate format-blocked or unblocked.

1.1.1 Macro Assembler Features

General operating features and capabilities of the Macro Assembler include:

. A comprehensive set of Assembler directives.

. Mnemonic operation codes for all directives and instructions.

. Symbolic addressing.

. Decimal (integer and real), hexadecimal, and character representation of machine language binary values.

. Programs may be arbitrarily grouped into logical sections (subroutines) which may be assembled separately and combined into one executable program at load time. Linkage information between sections is provided through the EXT and DEF Assembler directives.

. Relocatable object programs.

. Macro instruction support.

. System services support.

. Listed output of source program and resulting object code.

1.1.2 Macro Assembler Operation

The Macro Assembler translates Assembler source program statements into binary-equivalent machine instructions, equates symbols to numeric values, assigns relocatable or absolute memory addresses for program instructions and data, and generates listed output.

After a source program has been assembled, the object module output can be placed in a subroutine library or a permanent file, output to a device medium, or cataloged into a load module. Object modules can be linked together to form a single task by assembling and cataloging the modules in the same job, by accessing the Subroutine Library during a separate Cataloger run, or by using a $SELECTx job control statement in batch mode prior to cataloging the object modules.

1.2 Directive Summary

Macro Assembler directives provide program control, symbol and data definition, listed output control, conditional assembly, macro support, and special usage functions. Directives are summarized by function below. Section 4 contains detailed directive descriptions in alphabetical order. Extended mnemonic instructions are listed in Appendix B.

Program Control

ABS	Assembles source code in absolute mode (CATALOG does not process)
REL	Assembles source code in relocatable mode
CSECT	Assembles source code in code section mode
DSECT	Assembles source code in data section mode
ORG	Assigns a value to the location counter
BOUND	Advances the location counter to represent a byte multiple of a specified value
RES	Reserves memory locations
REZ	Reserves and zeroes memory locations
END	Indicates the end of Assembler source code

Symbol Definition

EQU	Defines a symbol by equating it to an expression
EXT	Declares an external reference
DEF	Declares an external definition

Data Definition

GEN	Constructs a hexadecimal value by generating a bit pattern
DATA	Generates a value
AC	Generates an address constant

Conditional Assembly

ANOP	No operation
GOTO	Branches unconditionally
Computed GOTO	Branches conditionally based on indexed argument list
IFF	Branches if the specified expression is evaluated as true
IFT	Branches if the specified expression is evaluated as false
SET	Assigns the value of an expression to a label
SETF	Assigns the value false to a label
SETT	Assigns the value true to a label

Listed Output Control

PAGE	Causes a page eject on listed output
SPACE	Skips lines on listed output
TITLE	Prints a title at the top of each page of listed output
LIST	Controls listed output by requesting or inhibiting parts of source output

Macro Support

DEFM	Defines a macro by specifying its name and arguments
ENDM	Terminates a macro definition
EXITM	Terminates processing of a macro structure
IFA	Branches on presence of arguments
IFP	Branches on absence of arguments

Special Usage

PROGRAM	Specifies the name of an Assembler program
COMMON	Defines, manipulates, and initializes common communication areas.
LPOOL	Inserts literals into object code
REPT	Generates a repeat loop
ENDR	Terminates a repeat loop
FORM	Defines variable length data subfields

SECTION 2 - USAGE

2.1 Accessing the Macro Assembler

The Assembler can be accessed in batch or interactive mode in one of three ways:

```
$ASSEMBLE
$RUN ASSEMBLE
$EXECUTE ASSEMBLE
```

$RUN ASSEMBLE is valid only from the system directory.

When accessing the Assembler interactively, the ASS> prompt is displayed:

```
TSM > $ASSEMBLE
ASS >
```

2.2 Logical File Code Assignments

Logical file code (LFC) assignments are made by the $ASSIGN job control language statement. In an Assembler program, place the assignment statements before the $ASSEMBLE statement. If no assignments are made, the Assembler uses default assignments.

Figure 2-1 shows the Assembler's flow of operation and default assignments for logical file codes.

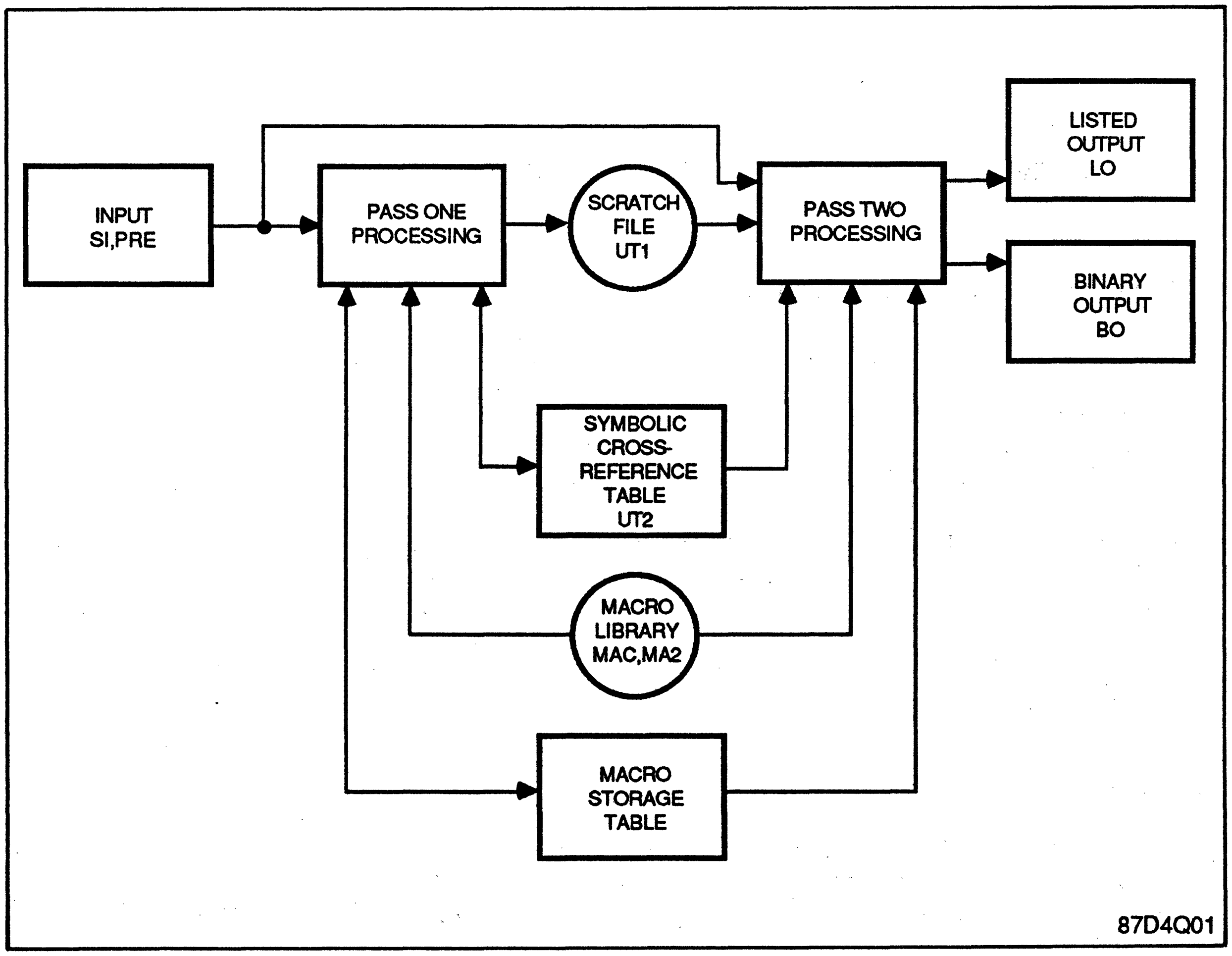


Figure 2-1. Macro Assembler Flow of Operation

The default assignments are not valid in independent mode. If an LFC is assigned to a file, that file must have been previously created.

During Pass One, the Assembler scans the source code for macro calls, referenced symbols, and errors which will not be resolved in Pass Two. Unresolvable errors include multiply defined symbols and illegal operations.

If the source program contains macro calls or referenced symbols, the Assembler dynamically establishes macro storage and symbol cross-reference tables in memory. Additional memory for these tables is allocated when they are accessed during Pass Two. However, the Assembler deallocates statically allocated memory before dynamically allocating additional memory.

During Pass Two, the Assembler reads the source code from the scratch file created during Pass One. The macro storage and symbol cross-reference tables are accessed to resolve macro and symbol references. If necessary, a macro library is also searched.

After the scratch file is processed, the output is sent to the device or file assigned for listed or binary output.

The following sections describe the LFC assignments used by the Assembler.

2.2.1 Source Input (PRE and SI)

Source code is assigned to logical file codes PRE and SI for input. Source code can be input from any device or file and can be in compressed format. User program source code should be assigned to SI. Source code containing nonexecutable Assembler directives (such as SET directives) can be assigned to PRE. During processing, the source code assigned to PRE is input until an end-of-file (EOF) is reached, then the source code assigned to SI is input.

The Macro Assembler accepts source input in upper and lower case. All input except the text specified in a TITLE directive, G-character strings, and C-character strings are converted to upper case by the Macro Assembler.

SI Default and Optional Assignments

The default assignment for Assembler source input is:

```
    $ASSIGN SI TO SYC
```

There are two optional assignments for SI:

```
    $ASSIGN SI TO  {pathname   }
                   {DEV=devmnc }
```

pathname	is the pathname of the file containing Assembler source input
devmnc	is the device mnemonic of a device containing Assembler source input

PRE Default and Optional Assignments

There is no default assignment for PRE.

There are two optional assignments for PRE:

```
    $ASSIGN PRE TO  {pathname   }
                    {DEV=devmnc }
```

pathname	is the pathname of the file containing nonexecutable Assembler source input
devmnc	is the device mnemonic of a device containing nonexecutable Assembler source input

2.2.2 Macro Libraries (MAC and MA2)

If macros are called in the source program, the Assembler searches a macro library during Pass One processing. The system macro library (M.MPXMAC) provides a collection of macro definitions which can be used by source programs. Users can also add macros to the library using the MACLIBR utility.

In addition to M.MPXMAC, the Assembler can support another macro library referenced by logical file code MA2. If assigned, MA2 is searched before MAC and the permanent symbol table for all names that appear in the opcode/instruction field of Assembler statements. Assignment to MA2 is useful for overriding an existing opcode or Assembler directive. However, use of MA2 is not recommended unless this override feature is needed.

Logical file codes MAC and MA2 are forced unblocked by the Macro Assembler.

MAC Default and Optional Assignments

The default assignment for MAC is to the system macro library (M.MPXMAC):

```
$ASSIGN MAC TO @SYSTEM(SYSTEM)M.MPXMAC
```

There are two optional assignments for MAC:

```
$ASSIGN MAC TO{@SYSTEM(SYSTEM)M.MACLIB}
              {pathname               }
```

M.MACLIB	contains RTM-compatible macros
pathname	is the pathname of the file containing the macro library

MA2 Optional Assignment

There is no default assignment for MA2 and one optional assignment:

```
$ASSIGN MA2 TO pathname
```

pathname	is the pathname of the file containing the macro library

If this assignment is made, the library assigned to MA2 is searched before the system macro library.

2.2.3 Temporary Files (UT1 and UT2)

Logical file code UT1 is used to store the source and expanded macro text for processing during Pass Two. In Pass One, the Assembler writes the text to UT1; in Pass Two, the Assembler reads UT1.

Logical file code UT2 is used for the symbol cross-reference table during assembly.

UT1 and UT2 Default Assignments

Both UT1 and UT2 are assigned to temporary scratch files. UT1 is forced to blocked format; UT2 is forced to unblocked format. The default assignments for UT1 and UT2 are:

```
$ASSIGN UT1 TO TEMP SIZE=800

$ASSIGN UT2 TO TEMP SIZE=400
```

The file size in the SIZE= parameter can be increased if an AS04 or AS05 abort occurs.

2.2.4 Listed Output (LO)

The Assembler produces listed output that pairs the hexadecimal representation of object code with the corresponding source program statements. Also included in the listing are symbol cross-reference tables and error diagnostics. Refer to Section 6 for an example and explanation of Assembler listed output. Refer to Section 5 for a list of error codes and their meanings.

LO Default and Optional Assignments

The default assignment for LO is to the System Listed Output file (SLO):

```
$ASSIGN LO TO SLO
```

There are three optional assignments for LO:

$$\text{\$ASSIGN LO TO}\left\{\begin{array}{l}\text{pathname}\\ \text{DEV=devmnc}\\ \text{LFC=UT}\end{array}\right\}$$

pathname	is the pathname of the file to contain listed output
devmnc	is the device mnemonic of the device to contain listed output
LFC=UT	assigns output to the user terminal

2.2.5 Object Code - BO (Binary Output) and GO (General Object)

Object code is output to the file or device assigned to logical file codes BO and GO. If binary output is desired, use BO; otherwise, use GO.

If Assembler option 2 is set, output to BO is suppressed.

BO Default and Optional Assignments

The default assignment for BO is the System Binary Output file (SBO):

```
$ASSIGN BO TO SBO
```

The SBO file is output to the device assigned as POD (Punched Output Device) at SYSGEN. The POD assignment can be overridden by the OPCOM SYSASSIGN directive.

The SBO file is a temporary file. If it is not accessed during the job in which it was generated, it will be lost. If necessary, make a permanent copy of the file or device assigned to SBO.

There are two optional assignments for BO:

```
$ASSIGN BO TO {pathname   }
              {DEV=devmnc }
```

pathname	is the pathname of the file to contain binary output
devmnc	is the device mnemonic of a device to contain the binary output

GO Default and Optional Assignments

The default assignment for GO is the System General Object file (SGO):

```
$ASSIGN GO TO SGO
```

The SGO file is a temporary file. If it is not accessed during the job in which it was generated, it will be lost. Some utilities, such as LIBED and CATALOG, access the SGO file automatically. If necessary, make a permanent copy of the file or device assigned to SGO.

There are two optional assignments for GO:

```
$ASSIGN GO TO {pathname   }
              {DEV=devmnc }
```

pathname	is the pathname of the file to contain object code
devmnc	is the device mnemonic of the device to contain object code

2.2.6 Compresssed Source (CS)

The Assembler can produce source output in compressed format. Compressed source output is assigned to the logical file code CS. There is no default assignment for CS.

CS Optional Assignments

To output compressed source, assign a file or device to logical file code CS:

```
$ASSIGN CS TO {pathname   }
              {DEV=devmnc }
```

If both BO and CS are assigned to SBO, BO output is generated first.

Refer to Appendix C for additional information on compressed source format.

2.2.7 LFC Summary

The following is a table of LFCs used by the Macro Assembler and their default and optional assignments.

Table 2-1
Macro Assembler LFC Summary

LFC	Default Assignment	Optional Assignment
SI	SYC	pathname DEV=devmnc
PRE	none	pathname DEV=devmnc
MAC	M.MPXMAC	M.MACLIB pathname
MA2	none	pathname
UT1	temporary file	N/A
UT2	temporary file	N/A

LO	SLO	pathname
		DEV=devmnc
		LFC=UT

BO	SBO	pathname DEV=devmnc
GO	SGO	pathname DEV=devmnc
CS (output)	none	pathname DEV=devmnc

2.3 Options

Options used by the Assembler include control and macro percentage parameters. Options are specified by number in an $OPTION job control language statement. The $OPTION statement should appear before the $ASSEMBLE statement in a jobstream. If no options are specified, the default in effect is option 5.

Option	Description
1	No Listed Output Suppresses source program listing on logical file code LO. Error messages are generated.

Option	Description
2	No Binary Output Suppresses binary output on logical file code BO.
3	Internally Generated Symbols Includes internally generated symbols in the symbol cross-reference listing. Does not apply if option 4 is set.
4	No Symbol Cross-Reference Suppresses listing of symbol cross-reference table.
5	Binary Output Directed to GO Directs binary output to logical file code GO.
6	Reserved
7	Compressed Source Output Generates source output in compressed format on logical file code CS.
8	SI Not Blocked Reads source input (SI) in unblocked format. The $ASSIGN job control language statement must also specify unblocked (BLOC = N).
9	LO, BO, and CS Not Blocked Writes logical file codes LO, BO, and CS in unblocked format. The $ASSIGN statements for these LFCs must also specify unblocked.
10	Nonreferenced Symbols Includes nonreferenced symbols in the symbol cross-reference listing. Does not apply if option 4 is set.
11-13	Reserved
14	Program Identification Information Processes up to 20 bytes of identification information from the id field of the PROGRAM directive and puts the information into the object code.
15	Date and Time Information Obtains the system date and time and puts it into the object code.
16-18	Reserved

Option	Description
19	Symbolic Information to Cataloger Outputs symbolic information to the Cataloger for use by the Symbolic Debugger (COMMON information is not included). Do not specify this option if creating object code that is to be processed by SYSGEN.
20	Call Monitor Compatibility Generates replacement SVC 15, X 'nn' instructions for Call Monitor (CALM) instructions.

2.4 Exiting the Assembler

In batch mode, the Assembler exits when it encounters the first job control language statement with a $ in column one following the $ASSEMBLE, $RUN ASSEMBLE, or $EXECUTE ASSEMBLE statement.

To exit the Assembler in interactive mode, enter CNTRL C in response to the Assembler prompt.

2.5 Using Macros

A macro is a named set of program instructions that occur frequently in a program. Once a macro has been defined, subsequent use of the macro name is all that is needed to generate the instruction sequence. The use of macros can simplify source program coding, minimize programming errors in repetitive instructions, and standardize coding sequences associated with similar functions.

The System Macro Library provides a collection of frequently used macros for use by Assembler source programs. If macros are used within a source program, the system library is assigned by default unless this assignment is specifically overridden. Refer to Section 2 (Logical File Code Assignments) for the LFC assignments used for macro processing.

Users can also create and maintain macro libraries using the Macro Library Editor (MACLIBR) utility.

2.5.1 Macro Components

The instruction sequence that comprises a macro is the macro definition. The variable components within a macro definition are called arguments. Use of a macro is referred to as a macro call, which results in the substitution of the macro instruction sequence for the macro name. The process of assembling the instruction sequence generated by a macro call is referred to as macro expansion.

The macro structure is defined by a set of source statements that specify the legal symbolic macro name, parameters used in the macro, and the sequence of instructions to be generated when a macro call is specified in the source program. Thus, every macro definition consists of the following three elements:

- Macro Definition Directive (DEFM)
- Macro Prototype
- Macro Termination Directive (ENDM)

2.5.2 Symbolic Parameters

Symbolic parameters in the macro definition, also called "dummy" arguments, represent symbolic arguments supplied with the macro call. Parameters specified by the macro call are substituted for the symbolic parameters in the code generated for macro expansion.

All symbolic parameters consist of a percent sign followed by a unique symbolic name of one to eight alphanumeric characters. Examples of valid symbolic parameters are:

```
%ABCDEFGZ
%$50F
%B
%4FX
%C207
```

This syntax is also used to define local symbols within a macro structure.

Use of symbolic parameters in the macro definition is shown in the following sequence:

```
SAMPL        DEFM         LABEL1,A,B
             LW           3,%A
             STW          3,%B
%LABEL1      BCT          2,TESTX
             ENDM
```

The percent sign must not be used as the leading character for a symbolic parameter definition included in the operand field of the DEFM directive.

Macros that are stored in a macro library by the MACLIBR utility retain only those dummy arguments that are actually referenced in the macro body.

2.5.3 Macro Definition

The macro definition is the instruction sequence to be generated in response to a macro call.

Rules of syntax and usage for the source statements comprising a macro are the same as described in the Macro Assembler Language section, with the following exceptions:

- The DEFM, ENDM, and FORM directives may not be used within the range of a macro definition delimited by DEFM and ENDM directives.
- Labels of the form %xxxxxxxx are valid as shown above.

Comment lines and comment fields of macro definition instructions are not included in the macro expansion or the macro definition storage.

The macro definition may include calls to other macro structures.

2.5.4 Macro Call

A macro is called by placing its name in the operation field and associated arguments in the operand field of an instruction.

Syntax:

Label	Operation	Operand
symbol	name	a1,a2,...an

symbol	is an optional symbolic label that is assigned the current value of the location counter
name	specifies the name of the macro definition being referenced. This name corresponds to name specified in the label field of the DEFM directive.
a1,a2,...an	specifies the arguments, if any, to replace the operand field parameters of the DEFM directive

Rules of syntax and usage for entries in the macro instruction are the same as described in the Macro Assembler Language section, with the following exceptions:

- The number of entries specified in the operand field (macro call arguments) must be in the range 0 to 254 and separated by commas.
- The operand field (argument list) must be terminated by a blank.
- Each parameter (argument) in the operand field is limited to 24 characters in length.
- The operand field may be continued to the next source statement by using the continuation character (;).
- Operand field entries (macro call arguments) replace symbolic parameters in the macro definition on a positional basis.
- If no symbolic parameters are specified in the macro definition, the operand field for the corresponding macro call instruction must contain at least 12 blank spaces; i.e., a minimum of 12 blank spaces must be embedded between the operation and comment fields.

The macro call argument list (operand field) may specify null arguments, which can be tested and referenced like an actual argument. Null argument specification results in no source replacement and thus may be used as an optional field. Null arguments are indicated by the omission of an argument in the list.

A typical macro call instruction for the macro defined as SAMPL with an argument list is:

```
EXMAC        SAMPL        XYZABZ,TEST,7
```

A call for the same macro expansion with null argument specification could be:

```
        EXMAC           SAMPL           XYZAB2,,TESTX
```

2.5.5 Macro Expansion

A macro call generates the instruction sequence defined in the macro definition. Symbolic parameters are replaced by the actual arguments supplied with the call.

The following example illustrates a typical macro expansion sequence.

Macro Definition

```
        SAMPL           DEFM            LABEL1,A,B
                        LW              3,%A
                        STW             3,%B
        %LABEL1         BCT             2,TESTX
                        ENDM
```

Macro Call

```
        FIG61           SAMPL           TEST,XLOC,YLOC
```

Macro Expansion

```
                        LW              3,XLOC
                        STW             3,YLOC
        TEST            BCT             2,TESTX
```

2.5.6 Label Generation Within Macros

A unique symbol is generated for any symbolic parameter in the macro definition for which no corresponding actual parameter is passed by the macro call.

A unique symbol is generated each time a macro is called and an actual parameter is not specified. These unique symbols are generated in the form:

```
    !xxxx
```

xxxx	is a hexadecimal value in the range 0000 to FFFF

The following example illustrates the result of unspecified actual parameters:

Macro Definition

```
TEST        DEFM      LABEL, VALUE
%LABEL      LW        3,%VALUE
            BL        TESTX
            BCT       2,$+2W
            BU        %LABEL
            ENDM
```

Macro Call

```
            TEST      ,ALPHA
```

Macro Expansion

```
!0000       LW        3,ALPHA
            BL        TESTX
            BCT       2,$+2W
            BU        !0000
```

Note that the generated code for the macro expansion is exactly the same as for a macro definition of:

Label	Operation	Operand
TEST	DEFM	VALUE

and the macro call:

Label	Operation	Operand
	TEST	ALPHA

The Assembler-generated symbol !XXXX, should not be confused with a user-coded symbol of the similar format, !YYYY. These two types of symbols are treated uniquely and are listed separately in any symbol cross-reference.

In addition to a standard symbol cross-reference, a cross-reference of internally generated symbols may be optionally requested. If specified, the internal symbol cross-reference is listed immediately following the symbol cross-reference.

2.5.7 Symbol Concatenation

Concatenation is the process of combining symbolic parameters within a macro definition with symbolic strings or other symbolic parameters. Concatenation may be performed in any field of a source statement.

Concatenation of symbolic parameters as a suffix to a symbolic string or to another symbolic parameter is indicated by a percent sign (%). The percent sign indicates the presence of the symbolic parameter. If a symbolic parameter is to be concatenated as a prefix to a symbolic string, it must be delimited by a colon (:).

Be sure that symbols to be generated from concatenated strings are syntactically correct and uniquely defined.

The following example illustrates the use of concatenation:

Macro Definition

```
MOVE        DEFM        TO,FROM,LABEL,OP1,OP2,LAB1,LAB2
X%LABEL     %OP1:W      3,A%FROM
            %OP2:W      3,%TO:FIELD
            BU          %LAB1%LAB2
            ENDM
```

Macro Call

```
            MOVE        X,FIELD,5,L,ST,CON,T1
```

Macro Expansion

```
X5          LW          3,AFIELD
            STW         3,XFIELD
            BU          CONT1
```

2.5.8 Nested Macros

A macro call used within the body of a macro definition is referred to as an inner macro call.

The macro definition structure that includes the inner macro call is called the outer macro.

Such inner macro/outer macro constructions are commonly referred to as nested macros. The inner macro call refers to the innermost, or nested, macro structure.

The nested macro is not defined until the outer macro is expanded. Likewise, a macro call in the text of another macro structure is not expanded until the outer macro is called.

If a macro expansion contains a nested macro call, the expansion of the outer macro is suspended until the inner macro is completely expanded. The expansion of the innermost macro in a structure of nested macros always precedes that of the next outermost macro.

Symbolic parameters specified with the inner macro call are replaced by the corresponding values of the outer macro call before the nested macro is processed.

A macro definition that corresponds to an outer macro call instruction may contain any number of inner macro calls.

The depth to which macro calls may be nested is a function of the macro definition's complexity, the number and length of actual parameters, and the amount of storage available.

The following example illustrates the use of a nested macro structure:

Macro Definitions

```
AAA     DEFM    JA,JB,JC          BBB     DEFM    JA,JB,JC
        ZMW     %JA                       LW      6,%JA
        LI      %JC,-47                   ANMW    6,%JB
%INT1   TRR     6,6                       AAA     D,LT,%JC
        BCT     %JB,%INT3                 ARMW    6,TOT
%INT2   SLL     6,1                       ENDM
        BIB     %JC,%INT1
        BU      $+2W
%INT3   ABM     31,%JA
        BU      %INT2
        ENDM
```

Macro Call

```
        BBB     SY,KEN,2
```

Macro Expansion

```
        BBB     SY,KEN,2  }
        LW      6,SY      }
        ANMW    6,KEN     }     Outer Macro
        AAA     D,LT,2    }
        ZMW     D         }
        LI      2,-47     }
!0000   TRR     6,6       }
        BCT     LT,!0002  }
!0001   SLL     6,1       }     Inner Macro
        BIB     2,!0000   }
        BU      $+2W      }
!0002   ABM     31,D      }
        BU      !0001     }
        ENDM              }
        ARMW    6,TOT     }
        ENDM              }     Outer Macro
```

2.6 Datapool

The Datapool feature defines variables for inclusion in a memory partition with the name DATAPOOL or DPOOL00 through DPOOL99. These partitions are similar in use and function to a global common partition. However, Datapool variables are unique in that their placement in a Datapool partition is not order critical like common usage. This eliminates the need to ensure a given order for proper address generation.

Ordering of Datapool variables is defined in the Datapool dictionary. The Datapool Editor (DPEDIT) utility creates and maintains Datapool dictionaries.

Variables used in Assembler language source programs can be typed as Datapool items by defining them as elements of labeled common DATAPOOL or DPOOL00 through DPOOL99.

References to Datapool variables result in the character A being appended to the location counter address field of Assembler listed output. Similarly, the character A is used to denote a Datapool variable in the symbol cross-reference.

The following rules apply to the use of Datapool variables in Assembler language source programs:

- A program variable cannot be equated to a Datapool variable using the EQU directive.
- Four bytes of data must be generated for each reference to a Datapool variable.
- Only one Datapool variable may be referenced in any one instruction.
- For instances where the same variable name is used for both a program variable name and a Datapool variable name, the program variable name will take precedence.

The Datapool Editor section of this manual provides additional information regarding the use of the Datapool feature and operation of the Datapool Editor (DPEDIT).

2.7 Global Common

The labeled common areas GLOBAL00 through GLOBAL99 are treated by the Assembler as externally defined memory partitions. Consequently, variables allocated to these areas may not be initialized.

SECTION 3 - MACRO ASSEMBLER LANGUAGE

3.1 Introduction

This section provides details of Assembler language coding conventions, methods of data representation, and addressing techniques.

3.2 Source Statement Format

Assembler source statements consist of five elements: label, operation, operand, comment, and sequence identification. There are only two syntax requirements: the first four elements must occur in the first 72 input columns, and sequence identification must occur in input columns 73 to 80 (see Figure 3-1). All lower case characters are converted to upper case internally, except for titles, and C and G character strings.

3.2.1 Label Field

The label field is an optional entry that identifies a source statement. The entry consists of a string of alphanumeric characters, of which the first eight characters must be unique. The first character must be an alphabetic character. An error will not be generated if the first character in the label field is not alphabetic. However, statements that reference that label are flagged as errors. The label field begins in input column one and terminates at the first blank column. A blank in input column one indicates the absence of a label. Embedded blanks are not permitted within the label field.

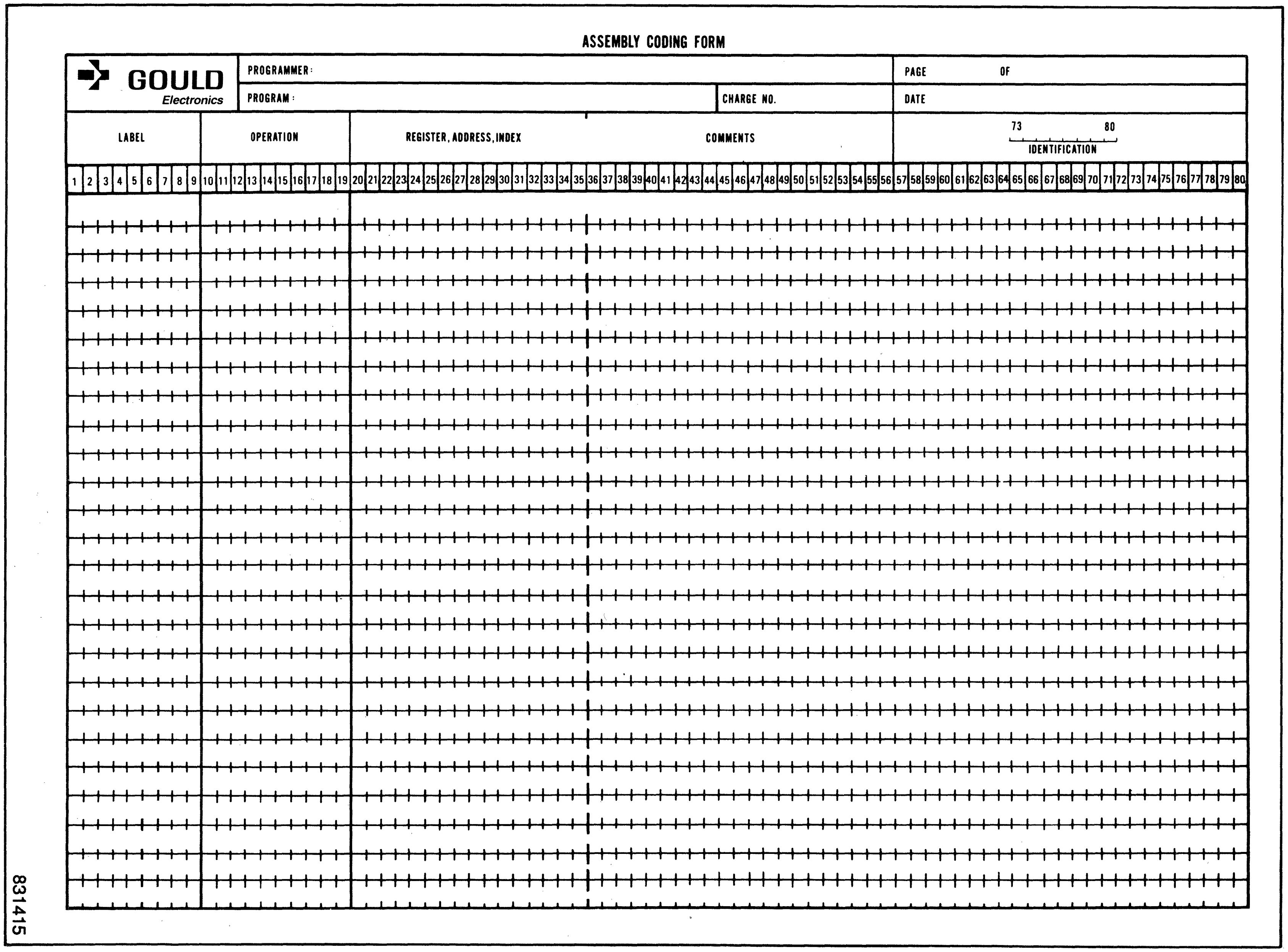
3.2.2 Operation Field

The operation field is required and specifies the mnemonic operation code, Assembler directive or macro name. The operation field begins in the first nonblank input column following the label field or in the first nonblank input column following input column one if the label field is blank. The operation field may be separated from the last nonblank character of the label field by no more than eleven blanks. Only the first eight characters of the operation field are interpreted by the Macro Assembler. Embedded blanks are not permitted in the operation field.

3.2.3 Operand Field

The operand field is required and specifies operands associated with the current operation. Operands may identify storage locations, masks, storage area size, or data types and may take the form of a single expression, a series of expressions separated by commas, a constant, or a constant expressed as a literal. Multiple operands must be separated by commas.

The operand field begins in the first nonblank input column following the operation field and is terminated by a blank. If there are eleven blank columns after the last nonblank character of the operation field or input column 72 is encountered, the operand field is treated as empty.



ASSEMBLY CODING FORM

GOULD Electronics

PROGRAMMER:

PAGE OF

PROGRAM:

CHARGE NO.

DATE

LABEL

OPERATION

REGISTER, ADDRESS, INDEX

COMMENTS

73 80

IDENTIFICATION

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80

831415

Figure 3-1. Assembler Coding Form

Operand field entries may not contain embedded blanks except when an entry is used to specify a null character string or when an operative special character is being used.

Embedded blanks cannot be used between multiple operands separated by commas.

3.2.4 Comment Field

The comment field is optional and provides descriptive information to be included with program listed output. Typical usage is for program documentation. All valid characters, including blanks, may be used in the comment field. The comment field begins in the first nonblank input column following the operand field and terminates with input column 72. If the last entry in the operand field is blank, eleven blank columns must separate the comment field from the comma which is the last nonblank character of the operand field.

A source statement with an asterisk (*) in input column one is treated as a comment line. Each comment line is assigned a line number as part of the assembly process and appears as part of the program's listed output.

3.2.5 Sequence Field

The sequence field specifies program identification information and/or sequential ordering for source statements. The sequence field occupies input columns 73 to 80 and is not examined during Assembler processing. The entry is listed on all Assembler printed output.

3.2.6 Continuation Lines

When coding source statements that must be continued on the next line, the following rules apply:

- A semicolon (;) within or following the label or operation fields, or within an operand field, indicates continuation of the field to the next source statement line. Continuation starts in column one of the next line.

- Blanks cannot be imbedded within the label, operation, or operand fields when they are continued. The blanks between each field must either precede the semicolon or appear in column one of the continued line.

- Comments can be placed after the semicolon but cannot continue to the next line. The next line must contain the continuation of the source statement, beginning in column one.

- Comments cannot be continued by appending the semicolon continuation indicator.

- The number of character positions allowed for each field is the same as in noncontinued source statements.

3.2.7 Character Set

Valid characters for coding source statements are:

Character Name	Character Representation
Alphabetic	Letters A through Z
Numeric	Digits 0 through 9
Plus Sign (Addition)	+
Minus Sign (Subtraction)	-
Asterisk (Multiplication/Indirect)	*
Slash (Division)	/
Dot/Period (Decimal Point)	.
Comma (Subfield Delimiter)	,
Left Parenthesis (Attribute Delimiter)	(
Right Parenthesis (Attribute Delimiter)	)
Single Quote (Constant Delimiter)	'
Double Quote (Escape Control Character)	"
Equal Sign (Literal Definition)	=
Blank (Field Delimiter/Spacing)	
Semicolon (Continuation Indicator)	;
Colon (Dummy Parameter Concatenation)	:
Percent Sign (Dummy Argument Identifier)	%

Source statements are usually coded from a combination of these 51 characters. However, the use of character constants in character strings permits the use of any of the 128 ASCII character codes supported by the symbolic input device. The ASCII codes are listed in Appendix D.

3.2.7.1 Escape Character

The escape character (") causes the Macro Assembler to generate the ASCII control characters listed in Table 3-1. The character immediately following the escape character is logically ANDed with 3F (hexadecimal), then ASCII-coded, resulting in one of the control characters.

Use the escape character to generate the following characters within a C or G-character string:

%	(percent)
'	(single quotation mark)
"	(double quotation mark)
;	(semicolon)

Table 3-1

ASCII Control Characters

Control Character	Character String Entry	Description
NUL	"@	Null
SOH	"A	Start of Heading
STX	"B	Start of Text
ETX	"C	End of Text
EOT	"D	End Of Transmission
ENQ	"E	Inquiry
ACK	"F	Acknowledge
BEL	"G	Bell
BS	"H	Backspace
HT	"I	Horizontal Tabulation (punch card skip)
LF	"J	Line Feed
VT	"K	Vertical Tabulation
FF	"L	Form Feed
CR	"M	Carriage Return
SO	"N	Shift Out
SI	"O	Shift In
DLE	"P	Data Link Escape
DC1	"Q	Device Control 1
DC2	"R	Device Control 2
DC3	"S	Device Control 3
DC4	"T	Device Control 4
NAK	"U	Negative Acknowledge
SYN	"V	Synchronous Idle
ETB	"W	End Of Transmission Block
CAN	"X	Cancel
EM	"Y	End Of Medium
SUB	"Z	Substitute
ESC	"[	Escape
FS	"/	File Separator
GS	"]	Group Separator
RS	"^	Record Separator
US	"-	Unit Separator

3.3 Data Representation

Most source program statements include one or more operands composed of one or more expressions. Expressions are composed of a term or a valid combination of terms. Every term represents a value which may be assigned by the Macro Assembler (symbol) or which may be inherent to the term itself (literal, constant).

Arithmetic combinations of terms are reduced to a single value in the assembly process.

3.3.1 Symbols

A symbol is a character or combination of characters that references program elements. Symbols are typically used in the source statement label field and/or operand field.

Symbols consist of a string of alphanumeric characters, of which the first eight must be unique. The first character must be an alphabetic character. The symbol may not include embedded blanks. Only the first eight characters of the symbol name are printed in the symbol cross-reference listing.

Examples of valid symbols are:

```
MACRO2
BP1234XX
H204
```

Examples of invalid symbols are:

```
7GPI  (first character not alphabetic)
AB D  (contains embedded blank)
```

Each defined symbol must be unique within an assembly job step. Multiply defined symbols are denoted by the error flag M in the listed output.

3.3.2 Literals

Literal terms are used to enter numeric values, addresses, or alphabetic character strings for phrases or message output to the source program. Literals specify a constant or executable address attribute preceded by an equal sign (=).

Literals represent data rather than a reference to data. In general, literals may be used wherever a storage address is permitted as a valid operand. Literal terms are relocatable since the address of the literal, rather than the literal itself, is assembled.

The following rules apply to the use of literals:

- Literals may not be combined with other terms.
- Literals may not be used in any statement that requires a previously defined symbol.
- Literals may not contain external references.
- Symbols used in literals must be previously defined.

To process a literal, the Assembler stores the literal's value in the literal pool. The address of the literal pool containing the value is placed in the operand field of the assembled source statement. All literals generate a 32-bit value.

The literal pool begins on the first available word boundary location following the program counter location for the Assembler END or LPOOL directive. Only one entry is made for the same literal term in the literal pool.

The LPOOL directive controls the placement of the literal pool contents.

Examples of literal terms are:

As written in source	As written in literal pool
=A(TAG1)	(Address of TAG1)
=B(TAG1)	(Byte Address of TAG1)
=W(TAG1)	(Word Address of TAG1)
=5	(Decimal Value=5)
=C'END'	(Data=END)
=X'3A7'	(Hexadecimal Value=3A7)
=A+B+C	(Value of A+B+C)

3.3.3 Constants

A constant is used to enter data into storage. The Macro Assembler supports constants used in data statements and as operands in immediate type instructions. Constant length is limited to one doubleword (eight bytes), with the exception of C-character strings, which may be any length.

The Macro Assembler recognizes seven types of constants:

- C - Character String
- G - Character String
- X - Hexadecimal Constant
- N - Fixed Point Decimal Word
- F - Fixed Point Decimal Doubleword
- E - Floating Point Decimal Word
- R - Floating Point Decimal Doubleword

3.3.3.1 C-Character String

A C-character string consists of any number and any combination of ASCII characters enclosed in single quotation marks preceded by the letter C. A C-character string is left-justified to the boundary defined by the operation code. For C-character strings, all ASCII characters are stored in hexadecimal form. An internal lower to upper case conversion does not occur for C-character strings.

Usage:

A typical C-character string entry is:

DATAW C'NORMAL STRING 1'

This entry is transferred to memory on the first available word boundary as follows:

Memory Location	Contents							
	0	7	8	15	16	23	24	31
	N		O		R		M	
Word 1	0100	1110	0100	1111	0101	0010	0100	1101
	A		L		(Space)		S	
Word 2	0100	0001	0100	1100	0010	0000	0101	0011
	T		R		I		N	
Word 3	0101	0100	0101	0010	0100	1001	0100	1110
	G		(Space)		1		(Space)*	
Word 4	0100	0111	0010	0000	0011	0001	0010	0000

*This space was added by the Assembler to complete the word boundary.

The entry for a C-character string with ASCII control characters that will generate 'MESSAGE' followed by a carriage return and line feed is:

```
DATAH C'MESSAGE"M"J'
```

M and J are the ASCII control characters for a carriage return and line feed.

This entry will be transferred to memory on the first available halfword boundary as follows:

Memory Location	Contents							
	0	7	8	15	16	23	24	31
	Previous Entry				M		E	
Word 1	XXXX	XXXX	XXXX	XXXX	0100	1101	0100	0101
	S		S		A		G	
Word 2	0101	0011	0101	0011	0100	0001	0100	0111
	E		CR*		LF*		(Space)**	
Word 3	0100	0101	0000	1101	0000	1010	0010	0000

*The upper case alphabetic character is logically ANDed with a 3F (hexadecimal) to produce the ASCII control character when preceded by an escape character (").

**This space was added by the Assembler to complete the halfword boundary.

An entry for a C-character string using the symbols ("), ('), and (;) is:

DATAB C'AB" "C'''D";'

This entry is transferred to memory on the first available byte boundary as follows:

Memory Location	Contents							
	0	7	8	15	16	23	24	31
	Previous Entry						A	
Word 1	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX	0100	0001
	B		"		C		'	
Word 2	0100	0010	0010	0010	0100	0011	0010	0111
	D		;		Available Memory			
Word 3	0100	0100	0011	1011				

An entry for a C-character string with continuation is:

```
     DATAD C'ABC;
DEFG'
```

This entry is transferred to memory on the first available doubleword boundary as follows:

Memory Location	Contents							
	0	7	8	15	16	23	24	31
	A		B		C		D	
Word 1	0100	0001	0100	0010	0100	0011	0100	0100
	E		F		G		(Space)*	
Word 2	0100	0101	0100	0110	0100	0111	0010	0000

*This space was added by the Assembler to complete the doubleword boundary.

3.3.3.2 G-Character String

A G-character string has the same format as a C-character string except it is right-justified, limited to the size of its defined boundary (a maximum of eight bytes), and zero-filled on the left for bounding purposes. For G-character strings, all ASCII characters are stored in hexadecimal form. The G-character string constant type can be used as the operand field of immediate type instructions and should be used in preference to the left-justified C-type constant. An internal lower to upper case conversion does not occur for G-character strings.

For example:

LI 4, C'A'	generates	CA002020
LI 4, G'A'	generates	CA000041
LI 4, G'AB'	generates	CA004142

An entry for a typical G-character string is:

DATAW G'A G'

This entry is transferred to memory on the first available word boundary (right-justified) as follows:

Memory Location	Contents							
	0	7	8	15	16	23	24	31
	(00)*		(A)		(Space)		G	
Word 1	0000	0000	0100	0001	0010	0000	0100	0111

*These zeros were added by the Assembler to complete the word boundary.

3.3.3.3 Hexadecimal Constant (X)

A hexadecimal constant consists of an optionally signed hexadecimal number enclosed in single quotation marks and preceded by the letter X. No sign indicates a positive number. If the hexadecimal constant is preceded by a minus sign, a two's complement of the hexadecimal number will be generated. A hexadecimal constant is right-justified and is limited to the size of its defined boundary (a maximum of eight bytes). Hexadecimal constant definitions must not contain embedded blanks.

Usage:

The following hexadecimal constant definitions generate the indicated constants:

DATAB X'E'	0E
DATAW X'C2DA'	0000C2DA
DATAD X'B123F6C'	000000000B123F6C
DATAH X'FFC213D'	213D (the three most significant bytes are lost because of the defined boundary size)

The following negative hexadecimal constant definitions generate the indicated constants:

DATAB X' -E2'	1E
DATAH X' -E2'	FF1E

3.3.3.4 Fixed Point Decimal Word (N)

A fixed point decimal word string consists of a decimal number up to 16 digits on either side of the decimal point. A fixed point decimal word is right-justified and limited to its defined boundary (a maximum of eight bytes). Embedded blanks may be used to improve legibility.

Syntax:

```
N'[{±}]m[E[{±}]xx][Byy]'
```

+	indicates a positive decimal number. If no sign is present, the default is positive.
-	indicates a negative decimal number
m	is a constant in the form of a decimal fraction, decimal integer, or mixed number. A number such as 1234567890123456.123401234 is valid even though truncation would occur and significant digits would be lost. Larger numbers, such as the above, may be significantly encoded by use of the optional binary scaling point.
E	indicates that the following number represents an exponent

xx	is a one or two-digit exponent in the range of +77 through -77. A positive integer is the default if no sign is present.
B	indicates that the following number represents a binary scale
yy	is a one or two-digit binary scale specification. If Byy is omitted, the default is B31, as this is the register scaling for 16 digits on either side of the decimal point.

Usage:

An entry for a positive decimal number is:

DATAB N'105'

This entry is transferred to memory on the first available byte boundary as follows:

Memory Location	Contents							
	0 7		8 15		16 31			
	Previous Entry		69		Available Memory			
Word 1	XXXX	XXXX	0110	1001				

An entry for a negative decimal number is:

DATAH N'-6'

This entry is transferred to memory on the first available halfword boundary as follows:

Memory Location	Contents							
	0 7		8 15		16 31			
	FF		FA		Available Memory			
Word 1	1111	1111	1111	1010				

An entry for an exponentiated fixed point decimal number is:

DATAH N'9.2E2'

This entry is transferred to memory on the first available halfword boundary as shown below. Truncation has occurred since 9.2 cannot be represented as an exact binary quantity.

Memory Location	Contents							
	0	7	8	15	16			31
	03		97		Available Memory			
Word 1	0000	0011	1001	0111				

An entry for a fixed point decimal string with binary specification is:

DATAW N'19B7'

This entry is transferred to memory on the first available word boundary as follows:

Memory Location	Contents							
	0	7	8	15	16	23	24	31
	13		00		00		00	
Word 1	0001	0011	0000	0000	0000	0000	0000	0000

Note: The arrow denotes the binary scale specification which specifies the bit position to the right of the decimal integer in a fixed point decimal word string.

An entry for a fixed point decimal string with binary specification is:

```
DATAD N'.25B14'
```

This entry is transferred to memory on the first available doubleword boundary as follows:

Memory Location	Contents							
	0	7	8	15	16	23	24	31
	00		00		00		00	
Word 1	0000	0000	0000	0000	0000	0000	0000	0000
	00		00		80		00	
Word 2	0000	0000	0000	0000	1000	0000	0000	0000
				↗				

Note: The arrow denotes the binary scale specification which specifies the bit position to the right of the decimal integer in a fixed point decimal word string. Because a doubleword data definition has been specified, both the value and binary scale specification apply to word two.

3.3.3.5 Fixed Point Decimal Doubleword (F)

A fixed point decimal doubleword string consists of a decimal number with up to 16 digits on either side of the decimal point. A fixed point decimal doubleword is right-justified. Embedded blanks may be used to improve legibility.

Syntax:

```
F'[{±}]m[E[{±}]xx][Byy]'
```

+	indicates a positive decimal number. If no sign is present, the default is positive.
-	indicates a negative decimal number
m	is a constant in the form of a decimal fraction, decimal integer, or a mixed decimal number
E	indicates that the following number represents an exponent
xx	is a one or two-digit exponent in the range +77 through -77
B	indicates that the following number represents a binary scale
yy	is a one or two-digit binary scale specification. If Byy is omitted, the default is B63 as this is the register scaling for 32 digits on either side of the decimal point.

Usage:

An entry for a positive fixed point decimal number is:

DATAD F'16E2'

This entry is transferred to memory on the first available doubleword boundary as follows:

Memory Location	Contents							
	0	7	8	15	16	23	24	31
	00		00		00		00	
Word 1	0000	0000	0000	0000	0000	0000	0000	0000
	32	39	40	47	48	55	56	63
	00		00		06		40	
Word 2	0000	0000	0000	0000	0000	0110	0100	0000

Note: The arrow denotes the binary scale specification which specifies the bit position to the right of the decimal integer in a fixed point decimal doubleword string.

3.3.3.6 Floating Point

Floating point constants can be expressed as word or doubleword decimal values. The Macro Assembler converts the specified constant value to its hexadecimal equivalent to produce a floating point hexadecimal number.

The floating point number (either word or doubleword) is comprised of three parts: sign, fraction, and exponent. Floating point data formats are shown in Figure 3-2.

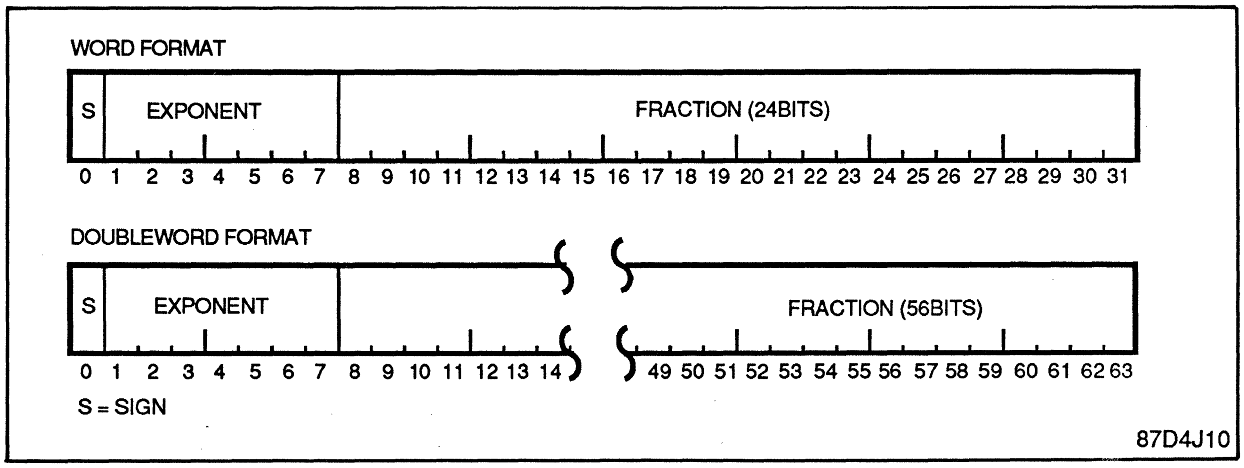


Figure 3-2. Floating Point Data Formats

The sign (bit 0) applies to the fractional part of the value and denotes positive or negative. The fraction is a hexadecimal normalized number with a radix point to the left of the highest order fraction bit (bit 8). The exponent (bits 1-7) is a seven-bit binary value to which the base 16 (decimal) is raised.

Negative exponents are carried in two's complement format by the hardware. To remove the sign and provide for direct comparison of exponents, both positive and negative exponents are biased upward by 40 (hexadecimal). Thus, the quantity a floating point value represents is derived by multiplying the hexadecimal fraction by the number 16 (decimal) raised to the power specified by the exponent minus 40 (hexadecimal).

For example, to convert the value 31 (decimal) to a hexadecimal fraction:

$$.1F \times 16^{2}$$

The power of 16 (i.e., 2) is added to the exponential bias 40 (hexadecimal) to yield an operative exponent of 42 (hexadecimal).

Thus, the range of values that can be represented in floating point format is:

$$[(F)\,(16^{-64}) \leq N \leq (F)\,(16^{63})]$$

F is a hexadecimal fraction and N is the range of values.

The range of the fraction F in nomalized format is:

(Word)	$(2^{-4}) \leq F \leq 1-(2^{-24})$
(Doubleword)	$(2^{-4}) \leq F \leq 1-(2^{-56})$

Hardware converts a positive floating point number to a negative floating point number by taking the two's complement of the positive fraction and the one's complement of the biased exponent. All floating point operands are normalized before the floating point instruction is performed. A positive floating point value is normalized when the fraction is in the range ($1 \geq F \geq 1/16$). A negative floating point value is normalized when the fraction is in the range ($-1 \leq F \leq -1/16$). All floating point results are normalized by floating point hardware.

Details of floating point hardware operation, including instruction formats, are given in the appropriate CPU reference manual.

3.3.3.7 Floating Point Decimal Word (E)

A floating point word string consists of a decimal number with up to 16 digits on either side of the decimal point. A floating point word string is right-justified and limited to its defined boundary (a maximum of eight bytes). Embedded blanks may be used to improve legibility.

Syntax:

```
E'[{±}]m[E[{±}]xx]'
```

+	indicates a positive decimal number. If no sign is present, the default is positive.
-	indicates a negative decimal number
m	is a constant in the form of a decimal fraction, decimal integer, or mixed decimal number
E	indicates that the following number represents an exponent
xx	is a one or two-digit exponent

Usage:

The processing involved in producing the floating point hexadecimal representation for:

```
DATAW E'12'
```

includes the following sequence:

(1) Decimal 12 is converted to hexadecimal C.

(2) The decimal point is moved until only a fraction remains:

$$.C \times 16^1$$

The exponent indicates the number of places the decimal point was moved.

(3) A biased exponent is produced by adding the power of 16 (i.e., 1) to the biased exponential representation (40):

$$16^{40} + 16^1 = 16^{41}$$

which produces the biased exponent 41.

(4) The biased exponent and hexadecimal fraction are stored in memory right-justified and bounded. The internal representation for this example is:

41C00000

The following are sample coding entries for floating point decimal word constants and their generated internal representation:

Coding Entry	Internal Representation
DATAW E'1'	41100000
DATAW E'-1'	BEF00000
DATAW E'15'	41F00000
DATAW E'-15'	BE100000
DATAW E'16'	42100000
DATAW E'-16'	BDF00000
DATAW E'2E3'	437D0000
DATAW E'3.14159265359'	413243F6

3.3.3.8 Floating Point Decimal Doubleword (R)

A floating point doubleword string follows the same rules as a floating point word string except for the range of the hexadecimal fraction $[(2^{-4}) \leq F \leq 1-(2^{-56})]$.

Syntax:

R' [{±}]m[E[{±}]xx]

+	indicates a positive decimal number. If no sign is present, the default is positive.
-	indicates a negative decimal number
m	is a constant in the form of a decimal fraction, decimal integer, or mixed number
E	indicates that the following number represents an exponent
xx	is a one or two-digit exponent

The following are sample coding entries for floating point decimal doubleword constants and their generated internal representation:

Coding Entry	Internal Representation
DATAD R'1'	41100000 00000000
DATAD R'17'	42110000 00000000
DATAD R'3.14159'	413243F3 E0370CDC
DATAW R'3.14159'	E0370CDC

For the last entry (DATAW R'3.14159'), only the least significant 32 bits are retained because of the word constant size.

3.3.4 Expressions

An expression is a single term, i.e., symbol or constant, or an arithmetic combination of terms. Examples of legal expressions are:

```
PRNT
IBUF+6
TBL1-TBL2
CD*7
$+4
```

The following rules apply to the use of expressions:

- Expressions may not begin with an arithmetic operator (+ - * /).
- Expressions may not contain two terms or two arithmetic operators in succession.
- A multiterm expression may not contain a literal or attributed term.
- A multiterm expression in a DATA or GEN directive may not contain an entry C-character string as a first term.
- Negative results cannot be generated because evaluation is done in 23 bits.
- Floating point cannot be used in expressions because evaluation is done in 23 bits.

3.3.4.1 Expression Evaluation

Expressions are evaluated from left to right with arithmetic operations performed as encountered. Thus, the leftmost operator has the highest hierarchical priority and the rightmost operator the lowest. Expression evaluation cannot be altered by using nested expressions in parentheses. Parentheses are used to delineate attributed expressions. A missing expression on either side of an arithmetic operator is evaluated as zero.

Division by zero is valid and yields a zero result. If the value of the symbol to be used as a divisor is zero, the expression is flagged as an error by a backslash. Division always yields an integer result and any remainder is truncated. For example, the expression:

```
5 - 4/2
```

yields a result of zero. For arithmetic operations, each term in an expression is stored internally in 23 bits.

3.3.4.2 Expression Types

Multiterm expressions are relocatable, absolute, or common. An expression is relocatable if its value is changed by program relocation. An absolute expression is unaffected by program relocation.

Single term expressions can be any one of four modes: absolute, relocatable, common, or external.

Tables 3-2, 3-3, and 3-4 illustrate the types of terms which may be combined arithmetically and the mode of the result.

Table 3-2
Addition Operations

Addition

(+)	ABSOLUTE	RELOCATABLE	COMMON	EXTERNAL
ABSOLUTE	Absolute	Relocatable	Common	Illegal
RELOCATABLE	Relocatable	Illegal	Illegal	Illegal
COMMON	Common	Illegal	Illegal	Illegal
EXTERNAL	Illegal	Illegal	Illegal	Illegal

Table 3-3
Subtraction Operations

Subtraction

(-)	ABSOLUTE	RELOCATABLE	COMMON	EXTERNAL
ABSOLUTE	Absolute	Illegal	Illegal	Illegal
RELOCATABLE	Relocatable	Absolute	Illegal	Illegal
COMMON	Common	Illegal	Absolute(*)	Illegal
EXTERNAL	Illegal	Illegal	Illegal	Illegal

* All references to common within an expression must refer to the same common block

Table 3-4
Multiplication/Division Operations

Multiplication
Division

(* /)	ABSOLUTE	RELOCATABLE	COMMON	EXTERNAL
ABSOLUTE	Absolute	Illegal	Illegal	Illegal
RELOCATABLE	Illegal	Illegal	Illegal	Illegal
COMMON	Illegal	Illegal	Illegal	Illegal
EXTERNAL	Illegal	Illegal	Illegal	Illegal

3.4 Addressing Techniques

Memory is addressable in byte, halfword, word, and doubleword entities. Most memory reference instructions require that the operand field specify a general purpose register and an effective address. The effective address may be derived from a 20-bit memory address modified by indirect addressing and/or indexing. Symbolic addressing is allowed by the Macro Assembler.

Addressing in the Macro Assembler defaults to the data section (DSECT) mode. All data and program source code is generated as part of a DSECT unless the addressing mode is changed using the CSECT directive.

The code section (CSECT) mode is an optional mode used for data and source code that will not be changed during program execution. CSECT mode, also called control, pure, or read-only, allows the creation of shared or reentrant programs. CSECT mode also provides additional memory protection because it allows read only access.

Variables, buffers, and areas of code subject to self-modification should be addressed in DSECT mode when creating shared tasks. Constants and pure data can be addressed in CSECT mode. At catalog time, a shared environment is declared. The CSECT is loaded into the system once. The DSECT will be loaded into the system once for each user of the task. The amount of memory saved by using a combination of CSECT and DSECT depends on the sizes of the CSECT and DSECT and the number of users sharing the task.

3.4.1 Location Counter

The Macro Assembler maintains an internal location counter to determine memory allocation for each assembled source program statement. This counter is a byte counter that provides for location assignment to bytes in a memory word. For example:

Location Counter	Label	Operation	Operand
00000	W	RES	1
00001	X	RES	2
00003	Y	RES	1
00004	Z	RES	8
0000C	(Next Source Statement)		

W	is assigned to byte zero of word zero
X	is assigned to bytes one and two of word zero. Even though X is a halfword (two bytes), it is entered across a halfword boundary.
Y	is assigned to byte three of word zero
Z	is assigned to bytes one through three of words one and two

The contents of the location counter is expressed in hexadecimal format for all Assembler listed output.

When an instruction sequence is encountered and the location counter is not positioned at a byte multiple of the number of bytes required, the location counter is advanced for proper boundary alignment. In the case of a full word instruction being assembled after a halfword instruction, a halfword may not be usable. In this case, a no operation (NOP) instruction is entered into the halfword. If a label appears in the label field, the label is assigned after the NOP instruction. For example:

Location Counter	Generated Code	Label	Operation	Operand
			ABS	
			ORG	X'00000'
00000			RES	1B
00002	0E40	START	ZR	4
00004	2F40		TRR	4,6
00006	0002			
00008	AD880427	STOP	LB	3,A

The halfword instruction, ZR, forces the location counter to a halfword boundary. The symbolic address, START, is defined at location two. When the instruction LB is encountered, a NOP instruction (0002) is generated for the halfword location six prior to assigning a location for STOP and processing the LB operation.

If the symbolic address does not fall on a proper boundary because of the operand size, the least significant bit(s) are interpreted as zero and an error condition flag is set.

3.4.2 Self-Relative Addressing

Self-relative addressing implicitly defines a symbolic name which has an address equal to the current value of the location counter, or constructs a reference to a memory location in relation to the location counter. References to the location counter may be made by use of the special symbol $ as follows:

```
${±}n[s]
```

$	is the current value of the location counter. For multiword instructions, $ always refers to the first word.
+	increments the location counter
-	decrements the location counter
n	is an integer specifying the count of the size attribute
s	is an address size attribute indicating one of five types of addressing:

B - Byte
H - Halfword
W - Word
D - Doubleword
F - File (8 Words)

If s is not specified, a byte count is assumed.

For multiword instructions, $ always refers to the first word.

Usage:

In the following example, the symbolic address ZET has an address equal to location three:

Location Counter	Label	Operation	Operand
00003	ZET	EQU	$

In the following example, the LW instruction loads the contents of location 00C into register two:

Location Counter	Label	Operation	Operand
00000	STORE	LW	2,$+3W
.		.	.
.		.	.
.		.	.
.		.	.
0000C		DATAW	2000

3.4.3 Symbolic Addressing

Symbolic addressing allows user-defined symbols to represent the location of a particular constant, instruction, or storage location. Symbols are defined in the label field of a program source statement. A symbol may be referenced by any operand field entry. The value assigned to a symbol is the address of the most significant byte of the constant, instruction, or referenced storage location.

Usage:

The following example illustrates symbolic addressing:

Label	Operation	Operand
TAG1	DATAW	X'1000'
	LW	1,TAG1

3.4.4 Relative Addressing

Relative addressing allows the of addressing instructions, constants, or storage locations by designating their location relative to a symbolic location.

Usage:

In this example, the LW instruction loads the second word of doubleword A into register three. The LB loads the fourth byte of A into register two. Signed address attributes can be used similar to self-relative addressing.

Label	Operation	Operand
A	RES	1D
	LW	3,A+1W
	LB	2,A+3

3.4.5 Absolute Addressing

Absolute addressing explicitly defines the location of a particular constant, instruction, or storage location to be used in the operation.

Usage:

Label	Operation	Operand	Comments
A	LW	4,X'1000'	Loads the contents of location 1000 (hexadecimal) into register four.
B	LH	2,200	Loads the contents of location 200 (decimal) into register two.
X	STW	1,N'16B29'	Stores the contents of register one in the word beginning at location 40 (hexadecimal).

3.4.6 Literal Addressing

Literal addressing allows a defined literal term to specify an address to be used in the operation.

Usage:

The use of the literal GEE with the Load Word (LW) instruction loads the byte address of the symbol GEE (190 hexadecimal) into register two:

Label	Operation	Operand	Comments
GEE	EQU	400	The generated internal data representation for the literal address B(GEE) is: 00080190
	LW	2,=B(GEE)	

3.4.7 Blank Addressing

Blank addressing allows the symbolic representation $$ to inform the Macro Assembler that an address will be inserted at program execution time. When the $$ specification is used, zeros will be assembled into the address field of the instruction. The F and C bits of the address field depend on the operation to be performed.

3.4.8 Addressing Attributes

Addressing attributes allow the specification of a 20-bit operand address other than that which would normally be used for a particular mnemonic instruction.

The resulting 20-bit effective address incorporates the user-specified address expression with F and C bits corresponding to standard operand format codes (see Table 3-5).

Table 3-5
Operand Format Code

F Bit	C Bits	Designated Format
0	00	Word
0	01	Left Halfword
0	10	Doubleword
0	11	Right Halfword
1	00	Byte 0
1	01	Byte 1
1	10	Byte 2
1	11	Byte 3

Addressing attributes have the following format:

a (address)

a is the addressing attribute as follows:

Attribute	Result
B	Byte address
H	Halfword address
W	Word address
D	Doubleword address
A	Byte address with F bit set to zero

Usage:

In the following example of addressing attributes, the statements with labels X, Y, and Z produce the same halfword address (00003) which contains C and D:

Location Counter	Label	Operation	Operand
00000	A	RES	1
00001	B	RES	1
00002	C	RES	1
00003	D	RES	1
00004	X	DATAW	H(C)
00008	Y	DATAW	H(A+1H)
0000C	Z	DATAW	H(A+2B)

For the following operation, only the first byte of location ALPHA is loaded into register three because the address attribute of the expression overrides the operation:

```
LW 3,B(ALPHA)
```

SECTION 4 - DIRECTIVES

4.1 Introduction

Macro Assembler programs contain source statements consisting of Assembler instructions and Macro Assembler directives.

Assembler instructions are categorized by the type of function they perform. Assembler instructions are provided for the following types of operations:

Load/Store
Branch
Compare
Register Transfer
Memory Management
Logical
Shifting
Bit Manipulation
Fixed & Floating Point Arithmetic
Control
Interrupt Control
Writable Control Storage
Input/Output

These instructions are documented in the CPU hardware reference manual corresponding to the machine type.

Macro Assembler directives are provided for program control, data and symbol definition, listed output control, conditional assembly, macro support, and special usage.

For some directives described in this section, the syntax may include an optional label entry in the first field of an Assembler source statement. Thus, where the syntax is shown as:

Label	Operation	Operand
label		

label may be any valid entry used for convenience to identify a source statement as described in the Macro Assembler Language section. If the label field has significance other than as an optional identifier, it will be identified with another variable name in the syntax and explained in the usage notes for that directive.

4.2 ABS Directive

The ABS directive indicates a section of program coding to be assembled in absolute mode. Generated object code cannot be relocated by the loader. All symbolic names are assigned absolute memory addresses relative to location zero (absolute). Subsequent use of this directive causes the location counter to be assigned a value equal to the next absolute memory address to be allocated.

Syntax:

Label	Operation	Operand
	ABS	

Usage:

The following example assembles code in the absolute mode beginning at location 1000 (hexadecimal):

```
TAG             ABS
                ORG                 X'1000'
```

NOTE: Object code containing absolute sections cannot be processed by CATALOG.

4.3 AC Directive

The AC directive allows for address generation within a source program. The address may incorporate indexing and/or indirect modification. The directive may also be used to reserve a word of storage.

Use of this directive results in word bounding of the location counter.

Bits zero to eight of the generated word address are always zero.

Syntax:

Label	Operation	Operand
label	ACx	address

x specifies the address field as follows:

Variable	Field
B	Byte
H	Halfword
W	Word
D	Doubleword

If not specified, the default is B.

address specifies a valid address that may include indexing and/or indirect modification

Usage:

The following example illustrates typical usage of the AC directive:

```
Location    Machine      Byte
Counter     Instruction  Address    Label   Operation   Operand

P0140C      000017B8     P017B8     KTXT    ACW         TEXT
P01410      0040153C     P0153C     LNK1    ACW         LINK,2
P01414      00101978     P01978     PARS    ACW         *PPAK
P01418      000019A4     P019A4     PARE    ACW         PIAT
P0141C      000019A4     P019A4     BFOP    ACW         BFO
P01420      000029A4     P029A4     BFIP    ACW         BFI
  .                                   .       .           .
  .                                   .       .           .
  .                                   .       .           .
P0153C                              LINK    EQU         $
  .                                   .       .           .
  .                                   .       .           .
  .                                   .       .           .
P017B8                              TEXT    EQU         $
  .                                   .       .           .
  .                                   .       .           .
  .                                   .       .           .
P01978                              PPAK    EQU         $
  .                                   .       .           .
  .                                   .       .           .
  .                                   .       .           .
P019A4                              PIAT    EQU         $
P019A4      00000000                BFO     DATAW       0
  .                                   .       .           .
  .                                   .       .           .
  .                                   .       .           .
P029A4      000017B8     P017B8     BFI     GEN         32/A(TEXT)
```

4.4 ANOP Directive

The ANOP directive facilitates conditional and unconditional branching to source statements identified by labels that are defined by variable symbols. Typical usage involves branching to a source statement whose label is defined by a symbolic parameter. An ANOP directive coded immediately preceding a source statement provides the capability to generate conditional and unconditional branching to that statement at assembly time.

No code is generated for this directive.

No entry is made in the symbol table for the symbol specified in the label field.

Syntax:

Label	Operation	Operand
symbol	ANOP	

4.5 BOUND Directive

The BOUND directive advances the location counter until it represents a byte multiple of the bounding value specified in the operand field.

Symbols used in the operand field must have been previously defined.

Syntax:

Label	Operation	Operand
label	BOUND	value

Usage:

The following example sets the location counter to the first even multiple of four:

```
(LC-00000)      RES       1
                BOUND     4
(LC-00004)      DATA      C'A'
```

4.6 COMMON Directive

The COMMON directive, used in conjunction with data-generating directives, defines, initializes, and manipulates common communication areas. Common data storage areas can be shared between programs loaded at different locations in memory. The sharing programs may be coded in Assembler language or FORTRAN. Common areas are always generated in DSECT mode.

Syntax:

Label	Operation	Operand
symbol	COMMON	n1,n2,...,nn

symbol	is the symbolic name of the common block defined by the operand field. If specified, this symbolic name is unique and must be previously undefined. If no entry is specified in the label field, the directive defines BLANK common.
n1,n2,...,nn	are unique symbolic names which define the common area. The names can be subsequently referenced to define elements of common. Each entry in the operand field may optionally have the form: name(number) name is the location to begin allocation and number is the decimal number of words that are to be allocated. No embedded blanks are allowed between name and number.

The common block name specified in the label field of a COMMON directive may not be referenced in the operand field of other instructions or directives. Symbolic names used to define the common area result in the allocation of contiguous full word storage locations.

A maximum of 254 distinct common areas (including blank common) with a maximum 16K words per area can be defined.

Actual memory allocation for common areas in a subsequently generated task is decided by the CATALOG utility.

To initialize and manipulate data elements in common areas:

1. Set the location counter to the element of COMMON where initialization is to begin using the ORG directive. The symbol must have been defined by a preceding COMMON directive.

2. Define initialization data with the appropriate data generation directives.

3. Specify the appropriate assembly continuation mode with either the REL or ABS directive. Note: ABS is not supported by CATALOG.

The following example illustrates the sequence:

```
AB         COMMON        A(10),B(2)
           ORG           B
           DATAW         X'FF',X'3E8'
           REL
            .
            .
```

In this example, labeled common AB consists of twelve contiguous words of storage, ten words for area A and two for area B. The first word of area B is initialized to the value FF (hexadecimal) and the second word of area B is initialized to the value 3E8 (hexadecimal).

The following example defines a labeled common area XY consisting of eleven contiguous words of storage, ten words associated with area X and one word with common area Y. The second line defines the blank common area:

```
XY         COMMON        X(10),Y
           COMMON        A(20)
```

A COMMON directive may be used in conjunction with data generating directives and a PROGRAM directive to define a subprogram structure functionally equivalent to a FORTRAN block data subprogram.

COMMON block names GLOBAL00 through GLOBAL99, DATAPOOL, and DPOOL00 through DPOOL99 have the same special meaning as in FORTRAN. Elements within these areas may not be initialized.

4.7 Computed GOTO Directive

The Computed GOTO directive directs the assembly process to continue processing at the source statement whose label is given by a symbol in an indexed argument list in the operand field.

Syntax:

Label	Operation	Operand
label	GOTO	i,x1,x2,...,xn

i	is an arithmetic expression that represents an integer value. The value is used as an index pointer into the symbolic argument list.
x1,x2,...,xn	are symbols comprising the symbolic argument list. The symbols must represent forward references. The number of symbols in this list (n) should be equal to or greater than the integer value (i) used as an index.

If the index expression is evaluated as zero or an integer value greater than the integer value index, the assembly process continues at the source statement immediately following the COMPUTED GOTO directive.

4.8 CSECT Directive

The CSECT directive assembles a section of program source code in code section mode. All symbolic names (labels) are assigned relocatable memory addresses relative to the beginning of the code section.

Syntax:

Label	Operation	Operand
label	CSECT	

Usage:

The following example sets code section mode:

```
STRTC           CSECT
                LW              2,NUM
```

4.9 DATA Directive

The DATA directive allows specific data representation within a source program. Use of this directive results in automatic bounding for the location counter.

Syntax:

Label	Operation	Operand
symbol	DATAx	v1,v2,...vn

symbol — is an optional entry that may be any symbol. If specified, the symbol is equal to the address of the first operand.

x — specifies the bounding and length of the entries in the operand field as follows:

Variable	Field
B	Byte
H	Halfword
W	Word
D	Doubleword

If not specified, the default is B.

v1,v2,...vn — specifies the list of data values to be generated. These values may be any valid expressions.

Usage:

The following example illustrates attributed expression, address attribute, and constant string operands:

```
MPX-32 UTILITIES RELEASE 1.1 (ASSEMBLE R10.5.11)
   MAIN         12/19/83  16:57:51

 00001                                                 LIST      DATA
 00002     P00000                                      BOUND     1W
 00003     P00000  0064                       C.0      DATA      0,100
 00004     P00002      4142                   C.1      DATA      C'A',C'BC',C'DEF'
           P00004  43444546
 00005     P00008  01160808                   C.2      DATAB     N'1',22,8,8B,4H,2W
           P0000C  0808
 00006     P0000E      58                     C.3      DATAB     G'X'
 00007     P00010  0001                       C.4      DATAH     N'1'
 00008     P00012      FFFF                   C.5      DATAH     N'-1'
 00009     P00014  8001                       C.6      DATAH     32769
           P00016      0002
 00010     P00018  59202020                   C.7      DATAW     C'Y',G'Z'
           P0001C  0000005A
 00011     P00020  0007FFFC                   A.1      DATAW     X'0007FFFC'
 00012     P00024  00000002   P00002          A.2      DATAW     A(C.1)
 00013     P00028  00080002   P80002          A.3      DATAW     B(C.1)
 00014     P0002C  41200000                   A.4      DATAW     E'2.0'
 00015     P00030  52535420                   D.1      DATAD     C'RST'
           P00034  20202020
 00016     P00038  00000000                   D.2      DATAD     G'XYZ'
           P0003C  0058595A
 00017     P00040  00000000                   D.3      DATAD     1
           P00044  00000001
 00018     P00048  41100000                   D.4      DATAD     R'1.0'
           P0004C  00000000
 00019     P00050                                      END

*   0000  ERRORS IN MAIN
```

870027

4.10 DEF Directive

The DEF directive identifies linkage symbols within a given program which may be referenced by another program or subroutine as entry points or data.

The symbols referenced in the operand field must be defined in the same program in which the directive is used.

DEF directives must precede data definitions and executable statements in the source program.

Syntax:

Label	Operation	Operand
label	DEF	s1,s2,...,sn

s1,s2,...,sn	are symbolic names local to the program

See the EXT directive description.

4.11 DEFM Directive

The DEFM directive specifies the name of a macro. A macro definition must always begin with a labeled DEFM directive.

Syntax:

Label	Operation	Operand
name	DEFM	p1,p2,...pn

name	is a symbolic name which generates the macro when used in the operation field of a macro instruction
p1,p2,...pn	specify parameters that correspond to arguments supplied with the macro call

4.12 DSECT Directive

The DSECT directive assembles a section of program source code in data section mode. All symbolic names (labels) are assigned relocatable memory addresses relative to the beginning of the data section. DSECT is the default addressing mode used by the Macro Assembler.

Syntax:

Label	Operation	Operand
label	DSECT	

Usage:

The following example sets data section mode:

```
STARTD         DSECT
TCW1           RES           1W
```

4.13 END Directive

The END directive indicates the end of the source program and must always be the last Assembler language statement in a source program. END directive processing dumps all literals defined since the last LPOOL directive.

The operand field may contain an expression specifying a transfer address to which control is passed at load time. For a series of programs and subprograms, a transfer address is provided only with the main program. The operand field expression must not be literal.

Use of the label field is optional. If specified, the label is equal to the address of the first unused word location following the program.

Syntax:

Label	Operation	Operand
label	END	expression

4.14 ENDM Directive

The ENDM directive terminates a macro definition.

Syntax:

Label	Operation	Operand
label	ENDM	

There must be one ENDM directive for each macro definition. Entries in the label and/or operand fields are ignored.

The following sequence illustrates a macro definition that generates the load, store, and branch instructions each time the macro is called:

```
SAMPL       DEFM
            LW          3,ABC
            STW         3,XYZ
            BCT         2,TEST
            ENDM
```

4.15 ENDR Directive

The ENDR directive delineates the range of the repeat loop for the preceding REPT directive. This directive must be the last statement in a repeat loop. See the description of the REPT directive.

Syntax:

Label	Operation	Operand
label	ENDR	

4.16 EQU Directive

The EQU directive defines a symbol by assigning the attributes of the expression in the operand field to the symbol. This directive can be used to equate symbols to frequently used expressions such as register numbers, data, or arbitrary values.

The operand field must specify an evaluatable expression which may be absolute, relocatable, or common. External symbols or Datapool references cannot be specified. Symbols used in the operand field must have been previously defined if used in an arithmetic expression.

The label field must specify a symbolic name which cannot be redefined. The symbolic name assumes the same attributes as the expression. If the label field is not specified, the source statement is ignored.

Symbols used in the operand field must have been previously defined if BOUND, FORM, IFT, IFF, ORG, REPT, RES, or REZ directives reference the symbol in the label field of the EQU directive.

Syntax:

Label	Operation	Operand
symbol	EQU	expression

Usage:

The following example illustrates EQU directive usage when the operand is a constant expression:

Location Counter	Byte Address	Label	Operation	Operand
0002F	0002F	SLSH	EQU	X'2F'
00040	00040	ASGN	EQU	X'40'
0000A	0000A	CSZE	EQU	10
0005A	0005A	ZLET	EQU	G'Z'

4.17 EXITM Directive

The EXITM directive terminates processing of a macro structure. Label and operand field entries are ignored.

If this directive is used within a nested macro structure, assembly processing continues in the next outer macro, if applicable.

The EXITM directive should not be confused with the ENDM directive. The ENDM directive indicates the end of a macro definition and must be the last statement of any defined macro structure, including one that could contain one or more EXITM directives.

Syntax:

Label	Operation	Operand
label	EXITM	

4.18 EXT Directive

The EXT directive identifies linkage symbols which are entry points or data in another program or subroutine, but referenced by the given program.

The symbols referenced in the operand field must be defined in a different program than the one in which the EXT directive is used. The symbols are given defined addresses at load time if corresponding DEF directives in another program or subroutine are present.

Symbols defined by EXT directives may not be used within a common definition or in the operand field of the EQU directive.

Syntax:

Label	Operation	Operand
label	EXT	s1,s2,...sn

s1,s2,...sn	are symbolic names defined in another program or subroutine

Usage:

The following examples illustrate use of the EXT and DEF directives:

REFERENCING PROGRAM

Location Counter	Machine Instruction	Byte Address	Label	Operation	Operand
				PROGRAM	EXTDEF1
				EXT	CAL4
P00000			CAL5	EQU	$
P00000	D4000018	P00018		STW	0,CAL5R0
P00004	F8800001	X00000		BL	CAL4
P00008	F8800005	Y00004		BL	CAL4
P0000C	F8800009	Y00008		BL	CAL4
P00010	F880001D	P0001C		BL	CAL2
P00014	EC100019	P00018		BU	*CAL5R0
P00018	00000000		CAL5R0	DATAW	0
P0001C	D4000028	P00028	CAL2	STW	0,CAL2R0
P00020	C9800003			LI	3,3
P00024	EC100029	P00028		BU	*CAL2R0
P00028	00000000		CAL2R0	DATAW	0
P0002C				END	

REFERENCED PROGRAM

```
Location   Machine      Byte
Counter    Instruction  Address   Label    Operation   Operand

                                           PROGRAM     EXTDEF2
                                           DEF         CAL4
P00000                            CAL4     EQU         $
P00000     D4000014     P00014             STW         0,CAL4R0
P00004     D5800018     P00018             STW         3,WORD3
P00008     D600001C     P0001C             STW         4,WORD4
P0000C     D6800020     P00020             STW         5,WORD5
P00010     EC100015     P00014             BU          *CAL4R0
P00014     00000000               CAL4R0   DATAW       0
P00018     00000000               WORD3    DATAW       0
P0001C     00000000               WORD4    DATAW       0
P00020     00000000               WORD5    DATAW       0
P00024                                     END
```

4.19 FORM Directive

The FORM directive defines variable length data subfields. The bit size of each subfield is defined in the operand field of the directive. The data specification in the operand field is subsequently invoked when a source statement whose operation field matches the label field entry (symbol) of the FORM directive is encountered.

Syntax:

Label	Operation	Operand
symbol	FORM	f1,f2,...,fn

symbol	is a symbol that identifies the format definition given by the directive. When used as an operation in a subsequent source statement, this symbol invokes the format specification for the designated data constants.
f1,f2,...,fn	are positive integer values in the range 1 to 254 specifying the bit size of a given subfield. The number of subfields that may be defined is limited only by available storage. Symbols used in the operand field must be previously defined.

A subfield specification that exceeds 254 bits is flagged with the Assembler error code H. If this occurs in a FORM directive defining only a single subfield, the directive is flagged and processing continues with the next source statement. References to the erroneous specification are ignored with no code generated. If the 254-bit limit is exceeded in a FORM directive defining multiple subfields, the directive is flagged, directive processing is terminated, and the next source statement is processed. Subsequent references to the format specification result in correct code generation for subfields defined prior to the erroneous subfield.

A subfield specification of zero bits is ignored. No Assembler error code is generated and no code is generated for subsequent references to the format specification.

Data definitions that use fewer bits than specified for a subfield are aligned, justified, and/or zero-filled according to the conventions for the corresponding constant type (see Section 2, Data Representation).

For data definition entries exceeding the size of the specified subfield, high order bits are truncated. Alignment and justification are performed according to constant type.

Usage:

The following example generates one word with the following format:

- An 8-bit field containing the hexadecimal character A
- A 16-bit field containing the C-string characters YZ
- An 8-bit field containing the decimal number 15

Label	Operation	Operand
EXAMPL1	FORM	8,16,8
	.	
	.	
	.	
	EXAMPL1	X'A',C'YZ',N'15'

The following example illustrates excessive bit field specification, zero bit field specification, alignment, and justification for various constant types:

Error Line	Line Counter	Location Counter	Machine Instruction	Label	Operation	Operand
	00001			F1	FORM	32
H	00002			F2	FORM	4097
H	00003			F3	FORM	4096
	00004			F4	FORM	0
	00005			F5	FORM	1
	00006	P00000	41424320		F1	C'ABC'
	00007	P00004	00414243		F1	G'ABC'
	00008	P00008	41424344		F1	C'ABCDE'
	00009	P0000C	42434445		F1	G'ABCDE'
	00010				F2	1
	00011				F3	1
	00012				F4	X'44'
	00013	P00010	80		F5	1
	00014	P00014			END	

The following example is similar to the previous example. Note that the data specification format given in the FORM directive is serially reusable for cases where the actual number of subsequent data definitions is greater than the number of subfields specified.

```
Error  Line     Location  Machine
Line   Counter  Counter   Instruction   Label  Operation  Operand

       00001                            F1     FORM       32
       00002                            F2     FORM       254
H      00003                            F3     FORM       255
       00004                            F6     FORM       8,8,8,8
       00005                            F4     FORM       0
       00006                            F5     FORM       1
       00007    P00000    41424320             F1         C'ABC'
       00008    P00004    00414243             F1         G'ABC'
       00009    P00008    41424344             F1         C'ABCDE'
       00010    P0000C    42434445             F1         G'ABCDE'
       00011    P00010    00000000             F2         1
                P00014    00000000
                P00018    00000000
                P0001C    00000000
                P00020    00000000
                P00024    00000000
                P00028    00000000
                P0002C    00000004
       00012                                   F4         X'44'
       00013    P00030    80                   F5         1
       00014    P00031      040404             F6         4,4,4
       00015    P00034    04040404             F6         4,4,4,4,4,4,4
                P00038    040404
       00016    P0003C                         END
```

4.20 GEN Directive

The GEN directive constructs a hexadecimal value representing specified bit string combinations.

The operand specifies a field list comprised of subfields separated by commas. Each subfield is packed into a contiguous bit string. If the total length of all subfields in bits is not a multiple of eight, the bit string is zero-filled to achieve byte multiplicity.

All relocatable, common, and external mode address fields must be 20 to 32 bits in length and right-justified within a word. Fields of other types may range in length from 0 to 4096 bits.

Precise bounding can be achieved through use of the BOUND directive immediately preceding the GEN directive.

Syntax:

Label	Operation	Operand
label	GEN	f1/e1,f2/e2,...fn/en

f1,f2,...fn	is a positive integer specifying the bit length of a given subfield
/	separates subfield bit length and subfield contents
e1,e2,...en	is an expression specifying the contents of a given subfield

Usage:

The following GEN directive specification generates a 32-bit word. The first 12 bits contain the hexadecimal value FFF and the remaining 20 bits contain the word address location for ALPHA.

```
GEN     12/X'FFF',20/W(ALPHA)
```

The following GEN directive specification generates a 64-bit field. The first 24 bits contain the decimal value 1, the next 24 bits contain the hexadecimal value 374AC1, and the last 16 bits contain character codes for the representation XY:

```
GEN     24/1,24/X'374AC1',16/C'XY'
```

4.21 GOTO Directive

The GOTO directive directs the Macro Assembler to continue processing at the source statement whose label is indicated by the symbol in the operand field. Source statements between the GOTO directive and the specified source statement are not processed.

The symbol entry in the operand field must be a forward reference.

Syntax:

Label	Operation	Operand
label	GOTO	symbol

4.22 IFA Directive

The IFA directive can only be used within a macro structure.

The IFA directive checks the symbolic parameter list of the expression in the operand field for the existence of actual arguments. If absent, processing continues at the source statement immediately following the IFA directive. If arguments are present, processing branches to the source statement with a label specified by symbol2.

Symbols defined by the SET, SETF, and SETT directives may be used in IFT, IFP, IFA, and IFF directives. These symbols must not represent forward references.

Symbols used in the label field of IFT, IFF, IFP, and IFA directives are not entered into the symbol table.

Syntax:

Label	Operation	Operand
label	IFA	exp,symbol2

exp	is a valid expression. This expression typically specifies a symbolic parameter of the form %xx...x, or a string of symbolic parameters combined with arithmetic and/or logical operators (e.g., %AB+%CB+%XY+...). This expression could also represent a value corresponding to a locally generated label within a macro.
symbol2	is a valid symbol specifying the label of the source statement to branch to if the expression contains internally generated symbols

4.23 IFF Directive

The IFF directive evaluates the expression in the operand field. If the expression is evaluated as a FALSE (0), assembly processing continues with the source statement immediately following the directive. If the expression is evaluated as TRUE (1), assembly processing continues at the source statement with the label specified by symbol2.

The following information applies to the IFF and IFT directives.

The expression specified in the operand field may consist of a string combined with arithmetic operators (+ - * /) and/or logical operators (.AND. .OR. .EQ. .NE. .LT. .GT. .LE. .GE.). Logical operators must be preceded and followed by a blank (e.g., XYZ .NE. XYW).

Only the least significant 23 bits of the expression or logical subexpression are used in determining the logical value for the entire expression.

Syntax:

Label	Operation	Operand
label	IFF	exp,symbol2

exp	is a valid expression. This expression typically specifies a symbolic parameter of the form %xx...x, or a string of symbolic parameters combined with arithmetic and/or logical operators. This expression could also represent a value corresponding to a locally generated label within a macro. Symbols used in exp must have been previously defined.
symbol2	is a valid symbol specifying the label of the source statement to branch to if the expression is evaluated as TRUE (1).

4.24 IFP Directive

The IFP directive can only be used within a macro structure.

The IFP directive checks the symbolic parameter list of the expression in the operand field for the existence of actual arguments. If present, processing continues with the source statement immediately following the IFP directive. If arguments are absent, processing branches to the source statement with a label specified by symbol2.

See the IFA directive description for further details.

Syntax:

Label	Operation	Operand
label	IFP	exp,symbol2

exp	is a valid expression (see IFA directive description)
symbol2	is a valid symbol specifying the label of the source statement to branch to if the expression does not contain internally generated symbols

4.25 IFT Directive

The IFT directive evaluates the expression in the operand field. If the expression is evaluated as TRUE (1), assembly processing continues with the source statements immediately following the directive. If the expression is evaluated as FALSE (0), assembly processing continues at the source statement with the label specified by symbol2.

See the IFF directive description for further details.

Syntax:

Label	Operation	Operand
label	IFT	exp,symbol2

exp	is a valid expression (see IFF directive description)
symbol2	is a valid symbol specifying the label of the source statement to branch to if the expression is evaluated as FALSE (0)

4.26 LIST Directive

The LIST directive controls listed output. This directive is meaningful only when listed output has not been inhibited through the use of option l.

Syntax:

Label	Operation	Operand
label	LIST	term1,...,term5

term1,...,term5 specify one to five terms from the following operator pairs:

List Option	Results
ON	Generate listed output (default).
OFF	Suppress listed output.
DATA	Print data in full with listed output. Additional print lines may be required (default).
NODATA	Restrict data printing to single source image. Suppress additional lines.
NGLIST	List all source statements (default).
NONG	Suppress listed output for the following types of statements: (1) ANOP, GOTO, SET, SETT, SETF, IFT, IFF, IFP, IFA DEFM, ENDM, EXITM, REPT, END, and FORM directives (2) Statements within macro prototype definitions (3) Statements within REPT loop definitions (4) Statements not assembled because of conditional assembly
MAC	List macro expansions (default).
NOMAC	Suppress listed output of macro expansions.
REP	List REPT loop expansions (default).
NOREP	Suppress listed output of REPT loop expansions.

If multiple or conflicting option are specified, the last option specified is the controlling element. This includes specification of LIST options within macros.

If a LIST directive is not specified, Macro Assembler operation assumes the following defaults for listed output control:

```
ON,DATA,NGLIST,MAC,REP
```

LIST directives specifying the ON option while under control of a LIST OFF directive are not listed.

4.27 LPOOL Directive

The LPOOL directive inserts literals into the generated object code beginning at the current location within the program. Processing of the LPOOL directive causes all literals since the previous LPOOL directive (or start of the program) to be assembled at word boundaries starting at the first full word boundary following the directive. If no literals are assembled as the result of a given LPOOL directive, the location counter is advanced to the next full word boundary following the LPOOL directive.

Syntax:

Label	Operation	Operand
symbol	LPOOL	

symbol is an optional symbolic entry. If specified, this entry represents the address of the first unused full word location following literal pool output.

Only the first 31 LPOOL directives are processed. All others are not processed, and are flagged with the Assembler error code Z.

All literals encountered between the last LPOOL directive of a program and the END directive are dumped as part of the Assembler's END directive processing.

```
MPX-32 UTILITIES RELEASE 1.1 (ASSEMBLE R10.5.11)
  MAIN         02/28/84   16:20:24

 00001                                          *
 00002                                          * THIS EXAMPLE SHOWS THE USE OF THE LPOOL DIRECTIVE TO FORCE THE
 00003                                          * LITERALS TO BE INCLUDED IN THE OBJECT CODE AT THE CURRENT
 00004                                          * PROGRAM COUNTER LOCATION.
 00005                                          *
 00006                                          * THIS PERMITS THE LITERALS TO BE POSITIONED AWAY FROM THEIR NORMAL
 00007                                          * PLACE AT THE END OF THE PROGRAM. THIS PERMITS USERS TO USE
 00008                                          * EXTRA PROGRAM SPACE VIA SALLOCATE AND TESTING OF THE PRCGRAM'S
 00009                                          * ADDRESS LIMITS.
 00010                                          *
 00011                                          * IN THIS EXAMPLE THE LITERALS WOULD APPEAR IN THE GENERATED CODE
 00012                                          * BETWEEN M.EXIT AND THE START OF BUF. HENCE BY USING M.GADRL, THE
 00013                                          * PROGRAM CAN USE THE SPACE BETWEEN BUF+99W AND THE LAST LOCATION
 00014                                          * CURRENTLY AVAILABLE IN THE TASK'S CONTIGUOUSLY ALLOCATED DSECT.
 00015                                          *
 00016   P00000  C8800001                       STRT      LI        1,1
 00017   P00004  ADC00010  P00C10                         LW        2,=C'ABC'       A LITERAL WILL BE GENERATED
 00018   P00C08  EC00000D  P0000C                         BU        STP
 00019   P0000C                                 STP       M.EXIT                    (SVC 1,X'55')
         P0000C  C8061055                                 SVC       1,X'55'
                                                          ENDM
         P00010  41424320
 00020                                                    LPOOL                     INCLUDE LITERALS IN GENERATED CCDE
 00021   P00014                                 BUF       RES       100W
 00022   P001A4            P00000                         END       STRT

*  0000  ERRORS IN MAIN
```

870024

4.28 ORG Directive

The ORG directive assigns the value specified in the operand field to the location counter. Symbolic names are assigned absolute or relocatable values relative to the point of origin until a subsequent ABS, REL, or ORG directive is encountered.

Symbols used in the operand field must have been previously defined.

External references may not be used in the operand field.

Syntax:

Label	Operation	Operand
label	ORG	value

Usage:

The following example assigns the value 1000 (hexadecimal) to TAGA and START:

```
TAGA          ORG           X'1000'
START         LW            2,TAGA
```

4.29 PAGE Directive

The PAGE directive causes a page eject on the listed output (LO) device. The current TITLE identification is printed on each new page. The PAGE directive is not printed but is assigned a line number.

Syntax:

Label	Operation	Operand
label	PAGE	

4.30 PROGRAM Directive

The PROGRAM directive identifies Macro Assembler generated programs and specifies the program name to be printed on each page of listed output. Only one PROGRAM directive can be specified within one program assembly.

If a PROGRAM directive is not specified, the program name defaults to MAIN. If the operand field of a PROGRAM directive is blank, the program module is assembled with no name.

Syntax:

Label	Operation	Operands
label	PROGRAM	[name [id]]

name	is a string of one to eight alphanumeric characters that specifies a program name. Embedded blanks are not permitted.
id	is a string of up to 20 alphanumeric characters for optional identification information. This field is inserted into the object code if option 14 is set. The program name field must be specified if the id field is specified.

4.31 REL Directive

The REL directive assembles a section of program coding in relocatable mode. Generated object code can be relocated by the loader. All symbolic names are assigned relocatable memory addresses relative to the program start location. Subsequent use of this directive causes the location counter to be assigned a value equal to the next relative memory address to be allocated.

Syntax:

Label	Operation	Operand
label	REL	

Usage:

The following example assembles the code in the relocatable mode beginning at location 1000 (hexadecimal):

```
TAG               REL
                  ORG               X'1000'
```

4.32 REPT Directive

The REPT directive allows the repetitive generation of a sequence of coding or data. The number of repetitions is specified by an expression in the operand field. This directive is typically used within repeat loops.

The special symbol $$$ can be used in coding within a repeat loop, and is always equal to the current repetition count. The value of the special symbol $$$ is initially equal to one.

REPT directives may not be utilized in nested loop structures.

The FORM directive cannot be used within the range of a repeat loop.

Syntax:

Label	Operation	Operand
label	REPT	expression

expression	specifies the repetition count. This expression is evaluated as an arithmetic expression. If the value specified is zero or negative, the assembly process is suspended until an ENDR directive is encountered. Symbols used in the operand field must have been previously defined.

Usage:

The following example illustrates typical usage of the REPT and ENDR directives.

Code:

```
        TABLE      REPT       3
                   GEN        16/A($-TABLE),16/A($$$)
                   ENDR
```

Result:

```
        TABLE      GEN        16/0,16/1
                   GEN        16/4,16/2
                   GEN        16/8,16/3
```

4.33 RES Directive

The RES directive reserves blocks of storage for use as tables, data arrays, or work areas.

Halfword, word, doubleword, and file unit specifications are adjusted for proper bounding.

If the operand field contains an evaluatable expression, the location counter is adjusted by the defined number of bytes with no bounding performed. Symbols used in the operand field must have been previously defined.

Syntax:

Label	Operation	Operand
symbol	RES	ns
symbol	RES	expression

symbol	is an optional symbol that is assigned the value of the location counter at the time the first location of the reserved block is allocated
n	is a decimal integer value designating the unit multiple to be reserved
s	specifies the size of the unit to be reserved as follows:

Variable	Field
B	Byte
H	Halfword
W	Word
D	Doubleword
F	File (8 words)

If not specified, the default is B.

Usage:

The following example reserves storage as indicated:

Code:

```
(LC=00000)       RES      3H
(LC=00006)       RES      1
(LC=00008)       RES      1D
```

Result:

```
0000        1HW                    1HW
0004        1HW         1B              Unused
0008                1DW
000C
```

The following sample program listing illustrates the use of the RES directive:

Line Number	Location Counter	Label	Operation	Operand
01410	P017A8	TXPR	RES	1W
01411	P017AC	PRAF	RES	1W
01412	P017B0	DCTV	RES	1W
01413	P017B4	EXTC	RES	1W
01414	P017B8	TEXT	RES	30W
01415	P01830	STAT	RES	10W
01416	P01858	HDR	RES	1D
01417	P01860	IPRF	RES	1D
01418	P01868	TDRT	RES	1H
01419	P0186A	KSRF	RES	1B
01420	P0186B	RCFG	RES	1B

4.34 REZ Directive

The REZ directive reserves and zeroes blocks of storage for use as tables, data arrays, and/or work areas.

Halfword, word, doubleword, and file unit specifications are adjusted for proper bounding.

If the operand field contains an evaluatable expression, the defined number of bytes is reserved and zeroed with no bounding performed. Symbols used in the operand field must have been previously defined.

Syntax:

Label	Operation	Operand
symbol	REZ	ns
symbol	REZ	expression

symbol — is an optional symbol that is assigned the value of the location counter at the time the first location of the reserved block is zeroed

n — is a decimal integer value designating the unit multiple to be reserved and zeroed

s — specifies the size of the unit to be reserved and zeroed as follows:

Variable	Field
B	Byte
H	Halfword
W	Word
D	Doubleword
F	File (8 Words)

If not specified, the default is B.

Usage:

The following example reserves and zeroes storage as indicated:

Code:

```
(LC=00000)          REZ          3H
(LC=00006)          REZ          1
(LC=00008)          REZ          1D
```

Result:

```
00000       1HW       0000         1HW        0000

00004       1HW       0000    1B       00     Unused

00008                 0000         0000
            1DW
0000C                 0000         0000
```

The following sample program listing illustrates the use of the REZ directive:

```
Line     Location   Generated
Number   Counter    Code        Label   Operation   Operand

00005                                   LIST        NODATA
00006    P017A8     00000000    TXPR    REZ         1W
00007    P017AC     00000000    PRAF    REZ         1W
00008    P017B0     00000000    DCTV    REZ         1W
00009    P017B4     00000000    EXTC    REZ         1W
00010    P017B8     00000000    TEXT    REZ         30W
00011    P01830     00000000    STAT    REZ         10W
00012    P01858     00000000    HDRT    REZ         1D
00013    P01860     00000000    IPRF    REZ         1D
00014    P01868     0000        TDRT    REZ         1H
00015    P0086A         00      KSRF    REZ         1B
00016    P0186B           00    RCFG    REZ         1B
```

4.35 SET Directive

The SET directive assigns the value of the expression in the operand field to the symbolic name specified in the label field. The expression is evaluated as an arithmetic expression and may consist of a string combined by arithmetic operators (+ - * /).

The symbol defined by the SET directive may be subsequently redefined any number of times.

Symbols used in the operand field must have been previously defined if BOUND, FORM, IFT, IFF, ORG, REPT, RES, or REZ directives reference the symbol in the label field of the SET directive.

Syntax:

Label	Operation	Operand
symbol	SET	expression

4.36 SETF Directive

The SETF directive assigns the Boolean value false (0) to the symbolic name specified in the label field.

Syntax:

Label	Operation	Operand
symbol	SETF	

4.37 SETT Directive

The SETT directive assigns the Boolean value true (1) to the symbolic name specified in the label field.

Syntax:

Label	Operation	Operand
symbol	SETT	

4.38 SPACE Directive

The SPACE directive skips a specified number of lines on listed output. The SPACE directive is not printed but is assigned a line number.

Syntax:

Label	Operation	Operand
label	SPACE	n

n	is an integer value in the range 1 to 59 specifying the number of lines to skip. If n is specified as a blank, zero, or negative integer, the output device skips one line.

4.39 TITLE Directive

The TITLE directive specifies a heading to be printed on each page of Macro Assembler listed output. The heading is printed until a subsequent TITLE directive is encountered.

When encountered, a TITLE directive causes a page eject on the listed output device prior to printing the new TITLE heading.

The TITLE directive is not printed but is assigned a line number.

Syntax:

Label	Operation	Operand
label	TITLE	cs

cs	is a string of ASCII characters comprising the heading identification. The text consists of all characters encountered between the blank input column immediately following the directive and input column 72, exclusive. This field may be blank.

SECTION 5 - ERRORS AND ABORTS

5.1 Error Codes

Errors detected by the Macro Assembler during Pass One and Pass Two processing are printed on the program's listed output.

During Pass One, errors which will not be resolved during Pass Two are detected. Pass One errors are printed before the source program listing on the LO file or device. The Pass One error statement contains the alphabetic error code, source program line number, and source image.

Pass Two errors are listed in the error flag field on the LO file or device. The total number of Pass Two errors is given in the error report line following the source program listing.

Refer to Section 6 for a sample Assembler output listing showing the location of error code flags.

Macro Assembler error codes are denoted by an alphabetic character as follows:

Error Code	Explanation
A	(1) Nested Macro Definition(s) (2) External Reference from Common (3) External Reference from Noncode-generating Statement (4) FORM Directive Encountered during Macro Expansion (5) FORM Directive Encountered within REPT Loop
B	(1) Addressing Boundary Error (2) Odd Register Specification in Double Register Operand (3) Improper Register Usage (4) Illegal Condition Code for BCF/BCT Instructions (5) Bit Value Out of Range (0-31) for External Variable
C	Character Constant String Improperly Terminated
D	Data Statement Field Error
E	Expression Improperly Terminated
F	(1) DEF Undefined (2) DEF Improperly Defined (3) DEF Does Not Precede Executable Code/Data
G	SPACE Directive Specifies Excessive Line Count
H	GEN or FORM Directive Specifies Excessive Field Size
I	(1) Internally Generated Symbol Specified as COMMON Block Name (2) Internally Generated Symbol Specified as External

Error Code	Explanation
	(3) Internally Generated Symbol Used in Concatenated String Used as Macro Argument
J	Attempt to Use Previously Defined Symbol as Common Block Name
K	Illegal Option Specified for LIST Directive
L	Data Constant Definition Error
M	Symbol Multiply Defined
N	Numeric Constant Field Error
O	Operation Field Error
P	Multiple PROGRAM Directives
Q	(1) Illegal Location Specified with ORG Directive (2) Illegal Boundary Specified with BOUND Directive
R	(1) Relocatable/Common/External Symbolic Field Too Small (2) COMMON size $\geq$ 16KW
T	(1) Phasing Error - Pass Two Symbolic Value Not Equal to Pass One Symbolic Value (2) Phasing Error - Symbol Detected In Pass Two that Was Not Defined by Pass One
U	Symbol Undefined
V	Illegal Combination of Symbols in Expression
W	Illegal Constant Type or Address Attribute
X	(1) Literal Not Defined during Pass One (2) END Directive Processing Error (3) Illegal Number of Arguments Specified (4) Illegal Type for Literal
Y	Violation of REPT Usage Conventions
Z	Excessive LPOOL Directives (Maximum 31)
\ (Backslash)	Attempted Division by Zero Is Illegal

5.2 Abort Codes

If the Macro Assembler aborts, an abort code or message is shown on the listed output assigned to logical file code LO. The following is a list of the codes, messages, and action to take when a program abort occurs.

AS01 PHYSICAL END-OF-FILE ENCOUNTERED ON WRITE TO THE GENERAL OBJECT (GO) FILE

If logical file code GO is assigned to SGO, the maximum number of extents was reached. Specify a larger SGO size to reduce the number of extents needed.

If logical file code GO is assigned to a file, the file is too small and could not be extended, or the maximum number of extends was reached. Recreate the file with a larger size so it can be extended.

AS02 PHYSICAL END-OF-FILE ENCOUNTERED ON WRITE TO THE BINARY OUTPUT (BO) FILE

If logical file code BO is assigned to SBO, the maximum number of extents was reached. Specify a larger SBO size to reduce the number of extents needed.

If logical file code GO is assigned to a file, the file is too small and could not be extended, or the maximum number of extents was reached. Recreate the file with a larger size so it can be extended.

AS03 PHYSICAL END-OF-FILE ENCOUNTERED ON WRITE TO THE LISTED OUTPUT (LO) FILE

If logical file code LO is assigned to SLO, the maximum number of extents was reached. Specify a larger SLO size to reduce the number of extents needed.

If logical file code LO is assigned to a file, the file is too small and could not be extended, or the maximum number of extents was reached. Recreate the file with a larger size so it can be extended.

AS04 PHYSICAL END-OF-FILE ENCOUNTERED ON WRITE TO THE SCRATCH (UT1) FILE (I.E., $AS UT1 TO TEMP SIZE=800 BLOC=Y)

The maximum number of extents was reached for the file assigned to UT1. Specify a larger size to reduce the number of extents needed.

AS05 PHYSICAL END-OF-FILE ENCOUNTERED ON WRITE TO THE CROSS-REFERENCE (UT2) FILE (I.E., $AS UT2 TO TEMP SIZE=400 BLOC=N)

The maximum number of extents was reached for the file assigned to UT2. Specify a larger size to reduce the number of extents needed.

AS06 Reserved

AS07 UNRECOVERABLE I/O ERROR ON THE BINARY OUTPUT (BO) FILE

Improper request for unblocked output. Correct the JCL and rerun the program.

AS08 UNRECOVERABLE I/O ERROR ON THE GENERAL OBJECT (GO) FILE

Possible hardware problem. Rerun the program.

AS09 UNRECOVERABLE I/O ERROR ON THE LISTED OUTPUT (LO) FILE

Possible hardware problem. Rerun the program.

AS10 UNRECOVERABLE I/O ERROR ON THE SOURCE INPUT (SI) FILE

Possible hardware problem. Rerun the program.

AS11 UNRECOVERABLE I/O ERROR ON THE INTERMEDIATE COMPRESSED SOURCE (UT1) FILE

Possible hardware problem. Rerun the program.

AS12 PHYSICAL END-OF-FILE ENCOUNTERED ON WRITE TO THE COMPRESSED SOURCE OUTPUT (CS) FILE

If CS is assigned to a file, the maximum number of extents was reached. Recreate the file at a larger size to reduce the number of extents needed.

AS13 CHECKSUM ERROR ON COMPRESSED SOURCE INPUT DURING PASS 1 WHILE READING COMPRESSED SOURCE FROM THE SOURCE INPUT (SI) FILE OR DURING PASS 2 WHILE READING THE INTERMEDIATE SCRATCH COMPRESSED SOURCE (UT1) FILE.

Rerun the program. If the program aborts a second time with this abort code, access the program's file with the Text Editor, SAVE the file, and rerun the program. If the program aborts with this code again, a hardware problem, such as a bad spot on a disc, probably exists.

AS14 THE FILE THE ASSEMBLER IS USING AS THE MACRO LIBRARY WAS NOT SUCCESSFULLY CREATED BY THE MACRO LIBRARY EDITOR. THE FILE IS INVALID.

AS15 UNRECOVERABLE I/O ERROR ON THE MACRO LIBRARY (MAC) FILE

Possible hardware problem. Rerun the program.

AS16 UNRECOVERABLE I/O ERROR ON THE CROSS-REFERENCE (UT2) FILE

Possible hardware problem. Rerun the program.

AS17 UNRECOVERABLE I/O ERROR ON THE COMPRESSED SOURCE OUTPUT (CS) FILE

Possible hardware problem. Rerun the program.

AS18 INVALID BLOCKING BUFFER CONTROL POINTER ENCOUNTERED ON THE BINARY OUTPUT (BO) FILE

AS19 INVALID BLOCKING BUFFER CONTROL POINTER ENCOUNTERED ON THE GENERAL OBJECT (GO) FILE

AS20 INVALID BLOCKING BUFFER CONTROL POINTER ENCOUNTERED ON THE LISTED OUTPUT (LO) FILE

AS21 INVALID BLOCKING BUFFER CONTROL POINTER ENCOUNTERED ON THE SOURCE INPUT (SI) FILE

AS22 INVALID BLOCKING BUFFER CONTROL POINTER ENCOUNTERED ON THE SCRATCH COMPRESSED SOURCE (UT1) FILE

AS23 INVALID BLOCKING BUFFER CONTROL POINTER ENCOUNTERED ON THE COMPRESSED SOURCE OUTPUT (CS) FILE

AS24 INVALID BLOCKING BUFFER CONTROL POINTER ENCOUNTERED ON THE CROSS-REFERENCE (UT2) FILE

AS25 THE MACRO LIBRARY (MAC) FILE IS UNBLOCKED

Make sure the assignment for MAC is unblocked. See Section 2.2.2.

AS26 END-OF-FILE ON MA2 FILE

The file assigned to MA2 is too small for the job. Increase the size specified by the assignment so that UT2 is large enough to hold the program.

AS27 UNRECOVERABLE I/O ERROR ON MA2 FILE

Possible hardware problem. Rerun the program.

AS28 INVALID BLOCKING BUFFER CONTROL POINTER ON MA2 FILE

AS29 MAC ASSIGNED TO ILLEGAL DEVICE

The Assembler probably encountered a problem opening the file assigned to MAC. The device may not be configured correctly for use by the Assembler.

AS30 MA2 ASSIGNED TO ILLEGAL DEVICE

The Assembler probably encountered a problem opening the file assigned to MA2. The device may not be configured correctly for use by the Assembler.

AS31 ERROR(S) (DESCRIBED ON LFC LO) DETECTED DURING EXECUTION

The program did not assemble. A problem exists within the source code and is flagged with an Assembler error code on the LO.

AS32 UNRECOVERABLE I/O ERROR ON THE PREFIX (LFC PRE) FILE

Possible hardware problem. Rerun the program.

AS33 INVALID BLOCKING BUFFER CONTROL POINTER ENCOUNTERED ON THE PREFIX (LFC PRE) FILE

5.3 Error Messages

The following situations do not report abort codes, although they cause Macro Assembler programs to abort and generate the following error messages:

Message: ** BAD MACRO ENCOUNTERED DURING MACRO SEARCH **

The Assembler attempted to read an improperly formatted macro library.

Message: ** XREF COULD NOT BE PERFORMED **

A cross-reference was not generated because there was insufficient memory available to sort the cross-reference information.

The message above is also produced if there is insufficient memory available to store the required macros and the symbol table.

Message: ** SYMBOL TABLE OVERFLOW **

The number of symbols in a program exceeded the number of symbols the symbol table can hold.

Message: ** UNABLE TO ALLOCATE MEMORY FOR MACRO STORAGE **

The macro table size is exceeded due to excessive bytes of in-line macros, FORM skeletons, repeated code, or macro call argument data.

Message: ** UNEXPECTED END STATEMENT, PROBABLY MISSING ENDM, ENDR OR CONDITION LABEL **

An END statement was detected while processing a macro prototype, repeat loop, or conditionally skipping code.

SECTION 6 - OUTPUT AND EXAMPLES

6.1 Introduction

The Macro Assembler optionally produces listed output, object program output, and/or compressed source output.

Listed output typically consists of a program source listing with a symbol cross-reference table and an error diagnostic report. The LIST, PAGE, SPACE, and TITLE Assembler directives provide control of the quantity and format of listed output.

Object program output is a binary representation of assembled machine instructions, data, and encoded loader function codes. The object program serves as input for a cataloger or linking-type loader which physically loads the program into memory. The loading process involves allocation of memory space for the program, resolution of external and internal symbolic references, and relocation of address-dependent locations.

The Macro Assembler optionally produces source output in compressed format on cards, magnetic tape, or disc. The option is specified at assembly time by a JCL statement. Details of the compressed source format are in Appendix C.

6.2 Source Listing

The source listing produced by the Macro Assembler pairs a hexadecimal representation of object code with the corresponding source program statement.

Figure 6-1 illustrates typical Assembler listed output. The basic format is organized in columns. The following paragraphs provide details for each of the circled entries in Figure 6-1.

1 Program Name - The name specified in the PROGRAM directive is printed at the top of each listed output page. If a PROGRAM directive is not specified, the program name defaults to MAIN. If a PROGRAM directive is specified with a blank operand field, no program name is printed.

2 Statement Counter - The five-digit statement counter numbers each source code statement in the program.

3 Error Flag - Macro Assembler error flags are indicated by one alphabetic character per error condition. If more than one error of the same type occurs within a statement, the error flag is printed only once. If more than one different type of error occurs within the same statement, an error code is printed for each type; if more than two types of errors occur within the same statement, they are continued on the next line.

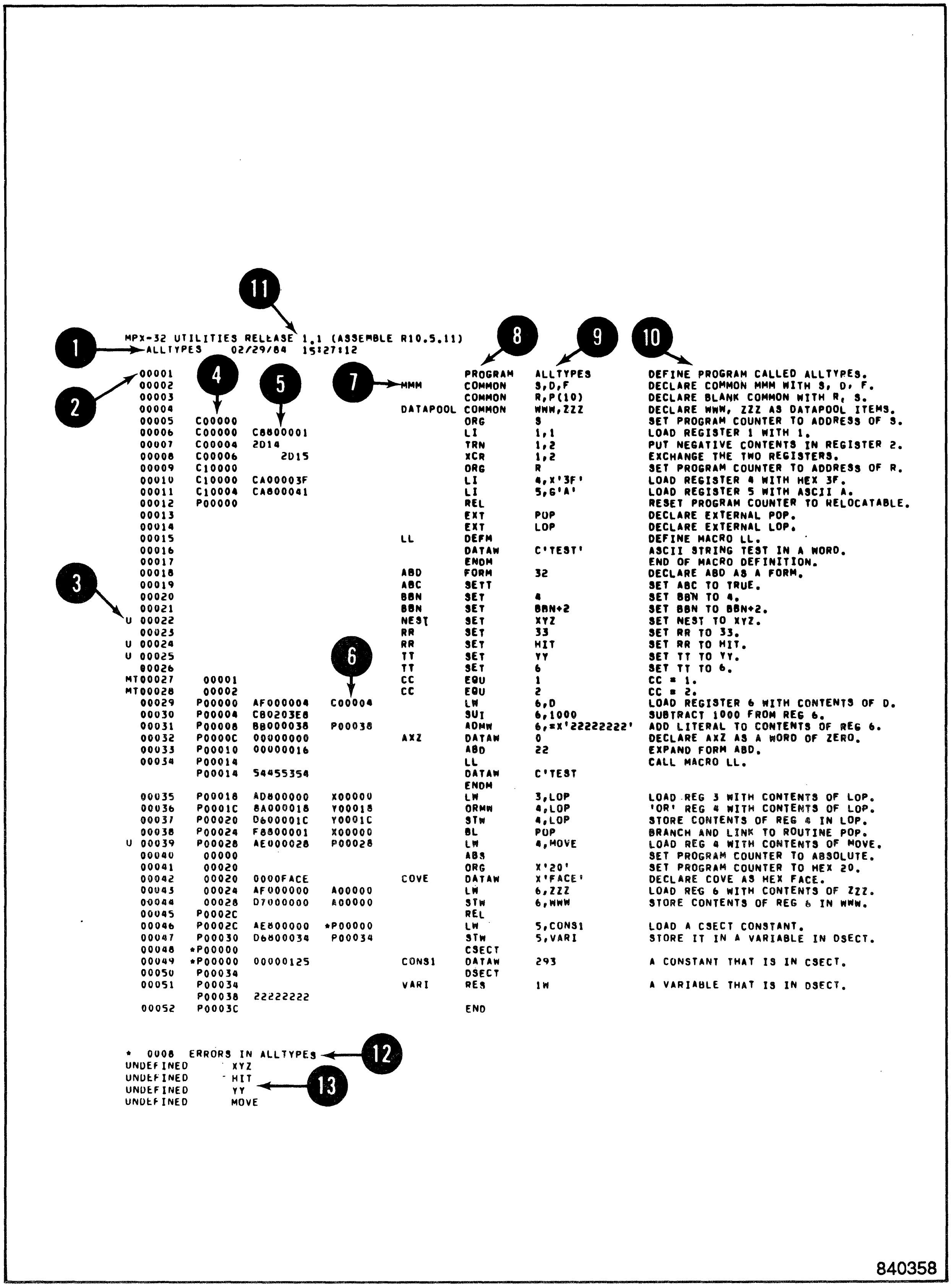
4 Location Counter - Specifies the mode and hexadecimal value of the location counter in byte increments. The five least significant positions represent the location counter. The most significant character indicates one of the following address modes:

Character	Mode	Location Counter Significance
Blank	ABSOLUTE	Absolute address of the generated code
C	COMMON	Leftmost digit is the least significant digit of the common block number into which code is being generated. Least significant four digits give the relative address within the common block at which code is being generated.
P	RELOCATABLE	Relocatable address of the generated code
*	CSECT	Address reference is within a code section

5 Object Code - Contains the hexadecimal representation of object code for the corresponding source program instruction. The positioning indicates byte/halfword position within a word. In the case of multiword instructions, multiple lines of object code are generated.

6 Symbol Address - Specifies the type and hexadecimal address of a referenced symbol. The address is given in bytes by the five least significant positions. The address corresponds to the word address operand format code in the object output. The most significant character indicates the symbol type and address significance as follows:

Character	Symbol Type	Address Significance
Blank	ABSOLUTE	Absolute address or value of the referenced symbol
C	COMMON	Leftmost digit is the least significant digit of the common block number in which the referenced symbol occurs. Least significant four digits give the relative address of the referenced symbol within the common block.
P	RELOCATABLE	Relocatable address of the referenced symbol
X	EXTERNAL (Absolute Linked)	Zero on first reference to symbol. For all other occurrences, this is the program address of the last instruction that referenced the symbol (absolute programs only).
Y	EXTERNAL (Relocatable Linked)	Program address of the last instruction that referenced the symbol (relocatable programs only)
A	DATAPOOL	Value is always zero
*	CSECT	Address reference is within a code section



```
MPX-32 UTILITIES RELEASE 1.1 (ASSEMBLE R10.5.11)
  ALLTYPES     02/29/84  15:27:12

  00001                                              PROGRAM   ALLTYPES          DEFINE PROGRAM CALLED ALLTYPES.
  00002                                    MMM       COMMON    S,D,F             DECLARE COMMON MMM WITH S, D, F.
  00003                                              COMMON    R,P(10)           DECLARE BLANK COMMON WITH R, S.
  00004                                    DATAPOOL  COMMON    WWW,ZZZ           DECLARE WWW, ZZZ AS DATAPOOL ITEMS.
  00005   C00000                                     ORG       S                 SET PROGRAM COUNTER TO ADDRESS OF S.
  00006   C00000   C8800001                          LI        1,1               LOAD REGISTER 1 WITH 1.
  00007   C00004   2D14                              TRN       1,2               PUT NEGATIVE CONTENTS IN REGISTER 2.
  00008   C00006       2D15                          XCR       1,2               EXCHANGE THE TWO REGISTERS.
  00009   C10000                                     ORG       R                 SET PROGRAM COUNTER TO ADDRESS OF R.
  00010   C10000   CA00003F                          LI        4,X'3F'           LOAD REGISTER 4 WITH HEX 3F.
  00011   C10004   CA800041                          LI        5,G'A'            LOAD REGISTER 5 WITH ASCII A.
  00012   P00000                                     REL                         RESET PROGRAM COUNTER TO RELOCATABLE.
  00013                                              EXT       POP               DECLARE EXTERNAL POP.
  00014                                              EXT       LOP               DECLARE EXTERNAL LOP.
  00015                                    LL        DEFM                        DEFINE MACRO LL.
  00016                                              DATAW     C'TEST'           ASCII STRING TEST IN A WORD.
  00017                                              ENDM                        END OF MACRO DEFINITION.
  00018                                    ABD       FORM      32                DECLARE ABD AS A FORM.
  00019                                    ABC       SETT                        SET ABC TO TRUE.
  00020                                    BBN       SET       4                 SET BBN TO 4.
  00021                                    BBN       SET       BBN+2             SET BBN TO BBN+2.
U 00022                                    NEST      SET       XYZ               SET NEST TO XYZ.
  00023                                    RR        SET       33                SET RR TO 33.
U 00024                                    RR        SET       HIT               SET RR TO HIT.
U 00025                                    TT        SET       YY                SET TT TO YY.
  00026                                    TT        SET       6                 SET TT TO 6.
MT00027    00001                           CC        EQU       1                 CC = 1.
MT00028    00002                           CC        EQU       2                 CC = 2.
  00029   P00000   AF000004    C00004                LW        6,D               LOAD REGISTER 6 WITH CONTENTS OF D.
  00030   P00004   CB0203E8                          SUI       6,1000            SUBTRACT 1000 FROM REG 6.
  00031   P00008   BB000038    P00038                ADMW      6,=X'22222222'    ADD LITERAL TO CONTENTS OF REG 6.
  00032   P0000C   00000000                AXZ       DATAW     0                 DECLARE AXZ AS A WORD OF ZERO.
  00033   P00010   00000016                          ABD       22                EXPAND FORM ABD.
  00034   P00014                                     LL                          CALL MACRO LL.
          P00014   54455354                          DATAW     C'TEST
                                                     ENDM
  00035   P00018   AD800000    X00000                LW        3,LOP             LOAD REG 3 WITH CONTENTS OF LOP.
  00036   P0001C   8A000018    Y00018                ORMW      4,LOP             'OR' REG 4 WITH CONTENTS OF LOP.
  00037   P00020   D600001C    Y0001C                STW       4,LOP             STORE CONTENTS OF REG 4 IN LOP.
  00038   P00024   F8800001    X00000                BL        POP               BRANCH AND LINK TO ROUTINE POP.
U 00039   P00028   AE000028    P00028                LW        4,MOVE            LOAD REG 4 WITH CONTENTS OF MOVE.
  00040    00000                                     ABS                         SET PROGRAM COUNTER TO ABSOLUTE.
  00041    00020                                     ORG       X'20'             SET PROGRAM COUNTER TO HEX 20.
  00042    00020   0000FACE                COVE      DATAW     X'FACE'           DECLARE COVE AS HEX FACE.
  00043    00024   AF000000    A00000                LW        6,ZZZ             LOAD REG 6 WITH CONTENTS OF ZZZ.
  00044    00028   D7000000    A00000                STW       6,WWW             STORE CONTENTS OF REG 6 IN WWW.
  00045   P0002C                                     REL
  00046   P0002C   AE800000   *P00000                LW        5,CONS1           LOAD A CSECT CONSTANT.
  00047   P00030   D6800034    P00034                STW       5,VARI            STORE IT IN A VARIABLE IN DSECT.
  00048  *P00000                                     CSECT
  00049  *P00000   00000125                CONS1     DATAW     293               A CONSTANT THAT IS IN CSECT.
  00050   P00034                                     DSECT
  00051   P00034                           VARI      RES       1W                A VARIABLE THAT IS IN DSECT.
          P00038   22222222
  00052   P0003C                                     END


*  0008  ERRORS IN ALLTYPES
UNDEFINED      XYZ
UNDEFINED      HIT
UNDEFINED      YY
UNDEFINED      MOVE
```

Figure 6-1. Sample Assembler Listed Output

7 Label Field - Source statement label field

8 Operation Field - Source statement operation (directive) field

9 Operand Field - Source statement operand field

10 Comment Field - Source statement comment field

11 Release Number - Release number of the MPX-32 utilities followed by the Macro Assembler internal release number.

12 Error Report - The error report line gives the total number of errors detected in the assembly process.

13 Undefined Symbols - If undefined symbols were referenced by the source program, they are listed below the error report line.

6.3 Symbol Cross-Reference

A symbol cross-reference is optionally produced at the end of each source program assembly.

In addition to the standard symbol cross-reference, a cross-reference of internally generated symbols may be optionally requested by specifying option 3. If specified, the internal cross-reference is listed immediately following the symbol cross-reference in descending alphanumeric order. Symbols that are not referenced are not included unless option 10 is specified.

The symbol cross-reference for the ALLTYPES program used in Figure 6-1 is shown in Figure 6-2. The following paragraphs detail each of the circled entries in Figure 6-2.

1 Program Name - The program name specified in the PROGRAM directive is printed on each page of listed output.

2 Symbol Type - Specifies the symbol type and its value or address. The value or address is given in bytes by the five least significant positions. The most significant character indicates the symbol type and significance of the value as follows:

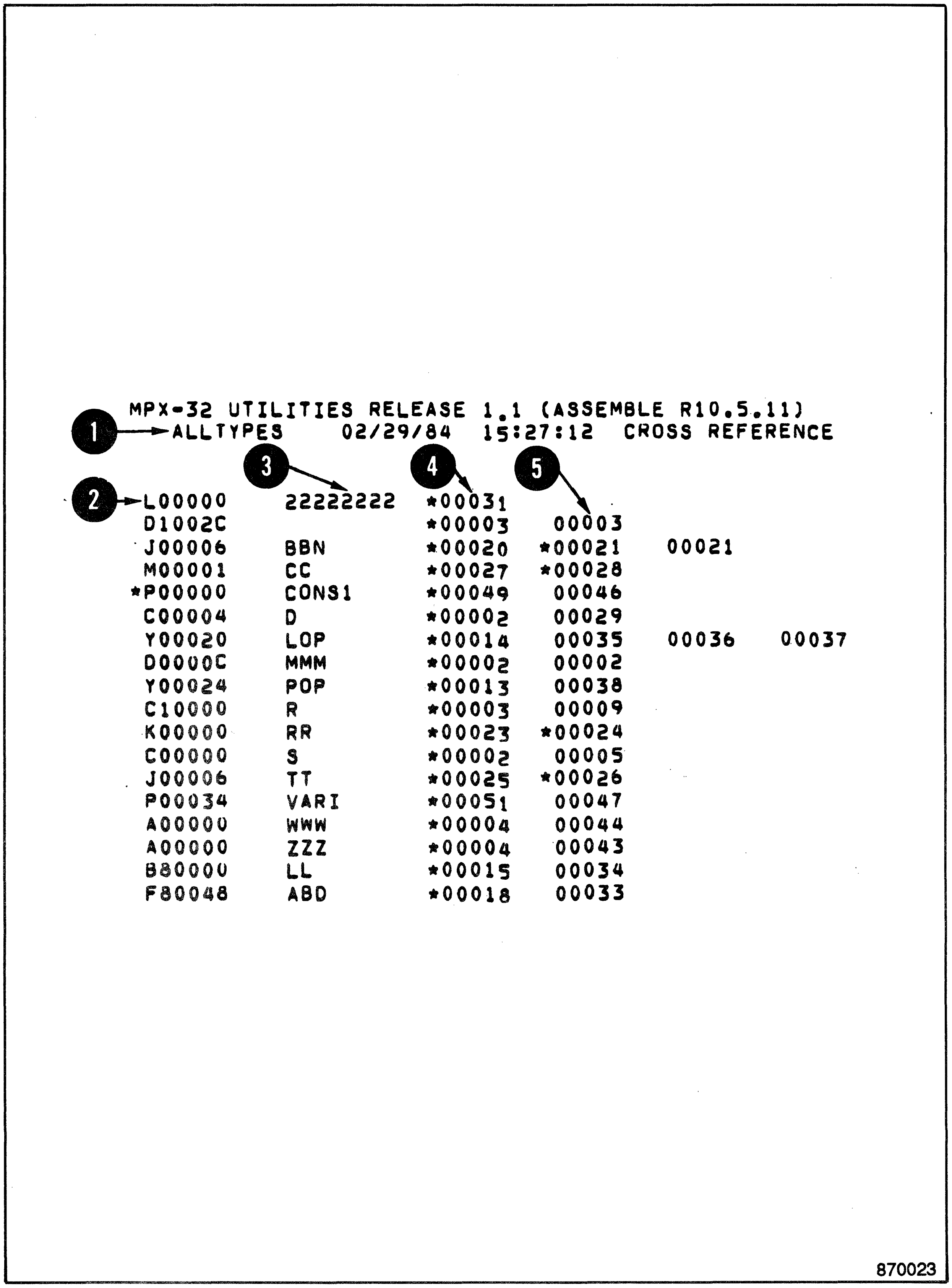
Character	Symbol Type	Value Significance
Blank	ABSOLUTE	Absolute address or value of the symbol
A	DATAPOOL	Value is always zero
B	MACRO	Relative address of the macro prototype in the macro storage table
C	COMMON ITEMS	Most significant digit is the least significant digit of the common block number in which the referenced symbol occurs. Least significant four digits are the relative address of the symbol within the common block.

Character	Symbol Type	Value Significance
D	COMMON BLOCK	Leftmost digit is the least significant digit of the common block number. Least significant four digits are the common block size in bytes.
F	FORM	Relative address of the form prototype in the macro storage table
J	SET (ABSOLUTE)	Last absolute value to which the symbol was set
K	SET (RELOCATABLE)	Last relocatable value to which the symbol was set
L	LITERAL	Leftmost digit is the least significant digit of the literal pool number. Least significant four digits are the number of the literal within the literal pool.
M	MULTIPLE DEFINITION	Initial value the symbol was assigned
P	RELOCATABLE	Relocatable address of the symbol
U	UNDEFINED	Contents of the location counter when the symbol was last referenced
X	EXTERNAL	The program address at which the symbol was last referenced (absolute programs only). Value is zero if the symbol was never referenced.
Y	EXTERNAL	Program address at which the symbol was last referenced (relocatable programs only)
*	CSECT	The symbol is within a code section

3 Symbol - Lists the symbol being cross-referenced. A blank entry indicates blank COMMON.

4 Symbol Defined - Lists the line number of the source statement in which the symbol is defined. The line number specifying symbol definition is always preceded by an asterisk (*).

5 Symbol Referenced - Multiple entries specifying line number(s) in the source program where the symbol was referenced and/or redefined.


1
2
3
4
5
870023

```
MPX-32 UTILITIES RELEASE 1.1 (ASSEMBLE R10.5.11)
  ALLTYPES    02/29/84  15:27:12  CROSS REFERENCE

 L00000    22222222  *00031
 D1002C              *00003    00003
 J00006    BBN       *00020   *00021    00021
 M00001    CC        *00027   *00028
*P00000    CONS1     *00049    00046
 C00004    D         *00002    00029
 Y00020    LOP       *00014    00035    00036    00037
 D0000C    MMM       *00002    00002
 Y00024    POP       *00013    00038
 C10000    R         *00003    00009
 K00000    RR        *00023   *00024
 C00000    S         *00002    00005
 J00006    TT        *00025   *00026
 P00034    VARI      *00051    00047
 A00000    WWW       *00004    00044
 A00000    ZZZ       *00004    00043
 B80000    LL        *00015    00034
 F80048    ABD       *00018    00033
```

Figure 6-2. Sample Symbol Cross-Reference

6.4 Error Diagnostics

A comprehensive set of error diagnostics is defined for error conditions detected by the Macro Assembler during source code processing. Refer to Section 5 for a list of error codes.

Pass One errors are printed on the listed output page immediately preceding the source program listing. Pass One error information includes the error code, the source program line number of the error, and the source image. Figure 6-3 shows the Pass One error list for the program ALLTYPES used in Figures 6-1 and 6-2.

```
MPX-32 UTILITIES RELEASE 1.1 (ASSEMBLE R10.5.11)
   ALLTYPES     02/29/84  15:27:12

M 00028       CC         EQU        2                    CC = 2.
                                                                  870026
```

Figure 6-3. Pass One Error List

The total number of Pass Two errors is listed in the error report line. The error report line also lists the program name, if specified, or the default name MAIN.

The error report line is followed by a listing of undefined symbols for a particular program step.

Figure 6-1, items 12 and 13, depict Pass Two error reporting.

6.5 Object Output

Macro Assembler object output is produced in word format. Object records are variable length depending on the peripheral device to which object output is directed. All object records, regardless of the peripheral device used for output, contain the following information in this order:

(1) RECORD TYPE (1 byte) -- Defines the type of record being processed. The hexadecimal value FF signifies a binary record. The last record of a program is signified by the hexadecimal value DF.

(2) BYTE COUNT (1 byte) -- Specifies the number of data bytes in the object record, exclusive of checksum and sequence number

(3) CHECKSUM (2 bytes) -- Is a halfword additive checksum of data bytes within the object record, exclusive of byte count and sequence number

(4) SEQUENCE NUMBER (2 bytes) -- Is the binary sequence number of the object record. The sequence number count is initialized to one for each new program. If more than 65,536 records occur in a particular program, the sequence number count resets to one and continues.

(5) OBJECT PROGRAM -- Is of a series of data blocks. Each data block is preceded by a control byte specifying a loader function code and byte count. The format of the control byte is:

```
xxxxnnnn
```

xxxx specifies a loader function code as follows:

Code	Definition
0000	Absolute
0001	Program Origin
0010	Absolute Data and Repeat Load
0011	Transfer Address to Start Execution
0100	Relocatable Data
0101	Program Name
0110	Relocatable Data and Repeat Load
0111	External Definition
1000	Forward Reference Stringback
1001	External Reference
1010	Common Block Definition
1011	Common Reference
1100	Datapool Reference
1101	Extended Codes
1110	Common Origin
1111	Last Byte of Object Output

nnnn specifies the byte count for the data block being processed by the loader. This entry is the least significant four bits of the control byte. For a data block of 16 bytes, the bit representation for nnnn is 0000.

Data blocks comprising the object program are grouped and output in the following order:

(1) Program Name (with maximum boundary required for loading)

(2) Common Block Names and Sizes (if any)

(3) Defined Entry Points (DEFs, if any)

(4) Binary Object Code

(5) External Stringbacks (EXTs, if any)

(6) Transfer Address (if any)

(7) Termination Code

A relocation offset of zero is assumed by the loader for binary object code, unless otherwise specified. Object code must end with a final origin to the next available location.

6.6 Macro Assembler Programming Examples

The following section provides sample programming sequences illustrating the use of various Macro Assembler directives.

Example 1 shows the use of conditional assembly directives. Examples 2 through 11 show the use of macros. The same macro definition is used for Examples 2 through 5. The code generated is different for each example based on the different usages of the macro call. Examples 6 through 11 show the use of recursive macros. The same macro definition is used for Examples 6 through 11. The code generated is different for each example based on the different usages of the macro call.

Example 12 shows a program that assembles source code from a user file, catalogs the object output into a load module, and directs the job output to a user file.

Example 1

The following coding sequence:

```
                    LIST          NONG
          STEP      SET           45
          LEVEL     SET           0
                    REPT          6
                    DATAH         A(LEVEL)
          LEVEL     SET           LEVEL+STEP
                    IFT           LEVEL.GE.180,ENDLOOP
          STEP      SET           55
          ENDLOOP   ANOP
                    ENDR
                    END
```

generates the following sequence of instructions:

```
P00000  0000        DATAH         0
P00002       0020   DATAH         45
P00004  005A        DATAH         90
P00006       0087   DATAH         135
P00008  00B4        DATAH         180
P0000A       00EB   DATAH         235
```

Example 2

Macro Definition for Examples 2 through 5

```
          MOVE      DEFM          LABEL,A
                    LW            3,FLAG
                    IFT           %A,LABEL1
          %LABEL    STW           3,RESET1
                    EXITM
          LABEL1    ANOP
                    IFF           %A,LABEL2
          %LABEL    STW           3,RESET2
          LABEL2    ANOP
                    ENDM
```

Usage

```
        LIST    NONG
ONE     SETT
         .
         .
         .
        MOVE    X,ONE
```

Result

```
        LW      3,FLAG
X       STW     3,RESET1
```

Example 3

Usage

```
        LIST    NONG
ONE     SETF
         .
         .
         .
        MOVE    X,ONE
```

Result

```
        LW      3,FLAG
X       STW     3,RESET2
```

Example 4

Usage

```
I       SET     1
         .
         .
        MOVE    X,I
```

Result

```
        LW      3,FLAG
X       STW     3,RESET1
```

Example 5

Usage

```
I       SET     2
         .
         .
        MOVE    X,I
```

Result

```
        LW      3,FLAG
X       STW     3,RESET2
```

Example 6

Macro Definition for Examples 6 through 11

```
RECURS      DEFM       ARG1,ARG2,ARG3
            IFP        %ARG1,%LAB1
            STW        1,%ARG1
            RECURS     ,%ARG2,%ARG3
            EXITM
%LAB1       IFP        %ARG2,%LAB2
            STW        2,%ARG2
            RECURS     ,,%ARG3
            EXTIM
%LAB2       IFP        %ARG3,%LAB3
            STW        3,%ARG3
%LAB3       ANOP
            ENDM
```

Usage

```
RECURS      WORD1,WORD2,WORD3
```

Result

```
STW         1,WORD1
STW         2,WORD2
STW         3,WORD3
```

Example 7

Usage

```
RECURS      WORD1,WORD2
```

Result

```
STW         1,WORD1
STW         2,WORD2
```

Example 8

Usage

```
RECURS      WORD1
```

Result

```
STW         1,WORD1
```

Example 9

Usage

```
        RECURS          WORD1,,WORD3
```

Result

```
        STW             1,WORD1
        STW             3,WORD3
```

Example 10

Usage

```
        RECURS          ,,WORD3
```

Result

```
        STW             3,WORD3
```

Example 11

Usage

```
        RECURS
```

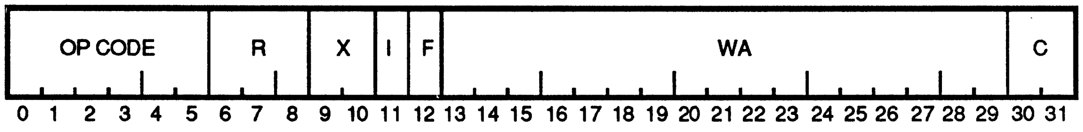
Result

No Code Generated

APPENDIX A

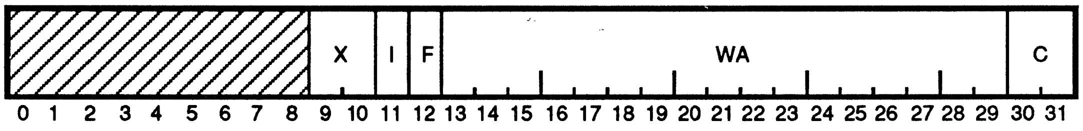
INSTRUCTION FORMATS

MEMORY ADDRESS

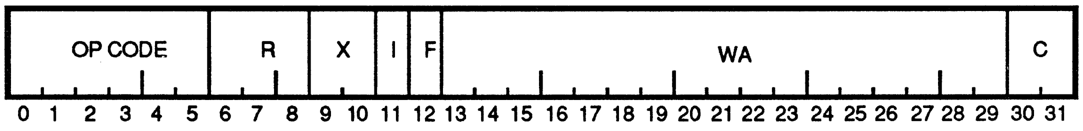


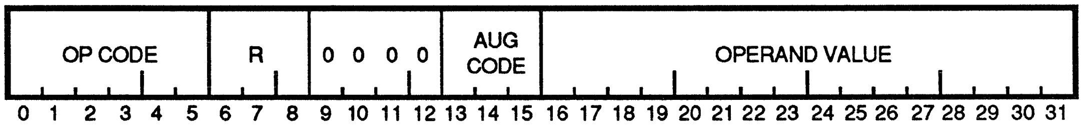
OP CODE	OPERATION CODE
R	GENERAL PURPOSE REGISTER (0-7)
X	INDEX REGISTER (1-3)
I	INDIRECT ADDRESSING SPECIFICATION
F	FORMAT BIT
WA	WORD ADDRESS
C	ADDRESS CODE (INCLUDING BYTE ADDRESS)

INDIRECT/EFFECTIVE ADDRESS



MEMORY REFERENCE INSTRUCTION

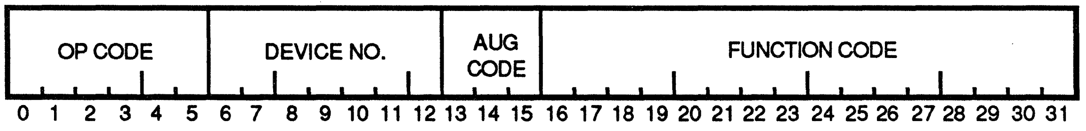




BITS 13-15	AUGMENTING OPERATION CODE
BITS 16-31	16-BIT OPERAND VALUE

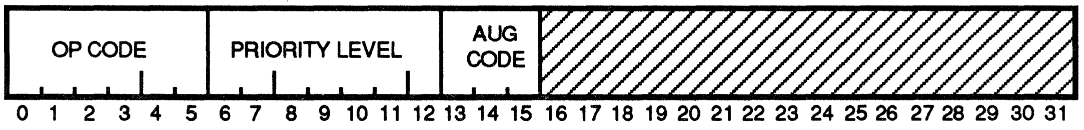
87D4J04

I/O INSTRUCTION



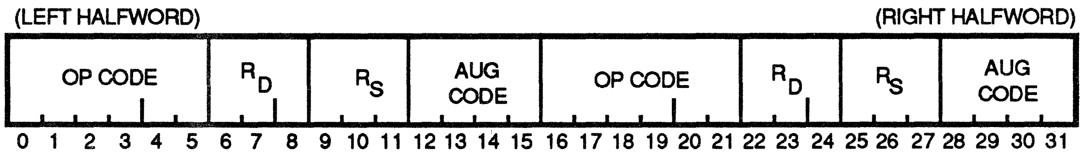
BITS 6-12	I/O DEVICE NUMBER
BITS 16-31	16-BIT I/O FUNCTION CODE

INTERRUPT CONTROL INSTRUCTION



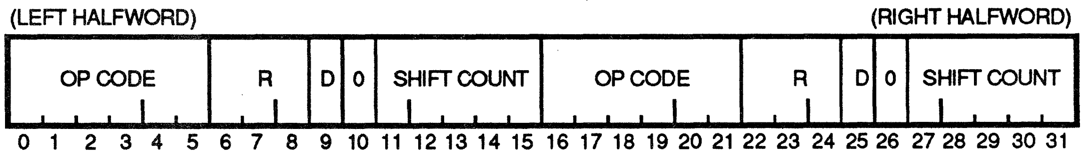
BITS 6-12	INTERRUPT PRIORITY LEVEL

INTER-REGISTER INSTRUCTION



BITS 6-8/22-24	DESTINATION REGISTER (0-7) FOR RESULT OF OPERATION
BITS 9-11/25-27	SOURCE REGISTER OF OPERAND

SHIFT INSTRUCTION

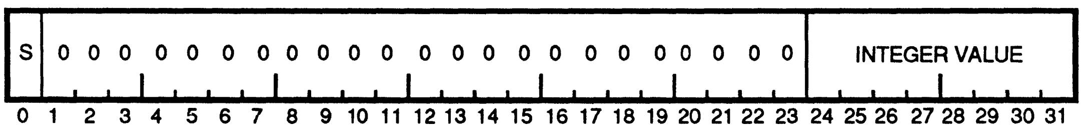


BITS 6-8/22-24	GENERAL PURPOSE REGISTER (0-7)
BITS 9/25	SHIFT DIRECTION (0=RIGHT/1=LEFT)
BITS 10/26	UNUSED (=0)
BITS 11-15/27-31	SHIFT COUNT

87D4J05

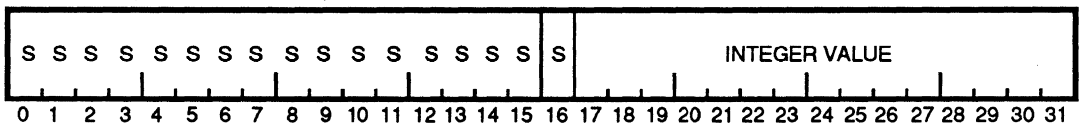
FIXED POINT FORMATS

BYTE

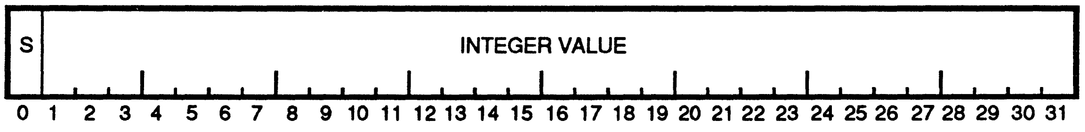


(BIT 0=SIGN)

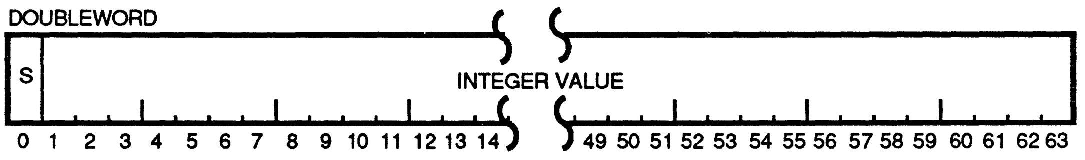
HALFWORD (SIGN - EXTENDED)



FULLWORD

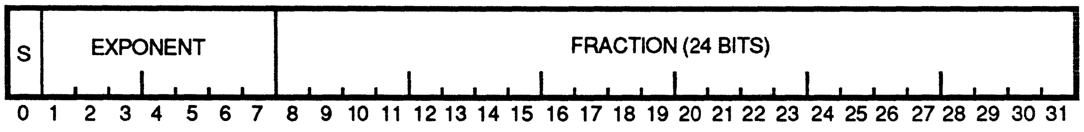


DOUBLEWORD

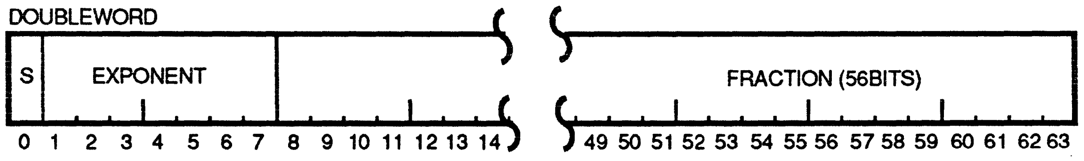


FLOATING POINT FORMATS

WORD



DOUBLEWORD



87D4J03

APPENDIX B

EXTENDED MNEMONIC CODES

Assembler language extended mnemonic codes are used to specify conditional branch instructions. Included within the extended mnemonic codes are both the branch instruction and the branch condition. The Macro Assembler translates the extended mnemonic code into the appropriate machine instruction -- Branch Condition True (BCT) or Branch Condition False (BCF). The extended mnemonic codes, together with their operand formats and equivalent machine instruction, are shown below. The symbols used to indicate entries in the operand field are:

*	Indirect Addressing
m	Memory Address
x	Index Register

USED AFTER COMPARE INSTRUCTIONS

Extended Mnemonic Code	Operand Format	Machine Instruction	Description
BGT	*m,x	BCT 2,*m,x	Branch If Greater Than
BLT	*m,x	BCT 3,*m,x	Branch If Less Than
BEQ	*m,x	BCT 4,*m,x	Branch If Equal To
BGE	*m,x	BCT 5,*m,x	Branch If Greater Than or Equal To
BLE	*m,x	BCT 6,*m,x	Branch If Less Than or Equal To
BNE	*m,x	BCF 4,*m,x	Branch If Not Equal To

USED AFTER ARITHMETIC, LOGICAL, AND LOAD INSTRUCTIONS

Extended Mnemonic Code	Operand Format	Machine Instruction	Description
BOV	*m,x	BCT 1,*m,x	Branch If Overflow
BP	*m,x	BCT 2,*m,x	Branch If Positive
BN	*m,x	BCT 3,*m,x	Branch If Negative
BZ	*m,x	BCT 4,*m,x	Branch If Zero
BNOV	*m,x	BCF 1,*m,x	Branch If No Overflow
BNP	*m,x	BCF 2,*m,x	Branch If Nonpositive
BNN	*m,x	BCF 3,*m,x	Branch If Non-negative
BNZ	*m,x	BCF 4,*m,x	Branch If Nonzero

USED AFTER TEST INSTRUCTIONS

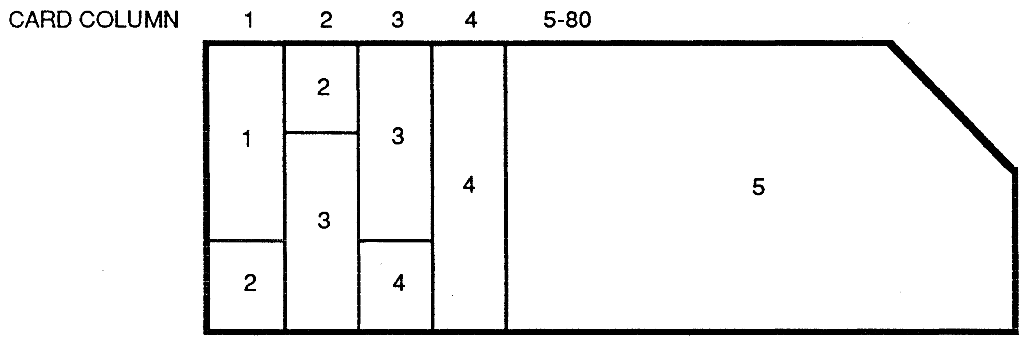
Extended Mnemonic Code	Operand Format	Machine Instruction	Description
BS	*m,x	BCT 1,*m,x	Branch If Set
BNS	*m,x	BCF 1,*m,x	Branch If Not Set
BANY	*m,x	BCT 7,*m,x	Branch If Any One
BAZ	*m,x	BCF 7,*m,x	Branch If All Zeros

APPENDIX C

COMPRESSED SOURCE FORMAT

The Macro Assembler optionally accepts source program input or produces source output in compressed format on cards, magnetic tape, or disc. The records are 120 bytes in length.

The compressed source format for card input media is:



87D4I06

The format fields are specified as follows:

1 - Data Type Code (8 bits) - Hexadecimal values BF or 9F specify compressed format; otherwise noncompressed. Hexadecimal value 9F indicates last record of a compressed source module.

2 - Byte Count (8 bits) - Specifies number of bytes remaining in record.

3 - Checksum (16 bits)

4 - Sequence Number (16 bits)

5 - Compressed Source Data (n bytes) - Contiguous bytes of source data to be assembled. Record length varies depending on peripheral device from which data is being input.

Compression is accomplished by squeezing blanks and inserting field and string counts. A string is broken only when three or more blanks are encountered. For the following 80-character source record:

```
    ALPHA          LW 3,BETA
```

the resulting compressed source image is the following byte string:

Blank Count	- 0
String Count	- 5
Data -	ALPHA
Blank Count	- 5
String Count	- 2
Data -	LW
Blank Count	- 4
String Count	- 6
Data -	3,BETA
Terminator	- FF (Hexadecimal)
.	.
.	.
.	.

(Remainder Of Compressed Source Record)

The compression ratio for a source program is dependent on the number and size of source comments. Compression is normally in the range from 3-2 to 4-1.

APPENDIX D

ASCII CODE SET

Row	Col	0	1	2	3	4	5	6	7
Bit Positions 4	0	0	0	0	0	0	0	0	0
5	1	0	0	0	0	1	1	1	1
6	2	0	0	1	1	0	0	1	1
7	3	0	1	0	1	0	1	0	1
0000	0	NUL 12-0-9-8-1	DLE 12-11-9-8-1	SP No Punch	0 0	@ 8-4	P 11-7	` 8-1	p 12-11-7
0001	1	SOH 12-9-1	DCI 11-9-1	! 12-8-7	1 1	A 12-1	Q 11-8	a 12-0-1	q 12-11-8
0010	2	STX 12-9-2	DC2 11-9-2	" 8-7	2 2	B 12-2	R 11-9	b 12-0-2	r 12-11-9
0011	3	ETX 12-9-3	DC3 11-9-3	# 8-3	3 3	C 12-3	S 0-2	c 12-0-3	s 11-0-3
0100	4	EOT 9-7	DC4 9-8-4	$ 11-8-3	4 4	D 12-4	T 0-3	d 12-0-4	t 11-0-3
0101	5	ENQ 0-9-8-5	NAK 9-8-5	% 0-8-4	5 5	E 12-5	U 0-4	e 12-0-5	u 11-0-4
0110	6	ACK 0-9-8-6	SYN 9-2	& 12	6 6	F 12-6	V 0-5	f 12-0-6	v 11-0-5
0111	7	BEL 0-9-8-7	ETB 0-9-6	´ 8-5	7 7	G 12-7	W 0-6	g 12-0-7	w 11-0-6
1000	8	BS 11-9-6	CAN 11-9-8	(12-8-5	8 8	H 12-8	X 0-7	h 12-0-8	x 11-0-7
1001	9	HT 12-9-5	EM 11-9-8-1	) 11-8-5	9 9	I 12-9	Y 0-8	i 12-0-9	y 11-0-8
1010	A	LF 0-9-5	SUB 9-8-7	* 11-8-4	: 8-2	J 11-1	Z 0-9	j 12-11-1	z 11-0-9
1011	B	VT 12-9-8-3	ESC 0-9-7	+ 12-8-6	; 11-8-6	K 11-2	[12-8-2	k 12-11-2	{ 12-0
1100	C	FF 12-9-8-4	FS 11-9-8-4	, 0-8-3	< 12-8-4	L 11-3	\ 0-8-2	12-11-3	¦ 12-11
1101	D	CR 12-9-8-5	GS 11-9-8-5	— 11	= 8-6	M 11-4	] 11-8-2	m 12-11-4	} 11-0
1110	E	SO 12-9-8-6	RS 11-9-8-6	. 12-8-3	> 0-8-6	N 11-5	∧ 11-8-7	n 12-11-5	~ 11-0-1
1111	F	SI 12-9-8-7	US 11-9-8-7	/ 0-1	? 0-8-7	O 11-6	— 0-8-5	o 12-11-6	DEL 12-9-7

840813

Some positions in the ASCII code chart may have a different graphic representation on various devices as:

ASCII	IBM 029
!	I
[	¢
]	!
^	>

Control Characters:

NUL	-	Null
SOH	-	Start of Heading (CC)
STX	-	Start of Text (CC)
ETX	-	End of Text (CC)
EOT	-	End of Transmission (CC)
ENQ	-	Enquiry (CC)
ACK	-	Acknowledge (CC)
BEL	-	Bell (audible or attention signal)
BS	-	Backspace (FE)
HT	-	Horizontal Tabulation (punch card skip) (FE)
LF	-	Line Feed (FE)
VT	-	Vertical Tabulation (FE)
FF	-	Form Feed (FE)
CR	-	Carriage Return (FE)
SO	-	Shift Out
SI	-	Shift In
DLE	-	Data Link Escape (CC)
DC1	-	Device Control 1
DC2	-	Device Control 2
SUB	-	Substitute
DC4	-	Device Control 4 (stop)
NAK	-	Negative Acknowledge (CC)
SYN	-	Synchronous Idle
ETN	-	End of Transmission Block (CC)
CAN	-	Cancel
EM	-	End of Medium
SS	-	Start of Special Sequence
ESC	-	Escape
FS	-	File Separator (IS)
GS	-	Group Separator (IS)
RS	-	Record Separator (IS)
US	-	Unit Separator (IS)
DEL	-	Delete
SP	-	Space (normally nonprinting)
(CC)	-	Communication Control
(IS)	-	Information Separator

Macro Library Editor (MACLIBR)

MPX-32 Utilities

CONTENTS

MACRO LIBRARY EDITOR (MACLIBR)

SECTION 1 - OVERVIEW

1.1 General Description

The Macro Library Editor (MACLIBR) utility creates and maintains system or user macro libraries.

MACLIBR can be used to:

. Delete or replace macros by name

. Insert macros

. Build a new macro library

. List or print a macro in a macro library

. Log the names of macros in a macro library

Macros are sequences of Assembly language instructions with unique names that can be stored in a library maintained by MACLIBR. Up to 65,535 macros can be contained in a macro library. When a macro name is used in source code, the Macro Assembler retrieves the macro from the macro library and expands the macro. Up to 255 variable parameters to pass to the macro can be defined in a macro. The task using a macro supplies the parameters. The parameters are used for macro expansion.

M.MPXMAC is the system macro library. Supplied with the operating system, M.MPXMAC contains all of the macros required to expand references to system services into Assembler level code, plus the definitions of all MPX-32 data structures.

Macros begin with a DEFM statement, which can be preceded by Assembler comment lines describing the macro. The /MACLIST directive lists the DEFM statements contained in a library.

Parameters to be passed to the macro can be defined within the macro. The parameters are dummy symbols preceded by percent signs. Other dummy symbols are labels used for conditional processing. The /MACLIST directive lists all dummy symbols used in a macro.

NOTE: Only parameters actually referenced in the macro body are retained in the DEFM.

MACLIBR processes files sequentially. A macro specified with any directive must be located in the library file after the macro specified with the previous directive. For example, a macro cannot be added in the middle of a library before replacing a macro at the beginning of the same library. The only exception to this rule is the /DISPLAY directive, which may be placed anywhere within a MACLIBR directive stream. Prepare the directive file and the file assigned to logical file code SI so both follow the sequence of the macros in the library being updated.

MACLIBR recognizes 1 to 8 character macro names and 1 to 16 character file and library names. Unless specified, files assigned to logical file codes will be forced to the appropriate format-blocked or unblocked.

MACLIBR does not check for duplicate names. If two macros have the same name and one is to be deleted, the first one encountered is deleted. A log of the library shows the relative position of each macro in the library.

1.2 Directive Summary

The following list summarizes the MACLIBR directives. Underlining indicates accepted abbreviations. Each directive is described in more detail in Section 3.

Directive	Function
/APPEND	Adds one or more macros to the end of a library file
/CREATE	Generates a macro library
/DELETE	Deletes a macro from a library
/DISPLAY	Lists a macro
/END	Defines the end of an INSERT, REPLACE, APPEND, or CREATE sequence. After APPEND or CREATE, END has the same effect as EXIT.
/EXIT	Performs update and returns control to the calling task. This is the last update directive.
/INSERT	Inserts one or more macros before a specified macro
/LOG	Lists names and numbers of all macros after all updates are complete
/MACLIST	Lists all, part, or none of the source for each macro in a library
/REPLACE	Replaces an existing macro with a new macro of the same name

SECTION 2 - USAGE

2.1 Accessing MACLIBR

MACLIBR can be accessed from the batch or interactive modes in one of three ways:

```
$MACLIBR
$RUN MACLIBR
$EXECUTE MACLIBR
```

$RUN MACLIBR is valid only from the system directory.

When accessing MACLIBR interactively, the MAC> prompt is displayed:

```
TSM> $MACLIBR
MAC>
```

2.2 Logical File Code Assignments

There are four logical file codes (LFCs) associated with MACLIBR: Macro Library (MAC), Macro Input File (SI), Directives (DIR), and Listed Output (LO).

2.2.1 Macro Library (MAC)

A macro library can reside in either a permanent disc file or on a magnetic tape file. If the macro library file is a disc file, the Volume Manager (VOLMGR) must be used to create the macro library file before the macro library is generated. The macro library is assigned to logical file code MAC. Logical file code (LFC) MAC is forced unblocked by MACLIBR. (This LFC must not be assigned by the user).

A temporary file which is the same size as the macro library being used is assigned to MAC. This file is dynamically allocated by MACLIBR and is used to build and edit macros from the source into an existing or new library.

MACLIBR checks the access rights associated with a macro library file. Depending on how the macro library was created, there may be access limitations. An error message is generated if the user does not have the appropriate access rights to the library. A user with read only access can use only the /DISPLAY, /EXIT, /LOG, and /MACLIST directives. If an access error occurs, it may be necessary to create the file with different access rights or to change the owner name and/or project name before attempting to access it.

MAC Default and Optional Assignments

The default assignment for MAC is to the system macro library (M.MPXMAC):

```
$ASSIGN MAC TO M.MPXMAC BLOC=N
```

There are two optional assignments for MAC:

```
$ASSIGN MAC TO {pathname   }
               {DEV=devmnc }
```

pathname	is the pathname of a macro library file
devmnc	is the device mnemonic of a device containing a macro library file

2.2.2 Macro Input File (SI)

The macro input file is a file of macros in uncompressed format. Each macro may have a maximum of 255 parameters. The maximum number of macros is 65,535. The macro input file is assigned to logical file code SI.

SI Default and Optional Assignments

The default assignment for SI is to the System Control file (SYC):

```
$ASSIGN SI TO SYC
```

There are two optional assignments for SI:

```
$ASSIGN SI  TO   {pathname   }
                 {DEV=devmnc }
```

pathname	is the pathname of a file containing macro input
devmnc	is the device mnemonic of a device containing macro input

2.2.3 Directives (DIR)

The directives file is a file of MACLIBR directives to be performed. The directives file is assigned to logical file code DIR.

DIR Default and Optional Assignments

The default assignment for DIR is to the System Control file (SYC):

```
$ASSIGN DIR TO SYC
```

There are two optional assignments for DIR:

```
$ASSIGN DIR TO   {pathname   }
                 {DEV=devmnc }
```

pathname	is the pathname of a file containing MACLIBR directives
devmnc	is the device mnemonic of a device containing MACLIBR directives

2.2.4 Listed Output (LO)

The listed output file contains a MACLIBR audit trail. The listed output file is assigned to logical file code LO.

LO Default and Optional Assignments

The default assignment for LO is to logical file code UT:

```
$ASSIGN LO TO LFC=UT
```

In the interactive mode, output is generated on the user terminal. In the batch mode, output is generated on the SLO device.

There are two optional assignments for LO:

```
$ASSIGN LO  TO   {pathname   }
                 {DEV=devmnc }
```

pathname	is the pathname of a file to contain listed output
devmnc	is the device mnemonic of a device to contain listed output

2.2.5 LFC Summary

The following is a table of LFCs used by MACLIBR and their default and optional assignments.

Table 2-1
MACLIBR LFC Summary

LFC	Default Assignment	Optional Assignment
MAC	M.MPXMAC	pathname DEV=devmnc
SI	SYC	pathname DEV=devmnc
DIR	SYC	pathname DEV=devmnc
LO	LFC=UT	pathname DEV=devmnc
UT1	TEMP	none

2.3 Options

Two options can be specified for MACLIBR. Options are specified by number on an $OPTION job control language statement.

Option	Description
7	DIR Unblocked The file assigned to logical file code DIR is unblocked. If this option is specified, the $ASSIGN statement for DIR must also be unblocked (i.e., BLOC=N).
8	SI Unblocked The file assigned to logical file code SI is unblocked. If this option is specified, the $ASSIGN statement for SI must also be unblocked (i.e., BLOC=N).

2.4 MACLIBR Listings

MACLIBR listed output is an audit trail including directives, a list of all macros, the contents of each macro, and the following MACLIBR operation counters:

Counter	Description
BR	Number of 192-word blocks read from the file assigned to logical file code MAC
BW	Number of 192-word blocks written to the scratch file
MD	Number of macros deleted
MR	Number of macros replaced
MI	Number of macros inserted and appended
BU	Number of 192-word blocks used on the file assigned to logical file code MAC after updating
NM	Number of next macro

The counter values appear at the end of the listed output.

2.5 Exiting MACLIBR

To exit MACLIBR from the batch and interactive modes, specify the /EXIT directive. In addition, MACLIBR exits when the /END directive follows either the /APPEND or /CREATE directive.

SECTION 3 - DIRECTIVES

3.1 Introduction

MACLIBR directives can be abbreviated to the first four characters, including the preceding slash. If a directive or parameter can be abbreviated, the acceptable abbreviation is underlined in the syntax.

Both a comma and blanks between parameters are valid delimiters. Commas need be used only where shown.

Only upper case is permitted for directives, the MACRO name, DEFM, or ENDM.

Directives are processed sequentially until an /EXIT directive or an end-of-file is encountered. At least one blank must separate the end of a directive verb and a required parameter. A required parameter must not be placed beyond column 21 or exceed eight characters. Directives referencing a macro by name must be in the same order as the macros in the macro library file. The only exceptions are the /DISPLAY and /LOG directives, which may occur anywhere in the MACLIBR directive stream.

MACLIBR writes the updated macro library to a dynamically allocated temporary file (UT1). When the updating sequence is complete, the /EXIT directive causes the scratch file to be copied to the file assigned to MAC. An /END directive or an end-of-file serves as an /EXIT directive for /APPEND and /CREATE directives, and causes the scratch file to be copied to the file assigned to MAC. If the message "NAME NOT FOUND" is displayed after a /DELETE, /DISPLAY, /INSERT, or /REPLACE, the scratch file is not copied and no updating occurs.

3.2 /APPEND Directive

The /APPEND directive adds macros to the end of a macro library. All macros from the current file position to the end of the macro library remain the same. Macros are read from the file assigned to logical file code SI until an /END directive or an end-of-file is encountered. MACLIBR then terminates.

Syntax:

<u>/APP</u>END

3.3 /CREATE Directive

The /CREATE directive generates a macro library. Macros are read from the file assigned to logical file code SI until an /END directive or an end-of-file is encountered. MACLIBR then terminates.

Syntax:

```
/CREATE
```

3.4 /DELETE Directive

The /DELETE directive deletes the specified macro from the macro library. All macros from the current file position to the named macro remain the same. The next directive is processed after the macro is deleted.

Syntax:

```
/DELETE  macro
```

macro	is the one to eight ASCII character name of the macro to be deleted

3.5 /DISPLAY Directive

The /DISPLAY directive lists the statements of the specified macro or of all macros if a name is not specified. This directive may be placed anywhere in the directive stream. After this directive is processed, the macro library is repositioned to the point where the display began. The next directive is then processed.

Syntax:

```
/DISPLAY [macro]
```

macro	is the one to eight ASCII character name of the macro to be displayed. If not specified, all macros in the library are displayed.

3.6 /END Directive

The /END directive defines the end of an /INSERT, /REPLACE, /APPEND, or /CREATE sequence. After an /INSERT or /REPLACE sequence, the next MACLIBR directive is processed. When /END occurs after an /APPEND or /CREATE sequence, processing is the same as an /EXIT directive.

Syntax:

```
/END
```

3.7 /EXIT Directive

The /EXIT directive is the last directive of a MACLIBR session. If updates were performed, the scratch file is copied to the file assigned to MAC. If a /LOG END directive was included, the updated library is logged. If no updates were performed, MACLIBR terminates. If MACLIBR was run interactively, control returns to TSM.

Syntax:

```
/EXIT
```

3.8 /INSERT Directive

The /INSERT directive inserts one or more macros ahead of the specified macro. All macros from the current file position to the specified macro remain unchanged. Macros are read from the file assigned to logical file code SI until an /END directive or an end-of-file is encountered. The next directive is then processed.

Syntax:

```
/INSERT macro
```

macro	is the one to eight ASCII character name of the macro before which the new macro will be inserted

3.9 /LOG Directive

The /LOG directive writes the name and number of each macro to the file or device assigned to logical file code LO. This directive may be placed anywhere in the MACLIBR directive stream. If the END parameter is specified, logging is performed after all updates are complete. If the END parameter is not specified, the macro library is logged without updates. When logging is complete, the macro library is repositioned to the point where logging began. The next directive is then processed.

Syntax:

```
/LOG [END]
```

END	writes the log after all updates are complete. If not specified, the updates are not written.

3.10 /MACLIST Directive

The /MACLIST directive allows the listing of all, part, or none of each source macro. This directive does not affect listed output already formatted by the /DISPLAY and /LOG directives. When dummy symbols are listed, their corresponding hexadecimal assignments are included.

Syntax:

/MACLIST [option]

option is one of the following parameters:

Parameter	Definition
ON	Complete listing
OFF	Suppress listing
ID=	List each macro DEFM statement
BODY	List each macro but exclude output of dummy symbols
SYMS	List each macro DEFM statement including output of dummy symbols

If no parameters are specified, the default is ON.

3.11 /REPLACE Directive

The /REPLACE directive replaces the specified macro with a new macro. All macros from the current file position to that of the specified macro remain the same. New macros are read from the file assigned to logical file code SI until an /END directive or an end-of-file is encountered. The next directive is then processed.

Syntax:

/REPLACE macro

macro is the one to eight ASCII character name of the macro to be replaced

SECTION 4 - ERRORS AND ABORTS

4.1 Abort Codes

The following is a MACLIBR abort code and its corresponding message:

ME99	ERROR(S) (DESCRIBED ON LFC LO) DETECTED DURING EXECUTION

4.2 Error Messages

When one of the following error messages is displayed, MACLIBR processing is inhibited until the error condition is corrected.

ARGUMENT 'N1' MATCHES ARGUMENT 'N2'

Macro parameters in the N1 and N2 positions of the parameter list are equal.

DIRECTIVE FILE READ ERROR

Error condition detected while reading the directive file.

DUMMY PARAMETERS OVERFLOW

A macro has exceeded the maximum of 255 parameters.

DYNAMIC ALLOCATION OF *U1 SCRATCH FILE FAILED

A scratch file the same size as the MAC file could not be allocated. There may be insufficient disc space.

EOF/EOM ON DIRECTIVE FILE

An end-of-file on the directive file was encountered before normal termination by an /EXIT or /END directive.

ILLEGAL DIRECTIVE

The directive is not a legal directive.

MAC FILE SIZE INCREASE REQUIRED

The updated macro library is larger than the macro library file.

NAME NOT FOUND

A macro specified on a /REPLACE, /INSERT, /DELETE, or /DISPLAY directive was not found in the library file assigned to MAC. The macro may not exist or the file may be positioned beyond that macro. /INSERT, /REPLACE, and /DELETE directives must be entered in the sequence in which the specified macros are found in the macro library file.

WARNING: ACCESS RIGHTS ARE LIMITED. OWNER AND/OR PROJECT MUST BE CHANGED OR THE FILE ASSIGNED TO "MAC" MUST BE RECREATED TO ALLOW FOR DESIRED ACCESS PRIVILEGES.

The owner and/or project attempting to access the file assigned to MAC does not have the required access rights to the file.

4.3 Information Messages

The following messages display information about a previous MACLIBR operation. They are not error or abort conditions.

MAC UPDATE COMPLETE

The editing of the macro library assigned to logical file code MAC is complete. Updates have been written from the scratch file to the library.

CURRENT MAC POSITION: location

This message is issued, followed by the current position of the macro library file, when a /LOG directive is encountered following a /DELETE, /INSERT, or /REPLACE directive.

REPOSITIONED TO: location

When a /DELETE, /REPLACE, or /INSERT directive is followed by a /DISPLAY or /LOG directive, processing of the /DISPLAY or /LOG completes and this message lists the current location of the macro library file.

SECTION 5 - EXAMPLES

The following example generates a new macro library with source input from magnetic tape:

```
$JOB CREATE  OWNER
$ASSIGN SI TO DEV=MT          Assign source file to tape
$ASSIGN MAC TO MYMACS         Assign macro library to MYMACS
$EXECUTE MACLIBR
/CREATE
(Macro Source)
/END
$EOJ
$$
```

The following example logs each macro by number and name:

```
$JOB LOG OWNER
$EXECUTE MACLIBR
/LOG
/END
/EXIT
$EOJ
$$
```

The following example displays the macro named M.EQUS:

```
$JOB DISPLAY OWNER
$EXECUTE MACLIBR
/DIS M.EQUS
/END
/EXIT
$EOJ
$$
```

The following example appends the macro named M.TEST to the macro library and lists it with no output of dummy symbols:

```
$JOB APPEND OWNER
$EXECUTE MACLIBR
/MAC BODY                     List with no dummy symbol output
/LOG END
/APPEND
(M.TEST Source)
/END
$EOJ
$$
```

The following example updates the macro library using the /REPLACE, /DELETE, and /INSERT directives:

```
$JOB UPDATE OWNER
$ASSIGN MAC TO MYMACS
$EXECUTE MACLIBR
/LOG                              List current macros
/LOG END                          List updated macro library
/REP  M.EQUS                      Replace M.EQUS
(Replacement Macro Source)
/END
/DELETE  M.EXIT                   Delete M.EXIT
/INS M.FADD                       Insert macro before M.FADD
(Macro Source to be Inserted)
/END
/EXIT                             Copy from scratch to MAC
$EOJ
$$
```

Media Conversion (MEDIA)

MPX-32 Utilities

CONTENTS

MEDIA CONVERSION (MEDIA)

SECTION 1 - OVERVIEW

1.1 General Description

The Media Conversion (MEDIA) utility implements a procedural language used to manipulate the contents of data files contained on various media. MEDIA allows the user to: copy one file to another file, copy a file from one type of medium to another (e.g., copy a magnetic tape file to a disc file), or dump a file from one type of medium to another (e.g., from magnetic tape to a line printer).

MEDIA can also manipulate data by rearranging one group of columns on an input file to another group of columns on an output file or merging data from multiple files into an output file. Another feature allows data to be converted from one type of code to another, such as from EBCDIC to ASCII.

MEDIA recognizes 1 to 16 character file names. Unless specified, files assigned to logical file codes will be forced to the appropriate format - blocked or unblocked.

1.2 Directive Summary

The following list summarizes the MEDIA directives. Refer to Section 3 for detailed descriptions of each directive.

Directive	Function
BACKFILE	Positions file or device back n files
BACKREC	Positions file or device back n records
BUFFER	Names a buffer (B03-B09) and specifies its size or resets a buffer's current read address to the buffer's start address
CONVERT	Converts contents of a buffer from ASCII to BCD, BCD to ASCII, ASCII to EBCDIC, EBCDIC to ASCII, or 026 to 029
COPY	Copies input records from a file or device to an output file or device
DUMP	Copies a file by converting it to ASCII-coded hexadecimal. Output is to the line printer or SLO.
END	Indicates the end of a MEDIA directive stream
EXIT	Terminates MEDIA processing
GOTO	Specifies conditional transfer to another directive based on counter value, error, or EOF. Transfer can also be unconditional.

INCR	Adds a specified value to counter (K01-K20)
MESSAGE	Sends a message to operator's console
MOVE	Moves bytes from one buffer to another buffer
OPTION	Modifies default output characteristics for devices
READ	Copies data to a buffer. Provides count by bytes, halfwords, or words.
REWIND	Rewinds the specified logical file code
SETC	Sets counter (K01-K20) to a specified value
SKIPFILE	Positions file or device forward n files
SKIPREC	Positions file or device forward n records
VERIFY	Compares records on one file or device to records on another file or device
WEOF	Writes an end-of-file (EOF) on a file
WRITE	Copies data from a buffer. Provides count by bytes, halfwords, or words.

SECTION 2 - USAGE

2.1 Accessing Media

MEDIA can be accessed in the batch or interactive modes in one of three ways:

```
$MEDIA
$RUN MEDIA
$EXECUTE MEDIA
```

$RUN MEDIA is valid only from the system directory.

When accessing MEDIA interactively, the MED> prompt is displayed:

```
TSM>$MEDIA
MED>
```

2.2 Logical File Code Assignments

MEDIA provides two default logical file code (LFC) assignments: *IN for MEDIA source input and *OT for listed output. All other LFC assignments for input and output files are user-defined. All assignments are made by $ASSIGN job control language statements before MEDIA is executed.

The following sections describe the LFC assignments used by MEDIA. The default and optional LFC assignments are summarized in Table 2-1.

2.2.1 Source Input (*IN)

MEDIA directives are assigned to logical file code *IN for input.

***IN Default and Optional Assignments**

The default assignment for *IN is to the System Control file (SYC):

```
$ASSIGN *IN TO SYC
```

There are two optional assignments for *IN:

```
                           {pathname   }
$ASSIGN *IN TO             {DEV=devmnc }
```

pathname	is the pathname of a file containing MEDIA directives
devmnc	is the device mnemonic of a device containing MEDIA directives

2.2.2 Listed Output (*OT)

Listed output contains a list of the MEDIA directives processed during the MEDIA session. Listed output is assigned to logical file code *OT.

***OT Default and Optional Assignments**

The default assignment for *OT is to logical file code UT:

```
$ASSIGN *OT TO LFC=UT
```

In the interactive mode, output is generated on the user terminal. In the batch mode, output is generated on the SLO device.

There are two optional assignments for *OT:

```
                           {pathname   }
$ASSIGN *OT TO             {DEV=devmnc }
```

pathname	is the pathname of the file to contain listed output
devmnc	is the device mnemonic of the device to contain listed output

2.2.3 Input Files

There is no default assignment for input files; they are user-defined. Input files can be from cards, permanent disc files, or magnetic tape. Input files are specified in MEDIA directives by referring to the LFC used in the $ASSIGN statement. Assignments for input files must specify blocked or unblocked format, as appropriate to the data contained in the assigned disc or tape file.

For unblocked input, the following transfer count limitations are in effect: 192 words for a disc, 4095 half words for magnetic tape, and 120 bytes for a card device. For blocked input, the transfer count is limited to 254 bytes.

For input and output files, MEDIA recognizes a hexadecimal 0F (EOF indicator) record only on card reader and card punch devices.

2.2.4 Output Files

There is no default assignment for output files; they are user-defined. If output is directed to a disc file, that file must have been previously created. Output files are specified in MEDIA directives by referring to the LFC used in the $ASSIGN statement. Assignments for output files should specify blocked or unblocked format. For unblocked output, transfer counts are limited to 133 bytes for line printers and the actual write count for files and magnetic tapes.

2.2.5 LFC Summary

The following is a table of LFCs used by MEDIA and their default and optional assignments.

Table 2-1
MEDIA LFC Summary

LFC	Default Assignment	Optional Assignment
*IN	SYC	pathname DEV=devmnc
*OT	LFC=UT	pathname DEV=devmnc
Input Files	none	User-assigned
Output Files	none	User-assigned

2.3 Exiting MEDIA

To exit MEDIA directive input in interactive mode, enter:

```
    CNTRL C or END
```

If MEDIA execution input is from UT, enter:

```
    CNTRL C
```

To exit MEDIA directive input in batch mode, use the END directive.

If MEDIA execution input is from SYC, follow the last data line with a $ directive.

SECTION 3 - DIRECTIVES

3.1 Introduction

MEDIA source statements have three fields: label field, directive field, and parameter field. Each field must be separated by a comma. Embedded blanks within a MEDIA source statement are ignored.

The label field is optional. It is used to direct a branch to that source statement by specifying the label in a GOTO or END directive. Labels can be from 1 to 255, beginning in column one. Labels do not have to be numbered sequentially; they bear no relationship to the physical sequence of MEDIA directives in a control file.

Directives begin in the first input column following a label, or column one if a label is not used. A maximum of 256 MEDIA directives can be specified in one jobstream.

Parameters begin in the first input column following a directive. Numeric parameters are specified in decimal and are limited to eight digits. Multiple parameters are separated by commas.

Two types of predefined areas may be referenced within MEDIA directives: buffer areas and counters.

The two predefined buffer areas are referenced by the names B01 and B02. Each buffer is 2048 words in length. Additional buffer areas may be defined as B03 to B09 by using the BUFFER directive. A total of 3K bytes can be allocated for buffers B03 to B09. Buffer names other than B01 to B09 are not valid.

Twenty predefined counter cells exist within MEDIA. These counters are referenced by the names K01 to K20. Counters may be used as program flags, record counters, file counters, etc. They may contain any positive decimal value within the range of 0 to 99,999,999.

3.2 BACKFILE Directive

The BACKFILE directive positions the specified LFC backwards by the number of files specified in the count field. After the BACKFILE operation, the pointer is positioned at the end-of-file.

Syntax:

```
BACKFILE, lfc, count
```

lfc	is the LFC on which to perform the BACKFILE operation
count	is one to eight digits specifying the number of files to skip

3.3 BACKREC Directive

The BACKREC directive positions the specified LFC backwards by the number of records specified in the count field.

Syntax:

```
BACKREC, lfc, count
```

lfc	is the LFC on which to perform the BACKREC operation
count	is one to eight digits specifying the number of records to skip

3.4 BUFFER Directive

The BUFFER directive defines a buffer or resets the current buffer read address to the buffer starting address. The space allocated to the buffer is allocated from a 3000-byte buffer pool, which is the maximum allowable additional buffer.

Syntax:

```
                {nbytes}
BUFFER,buffer,  {R     }
```

buffer	is the name of the buffer, B03 to B09. Buffers B01 and B02 are predefined. Buffer names B03 to B09 must be defined by a BUFFER directive before being used.
nbytes	is the size of the buffer in decimal bytes
R	resets the buffer pointer. If specified, B01 and B02 can be specified in the buffer field.

3.5 CONVERT Directive

The CONVERT directive converts data in a buffer to the specified code.

Syntax:

```
CONVERT,buffer,code [,nbytes]
```

buffer — is the name of the buffer, B01 to B09. Buffer names B03 to B09 must be defined by a BUFFER directive before being used.

code — is the four-character code specifying the type of conversion:

2629	026 to 029
ASBC	ASCII to BCD
BCAS	BCD to ASCII
ASEB	ASCII to EBCDIC
EBAS	EBCDIC to ASCII

nbytes — is the number of bytes to be converted. If this parameter is omitted, the count is obtained from the total number of bytes read and/or moved into the buffer.

3.6 COPY Directive

The COPY directive copies the specified input file, record by record, to the specified output file until an end-of-file (EOF) is encountered on either file. The EOF is not copied to the output file. Use the WEOF directive to write an EOF mark on the output file. The input and output files must have been previously defined in a $ASSIGN statement.

Syntax:

```
    COPY, lfcl, lfc2
```

lfcl	is the LFC identifying the input file to be copied
lfc2	is the LFC identifying the output file to which the copy is made

3.7 DUMP Directive

The DUMP directive copies an input file to an output file and converts it to ASCII-coded hexadecimal with side-by-side ASCII translation. The dump is terminated when an EOF is encountered on either file or when the specified number of records is copied.

Syntax:

```
    DUMP, lfcl, lfc2 [,recordcount]
```

lfcl	is the LFC identifying the file to be dumped
lfc2	is the LFC identifying the output file
recordcount	is the number of records in the file to be dumped. If not specified, the dump terminates at EOF on either file.

3.8 END Directive

The END directive indicates the end of MEDIA directives. If no syntax errors are detected in the first pass, the END directive initiates processing of the MEDIA directives. Control is transferred back to the first MEDIA directive if a label is not specified. If a label is specified, control transfers to the source statement corresponding to the label.

Syntax:

```
    END [,label]
```

label	is a numeric label from 1 to 255 associated with a directive. Control is transferred to that directive. If label is not specified, control is transferred to the first directive in the stream.

3.9 EXIT Directive

The EXIT directive terminates the execution of MEDIA directives.

EXIT directives may appear anywhere in the directive stream. At least one EXIT directive must appear prior to the END directive.

Syntax:

```
     EXIT
```

3.10 GOTO Directive

The GOTO directive transfers control to another MEDIA directive. The branch is unconditional if no arguments are specified, or conditional if the specified condition is evaluated as true. If none of the conditional specifications is satisfied, processing continues with the next MEDIA directive. The following conditions can be specified:

. the counter is equal to a value

. an end-of-file is encountered on the specified file

. an I/O error occurs on the specified file

Syntax:

```
                  ┌ ,counter, value ┐
     GOTO, label  │ ,EOF, lfc       │
                  └ ,ERR, lfc       ┘
```

label	is the numeric label in the range 1 to 255 associated with a directive. Control is transferred to that directive.
counter	is the name of the counter, K01 to K20 (See the INCR and SETC directives)
value	is a counter value that specifies when control is to be transferred
EOF	specifies conditional transfer at end-of-file. When this parameter is used, the GOTO directive must directly follow an I/O directive, such as READ or WRITE.
ERR	specifies conditional transfer if an I/O error occurs
lfc	if the LFC indicating the file where EOF or ERR conditions apply

3.11 INCR Directive

The INCR directive adds a value to a counter.

Syntax:

INCR, counter, value

counter	is the name of the counter, K01 to K20
value	is the one to eight digit decimal number to be used as an increment

3.12 MESSAGE Directive

The MESSAGE directive displays a message on the operator's console. A maximum of 256 bytes of message information can be displayed in one MEDIA jobstream. Up to 72 characters per MESSAGE directive can be specified.

Syntax:

MESSAGE, 'message'

'message'	is a message of up to 62 alphanumeric characters to be displayed on the system console. The single quotes are required.

3.13 MOVE Directive

The MOVE directive moves the specified number of bytes from buffer1 to buffer2. The starting byte positions for buffer1 and buffer2 are specified as an absolute byte number or a counter.

Syntax:

$$\text{MOVE, nbytes, buffer1,}\left\{\begin{matrix}\text{counter}\\\text{startbyte}\end{matrix}\right\}\text{, buffer2}\left[\begin{matrix}\text{,counter}\\\text{,startbyte}\end{matrix}\right]$$

nbytes	is the number of bytes to be moved
buffer1	is the name of the buffer containing the data to be moved, B01 to B09. Buffer names B03 to B09 must be defined by a BUFFER directive before being used.
counter	is the name of a counter, K01 to K20, indicating the starting byte position
startbyte	is an absolute indicator of the starting byte position
buffer2	is the name of the buffer where data is to be moved, B01 to B09. Buffer names B03 to B09 must be defined by a BUFFER directive before being used. If both counter and startbyte are not specified, the current read address for buffer2 is used.

3.14 OPTION Directive

The OPTION directive specifies nonstandard options for a file. See Table 3-1 for option definitions and defaults.

Syntax:

```
    OPTION, lfc [,BLOCKED] [,B2OF] [,B8OF] [,E] [,H]
                           [,B2ON] [,B8ON] [,O] [,L]
```

lfc	is the LFC to which the options are assigned
BLOCKED	is the blocked option. Though syntactically correct, the MPX-32 default for an assigned file is blocked. The $ASSIGN statement must be used to indicate unblocked.
B2OF	inhibits option 2
B2ON	enables option 2
B8OF	inhibits option 8
B8ON	enables option 8
E	specifies even parity option
O	specifies odd parity option
H	specifies high density option
L	specifies low density option

Table 3-1
MEDIA Options

DEVICE	BLOCKED OPTION	BIT 2 OPTION	BIT 8 OPTION	PARITY	DENSITY
CARD READER READER/PUNCH	N/A	*B2OF-Automatic Mode Select B2ON-Interpret Bit 8	N/A B8OF-Forced ASCII B8ON-Forced Binary	N/A	N/A
PAPER TAPE READER	N/A	*B2OF-Read Formatted Skipping Leader B2ON-Read Unformatted	N/A B8OF-Do Not Skip Leader B8ON-Skip Leader	N/A	N/A
PAPER TAPE PUNCH	N/A	*B2OF-Punch in Formatted Mode B2ON-Punch Unformatted	N/A	N/A	N/A
LINE PRINTER, TELETYPE	N/A	*B2OF-First Character is Carriage Control B2ON-No Carriage Control	N/A	N/A	N/A
9-TRACK MAG TAPE	Blocked I/O	N/A	N/A	N/A	N/A
7-TRACK MAG TAPE	Blocked I/O	*B2OF-Read/ Write Packed (Binary) Mode B2ON-Interpret Bit 8 (BCD)	N/A B8OF-Inter-Change Parity) B80N-Packed (Binary)	E (Even Parity) O (Odd)	H (800 BPI) L (556 LPI)
MOVING-HEAD, FIXED-HEAD DISC	Blocked I/O	B2OF-Report EOF for Unblocked Reads B2ON-No EOF Reporting for Unblocked Reads	N/A	N/A	N/A

*Default Options
N/A - Not Applicable

3.15 READ Directive

The READ directive reads one record from a file into buffer B01, or into the optionally specified buffer, starting at the current buffer address. The current buffer address is advanced after the read and is reset only by a write from the specified buffer or by a buffer reset via the BUFFER directive.

Syntax:

```
    READ,lfc[,buffer] [,count]
```

lfc	is the LFC identifying the file to be read
buffer	is the name of the buffer, B01 to B09. Buffer names B03 to B09 must be defined by a BUFFER directive before being used. If not specified, B01 is used by default.
count	is the number of bytes (B), halfwords (H), or words (W) to be read (e.g., B22, H10, H2048, W192). If not specified, one record is read.

3.16 REWIND Directive

The REWIND directive rewinds the specified logical file code. If the LFC is a magnetic tape, the tape is positioned at its beginning. If the LFC is a file, REWIND returns to the file's first record. If the file is being sent to the line printer, a top-of-form is performed.

Syntax:

```
    REWIND,lfc
```

lfc	is the LFC referencing the file or device to rewind

3.17 SETC Directive

The SETC directive sets a counter to a specified value.

Syntax:

```
    SETC,counter,value
```

counter	is the name of the counter, K01 to K20
value	is up to eight decimal digits specifying the value to which the counter is to be set

3.18 SKIPFILE Directive

The SKIPFILE directive skips forward the number of files specified in the count field on the specified logical file code.

Syntax:

```
     SKIPFILE,lfc,count
```

LFC	is the LFC on which to perform the SKIPFILE operation
count	is the decimal number of files to be skipped

3.19 SKIPREC Directive

The SKIPREC directive skips forward the number of records specified in the count field on the specified logical file code.

Syntax:

```
     SKIPREC,lfc,count
```

lfc	is the LFC on which to perform the SKIPREC operation
count	is the decimal number of records to skip

3.20 VERIFY Directive

The VERIFY directive compares the two files and displays the record numbers that differ. The verification is terminated when an EOF is encountered on either file. If records of unequal length are verified, the file specified by lfcl must contain the shorter record size.

Syntax:

```
     VERIFY,lfcl,lfc2
```

lfcl	is the LFC of the first file to be compared
lfc2	is the LFC of the second file to be compared

3.21 WEOF Directive

The WEOF directive writes an end-of-file (EOF) on the specified file.

Syntax:

```
    WEOF,lfc
```

lfc	is the LFC of the file on which to write the EOF

3.22 WRITE Directive

The WRITE directive writes one record from a buffer to a file. The WRITE directive resets the current buffer address and byte count for the output buffer.

Syntax:

```
    WRITE,lfc[,buffer] [,count]
```

lfc	is the LFC previously assigned for output files in an $ASSIGN statement
buffer	is the name of the buffer, B01 to B09. If buffer name is not supplied, B01 is used by default. Buffer names B03 to B09 must be defined by a BUFFER directive before being used.
count	is the number of bytes (B), halfwords (H), or words (W) to be written (e.g., B22, H10, H2048, W192). If count is not specified, the total number of bytes, halfwords, or words read and/or moved into the buffer is used as the output count.

SECTION 4 - ERRORS AND ABORTS

4.1 Error Codes

MEDIA errors are flagged with a two-digit diagnostic code. An abort is generated at the end of a MEDIA program that contains errors so that conditional batch processing directives can be used.

The status for a device is printed on the output file if an I/O error occurs.

If a loop is being executed where record or file information is accumulated, the information will be printed even if an abort occurs.

The following are MEDIA error codes and their explanations.

Code	Explanation
01	Control specification invalid
02	File code unassigned
03	Invalid conversion code specified
04	Invalid count specification
05	No available FCB, excessive file assignments
06	Statement number not between 1 and 255
07	Invalid buffer name
08	Buffer already defined
09	Insufficient buffer space available
10	Undefined buffer
11	Invalid device assignment
12	Invalid counter name specified
13	Insufficient message storage space available
14	Invalid byte number or number of bytes specifications
15	Invalid optional parameter
16	Missing parameter

Code	Explanation
17	Incorrect message format
18	Invalid decimal or hexadecimal character
20	No END statement
21	Excessive number of control statements specified
22	Fatal control statement error(s)
23	Undefined statement number encountered
24	Execution of END statement attempted
25	Length of READ/MOVE exceeds buffer size
26	WRITE statement which is not preceded by READ statement must specify count
27	End-of-medium encountered on input file
28	End-of-medium encountered on output file
29	CONVERT statement specified zero byte count
30	Duplicate statement number

4.2 Abort Codes

The following are MEDIA abort codes and their messages.

Code	Message
MD01	ERROR(S) (DESCRIBED ON LFC *OT) DETECTED DURING EXECUTION
MD02	AT EOF ON SLO FILE

SECTION 5 - EXAMPLES

The following example copies the tape assigned to IN to the output tape assigned to OT, and writes an EOF on OT; both tapes are rewound and then verified:

```
$JOB EXAMPLE1 OWNER
$ASSIGN IN TO DEV=MT
$ASSIGN OT TO DEV=MT
$EXECUTE MEDIA
COPY,IN,OT
WEOF,OT
REWIND,IN
REWIND,OT
VERIFY,IN,OT
EXIT
END
$EOJ
$$
```

The following example dumps the first 50 records of the second file on tape T132 to an SLO file:

```
$JOB EXAMPLE3 OWNER
$ASSIGN AB TO DEV=MT ID=T132
$ASSIGN DP TO SLO
$EXECUTE MEDIA
REWIND, AB
SKIPFILE, AB, 1
DUMP, AB, DP, 50
EXIT
END
$EOJ
$$
```

The following example outputs the first 40 columns of a maximum of 100 records to the line printer:

```
$JOB EXAMPLE4 OWNER
$ASSIGN IN TO FILE1
$ASSIGN OT TO DEV=LP
$EXECUTE MEDIA
OPTION, OT,,B2ON
SETC, K1, 0
3,READ, IN,,B40
GOTO,5,EOF,IN
WRITE,OT
INCR,K1,1
GOTO,5,K1,100
GOTO,3
5,EXIT
END
100 DATA RECORDS
$EOJ
$$
```

The following example reads two source program files. Columns 20 to 40 of FILE1 are moved to columns 10 to 30 of the output image. Columns 65 to 80 of FILE2 are moved to columns 31 to 46 of the output image. The output image is written to a tape in 120-byte records:

```
$JOB EXAMPLE5 OWNER
$ASSIGN IN1 TO FILE1
$ASSIGN IN2 TO FILE2
$ASSIGN OT TO DEV=MT ID=SAVE
$EXECUTE MEDIA
BUFFER,B04,120
4,READ,IN1,B01
GOTO,5,EOF,IN1
READ, IN2, B02
GOTO,5,EOF,IN2
MOVE, 21,B01, 19, B04, 9
MOVE, 16, B02, 64, B04
WRITE, OT, B04, B120
BUFFER, B01, R
BUFFER, B02, R
GOTO, 4
5, WEOF, OT
REWIND, OT
EXIT
END
$EOJ
$$
```

The following example copies FILE1 to FILE2 in the interactive mode:

```
TSM> $AS IN TO FILE1
TSM> $AS OUT TO FILE2
TSM> $MEDIA
MPX-32 UTILITIES RELEASE x.x (MEDIA Rx.x.x)
(C) COPYRIGHT 1983 GOULD INC., CSD, ALL RIGHTS RESERVED
MED> COPY,IN,OUT
  COPY,IN,OUT
MED> EXIT
  EXIT
MED> END
  END
MEDIA COMPILATION COMPLETE: EXECUTION STARTED
MEDIA EXECUTION COMPLETE
  00000001 FILES COPIED
TSM>
```

Source Update (UPDATE)

MPX-32 Utilities

CONTENTS

FIGURES

TABLES

SOURCE UPDATE (UPDATE)

SECTION 1 - OVERVIEW

1.1 General Description

The Source Update (UPDATE) utility adds, replaces, or deletes lines of source code within a particular file. It can also be used to maintain sets of source files by adding or deleting complete files.

UPDATE can be used to build and edit tapes containing multiple source files for software libraries into a single tape or disc file. Files can be positioned by specifying the number of files, or symbolically by referring to a header record. To symbolically position a file, the file must be in library format.

Library format is a structure in which source code is preceded by a header record and terminated with a single end-of-file mark. Any group of files in library format can be positioned symbolically using UPDATE. UPDATE also allows the insertion of header records during processing.

UPDATE recognizes 1 to 16 character file names. Unless specified, files assigned to logical file codes will be forced to the appropriate format - blocked or unblocked.

The following section summarizes the UPDATE directives in alphabetical order. Underlining indicates valid abbreviations. Section 3 describes the directives in detail.

1.2 Directive Summary

Directive	Function
/ADD	Adds source lines after the specified line of source
/AS1	Reassigns input or output logical file codes to another permanent disc file
/AS3	Reassigns input or output files to another configured peripheral device
/BKSP	Backspaces a specified number of files
/BLK	Blank fills the sequence field (columns 73 to 80) of each record
/COM	Places comments within a directive stream
/COPY	Copies all files up to specified header record
/DELETE	Omits the specified input source lines from the output file

Directive	Function
/END	Indicates the end of additions, deletions, and replacements. The remaining source lines from the input file are copied as-is through EOF.
/EXIT	Indicates the end of the UPDATE control stream
/INSERT	Copies one file from the current input medium and optionally updates the header text
/LIST	Controls the generation of listed output
/MOUNT	Allows the mounting and dismounting of tapes without exiting UPDATE
/NBL	Terminates a /BLK directive
/NOLIST	Resets the /LIST directive's options or terminates the /LIST request
/NOSEQN	Stops sequencing source statements
/REPLACE	Replaces source lines in an output file with source lines which follow, up to the next directive
/REWIND	Rewinds the specified input or output file
/SCAN	Sets the number of characters to scan on the remaining directives
/SELECT	Selects an alternate logical file code for input
/SEQUENCE	Numbers source statements of the current file or all files in sequential order
/SKIP	Skips files up to a specified header record. If the header is a numeric string, this directive skips the number of files in the string.
/USR	Permits the directory to be changed
/WEOF	Writes an EOF on the output medium. This directive must not be used when formatting a library file.

SECTION 2 - USAGE

Source updating is a two-pass process. In the first pass, UPDATE reads control directives and updating statements from the file or device assigned to logical file code SYC. All directives within the control stream are scanned for errors. The control stream, with error diagnostics, is copied to a work file for actual UPDATE processing. The work file is assigned to logical file code UTY and is normally a temporary file.

If directive errors are detected, UPDATE exits after it encounters an /EXIT directive in the first pass. A listing of the control stream and error diagnostics is written. When an /EXIT directive is encountered without detecting directive errors, the updating sequence begins. Updating continues until all of the stored directives have been sequentially processed.

Because UPDATE processes files sequentially, the line number specified with any directive must be equal to or greater than the line number specified in the previous UPDATE directive.

2.1 Accessing UPDATE

UPDATE can be accessed from the batch or interactive modes in one of three ways:

```
$UPDATE
$RUN UPDATE
$EXECUTE UPDATE
```

$RUN UPDATE is valid only from the system directory.

When accessing UPDATE interactively, the UPD> prompt is displayed:

```
TSM> $UPDATE
UPD>
```

2.2 Logical File Code Assignments

Logical file codes (LFCs) are assigned using the $ASSIGN job control language statement. Previously cataloged file assignments can be overridden by the $ASSIGN statement.

Input and output files can be in either compressed or standard source format. A listing can also be printed as the output file is generated.

Files assigned to logical file codes SO, SI1, SI2, and SI3 are assumed blocked unless otherwise specified with the $ASSIGN and $OPTION job control language statements.

The following sections describe UPDATE logical file code usage. Upon entry, UPDATE marks all unassigned file codes as unavailable for use.

2.2.1 Directive Input (SYC)

The directive input file contains UPDATE directives. Directive input is assigned to logical file code SYC.

SYC Default and Optional Assignments

The default assignment for SYC is to the System Control file (SYC):

```
    $AS SYC TO SYC
```

There are two optional assignments for SYC:

```
    $AS SYC TO {pathname   }
               {DEV=devmnc }
```

pathname	is the pathname of a source file containing UPDATE directives
devmnc	is the device mnemonic of a device containing a source file of UPDATE directives

2.2.2 Input Files (SI1, SI2, and SI3)

Input files to be manipulated by UPDATE are assigned to logical file codes SI1 (first input file), SI2 (second input file), and SI3 (third input file).

SI1, SI2, and SI3 Default and Optional Assignments

There are no default assignments for SI1, SI2, and SI3.

There are two optional assignments for input files:

```
       {SI1}
    $AS{SI2} TO{pathname   }
       {SI3}   {DEV=devmnc }
```

pathname	is the pathname of a source input file
devmnc	is the device mnemonic of a device containing the source input

2.2.3 Output File (SO)

The output file contains the input data that has been manipulated by UPDATE. The output file is assigned to logical file code SO.

SO Default and Optional Assigments

There is no default assignment for SO.

There are two optional assignments to SO:

```
$AS SO TO {pathname    }
          {DEV=devmnc  }
```

pathname	is the pathname of a file to contain UPDATE output
devmnc	is the device mnemonic of a device to contain UPDATE output

2.2.4 Work File (UTY)

A work file for intermediate storage and directive and error listings is assigned to logical file code UTY. The default assignment for UTY is to a temporary file. There are no optional assignments for UTY.

2.2.5 Output Image Listing (LO)

The history, history summary, and output image listing for the UPDATE process is assigned to logical file code LO.

LO Default and Optional Assignments

The default assignment for LO is to logical file code UT:

```
$AS LO TO LFC=UT
```

In the interactive mode, output is generated on the user terminal. In the batch mode, output is generated on the SLO device.

There are two optional assignments for LO:

```
$AS LO TO {pathname    }
          {DEV=devmnc  }
```

pathname	is the pathname of a user file to which UPDATE listings are written
devmnc	is the device mnemonic of a device to which UPDATE listings are written

When LO is assigned to SLO or a disc file and UPDATE reaches EOM, MPX-32 extends the file.

2.2.6 LFC Summary

The following is a table of LFCs used by UPDATE and their default and optional assignments.

Table 2-1
UPDATE LFC Summary

LFC	Default Assignment	Optional Assignment
SYC	SYC	pathname DEV=devmnc
SI1	N/A	pathname DEV=devmnc
SI2	N/A	pathname DEV=devmnc
SI3	N/A	pathname DEV=devmnc
SO	N/A	pathname DEV=devmnc
UTY	temporary file	N/A
LO	LFC=UT	pathname DEV=devmnc

2.3 Options

Options are specified by number in a $OPTION job control language statement. Specify the $OPTION statement before the $UPDATE statement in the job stream.

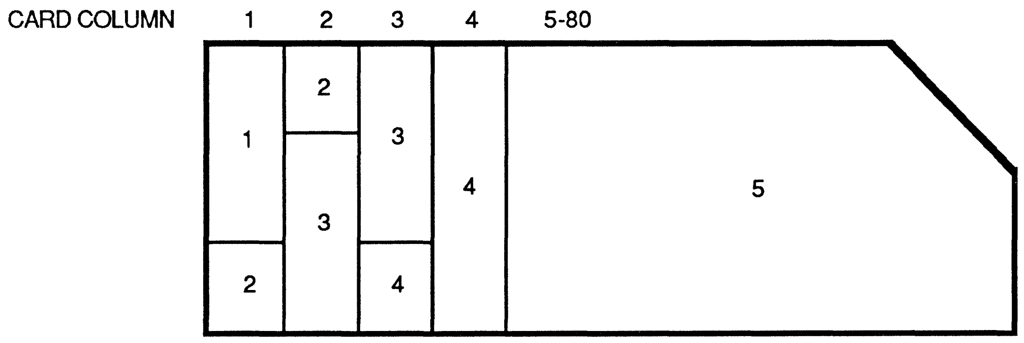
Once declared, an option remains in effect until changed by a directive.

Option	Description
1	Compress Source Output Generates the source output (SO) file in compressed format.
2	Library Source Output Generates the source output (SO) file in library format.
3	Print Control Stream Prints statements entered from the SYC file if SYC is assigned to a disc file. This option should not be specified if SYC is not assigned.

Option	Description
4	Inhibit EOF Inhibits writing end-of-file marks detected on the input file to the output file. This option can be used to consolidate multiple files that are to be assembled or compiled. This option cannot be used in conjunction with option 2.
8	SI1 Not Blocked Allows UPDATE to read the first input file (SI1) in unblocked format. If option 8 is used, unblocked must be specified in the assignment statement.
9	SO Not Blocked Creates an unblocked source output file. If option 9 is used, unblocked must be specified in the assignment statement. If this option is not specified, the file is assumed to be blocked.

2.4 Compressed Source Format

UPDATE can accept source input in compressed format. The source compression ratio varies depending on the number of comments in the source program. A compression ratio of 3:2 to 4:1 is normal for the average range of source decks. All compressed records written by UPDATE are 120 bytes long. The maximum input record size is 192 words. Compressed source can be written to or processed from disc, magnetic tape, paper tape, or cards. See Figure 2-1 for a description of compressed record card format.



(1)	8-BIT TYPE CODE X'BF' AND X'9F' DESIGNATE COMPRESSED SOURCE IMAGE X'9F' INDICATES LAST RECORD OF A COMPRESSED SOURCE MODULE
(2)	8-BIT COUNT OF BYTES REMAINING IN RECORD
(3)	16-BIT CHECKSUM OF RECORD
(4)	16-BIT SEQUENCE NUMBER
(5)	CONTIGUOUS BYTES OF DATA

87D4107

Figure 2-1. Compressed Record Card Format

2.5 Library Source Format

Library source format is a standard format for building library files. This format is used when option 2 is specified.

UPDATE accepts input files in any format, and creates output files in library format (see Figures 2-2, 2-3, and 2-4). Incorrectly formatted header records cause the job to be aborted. If consecutive end-of-file marks are encountered on the input file, one end-of-file mark is written on the output file. Upon completion of the library update, a unique end-of-library file record is written to the output file, and UPDATE exits.

2.6 Exiting UPDATE

To exit UPDATE from the batch and interactive modes, specify the /EXIT directive.

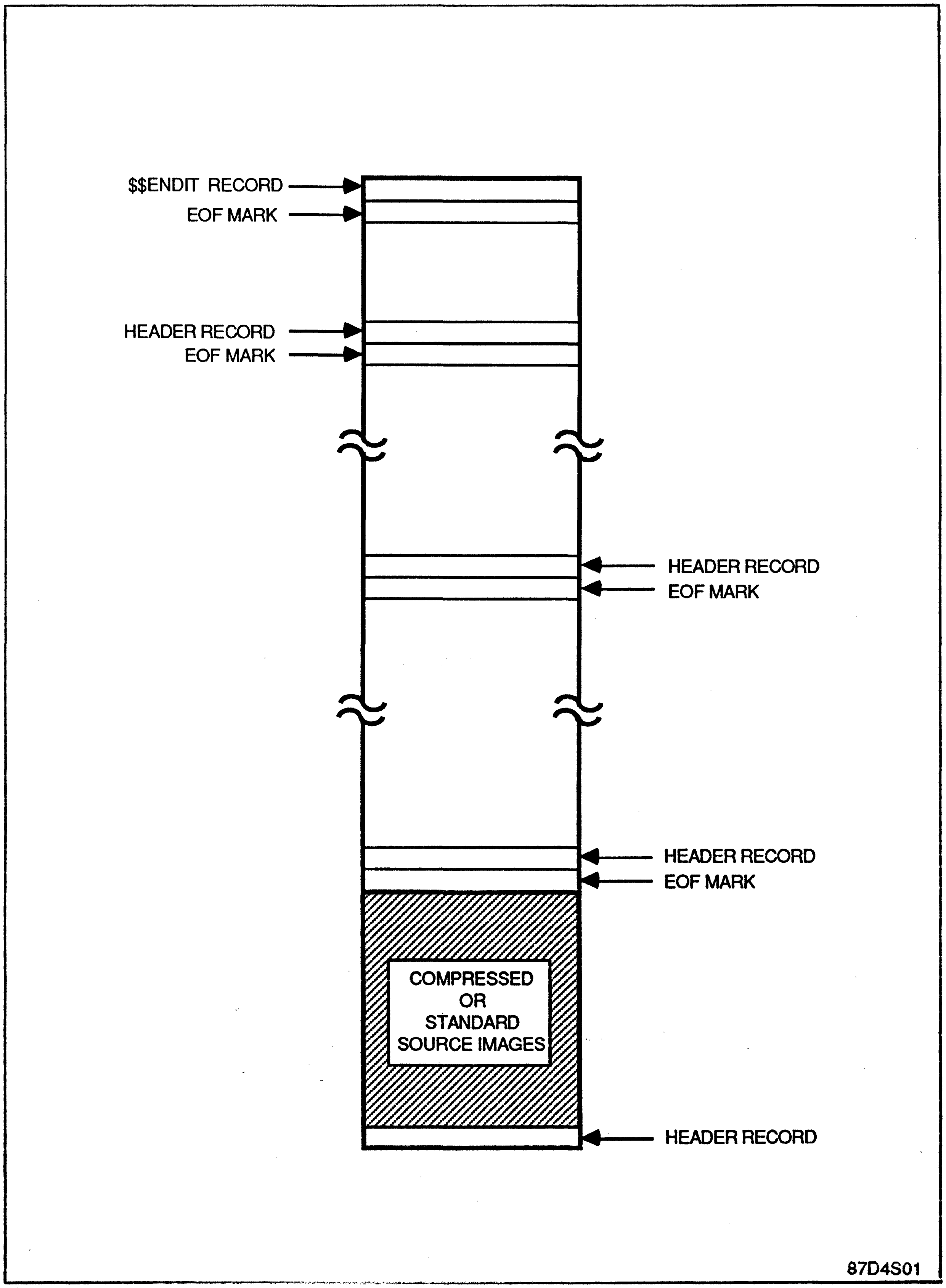


Figure 2-2. Library Format

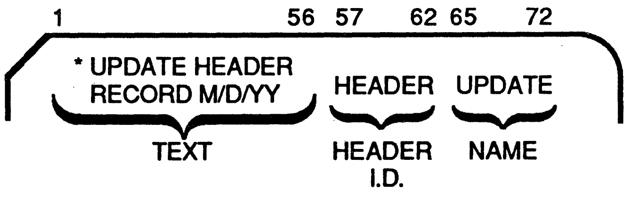


Figure 2-3. Header Record Format

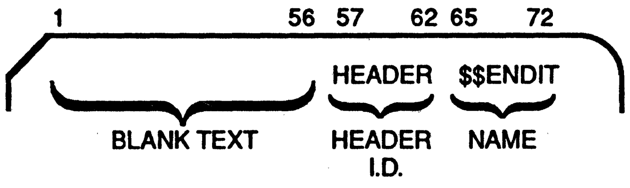


Figure 2-4. End-of-Library File Record

SECTION 3 - DIRECTIVES

3.1 Introduction

UPDATE directives have the following format:

    /directive parameter

A slash (/) in column one identifies the record as a directive. The directive field contains the name of the directive. The directive field may be abbreviated to four characters (including the slash) and must be followed by a blank. Valid abbreviations are indicated by underlining in the directive syntax.

The parameter field contains information that describes the operation to be performed. Parameters are separated by commas or blanks.

Special characters should not be specified in column one of a command file because they can be incorrectly interpreted, resulting in an abort condition. For example, a dollar sign ($) in column one is interpreted by MPX-32 as a job control language (JCL) directive and results in an UPDATE abort.

3.2 /ADD Directive

The /ADD directive adds statements immediately following a specified source line. All statements following the /ADD directive, up to the next directive, are added to the output file.

Syntax:

    /ADD start

start	identifies the source line after which the additions are to be inserted. A specification of zero (/ADD 0) adds statements to the beginning of an existing file.

3.3 /AS1 Directive

The /AS1 directive reassigns input or output logical file codes to another permanent disc file. The mode (blocked or unblocked) is not changed.

Only the first 19 characters of the /AS1 directive are printed on the output.

This directive reassigns only previously assigned logical file codes and does not assign unassigned file codes.

Syntax:

```
/AS1 lfc,filename [,password]
```

lfc	is the logical file code to be reassigned
filename	is the one to eight character name of a disc file
password	is ignored

3.4 /AS3 Directive

The /AS3 directive reassigns input or output logical file codes to another peripheral device. The mode (blocked or unblocked) is not changed.

Only the first 19 characters of the /AS3 directive are printed on the output.

This directive reassigns only previously assigned logical file codes and does not assign unassigned file codes.

Syntax:

```
/AS3 lfc,devmnc [,id,multivol]
```

lfc	is the logical file code to be reassigned
devmnc	is one of the following device mnemonics of a configured peripheral device: CD, CP, CR, M7, M9, or MT
id	is a one to four character reel identifier for magnetic tape devices
multivol	is the volume number for a multivolume magnetic tape operation

3.5 /BKSP Directive

The /BKSP directive backspaces a specified number of end-of-file marks. Detection of beginning-of-medium terminates backspacing and UPDATE proceeds to the next directive.

Syntax:

```
/BKSP lfc [,n]
```

lfc	is the logical file code assigned to the device or file to be backspaced
n	is the number of EOF marks to be backspaced. If not specified, the default is one.

3.6 /BLK Directive

The /BLK directive blank fills the sequence field, columns 73 to 80, of each following record. This directive remains in effect until a No Blank Sequence Field (/NBL) directive is read.

Syntax:

```
    /BLK
```

Usage:

```
/BLK
/COPY 1
/SKIP 1
/COPY 1
/BKSP SI1,1
/COPY 1
```

In this example, three files are copied without sequence numbers.

3.7 /COM Directive

The /COM directive places comments within an UPDATE directive stream. The comment following this directive is ignored by UPDATE.

Syntax:

```
    /COM comment
```

comment	is a user comment. It must be contained within the first 72 columns.

3.8 /COPY Directive

The /COPY directive copies complete input files. Library format files can be copied to the end of a library by the directive /COPY END.

Syntax:

$$\underline{\text{/COPY}} \quad \left\{\begin{matrix}\text{header}\\ \text{END}\end{matrix}\right\}$$

header	is a one to eight character delimiter. If header is an alphanumeric string, UPDATE copies all files up to the header record defined by header. If header is a numeric string, UPDATE copies the number of files represented by the string.
END	copies library format files to the end of the library

3.9 /DELETE Directive

The /DELETE directive deletes the specified sequential source lines from the output file.

Syntax:

/DELETE start [,end]

start	identifies the first source line to be deleted
end	identifies the last source line to be deleted. If not specified, the default is the same value as start.

3.10 /END Directive

The /END directive ends an /ADD, /DELETE, or /REPLACE sequence for a file. The file's remaining source lines are copied through the next EOF. UPDATE then proceeds to the next directive.

Syntax:

/END

3.11 /EXIT Directive

The /EXIT directive ends the UPDATE directive stream. If directive errors were detected, UPDATE exits. If directive errors were not detected, the updating sequence begins. When all directives have been sequentially processed, UPDATE checks for the library option. If the library option is set, a unique end-of-library file record is written to the file or device assigned to logical file code SO. The file is then rewound and UPDATE exits. If UPDATE was run interactively, control returns to TSM.

Syntax:

/EXIT

3.12 /INSERT Directive

The /INSERT directive copies one complete file from the current input file or device.

A maximum of 56 characters of text is allowed for new header text. Text begins at the next character after the first delimiter. If the file containing the new header text is to be submitted to a language translator, the first character of text should be the translator's comment character. For example, a header for a file to be translated by the Macro Assembler would contain an asterisk as the first text character.

The new header text replaces the old header text in a file that already has a header record.

Only the first 19 characters of the /INSERT directive are printed on the output.

Syntax:

```
/INSERT [name,text]
```

name,text — are optional parameters used to create header text in the following format:

1-56	57-62	65-72
text	header identification	name

If name and text are not specified and a header exists, the header is cleared. If name and text are not specified and a header does not exist, the first line of code is treated as the header.

3.13 /LIST Directive

The /LIST directive controls the generation of listed output. This directive may be placed anywhere in the control stream. If /LIST is not specified, no lists are generated during the updating sequence.

When /LIST is specified without optional parameters, an audit trail of the UPDATE session is generated. When a /LIST without parameters is followed by a /COPY directive, a top-of-form eject is performed before each file is listed.

More than one /LIST directive can be specified in a control stream. However, the additional /LIST directives do not reset options. The /NOLIST directive must be used to reset options.

When the FRST parameter is specified without the UPDT parameter, the file number, record count, and first record for each file are printed. /LIST FRST is intended only for checking file sizes or generating a header listing.

Syntax:

```
/LIST [FRST] [,UPDT]
```

FRST	lists the first record of each file. Header records are the first records in library formatted files.
UPDT	lists all control directives and records affected by /ADD, /DELETE, or /REPLACE directives

3.14 /MOUNT Directive

The /MOUNT directive allows the mounting and dismounting of tapes without exiting UPDATE. When entered, the /MOUNT directive and the following mount message are written to the operator console:

```
UPDATE-MOUNT INPUT VOL ON MT UNIT!  R,A,H?
```

The operator responds R for ready, A for abort, or H for hold.

Syntax:

```
/MOUNT lfc [,text]
```

lfc	is the input logical file code (SI1, SI2, or SI3) assigned to a magnetic tape
text	is a user comment to be written to the system console. It must be contained within the first 72 columns.

3.15 /NBL Directive

The /NBL directive terminates a /BLK directive. This directive may be placed anywhere following the /BLK directive in the UPDATE control stream.

Syntax:

```
/NBL
```

Usage:

```
/BLK
/COPY 1
/SKIP 1
/NBL
```

3.16 /NOLIST Directive

The /NOLIST directive controls the generation of listed output by resetting the options set by /LIST. This directive may be placed anywhere following the /LIST directive in the UPDATE control stream. If no parameters are specified, all /LIST options are reset and all listing output is terminated. If a parameter is specified, the corresponding /LIST option is reset.

Syntax:

```
/NOLIST [FRST] [,UPDT]
```

FRST	inhibits listing the first record of each file
UPDT	inhibits listing all directives and records affected by /ADD, /DELETE, or /REPLACE directives

3.17 /NOSEQN Directive

The /NOSEQN directive terminates the sequence numbering of source statements set by the /SEQUENCE directive. By default, statements are not sequenced.

Syntax:

```
/NOSEQN
```

3.18 /REPLACE Directive

The /REPLACE directive replaces a specified set of source lines from the input file with an alternate set. The alternate set may be smaller or larger than the original set of source lines. All statements following the /REPLACE directive, to the next control directive, are inserted into the output file.

Syntax:

```
/REPLACE start [,end]
```

start	identifies the first source line to be replaced
end	identifies the last source line to be replaced. If not specified, the default is the same value as start.

3.19 /REWIND Directive

The /REWIND directive rewinds input or output files.

If the library format option is set, logical file code SO must not be specified.

Syntax:

```
/REWIND lfc [,lfc] ...
```

lfc	is the logical file code assigned to the device to be rewound

3.20 /SCAN Directive

The /SCAN directive sets the number of characters to scan on the remaining directives. This directive should be first in a directive stream to ensure unwanted data is not scanned.

Syntax:

```
/SCAN n
```

n	is the number of characters in the range 6 to 72 to be scanned

3.21 /SELECT Directive

The /SELECT directive selects alternate logical file codes for input. UPDATE initially assumes input from the file or device assigned to SI1. When an input device or file is selected, all subsequent directives, with the exception of /BKSP, /MOUNT, and /REW, pertain to the selected device or file.

Syntax:

```
/SELECT lfc [,BLOCKED]
```

lfc	is the logical file code to which input is assigned (SI1, SI2, or SI3)
BLOCKED	is ignored. Blocking is controlled by the $ASSIGN and $OPTION JCL directives.

3.22 /SEQUENCE Directive

The /SEQUENCE directive sequentially numbers source statements. This directive may be placed anywhere in the control stream. Sequencing is terminated by the /NOSEQN directive. By default, statements are not sequenced.

Syntax:

```
/SEQUENCE [id], [inc], [HOLD]
```

id	is one to three alphanumeric characters to be placed in columns 73 to 75 of all source output records. At least one character must be nonnumeric. If not specified, the default is blanks.
inc	is one to five numeric characters to be placed as a sequence number in columns 76 to 80 of the first record. The value of inc is used to increment the sequence number for subsequent records. If not specified, the default is one.
HOLD	indicates sequencing continues until another /SEQUENCE or a /NOSEQN directive is encountered. If HOLD is specified, the beginning sequence number for all subsequent files is reset to zero and the file is sequenced as specified by the id and inc parameters. If HOLD is not specified, sequencing ends after the current file is processed.

3.23 /SKIP Directive

The /SKIP directive skips complete input files.

Syntax:

```
/SKIP   {header}
        {END   }
```

header	is a one to eight character delimiter. If header is an alphanumeric string, UPDATE skips all input files up to the header record defined by header. If header is a numeric string, UPDATE skips the number of files represented by the string.
END	skips files to the end of a library

3.24 /USR Directive

The /USR directive accesses files in another directory. This directive does not change the current working volume.

If a directory name changes during the UPDATE directive sequence, /USR must precede the first directive in the control stream that requires directory name identification. If a /USR directive is used anywhere in a directive stream to change a directory, the default directory as established by TSM or a JCL statement must be specified by a /USR directive at the beginning of the directive stream. This ensures that UPDATE is using the intended directory during pass two processing. Refer to Section 5--Examples.

Syntax:

```
/USR [dirname [,key] ]
```

dirname	is the one to eight character name of an existing directory on the current working volume. If not specified, defaults to SYSTEM.
key	is ignored

3.25 /WEOF Directive

The /WEOF directive writes an end-of-file (EOF) on the output file assigned to logical file code SO. This directive is not valid when output is in library format.

Syntax:

```
/WEOF
```

SECTION 4 - ERRORS AND ABORTS

4.1 Abort Codes

The following are UPDATE abort codes and their messages.

Code	Message
UD01	ERROR(S) (DESCRIBED ON LFC LO) DETECTED DURING EXECUTION
UD02	ABORT REQUESTED FOR /MOUNT DIRECTIVE
UD03	M.OPENR FAILED FOR LFC SYC
UD04	M.OPENR FAILED FOR LFC UTY
UD05	M.OPENR FAILED FOR LFC LO
UD06	M.OPENR FAILED FOR LFC SI1
UD07	M.OPENR FAILED FOR LFC SI2
UD08	M.OPENR FAILED FOR LFC SI3
UD09	M.OPENR FAILED FOR LFC SO

4.2 Error Messages

The following error messages are generated by UPDATE.

INVALID OPTION SPECIFICATION

Option 4 cannot be specified in conjunction with option 2.

lfc FILE IS NOT ASSIGNED

No file or device is assigned to the LFC named by this message.

INVALID UPDATE DIRECTIVE

The directive is not a valid UPDATE directive.

INVALID UPDATE DIRECTIVE SEQUENCE

An /END directive did not follow an /ADD, /REPLACE, or /DELETE sequence.

UPDATE SEQUENCE ERROR

The start or end address in an /ADD, /REPLACE, or /DELETE statement is not sequential.

xxxxxxxx NOT FOUND

The file represented by xxxxxxxx cannot be found.

IMPROPER HEADER FORMAT

The library file was not created because of an improperly formatted header record.

IMAGE NOT COMPRESSED

An illegal mixture of standard and compressed source exists in the image.

CHECKSUM ERROR

A checksum error was detected in the last record read.

lfc FILE AT END-OF-MEDIUM

The file or device assigned to the LFC named by this message is at EOM.

lfc FILE, UNRECOVERABLE I/O ERROR

An I/O error was encountered on the file or device assigned to the LFC named by this message.

EOF DETECTED BEFORE SPECIFIED LINE NUMBER

An EOF was detected before the line number indicated in an /ADD, /REPLACE, or /DELETE directive could be found.

LIBRARY MODE, DIRECTIVE NOT ALLOWED

A /WEOF directive was detected in the control stream while the library mode was in effect.

lfc FILE ASSIGNED TO UTY FILE

The UTY file code cannot be reassigned by the user.

lfc FILE ASSIGNED TO SLO FILE

An attempt was made to assign an input LFC to the SLO file.

lfc FILE, INVALID BLOCKING BUFFER CONTROL POINTER

An input operation for a blocked file assigned to the LFC named in this message was not successful. The file is either unblocked or does not exist.

PERMANENT FILE NONEXISTENT

A specified file name cannot be located.

INVALID DEVICE TYPE

The specified device type is not supported by UPDATE.

FILE ALLOCATION DENIED

Allocation for a file was denied.

DEVICE ALLOCATION DENIED

Allocation for a device was denied.

UNABLE TO ASSIGN DIRECTORY

The directory cannot be assigned.

UNABLE TO EXCLUSIVELY LOCK FILE REASSIGNED BY /AS1

The file being reassigned by /AS1 is allocated.

SECTION 5 - EXAMPLES

The following example updates the first file on the tape INPT and places the updated file in the file OUTFILE.

```
$JOB EXAMPLE1 OWNER
$ASSIGN SI1 TO DEV=MT ID=INPT
$ASSIGN SO TO OUTFILE
$EXECUTE UPDATE
/ADD 25
(statements to be added)
/DELETE 27, 28
/REPLACE 45,51
(replacement statements)
/END
/EXIT
$EOJ
$$
```

The following example adds and deletes statements. /ADD 0 makes additions to the beginning of an existing file. An updated and compressed copy of PROG11 is written to file OUTFILE, and a listing of all statements affected by the update is written. INPT is in library format, which allows a skip to the header PROG11. The output file OUTFILE is not in library format because option 2 is not specified.

```
$JOB EXAMPLE2 OWNER
$OPTION 1
$ASSIGN SI1 TO DEV=MT ID=INPT
$ASSIGN SO TO OUTFILE
$EXECUTE UPDATE
/LIST FRST,UPDT
/SKIP PROG11
/ADD 0
(statements to be added)
/DELETE 1,5
/ADD 5
(statements to be added)
/END
/EXIT
$EOJ
$$
```

The following example copies all files up to and including the eleventh file onto the tape OPUT and modifies and inserts source from the SI2 file. The primary input files are reselected, and 51 more files are copied to logical file code SO. A listing of the updates made to logical file code SO is written to logical file code SLO.

```
$JOB EXAMPLE3 OWNER
$ASSIGN SI1 TO DEV=MT ID=INPT
$ASSIGN SI2 TO DEV=MT ID=TAPE
$ASSIGN SO TO DEV=MT ID=OPUT
$EXECUTE UPDATE
/LIST UPDT
/COPY 11
/SELECT SI2
/REPLACE 24, 33
(replacement statements)
/END
/SELECT SI1
/COPY 51
/EXIT
$EOJ
$$
```

The following example converts a compressed format punched card deck to a standard format punched deck. The standard deck is sequenced by ten and the three-character id of ABC is placed in bytes 73 to 75 of each card during the conversion.

```
$JOB EXAMPLE4 OWNER
$ASSIGN SO TO SBO SIZE=1000
$ASSIGN SI1 TO SYC
$EXECUTE UPDATE
/SEQ ABC, 10
/COPY 1
/EXIT
(compressed deck)
$EOJ
$$
```

The following example copies the library format output tape to the SO file. The program NEW1 is sequenced by one and inserted, with a header record, on the output file. A new library end-of-tape marker is written on the tape.

```
$JOB EXAMPLE5 OWNER
$OPTION 2
$ASSIGN SI1 TO DEV=MT ID=INPT
$ASSIGN SO TO OUTFILE
$ASSIGN SI2 TO FILE1
$EXECUTE UPDATE
/COPY END
/SELECT SI2
/SEQ
/INSERT NEW1, *NEW 1 REV C JULY 12, 1983
/EXIT
$EOJ
$$
```

The following example lists all the header records on the library tape INPT. The file number and record count for each file is written.

```
$JOB EXAMPLE6 OWNER
$ASSIGN SI1 TO DEV=MT ID=INPT
$EXECUTE UPDATE
/LIST FRST
/COPY END
/EXIT
$EOJ
$$
```

The following example copies PROG13 and the file following PROG16 from INPT to OUTFILE. Output listings are generated for the three programs. The programs that are skipped are not listed.

```
$JOB EXAMPLE7 OWNER
$ASSIGN SI1 TO DEV=MT ID=INPT
$ASSIGN SO TO OUTFILE
$EXECUTE UPDATE
/LIST
/SKIP PROG13
/COPY 1
/SKIP PROG16
/COPY 2
/EXIT
$EOJ
$$
```

The following example copies the first, fifth, sixth, and twelfth files from INPT. Compressed source is written to the disc file PERMFILE without EOFs. All four files are then assembled.

```
$JOB EXAMPLE8 OWNER
$OPTION 1 4
$ASSIGN SI1 TO DEV=MT ID=INPT
$ASSIGN SO TO PERMFILE
$EXECUTE UPDATE
/COPY 1
/SKIP 3
/COPY 2
/SKIP 5
/COPY 1
/WEOF
/EXIT
$EOJ
$JOB ASSEMBLE OWNER
$ASSIGN SI TO PERMFILE
$EXECUTE ASSEMBLE
$EOJ
$$
```

The following example lists the first (header) records of the files on logical file code SI1 and copies PROG10 to logical file code SO. PROG10 from logical file code SI2 is inserted, with a new header record, into logical file code SO. Logical file code SI1 is reselected, files up to PROG11 are skipped, and files PROG11 through the last file on logical file code SI1 are copied to logical file code SO.

```
$JOB EXAMPLE9 OWNER
$OPTION 2
$ASSIGN SI1 TO DEV=MT ID=MAST
$ASSIGN SI2 TO FILE1
$ASSIGN SO TO OUTFILE
$EXECUTE UPDATE
/LIST FRST
/COPY PROG10
/SELECT SI2
/INSERT PROG10,*PROG10 REVB JULY 12, 1983
/SELECT SI1
/SKIP PROG11
/COPY END
/EXIT
$EOJ
$$
```

Subroutine Library Editor (LIBED)

MPX-32 Utilities

CONTENTS

SUBROUTINE LIBRARY EDITOR (LIBED)

SECTION 1 - OVERVIEW

1.1 General Description

The Subroutine Library Editor (LIBED) utility maintains sets of system and user nonbase object modules which can be linked at catalog time. Object modules are contained in library files. A directory file is maintained for each library file. Directory files are used by the LIBED and CATALOG utilities to locate object modules within library files.

LIBED can be used to create and maintain user libraries of object modules. These object modules can then be accessed by external reference from a calling task.

Depending upon the user's access rights to the library and directory files, LIBED can:

. Delete object modules from a library file
. Log the object modules in a library file
. Obtain library and directory file usage statistics
. Initialize library and directory files

A combined maximum of 2048 modules may be deleted, replaced, or added during any LIBED run. A user with read-only access to the files can use only the LOG directive.

References to object modules in a subroutine library are resolved when the referencing task is cataloged.

LIBED recognizes 1 to 16 character file names and 1 to 8 character module names. Unless otherwise specified, files assigned to logical file codes will be forced to the appropriate format, blocked or unblocked.

1.2 Directive Summary

The following list summarizes the LIBED directives. Underlining indicates accepted abbreviations. Each directive is described in more detail in Section 3.

Directive	Function
DELETE	Deletes a module from the library
EXIT	Terminates directive input
LOG	Provides a log of all modules and their global symbol definitions

SECTION 2 - USAGE

2.1 Accessing LIBED

LIBED can be accessed in the batch or interactive modes in one of three ways:

```
$LIBED
$EXECUTE LIBED
$RUN LIBED (valid only from the system directory)
```

When accessing LIBED interactively, the LIB> prompt is displayed:

```
TSM>$LIBED
LIB>
```

2.2 Logical File Code Assignments

There are five user-accessible logical file codes (LFC) associated with LIBED: Control directives (CTL), Directory (DIR), Library general object (LGO), Library (LIB), and LIBED listed output (LLO). Assignment statements must be made before LIBED is executed.

2.2.1 Control Directives (CTL)

The control directive file contains LIBED directives for the specified library. The control directive file is assigned to logical file code CTL. The contents of the file assigned to CTL are written to a temporary buffer. LIBED uses the contents of the buffer as input and performs the specified directives.

CTL Default and Optional Assignments

The default assignment for CTL is to the System Control file (SYC):

```
$ASSIGN CTL TO SYC
```

There is one optional assignment for CTL:

```
$ASSIGN CTL TO pathname
```

pathname	is the pathname of a user file containing LIBED directives for a specified library

2.2.2 Directory (DIR)

The directory file contains pointers for external references used by the specified library. It is assigned to logical file code DIR. The contents of the file assigned to DIR are written to a temporary buffer. LIBED uses the contents of the buffer as a directory for the specified library. The file assigned to logical file code DIR is forced unblocked by LIBED.

The file assigned to DIR must be created using the Volume Manager. Once the file is created, it can be initialized and maintained by LIBED.

The directory must be paired with the corresponding library when making LIBED or CATALOG assignments.

DIR Default and Optional Assignments

The default assignment for DIR is to the system directory MPXDIR:

```
$ASSIGN DIR TO @SYSTEM(SYSTEM)MPXDIR
```

There is one optional assignment to DIR:

```
$ASSIGN DIR TO pathname
```

pathname	is the pathname of a user file containing the directory information to be used for a specified library

2.2.3 Library General Output (LGO)

The library general output file contains one or more object modules and is assigned to logical file code LGO. The contents of the file or device assigned to LGO are written to a temporary buffer. LIBED manipulates the contents of the buffer during directive processing.

Logical file code LGO is forced unblocked by LIBED.

LGO Default and Optional Assignments

The default assignment for LGO is to the System General Object file:

```
$ASSIGN LGO TO SGO
```

There are two optional assignments for LGO:

$$\text{\$AS LGO TO } \begin{Bmatrix} \text{pathname} \\ \text{DEV=devmnc} \end{Bmatrix}$$

pathname	is the pathname of a user file containing one or more object files
devmnc	is the device mnemonic of a device containing one or more object modules

2.2.4 Library (LIB)

The library file contains one or more object modules that have been manipulated by LIBED. The library file is assigned to logical file code LIB. After LIBED processing completes, the final contents of the temporary buffer are written to the file assigned to logical file code LIB. The file assigned to logical file code LIB is forced unblocked by LIBED.

The file assigned to LIB must be created using the Volume Manager. Once the file is created, it can be initialized and maintained by LIBED.

Object modules are automatically replaced if they exist on both the file or device assigned to LGO and the file assigned to LIB. The copy of the object module in LGO replaces the copy of the object module in LIB.

LIB Default and Optional Assignments

The default assignment for LIB is to the system library MPXLIB:

```
$ASSIGN LIB TO @SYSTEM(SYSTEM)MPXLIB
```

There is one optional assignment for LIB:

```
$ASSIGN LIB TO pathname
```

pathname	is the pathname of a library file containing one or more object modules that have been manipulated by LIBED

2.2.5 Listed Output (LLO)

The listed output file contains an audit trail of LIBED processing. The file to be used for listed output is assigned to logical file code LLO.

LLO Default and Optional Assignments

The default assignment for LLO is to logical file code UT:

```
$ASSIGN LLO TO LFC=UT
```

In the interactive mode, output is generated on the user terminal. In the batch mode, output is generated on the SLO device.

There are two optional assignments for LLO:

$$\text{\$ASSIGN LLO TO} \begin{Bmatrix} \text{pathname} \\ \text{DEV=devmnc} \end{Bmatrix}$$

pathname	is the pathname of a user file where LIBED listings will be written
devmnc	is the device mnemonic of a device where LIBED listings will be written

2.2.6 Internal LFCs

The LFCs TLB and TDR are used internally by LIBED to reference the temporary library and directory during create and update operations. They must not be assigned by the user.

2.2.7 LFC Summary

The following is a table of the LFCs used by LIBED and their default and optional assignments:

Table 2-1
LIBED LFC Summary

LFC	Default Assignment	Optional Assignment
CTL	SYC	pathname
DIR	@SYSTEM(SYSTEM)MPXDIR	pathname
LGO	SGO	pathname DEV = devmnc
LIB	@SYSTEM(SYSTEM)MPXLIB	pathname
LLO	LFC = UT	pathname DEV = devmnc

2.3 Options

There are three options that can be specified when using LIBED. When no option is specified, LIBED processes directives from the file or device assigned to logical file code CTL.

If an object module name exists in both the file or device assigned to LGO and the file assigned to LIB, a copy of the object module in LGO replaces the object module in LIB.

When LIBED has completed updating a library, it automatically updates the directory.

Option	Description
1	New library and directory - Initializes a new library and directory, which are assigned to LIB and DIR. The new library contains all the object modules in the file assigned to LGO. CTL logical file code assignments are ignored. All object modules and externals in the new library are logged.
2	Library statistics - The amount of allocated and remaining disc space for the library and directory specified by LIB and DIR is written to LLO.
15	Log time/date - Enables the display of time, date, and program identification information as included in the object module.

If option one is specified, the following listed output is generated:

. a directive list

. a log of module names and external definitions

. statistical information on available library and directory disc space

If option two is specified, the listed output contains statistical information on available library and directory disc space. A list of modules which were specified for deletion but were not deleted is also generated.

If option 15 is specified, the data from logical file code LIB is read to find the data to display. This increases execution time because of the processing required for the I/O and search/display logic.

2.4 Directory File Format

A directory file consists of directory entries paired with the corresponding library file. Each directory entry is three words in length and has the following general format:

		0 15	16 31
Word	0 1	Entry point name	
	2	Starting block number	Starting record number

2.5 Exiting LIBED

To exit LIBED from the batch and interactive modes, specify the EXIT directive.

SECTION 3 - DIRECTIVES

All directives must begin in column one. User comments can be placed after the directive or its associated argument (if any). Directives are separated from user comments or arguments by a blank.

3.1 DELETE Directive

The DELETE directive deletes the specified object module from the library assigned to logical file code LIB. The deleted object module's external definitions are removed from the directory assigned to logical file code DIR.

Syntax:

```
DELETE  name  [comments]
```

name	is the one- to eight-character ASCII module name to be deleted

3.2 EXIT Directive

The EXIT directive terminates directive input, processes the previous directives, and returns control to TSM.

Syntax:

```
EXIT  [comments]
```

3.3 LOG Directive

The LOG directive outputs a log of all object module names and their external definitions to the device or file assigned to logical file code LLO.

Syntax:

```
LOG  [comments]
```

SECTION 4 - ERRORS AND ABORTS

4.1 Abort Codes

The following are LIBED abort codes and their messages:

Code	Message
LE98	UNRECOVERABLE I/O ERROR ON FILE OR DEVICE ASSIGNED TO LFC: LLO ST=RMXX IF ERROR ON OPEN; IOXX IF ERROR ON WRITE.
LE99	LIBED<<FATAL>>TYPE, ERROR AS DESCRIBED ON LLO AND/OR UT

4.2 Error Messages

LIBED issues error messages to LFCs LLO and UT (if interactive) preceded by a <<FATAL>> or (WARNING) heading.

The following information concerns interpretation of error messages:

. <<FATAL>> errors indicate that the requested operation could not be performed. If a library existed, it will remain unchanged. The task abort flag is always set.

. (WARNING) errors report conditions that the user should know about, but that are not catastrophic in nature. The library/directory is always updated when warnings are issued.

. Abnormal status returned by MPX-32 services is reported in the format XXnn, where XX indicates the MPX module involved and nn indicates the specific status. These codes may be interpreted by using the TSM ERR function. If the status is associated with a file operation, the LFC is displayed in the message.

. Object code records read from LGO and/or LIB are subjected to five correctness tests as follows:

Test	Description
HEADER	Record is not identifiable as an object code record (byte 0 does not equal X'DF' or X'FF')
SEQUENCE	Sequence number field of record does not contain expected value
CHECKSUM	Data of record fails checksum validation
INCMPLTE	End-of-file without object code termination item

. A program name of ???????? in an error message indicates that an error occurred between valid modules. A log operation indicates the preceding valid module.

SECTION 5 - EXAMPLES

The following section provides sample programming sequences illustrating the use of various LIBED directives.

Example 1 - Create Library and Directory

The following example creates a subroutine library in the file LIBA and a directory in the file DIRA. The library will contain all modules contained in the file USER.OBJ. A log of the module names and external definitions are produced on the LLO device.

```
$JOB CREATE OWNER
$NOTE CREATE NEW USER LIBRARY
$CREATE LIBA
$CREATE DIRA
$ASSIGN LGO TO USER.OBJ
$ASSIGN LIB TO LIBA
$ASSIGN DIR TO DIRA
$OPTION 1
$EXECUTE LIBED
$EOJ
$$
```

Example 2 - Display Log

The following example produces a log of all library module names and external definitions on the file LOGLIST. In addition, statistics on available space in the library and directory are printed.

```
$JOB LOG OWNER
$CREATE LOGLIST
$ASSIGN LLO TO LOGLIST
$ASSIGN LIB TO LIBA
$ASSIGN DIR TO DIRA
$EXECUTE LIBED
LOG
$EOJ
$$
```

Example 3 - Update Library and Directories

The following example updates the standard MPX-32 system subroutine library and directory MPXLIB and MPXDIR. The user must have access rights to MPXLIB and MPXDIR. The modules contained in NEWOBJECT either replace an old module of the same name or are added as a new module. The module name UMOD1 is deleted. A log of all module names and external definitions and an analysis of remaining disc space in MPXLIB and MPXDIR is produced.

```
$JOB UPDATE OWNER
$ASSIGN LGO TO NEWOBJECT
$EXECUTE LIBED
LOG
DELETE UMOD1
$EOJ
$$
```

Example 4 - Insert Modules

In the following example, the binary output from a compilation is taken directly from the SGO file and used as input to LIBED. Modules on SGO replace modules of the same names in LIBB; modules that do not match existing names are added.

```
$JOB UPDATE OWNER
$NOTE UPDATE OWNER LIB FROM COMPILATION
$OPTION 5
$EXECUTE FORTRAN
(Source program)
$AS LIB TO LIBB
$AS DIR TO DIRB
$EXECUTE LIBED
$EOJ
$$
```

Example 5 - Produce Log and Exit

The following interactive example produces a log and returns the user to the TSM prompt automatically when execution is complete.

```
TSM>$ASSIGN LIB TO LIBC
TSM>$ASSIGN DIR TO DIRC
   (If these ASSIGN statements are not specified, the resulting log details contents of
   the system subroutine library.)
TSM>$LIBED
MPX-32 UTILITIES RELEASE x.x (LIBED Rx.x.x)
(C) COPYRIGHT 1983 GOULD INC., CSD, ALL RIGHTS RESERVED
LIB>LOG
LIB>EXIT
TSM>
```

Symbolic Debugger (SYMDB)

MPX-32 Utilities

CONTENTS

CONTENTS

CONTENTS

FIGURES

TABLES

SYMBOLIC DEBUGGER (SYMDB)

SECTION 1 - OVERVIEW

1.1 General Description

The Symbolic Debugger (SYMDB) utility assists in locating program errors in all non-base languages supported by MPX-32. However, the symbolic capabilities are available only with FORTRAN 77+ and assembly languages.

SYMDB provides a stable environment to verify the correct execution of a program or to locate any logic errors that may prevent proper execution.

SYMDB provides execution trace and breakpoint (trap) capabilities. The traps can be established so that they occur only when certain conditions are met and that a predetermined sequence of instructions is automatically executed upon occurrence. The contents of memory can be changed or displayed in several different formats, including integer (decimal, hex), real, double precision and character.

In FORTRAN 77+ and assembly language, addresses may be accessed through the use of local and global symbols defined in the source program. These symbols represent memory addresses, therefore the user does not need to know exact numerical addresses. If the location is associated with a symbol, it can be accessed by using that symbol name in the appropriate address parameter or command expression of a SYMDB command. If the location to be accessed is not identified by a symbol, the name of the previous local symbol plus the offset to the desired location can be entered in the command expression.

1.2 Local and Global Symbols

The symbols used by SYMDB are divided into two groups, local symbols and global symbols. Local symbols are those symbols defined within a specific source program, and accessed by that program. Global symbols are symbols defined within a specific source program, and can be referenced by other programs to provide interprogram linkage.

In FORTRAN 77+, the following groups are local symbols:

. Array Names
. Variable Names
. Statement Names
. Internal Functions
. Statement Functions
. Symbolic Constants*
. Statement Numbers**

* A symbolic constant can be used as a local symbol only if its value is less than -2^{15}, greater than 2^{15}, or it is passed as an actual argument in a subroutine or function call.

** FORTRAN-77+ (Releases 3.0 and later) assigns a statement number to each executable statement. The format for the statement number is S.x where x is the sequential location of the statement from the beginning of the respective program. Statement numbers are treated as local symbols by SYMDB and can be used as address parameters or command expressions in SYMDB commands. Assembly language statements embedded in a FORTRAN program may be accessed by using the nearest FORTRAN statement number plus the offset to the appropriate assembly language statement.

In FORTRAN, the following groups are global symbols:

. Program Names
. Subroutine Names
. Function Names
. Entry Points

NOTE: In FORTRAN 77+, subprogram names occur as both local symbols (the start of the subprogram) and global symbols (the primary entry point). The user must exercise caution when specifying these symbols in partially qualified contexts as they are usually different locations in memory.

In assembly language, local symbols are all symbols used as address labels. Global symbols must be defined as linkage symbols through the DEF directive in the assembly stage of the defining program, and referenced as linkage symbol through the EXT directive in the assembly stage of the referencing program. Symbolic information for variables contained in common is not generated.

1.3 Accessing Program Symbols

To enable access to debugger symbols, option 19 must be set with the TSM OPTION command when compiling and cataloging. Setting Option 19 for the compiler (ASSEMBLE, FORTRAN 77+) causes symbol data to be produced in the object code.

Setting Option 19 for CATALOG causes global and local symbol tables to be produced in the load module (only global symbols appear for program elements which are compiled without Option 19).

Note:

Overlays cannot be accessed by SYMDB, since the Cataloger does not generate local symbols for overlays even though option 19 is set.

1.4 Summary of SYMDB Capabilities

SYMDB is capable of:

- debugging interactively or in batch. In either environment, SYMDB commands control the execution of the program.
- accessing program locations (memory addresses) by using the symbols defined in the source program. Addresses are displayed as symbolic expressions.
- displaying data in several formats (floating point, ASCII, integer, or instruction mnemonic).
- executing program instructions one at a time and showing the result after each is executed.
- printing a debugging session log.
- accessing commands from a SYMDB command file to alleviate the need of entering each command individually during the debugging session.

1.5 Directive Summary

Directive	Description
A	Displays the address of the specified expression
ABSOLUTE	Sets absolute mode. All subsequent address expressions are evaluated and displayed as absolute addresses until relative mode is set via the RELATIVE directive.
B	Evaluates and displays the specified expression in binary format
BASE	Creates, deletes or modifies a user base
BREAK	Transfers control to a user task's break receiver
CC	Displays or modifies condition codes in the user task's program status doubleword (PSD)
CLEAR	Clears all user defined bases or deletes all traps
CM	Changes memory contents to the 32-bit value(s) specified beginning at the address specified
CR	Changes register contents to the 32-bit value(s) specified beginning at the register specified

Directive	Description
DA	Displays the contents of the memory range specified in ASCII format
DD	Displays the contents of the memory range specified in double precision floating-point format
DELETE	Deletes the specified trap
DETACH	Detaches SYMDB from the user task and transfers control to the task at the address specified or at the last address executed in the task
DF	Displays the contents of the memory range specified in single precision floating point format
DI	Displays the contents of the memory range specified in instruction mnemonic format (assembly language)
DN	Displays the contents of the memory range specified in decimal integer format
DNB	Displays the contents of the memory range specified in decimal integer byte format
DNH	Displays the contents of the memory range specified in decimal integer halfword format
DNW	Displays the contents of the memory range specified in decimal integer word format
DUMP	Dumps the content of the memory range specified, the task's PSD and the general purpose registers to the line printer (interactive mode) or to the lfc #OT (batch mode) in a side-by-side hexadecimal and ASCII format
E	Evaluates and displays the result of the expression specified in single precision floating point format
END	Terminates a trap list and returns control to the lfc #IN
EXIT or X	Terminates SYMDB and the user task
FILE	Passes control to the directive file specified to read and execute the directives in the file, then return control to the lfc #IN
FORMAT	Sets the default radix to either decimal or hexadecimal for undesignated values in expressions
GO	Begins execution of the user's task at the address specified or at the current program counter value
IF	Establishes conditional trap list execution
LIST	Displays the trap list for the specified trap

Directive	Description
LOG	Writes the temporary log file to the line printer (interactive mode only, not available in batch mode)
MODE	Sets log/no log file, extended/no extended memory access, and FORTRAN/NOFORTRAN label field format
MSG	Designates a comment line
N	Evaluates and displays the result of the expression specified in signed decimal format
PGM	Establishes the program name specified as the default for local symbol searches or if no program name is specified, defaults to global symbol searches
RELATIVE	Sets relative mode and optionally establishes a new relative base or program name
REVIEW	Writes the temporary log file (one screen at a time) to the lfc #IN (interactive mode only, not available in batch mode)
RUN	Sets run mode (as opposed to single-step) for tracing or tracking. A full screen of program instructions is displayed before prompting for continuation or termination of the trace or track.
SET	Sets a trap at the word address specified and prompts for a trap list directive
SHOW	Displays trap addresses, base names and values, option settings and/or symbols
SNAP	Displays the contents of the memory range specified in side-by-side hexadecimal and ASCII format
STATUS	Displays the status of the user PSD and general purpose registers at the current address
STEP	Sets single-step mode for subsequent TRACE or TRACK directives. One program instruction is displayed before prompting for continuation or termination of the trace or track.
TIME	Displays the current date and time
TRACE	Transfers control to the user task and displays each instruction after it is executed
TRACK	Transfers control to the user task and displays each branch instruction after it is executed
WATCH	Transfers control to the user task and reports any erroneous branches into memory (no instructions are displayed)

Directive	Description
X	Evaluates and displays the result of the expression specified in hexadecimal format. If no expression is specified, X is interpreted as the EXIT directive.

SECTION 2 - USAGE

2.1 Accessing SYMDB

2.1.1 Accessing SYMDB in TSM

SYMDB can be accessed in interactive modes by placing a call to the M.DEBUG system service in the program to be debugged, by pressing the break key while the task is executing, or by using the following job control statement:

```
$DEBUG loadmod
```

The task is activated (JCL statement only) and SYMDB is attached as a co-resident module. Control is passed to SYMDB instead of to the task's transfer address when activation completes.

When accessing SYMDB interactively, or when logical file code #IN is assigned to a user terminal, SYMDB prints the current date and time, a status report, and prompts for an immediate directive with a dot (.):

```
MPX-32 UTILITIES 3.0 SYMDB R3.X.X.X mm/dd/yy TIME TASKNAME=xxxxx
(C) COPYRIGHT 1980 GOULD INC., COMPUTER SYSTEMS DIVISION, ALL RIGHTS RESERVED
...Status Report...
.
```

To access SYMDB for a task that was activated by a TSM RUN directive, press the break key. TSM responds:

```
** BREAK ** ON taskname AT location CPU TIME = n SEC.
CONTINUE, ABORT, DEBUG, OR HOLD?
```

Enter D to access SYMDB. SYMDB is then loaded into the task's address space and debugging begins at that point where the task was interrupted. The date, time, status report, and prompt are displayed as described above. The context of the task (PSD and general purpose registers) prior to the interrupt is retained.

Note: Tasks which configure their own break receiver disable this capability.

2.1.2 Accessing SYMDB via the Batch Stream

SYMDB can be accessed via the Batch Stream by entering the $DEBUG command in the job control. SYMDB cannot be accessed until the program has been assembled/compiled and cataloged. Therefore, the $DEBUG command must follow the $ASSEMBLE/ $FORTRAN and $CATALOG portions of the job control if the program is to be assembled/compiled, cataloged and debugged in one job stream. Otherwise, separate job control can be set up for each phase of the program development.

All the file assignments for the program to be debugged must be made prior to debugging the program. These file assignments can be made by establishing defaults through the use of the Cataloger's ASSIGN directives when the program is cataloged. If no defaults were assigned, the $ASSIGN commands must be entered in the job control preceding the $DEBUG command.

Default file assignments are made for all necessary SYMDB files (refer to Section 2.3.6 LFC Summary). Only SYMDB input (#IN) and output (#OT) files may be changed from their default.

SYMDB commands to be executed are entered following the $DEBUG command in the job control stream, unless the LFC #IN is assigned to another resource.

Note: If SYMDB commands and the data for the task being debugged are both in the job stream, the order of the commands and the data must match the order of reads that occur during exection.

Example 1

This is an example of job control to assemble, catalog and debug a program.

```
$JOB DBG.TST username
$OPTION 19
$EXECUTE ASSEMBLE
    .
    .
    .
source program
    .
    .
    .
$OPTION 19
$EXECUTE CATALOG
    .
    .
    .
Catalog directives
    .
    .
    .
$ASSIGN1 IN=infile
$ASSIGN2 OUT=SLO,1000
$DEBUG DBG.TST
command1
command2
    .
    .
    .
commandn
$EOJ
$$
```

infile	specifies the name of the permanent disc file which contains the input for the user program
SLO	specifies the user program output is to be written to the system listed output file (spooled output) which is then written to the line printer.

Example 2

This is an example job control for debugging only. This example assumes that the program has been assembled/compiled, and cataloged and that no default program file assignments were made during Cataloging.

```
$JOB DBG.TST username
$ASSIGN1 IN=infile
$ASSIGN2 OUT=SLO,1000
$DEBUG DBG.TST
command1
command2
      .
      .
      .
commandn
$EOJ
$$
```

infile	specifies the name of the permanent disc file which contains the input for the user program
SLO	specifies the user program output is to be written to the system listed output file (spooled output) which is then written to the line printer.

2.2 Using M.DEBUG

To access SYMDB with a call to a system service, code a call to the M.DEBUG system service in the task itself. The call loads SYMDB into the task's address space and passes control to SYMDB. The date, time, status report, and prompt are displayed as described above.

If the task is executing in the interactive or batch environments, the default assignments cause SYMDB I/O to occur to the user's terminal or SYC/SLO respectively. For independent tasks, the user must provide assignments for #IN and #OUT either at CATALOG time or prior to the TSM ACTIVATE. Usually, these assignments are to a specific terminal device (e.g. TY7EA0) or the operator's console. The specified device must be available for assignment (not logged in or in TSM WAIT) when the M.DEBUG call occurs.

Example

```
TSM>ASSIGN #IN TO DEV=TY7EA0
TSM>ASSIGN #OT TO LFC=#IN
TSM>ACTIVATE MYTASK
TASK NUMBER IS 21004EC1
TSM>WAIT
```

2.3 Logical File Code Assignments

When SYMDB gains control, it dynamically allocates logical file codes (LFCs) for input and output and creates a log file to record SYMDB session. The following sections describe the logical file code assignments used by SYMDB.

2.3.1 Source Input (#IN and #03)

SYMDB directive input is assigned to logical file code #IN.

#IN and #03 Default and Optional Assignments

In interactive mode, the default assignment for #IN is to logical file code UT.

In batch mode, the default assignment for #IN is to the System Control file (SYC).

SYMDB input can also be from a directive file by specifying the FILE directive. If the FILE directive is used, SYMDB uses the pathname specified in the directive to make the assignment to logical file code #03. After the directive file is read, input reverts to the file or device assigned to logical file code #IN.

There are two optional assignments for #IN in the batch mode:

```
    $AS  #IN  TO  {pathname   }
                  {DEV=devmnc }
```

pathname	is the pathname of a directive input file
devmnc	is the device mnemonic of a device containing directive input

There is no optional assignment to #03.

2.3.2 Listed Output (#OT)

Output generated by SYMDB directives is assigned to logical file code #OT.

#OT Default and Optional Assignments

In interactive mode, the default assignment for #OT is to logical file code UT.

In batch mode, the entire SYMDB session is output to the System Listed Output file (SLO).

There are two optional assignments for #OT in the batch mode:

```
    $AS  #OT  TO  {pathname   }
                  {DEV=devmnc }
```

pathname	is the pathname of an output file
devmnc	is a device mnemonic of a device to contain output

2.3.3 Temporary Log File (#01)

SYMDB uses logical file code #01 to dynamically allocate a temporary log file. The log file stores approximately 300 screens of terminal I/O or the batch equivalent. The REVIEW command replays the LOG file data to the #OT LFC. This file is copied to an output file (see logical file code #02) when a LOG directive is issued in the interactive mode. LOG is treated as a comment in batch because the SYMDB session is already output to an SLO file in batch mode.

2.3.4 Log Output (#02)

Alternate output assigned to logical file code #02. This is used to record the output from the LOG and DUMP directives in the interactive mode.

#02 Default and Optional Assignments

The default assignment for #02 is to the System Listed Output file (SLO).

There is no optional assignment for #02.

2.3.5 Symbol Table (#SM)

The load module is assigned to logical file code #SM. The load module is accessed to obtain the symbol table.

#SM Default and Optional Assignments

The default assignment for #SM to the load module is made only if Option 19 was set for the Cataloger when the load module was created.

There is no optional assignment to #SM.

2.3.6 LFC Summary

The following is a table of the LFCs used by SYMDB and their default and optional assignments.

Table 2-1
SYMDB LFC Summary

LFC	Default Assignment	Optional Assignment
#IN (interactive)	LFC=UT	pathname DEV=devmnc
#IN (batch)	SYC	pathname DEV=devmnc
#01	none	none
#02 (interactive)	SLO	none
#02 (batch)	N/A	none
#03	pathname	none
#OT (interactive)	LFC=UT	pathname DEV=devmnc
#OT (batch)	SLO	pathname DEV=devmnc
#SM	Load Module	none

2.4 Exiting SYMDB

To exit SYMDB in the batch and interactive modes, specify the EXIT directive.

2.5 Attaching SYMDB to a User Task

SYMDB functions as a co-resident module of the user task being debugged; the user neither catalogs SYMDB as part of the task nor provides any memory area for SYMDB when the task is cataloged.

The TSM DEBUG command and the M.DEBUG system service (H.REXS,29, SVC 1,X'63') attach SYMDB to the calling task as follows:

1. SYMDB is loaded at the beginning of the map block below the user task's pure code and data section (CSECT) and/or common areas as illustrated in Figure 2-1. The lower address of the user's CSECT, if any, is decreased by 8KW, the size of SYMDB.

2. The area T.CONTXT is initialized in the calling task's TSA. T.CONTXT contains eight words for the user register contents at the point of call and two words for the user PSD at the point of call. T.CONTXT is used by SYMDB to determine the last known context of the user task upon entry to any SYMDB entry point.

3. Control is passed to SYMDB's Entry Point 1 (startup entry point). Any task interrupt levels active at this point remain in effect. They are analyzed by SYMDB and displayed in a status report.

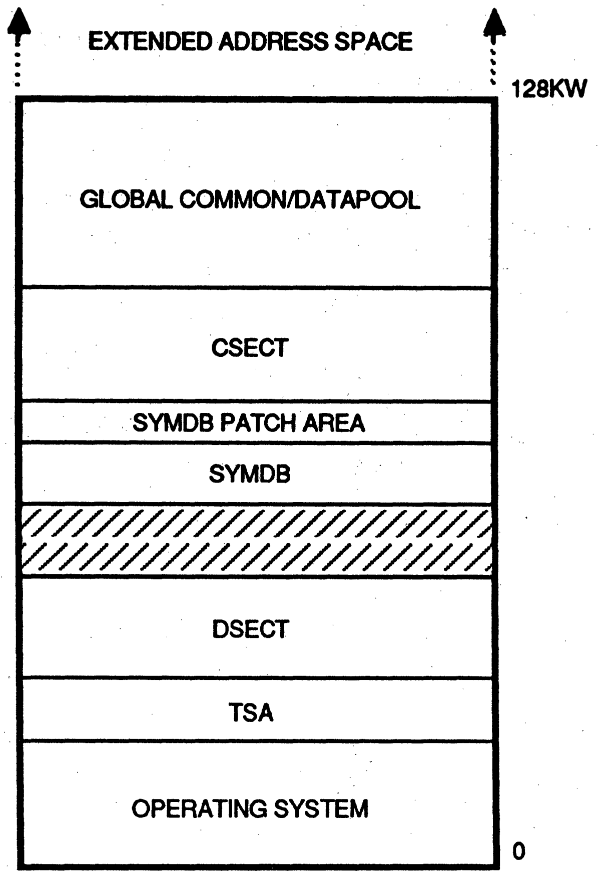
The combination of SYMDB and the user task is a single task, with a single TSA and a single dispatch queue entry.

When a task is activated from TSM and the break key is pressed while the task is processing, TSM provides the option to attach SYMDB. If SYMDB is attached, the task context is saved as described in step two.

If the Debugger is currently attached to a user task, and the task executes M.DEBUG,

- and the user is in GO mode, a break interrupt will occur.

- and the user is in TRACE mode, a SVC 1,X'63' will be echoed and execution will be transferred to the next instruction in the task.



87D4108

Figure 2-1. SYMDB Memory Map

2.6 Input/Output

2.6.1 Terminal I/O

When SYMDB is attached to a task, the screen size is obtained from the Unit Definition Table (UDT) for the terminal device assigned to logical file code UT. This is the screen size defined at SYSGEN. The screen size is used by SYMDB in the interactive mode and has no meaning in batch mode.

The screen size is used to calculate the size of a temporary disc log file large enough to contain approximately 300 full screens of terminal I/O. The log file is manipulated by the LOG and REVIEW commands. It contains a record of the most recent screens of I/O to the user's terminal and provides a complete audit trail of the debugging session. A warning is displayed ten screens before the end of the log file space.

The screen width (number of characters per line) is used to calculate how many words per line will fit into displays such as SNAPs. The minimum allowable screen width for debugging is 72 characters and the maximum is 132 characters, the width of a System Listed Output file.

The screen height (number of lines per screen) is used by SYMDB to enable or disable full-screen logic.

Full-screen logic enables SYMDB to pause when a full screen of lines is written to the terminal with no intervening terminal input. This prevents long displays from running off the top of the screen before they can be read. A SYSGEN-defined height of zero lines signifies that the terminal is a hardcopy device and disables full-screen logic. As a result, long displays cannot be terminated prematurely.

When at the end of a full screen of consecutive output, SYMDB prompts for a carriage return to continue displaying output. If anything other than a carriage return is entered, the current directive is terminated and SYMDB reads the next directive.

Terminal input and output are labeled with prefix characters (prompts) that indicate, both on the terminal and on the log file, user-issued directives and SYMDB responses. The prompt for an input directive from the terminal is one or two dots. A single prompt (.) signifies a request for an immediate directive; a double prompt (..) signifies a request for a deferred directive. In all other cases, the prefix characters are pseudoprompts in that they are only labels for terminal output lines. Table 2-2 identifies the various combinations of prompt characters. In batch mode, only the pseudoprompts are output.

Table 2-2
SYMDB Prompts and Labels

Prompt	Significance
.	Requests an immediate directive on logical file code #IN
..	Requests a deferred directive on logical file code #IN SYMDB pseudoprompts:
>	Immediate directive from logical file code #03 (FILE directive)
>>	Deferred directive from logical file code #03 (FILE directive)
!	Immediate directive from a trap list
!!	Deferred directive from a trap list
:	Follows any of the above prompts and pseudoprompts; labels output resulting from a directive

2.6.2 Command Files

A command file is a permanent disc file that contains SYMDB directives. Directives are in the form of 72-byte logical records. The Text Editor (EDIT) utility can be used to create a SYMDB command file. The STORE (not SAVE) directive should be used to write a command file. A command file is accessed by using the FILE directive.

The command directive can contain any number of SYMDB directives and all directives can be used in the command file except the FILE directive.

2.7 Control Transfers

During a debugging session, control can pass back and forth between SYMDB and the user task any number of times. Because SYMDB and the user task are parts of a single task, and it is important for the scheduler to know which part is executing at any given time, all such control transfers take place through scheduler (H.EXEC) service calls.

Each time SYMDB gains control, T.CONTXT in the TSA contains the user task's context as of its last executed instruction. T.REGS and T.REGP indicate the current task interrupt push-down level in effect for the user task (i.e., the stack is not pushed an additional level upon entry to SYMDB). SYMDB analyzes the TSA and DQE of the task in a status report to the terminal, indicating the user context (PSD and registers) for each active task interrupt level.

When SYMDB gains control, it runs privileged regardless of the state of the user task. When SYMDB passes control to the user task, H.EXEC restores the user task's state.

The following is a summary of the control transfers which take place between SYMDB and the user task. SYMDB always gains control as a result of system service M.DEBUG, whether it is called by the task activation service, by the task itself, or by TSM at the request of the terminal user after the task is running.

The user task gains control:

- when SYMDB executes a GO directive.
- when SYMDB executes a BREAK directive.
- for the execution of a user instruction during a TRACE, TRACK, or WATCH directive.
- when SYMDB executes a DETACH directive.

SYMDB gains control:

- when the user task executes SYMDB trap instruction.
- when IOCS recognizes a break from the user's terminal.
- when the user task calls the M.BRKXIT service after SYMDB executes a BREAK directive.
- after the execution of a user instruction during a TRACE, TRACK, or WATCH directive.
- when the user task would normally be aborted by MPX-32.
- when the user task executes an M.EXIT system service.

If SYMDB gains control on a trap instruction (SVC 1,X'66') which was user-coded (as opposed to one which was set by the SET directive), SYMDB interprets it as a break from the terminal instead of a trap.

2.8 Break Handling

A break occurs when:

- the terminal break key is pressed.
- any task uses the M.INT system service to simulate an interrupt and enter a break receiver.

SYMDB analyzes the user context in a status report when the break handling entry point is entered. It prompts for the next directive. If a command file is being used, command file processing terminates.

SYMDB recognizes breaks only when:

- it executes a GO directive and has not yet prompted for the next directive, i.e., the user task has control.
- it is executing a WATCH directive.

Breaks for the task are ignored if either of these conditions is not met.

If executing GO or WATCH as described above, the execution of a trap instruction that was not set by the SET directive appears to SYMDB as a break which occurred between the execution of the trap instruction and the next user instruction.

Breaks in batch mode are generated by an OPCOM BREAK directive or by an SVC 1,X'66' instruction coded as part of a task.

2.9 Setting the Default for Symbolic References

There are various symbol tables in which SYMDB searches for symbol names. These are the program name table, the global symbol table and a number of local symbol tables (one for each of the program names in the program name table). A default may be set to either the global or one of the local symbol tables through the use of the PGM directive. The default specifies that the symbol table to which the default is set will be the first table searched for the symbol name specified in a SYMDB command.

If the PGM directive is entered without an argument, the default is set to the global symbol table (this is the default condition when SYMDB is accessed). Global symbols may then be accessed by entering the symbol name in a command expression. Local symbols may be accessed in this default condition only if they are entered in a full pathname. A full pathname consists of the program name which defines the local symbol, a back slash (\ ) character and the local symbol name (progname \ locsym).

The default may be changed to one of the local symbol tables by entering the PGM directive and the name of the program which defines the desired local symbols. Local symbols defined in the program specified in the PGM directive may then be entered without specifying the program name or backslash (\ ) character. Local symbols defined in another program must be entered in the full pathname format. Global symbols may be entered in this default condition, but if a global symbol, default local symbol and/or program have the same name, the local symbol will be accessed. Therefore, if the global symbol was desired, the default must be returned to the global symbol table (enter PGM with no argument). If a program name was desired, it must be preceded with the pound sign (#).

Examples

```
    .PGM D.EXMPL
```

allows SYMDB to access all local symbols defined in the program D.EXMPL.

```
    .PGM
```

allows SYMDB to access all global symbols defined in the program to be debugged (local symbols are no longer accessible without entering a full pathname).

Note: In FORTRAN, #SUBR is the name of a subroutine and \SUBR is the location of the entry point. These locations are usually separated by a few words of data and do not refer to the same location in memory. The user must ensure that the wrong location is not implied when fully qualified symbols are not specified.

2.10 Program Execution

SYMDB has several directives for transferring control to the user program to begin program execution. These directives are the GO, TRACE, TRACK and WATCH directives.

After SYMDB is accessed, the execution of the program to be debugged can be started by entering the GO directive. This directive can also be used to continue execution after the program has been stopped. The GO directive has two optional parameters, the start address and the stop address. If the start address is not specified, SYMDB uses the current PSW as the start address. If the stop address is not specified, program execution continues until the program completes or until an abort or trap is encountered. If no parameters are specified, the program will execute as if SYMDB was not attached.

Program execution may also be started with the TRACE directive. The TRACE directive is used to single step through program execution. This directive has two optional parameters, the start address and the stop address. If a start address is specified, SYMDB starts execution at that address and displays the instruction located at that address. If no start address is specified, SYMDB starts execution at the address specified in the current PSW and displays the instruction located at that address. After each instruction is executed and displayed, a carriage return (cr) must be entered to execute and display the next instruction. The single step trace continues in this manner until reaching the stop address (if specified) or the end of the program (if no stop address is specified). The trace may be stopped at any time by entering any character other than a carriage return (cr) following the display of an instruction.

Note: When executing a program via the TRACE directive, all traps are ignored.

There are two other directives that can be used to start program execution, the TRACK and WATCH directives. Both of these directives are functionally like the TRACE directive. The TRACK directive differs from the TRACE directive in that it writes only branch instructions and their results to the lfc #OT. The WATCH directive differs from the TRACE directive in that it does not write any instructions or results.

The WATCH directive causes SYMDB to monitor program execution to detect erroneous branches into memory that is not within the program's address range. If a branching address error occurs, an error message will be written to the lfc #OT. There will be no other output during program execution in WATCH mode.

2.11 Traps and Trap Lists

Program execution may be stopped by setting traps at locations within the program. The SET directive is used to place traps at the desired locations. The SET directive requires one parameter, the trap (stop) address. When program execution is started by the GO directive, execution continues until a trap is encountered (or until the program finishes processing). If during execution a trap is encountered, execution of the program will stop at the trap address. This allows the user to execute sections of code that are known to be correct and stop at an address where errors are suspected. When execution is stopped because a trap was encountered, SYMDB will execute the directives in the trap list for the trap specified in the SET directive.

When the SET directive is entered, SYMDB defers the execution of subsequent directives and stores them in a trap list for the trap specified in the SET directive. A trap list can contain any number of SYMDB directives. The directives will be executed in the order they were entered in the trap list. All SYMDB directives can be entered in a trap list, except the LOG and REVIEW directives.

Each directive entered in a trap list, except the CLEAR, FORMAT, MODE and SHOW directives, is checked for validity before being stored. If a directive contains an error, the directive is not entered in the trap list and an appropriate error message is written to the lfc #OT (refer to Section 6.3 Trap Error Messages). Following the error message, the user may re-enter the directive or enter another directive as desired. The directives CLEAR, FORMAT, MODE, and SHOW will be validated only when they are to be executed.

A trap list directive may contain a user base parameter which has not yet been defined. This is not considered an error in a trap list directive. Therefore, care should be taken to define all user bases either before building the trap list or before executing the trap which contains an undefined user base reference.

Trap lists are ended by entering any one of the trap list terminator directives. If the trap list contains nested traps, each trap list terminator corresponds to the trap list following the most recent unterminated SET directive. Valid trap list terminators are the BREAK, END, EXIT, FILE, GO, TRACE, TRACK and WATCH directives.

The LIST command can be used to display the contents of a trap list.

Each trap maintains an attribute called COUNT. This special symbol is automatically incremented each time the trap is executed. The current value of COUNT is displayed if the trap is LISTed.

Traps can be established to occur only when certain conditions are met. This is done by using the IF directive as the FIRST directive of the trap list. When the trap is executed, the specified conditions are evaluated and if met, the trap is recognized and the trap list (if any) is executed.

Example

```
. SET S.23
..IF C(J)>n'300'&COUNT>=100
..DN J
..DF X
..CM Y = E'37.91'
..GO S.40
.
```

In the above example, a trap is set at the beginning of FORTRAN statement S.23. This trap will be recognized when the contents of variable J has a value greater than 300 (decimal) and the COUNT for this trap is greater than or equal to X'100'. Otherwise, COUNT is incremented and execution of the program continues. When the trap is recognized, the values of symbols J and X are displayed. The value of symbol Y is changed and execution is resumed at FORTRAN statement S.40.

2.12 Nested Traps

Traps can be set within a trap list. If a SET directive is entered in a trap list, a nested trap list is built within the original trap list. When the original trap is encountered during program execution, the SET directive in the trap list being executed will cause a second trap to be set at the address specified in the nested SET directive. Any number of trap lists may be nested within a trap list. Each nested trap will be set only after the trap list it is nested within is executed.

Each nested trap within a trap list must have a corresponding trap list terminator. Each trap list terminator corresponds to the trap list following the most recent unterminated SET directive.

The size of the trap table is a 16 by 20 word two-dimensional array. If single traps are set, each will take 16 words, giving a maximum of 20 traps.

If 1 trap contains more than 15 nested traps, the trap blocks will be linked together, giving a maximum of 18 remaining traps, each containing less than 15 nested traps.

Example

```
.   SET trap1     ┐    - trap is set at address specified by trap1
..  directive1-1  │    - directive1-1 is stored in the trap list for trap1
..  directive1-2  │    - directive1-2 is stored in the trap list for trap1
..  SET trap2    ┐│    - trap2 will be set when trap1 is encountered
..  directive2-1 ││    - directive2-1 is stored in the trap list for trap2
..  SET trap3   ┐││    - trap3 will be set when trap2 is encountered
..  directive3-1│││    - directive3-1 is stored in the trap list for trap3
..  terminator3┘││    - terminator3 is stored in and terminates the trap list for trap3
..  SET trap4   ┐││    - trap4 will be set when trap2 is encountered
..  directive4-1│││    - directive4-1 is stored in the trap list for trap4
..  terminator4┘││    - terminator4 is stored in and terminates the trap list for trap4
..  terminator2 ┘│    - terminator2 is stored in and terminates the trap list for trap2
..  terminator1  ┘    - terminator1 is stored in and terminates the trap list for trap1
```

2.13 Examining Memory and Registers

SYMDB provides directives which allow the user to examine the contents of memory or registers.

The directives to display memory are as follows:

Directive	Description
DA (Display ASCII)	Displays the contents of memory in ASCII format. For FORTRAN 77+ character strings, the decimal length of the symbol is used.
DD (Display Double Precision)	Displays the contents of memory in double precision floating point format.
DF (Display Floating Point)	Displays the contents of memory in single precision floating point format.
DI (Display Instruction)	Displays the contents of memory in instruction format.

Directive	Description
DN (Display Numeric)	Displays the contents of memory in a decimal integer format (for FORTRAN 77+ symbols, the data size is selected from the symbol table entry).
DNB (Display Numeric Byte)	Displays the contents of memory in a decimal integer byte format.
DNH (Display Numeric Halfword)	Displays the contents of memory in a decimal integer halfword format.
DNW (Display Numeric Word)	Displays the contents of memory in a decimal integer word format.
DUMP	Dumps the contents of memory to the line printer in a side-by-side hexadecimal and ASCII format.
SNAP	Displays the contents of memory in a side-by-side hexadecimal and ASCII format.

The eight general purpose registers can be displayed by entering the STATUS directive. This directive displays the contents of all general purpose registers in a side-by-side hexadecimal and ASCII format. The STATUS directive has no parameters.

2.14 Modifying Memory and Registers

SYMDB provides directives which allow the user to change memory or register values.

The contents of memory can be changed by entering the CM (Change Memory) directive. This directive requires two parameters separated by an equal sign (=). The first parameter is the starting address of the address values to be changed. The second parameter, which may be a list of values separated by commas, is the data to be entered in memory starting at the address specified in the first parameter. If there is only one entry in the second parameter (the data list), only the address specified will be changed. Two successive commas in the data list specify that the corresponding address word value will remain unchanged.

Example

```
.CM 100=1,2,,4
```

causes the values 1, 2, and 4 to replace the contents of addresses 100, 104 and 10C respectively. Address 108 remains unchanged.

The contents of registers can be changed by entering the CR (Change Register) directive. This directive requires two parameters separated by an equal sign (=). The first parameter is the starting register (R0-R7) of the register(s) to be changed. The second parameter, which may be a list of values separated by commas, is the data to be entered in the register(s) starting with the register specified in the first parameter. If there is only one entry in the second parameter (the data list), only the register specified will be changed. Two successive commas in the data list specify that the corresponding register will remain unchanged.

Example

```
.CR R1=1,2,,4
```

causes the values 1, 2, and 4 to replace the contents of registers R1, R2 and R4 respectively. Register R3 remains unchanged.

2.15 Selecting the Input Radix

The input radix can be selected using the FORMAT directive. The default radix is hexadecimal. To change the default radix to decimal, enter FORMAT N. To change the default radix back to hexadecimal, enter FORMAT X. The SHOW OPTIONS directive may be entered to display the current default input radix.

2.16 Establishing User Bases

To establish a base at the beginning of a data structure or a subroutine that will be referenced frequently during the debugging process, enter the BASE directive. This directive requires two parameters, the base name and the expression whose value is assigned to the base. Once a base is defined, it may be used as a term in an expression in SYMDB directives. To change the value of a base, enter the BASE directive. To remove all user defined bases from SYMDB's base table, enter the CLEAR BASES directive.

2.17 Selecting Relative or Absolute Addressing

To establish a relative reference point during debugging, use the RELATIVE directive. This directive uses one optional parameter, a base name or program name to be the relative reference point. If the parameter is omitted, SYMDB re-establishes the last relative name used in the program. The ABSOLUTE directive is used to make all subsequent address expressions absolute (relative to location 0 of the task's logical address space). The SHOW OPTIONS directive may be entered to display the current addressing mode (relative or absolute).

2.18 Selecting Log/No Log File

A temporary log file is allocated by default for SYMDB when SYMDB is accessed in interactive mode. The log file is used to store the last 300 screens of the directives and results of the debugging session until a LOG or REVIEW directive is entered. These directives display the log file to the line printer (LOG directive) or the lfc #OT (REVIEW directive) and then clear the log file. All subsequent directives will be entered in the log file until another LOG or REVIEW directive is entered or until the debugging session is ended.

The log file will not be maintained after the user enters the MODE NOLOG directive. The log file can be maintained again by entering the MODE LOG directive thus all subsequent directives will then be stored. The SHOW OPTIONS directive may be entered to display whether or not a log file is being maintained.

2.19 Selecting Label Field Format

The addresses which are displayed in the label field of all SYMDB directive results can be displayed in two formats. The first format is oriented to FORTRAN programs and displays the address as the program name, the symbol name, and the offset. Program name specifies the program in which the address to be displayed is located. Symbol name specifies the symbol name within the specified program which has the closest value less than or equal to the address to be displayed. Offset specifies the positive difference between the symbol name's value and the address to be displayed.

The second format is oriented to non-FORTRAN programs. This format displays the address as the program name plus the offset. Program name specifies the program in which the address to be displayed is located. Offset specifies the positive difference between the symbol name's value and the address to be displayed.

Entering the MODE FORTRAN directive causes SYMDB to select the FORTRAN oriented format. Entering the MODE NOFORTRAN directive causes SYMDB to select the non-FORTRAN oriented format. Both assembly language and FORTRAN programs may use either addressing format. The default setting of SYMDB is the non-FORTRAN mode. The SHOW OPTIONS directive may be entered to display the current label field format.

2.20 Selecting Extended Memory Access

If the program to be debugged uses extended memory addressing, SYMDB can access this portion of memory only when the extended memory bit is set in the program status word (PSW). The extended memory bit in the PSW can be set by executing the SEA (set extended addressing) instruction in the program being debugged or by the MODE EXTENDED directive.

The MODE EXTENDED directive allows the user access to extended memory without having to execute the SEA instruction within the program. This allows the user to examine or change extended memory at any time in the debugging session. If the MODE EXTENDED directive is not entered, the user would have to trace through the program location which contains the SEA instruction (setting the PSW extended memory bit) before attempting to access extended memory via a SYMDB directive.

If the program to be debugged does not require extended memory access, the MODE NOEXTENDED directive will inhibit user access to extended memory. This is the default condition in SYMDB. The SHOW OPTIONS directive may be entered to display the current extended memory access mode.

2.21 SYMDB Directive Expressions

Many SYMDB directives have required or optional parameters specified as expressions. SYMDB expressions are specified as arithmetic, logical, relational or single term expressions. The expressions are evaluated as 32-bit integer expressions. Each expression contains one or more valid terms. Valid terms used in expressions are integers (in the default input radix), constants, register and memory contents, base names, symbolic references, COUNT and period (.).

The rules for entering expressions are

. Operators are binary (arithmetic, logical or relational), requiring two operands.

. Expressions are evaluated left to right.

. Parentheses override left to right evaluation.

. Expressions are evaluated as 32-bit integer operations.

. Expressions contain one or more valid terms (a maximum of eight characters per symbolic name).

2.21.1 Arithmetic Expressions

The following is a description of valid arithmetic expressions (X and Y specify any valid term).

Expressions	Type	Description
X+Y	Addition	X is added to Y, overflow is ignored
X-Y	Subtraction	Y is subtracted from X, overflow is ignored
X*Y	Multiplication	X is multiplied by Y, overflow is ignored
X/Y	Division	X is divided by Y, remainder is ignored

2.21.2 Logical Expressions

The following is a description of valid logical expressions (X and Y specify any valid term).

Expression	Type	Description
X ^ Y	Logical Shift	X is shifted Y bits to the left if Y is positive or to the right if Y is negative
X&Y	Logical AND	X is logically ANDed with Y
X!Y	Logical OR	X is logically ORed with Y
X@Y	Exclusive OR	X is exclusively ORed with Y

2.21.3 Relational Expressions

The following is a description of valid relational expressions (X and Y specify any valid term). Comparisons are arithmetic, i.e., the 32-bit values being compared are assumed to be the signed numbers.

Expression	Type	Description
X=Y	Equal	if X is equal to Y, evaluated as TRUE or 1 (otherwise, FALSE or 0)
X < > Y	Not Equal	if X is not equal to Y, evaluated as TRUE or 1 (otherwise, FALSE or 0).
X > Y	Greater	if X is greater than Y, evaluated as TRUE or 1 (otherwise, FALSE or 0)
X < Y	Less	If X is less than Y, evaluated as TRUE or 1 (otherwise, FALSE or 0)
X >= Y	Greater or Equal	if X is greater than or equal to Y, evaluated as TRUE or 1 (otherwise, FALSE or 0)
X <= Y	Less or Equal	if X is less than or equal to Y, evaluated as TRUE or 1 (otherwise, FALSE or 0)

Note:

Single terms may be entered as expressions, and their value used as the expression result.

2.22 Terms used in SYMDB Expressions

SYMDB expressions contain one or more valid terms. The valid terms are integers, constants, register and memory contents, base names, symbolic references, COUNT and period (.).

2.22.1 Integers

Integers used as terms are entered in the default input radix. If the input radix is hexadecimal, the first digit of the integer must be 0 through 9. Therefore, if a hexadecimal integer beginning with A through F is to be entered, it must be preceded by a leading zero.

If the input radix is hexadecimal, any number of digits can be entered as a hexadecimal integer but only the last eight digits (the least significant digits) will be accepted by SYMDB as the integer value.

If the input radix is decimal, one to ten digits can be entered as the decimal integer. If more than ten digits are entered, SYMDB expects the eleventh digit to be a valid operator, and displays the message:

    MISSING OPERATOR

to the lfc #OT and reissues a prompt for another directive. The user should re-enter the directive with a one to ten digit decimal integer or enter another directive.

2.22.2 Constants

The following are six types of constants used as terms in SYMDB expressions:

- Hexadecimal Constant - A hexadecimal constant is a string of hexadecimal digits enclosed in apostrophes and preceded by the letter X (e.g., X '1EC'). If the default input radix is hexadecimal, the letter X and the apostrophes are unnecessary. If the X and apostrophes are omitted and the hexadecimal value begins with A-F, a leading zero must precede the hexadecimal constant (synonymous with hexadecimal integer). In either format, any number of digits can be entered, but only the last eight digits (the least significant) will be used as the constant.

- Decimal Constant - A decimal constant is a string of one to ten decimal digits enclosed in apostrophes and preceded by the letter N (e.g., N '193'). If the default input radix is decimal, the letter N is unnecessary (synonymous with decimal integer). If more than ten digits are entered in a decimal constant string, SYMDB expects the eleventh digit to be a valid operator, and displays the message:

  MISSING OPERATOR

  to the lfc #OT and reissues a prompt for another directive. The user should re-enter the directive with a one to ten digit decimal constant or issue another directive.

- Binary Constant - A binary constant is a string of one to 32 ones and zeros enclosed in apostrophes and preceded by the letter B (e.g., B'101011'). If fewer than 32 digits are entered, leading binary zeros are added to produce a 32-bit value.

- Floating Point Constant - A floating point constant is a string of one to 21 decimal digits enclosed in apostrophes and preceded by the letter E. The floating point string is entered in one of three formats, a single precision value without an exponent, a single precision value with an exponent (denoted by the letter E) or a double precision value with an exponent (denoted by the letter D). The mantissa and the exponent can optionally be designated as positive (+) or negative (-).

  Examples:

  A single precision floating point constant without an exponent.

  E'0.999'

  A positive single precision floating point constant with a negative exponent.

  E'+100.32E-10'

  A negative double precision floating point constant with an exponent

  E'-100.32D10'

- C-Character Constant - A C-character constant is a string of one to four characters enclosed in apostrophes and preceded by the letter C (e.g., C'A1?'). C-character constants are left justified and trailing blanks are added to produce a 32-bit value, if fewer than four characters are entered.

- G-Character Constant - A G-character constant is a string of one to four characters enclosed in apostrophes and preceded by the letter G (e.g., G'A1?'). G-character constants are right-justified and leading binary zeros are added to produce a 32-bit value, if fewer than four characters are entered.

2.22.3 Register and Memory Contents

The contents of registers and memory are used as terms in expressions by specifying the register name or the memory address of the contents to be used.

Register contents are used by entering any of the eight general purpose registers in the form Rn (n specifies a register number 0 through 7).

Memory contents are used by entering the address of the contents to be used in one of the following formats:

```
C (address)
C (address ± hex)
C (address ± dec)
C (hex)
C (dec)
```

C	specifies the contents of the term enclosed in parentheses is to be used in the expression
address	specifies a base name, program name, symbol, period (.) or explicit pathname (program name plus symbol name plus offset or program name plus offset).
hex	specifies a hexadecimal value
dec	specifies a decimal value

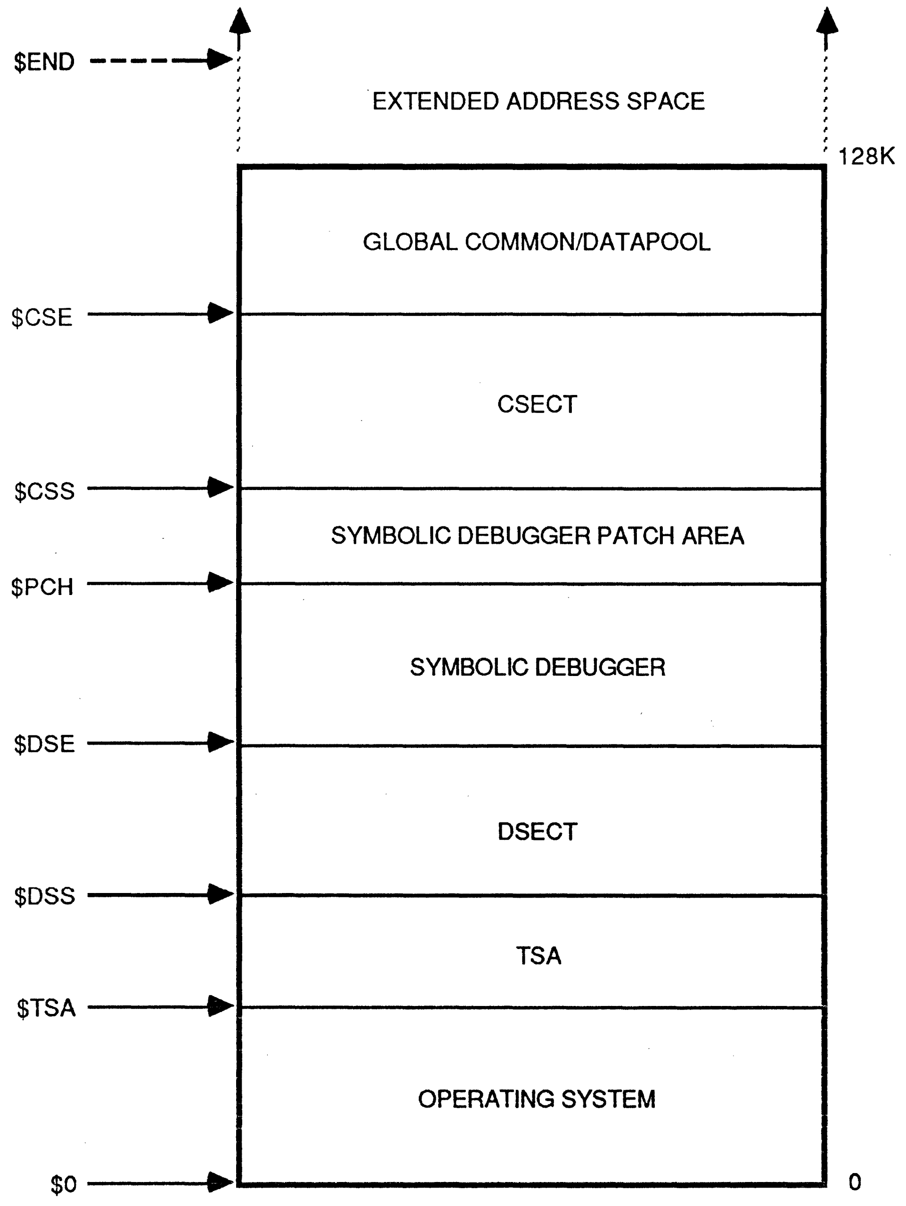
These expressions specify the 32-bit contents of the address or expression inside the parentheses. Bits 30 and 31 of the expression value are zeroed to determine the word address.

2.22.4 Bases

Bases are symbolic terms used in expressions. A base name is denoted by a $ as the first character. SYMDB defines nine bases when it is accessed. The nine SYMDB bases are

Base Name	Description
$	Bits 13-31 of the user task program status doubleword (PSD)
$PSD	Bits 0-31 of the user task PSD
$0	Constant zero
$TSA	Start address of the user task's task service area (TSA)
$DSS	Start address of the user task's DSECT
$DSE	End address of the user task's DSECT
$PCH	Start address of the SYMDB patch area
$CSS	Start address of the user task's CSECT
$CSE	End address of the user task's CSECT
$END	End address of the user task's extended memory

The relative position of some of the bases described above on a memory map of a user task which uses all possible memory areas (CSECT, DSECT, Global Common, and extended memory) is shown below.



SYMDB BASE NAMES

87D4102

Bases, other than the reserved SYMDB bases, can be defined through the use of the BASE directive. A user defined base name contains a maximum of nine characters. The first character must be a $ and the second an alphabetic character. Characters three through nine are optional and may be alphanumeric. User defined base names may not be the same as any of the reserved SYMDB bases.

2.22.5 Symbols

Programs assembled/compiled and cataloged with option 19 set allow SYMDB access to program names, global symbols and local symbols. If option 19 is set only for the Cataloger, only program names and global symbols can be accessed.

Program names are denoted by the special character # (pound sign) and symbol names (local and global) are denoted by the special character \ (backslash). Both special characters are optional, but if global symbols, local symbols and/or programs have the same name, the special characters should be entered for clarity.

NOTE: This is the case for FORTRAN 77+ subprograms.

When SYMDB is accessed, it defaults to searching for global symbols. If the default is not changed (via the PGM directive), local symbols must be preceded by the program name in which they are located and the backslash (\ ) character for SYMDB to access them.

If the PGM directive and a program name are entered, SYMDB then defaults to the local symbols within the specified program name. In this default condition, if a symbol is entered without the special character, SYMDB will first search the local symbol table (of the specified program). If the symbol is not found, the global symbol table will be searched. If the symbol is not found in the global symbol table, the program table will be searched. Therefore, the special characters should be used to avoid ambiguous cases (symbols and programs with the same name).

Local symbols which are not located in the program specified in the PGM directive must be preceded by the program name in which they are defined and the backslash (\ ) character.

The following syntax shows valid symbolic addresses:

Syntax

```
[#] progname
```

#	specifies the optional special character to denote a program name
progname	specifies the program name to be used as a symbolic address

Syntax

```
[ \ ] glosym
```

\	specifies the optional special character to denote a symbol name
glosym	specifies the global symbol to be used as a symbolic address.
Note:	If the default is to local symbols and a local symbol exists with the same name, the PGM directive must be entered without a program name to set the default to global symbols. Otherwise, the local symbol by that name will be accessed.

Syntax

    [[#] progname]] [\ ] locsym

#	specifies the optional special character to denote a program name
progname	specifies the program name. If the default is set to local symbols defined in that program name, progname is optional. If the default is set to global symbols (null pgm) or to local symbols defined under a different program name, then the program name must be specified followed by the backslash character and the local symbol name.
\	specifies the optional special character to denote a symbol name. This character is optional if the local symbol name it precedes is in the default local symbol table. Otherwise it must be specified.
locsym	specifies the local symbol to be used as a symbolic address.

2.22.6 COUNT

COUNT is a special term in expressions used to determine how many times a trap has been encountered since it was set. When a trap is set, a counter is established to track the number of times the trap is encountered. COUNT is always updated to reflect the number of times that the last trap in the program was encountered. Therefore, COUNT can be specified after each trap to determine how many times each trap has occurred.

COUNT is used only in conditional trap lists. If a program has a loop which executes properly a number of times and then encounters an error, a trap can be set at the beginning of the loop with a conditional trap list to execute only when COUNT equals the number of times the loop executed properly. Then, through the directives in the trap list, the user can examine memory or register contents during the iteration of the loop which contains the error.

Example

The user sets a trap

```
.   SET S.10
..  IF COUNT Y >=100
..  SNAP C(I)
.   GO
```

When Y is greater than or equal to 100, SYMDB traps and displays the value of the variable "I".

2.22.7 Period (.)

The special character period (.) is equal to the last address displayed by a memory related directive. The period (.) is used as a term in an expression in place of re-entering the last displayed address.

The period (.) is set by the execution of the CM, DA, DD, DF, DI, DN, DNB, DNH, DNW and SNAP directives.

Example

The user enters the directive

```
.DI #DBGTST \ SYMBOL1
```

SYMDB responds

```
DBGTST \ SYMBOL1        LW R5, DBGTST \ SYMBOL2
```

The user enters the directive

```
.SET .
```

SYMDB issues the trap list prompt and the user enters the directive

```
..END
```

SYMDB sets a trap at the address specified by period (.) which is DBGTST \ SYMBOL1 with no corresponding trap list directives.

SECTION 3 - DIRECTIVES

3.1 Using SYMDB Directives

The following rules apply to SYMDB directives, whether they are entered from the lfc #IN (batch mode or interactive mode) or from a command file (lfc #03) through the use of the FILE command.

. Each directive record read from the lfc #IN is placed in a 72-character buffer. If the record size of the file/device assigned to the lfc #IN is other than 72 characters, the command is left-justified and blank-filled or truncated to the 72-character buffer size.

. Compound directives and continuation of directive lines are not allowed.

. All commands have a directive verb. Some directive verbs may be abbreviated by entering the characters underlined in the syntax. If no directive verb is entered, SYMDB defaults to the SNAP directive.

. The directive verb is followed by a termination character (any non-alphabetic character) and the directive argument list (if required). Multiple arguments are separated by commas (,). Embedded blanks in a directive line are ignored except inside a G or C character constant.

. Error messages are written to the lfc #OT when an incorrect directive is entered. Refer to Section 4 for a description of the error messages.

. The response to each entered directive is written to the lfc #OT following that directive. (Some directives have no displayed response.)

3.2 A (Address) Directive

The A directive evaluates and displays an expression in address format. If extended addressing mode is not set, 19 bits are used. If extended addressing mode is set, 24 bits are used.

Syntax:

A expr

expr	specifies any valid SYMDB expression.

Response:

In relative (no FORTRAN) mode, the address is displayed as the closest base or program name to the value plus the positive offset, in hexadecimal.

In relative FORTRAN mode, the address is displayed as #PROGRAM\LOCSYM plus offset.

In absolute mode, the address is displayed as a hexadecimal number without leading zeros.

3.3 ABSOLUTE Directive

The ABSOLUTE directive sets the absolute mode. As a result, subsequent address expressions are interpreted as absolute and displayed as absolute hexadecimal logical addresses. This mode is in effect until a RELATIVE directive is executed.

Syntax:

ABSOLUTE

Response:

The directive is always valid.

There is no output.

SYMDB prompts for the next directive.

Usage:

```
.AB
.X C(100)
00000000
.REL
.X C(100)
DGE00008
```

3.4 B (Binary) Directive

The B directive evaluates an expression and display its result in binary format.

Syntax:

<u>B</u> expr

expr	specifies any valid SYMDB expression

Response:

The 32-character binary equivalent of the expression is displayed.

3.5 BASE Directive

The BASE directive defines a user base (add its name to the internal base definition table), delete a user base name from the base table, or redefine a user base (change the value specified in the base name's definition).

Up to 16 user bases are allowed. Refer to Section 2.16 Establishing User Bases.

Syntax:

<u>BA</u>SE base [,expr]

base	specifies a user base name. A user defined base name contains a maximum of nine characters. The first character must be a $ and the second an alphabetic character. Characters three through nine are optional and may be alphanumeric.
expr	specifies an address to be used as the base's value. If the expression is not specified, the base name is deleted. If expr is specified and base is already defined, base is redefined to the value specified by expr.

Response:

There is no output except error messages. Error messages inform the user if:

. the user tries to define a new base and the base table is full (16 user bases)

. base is not specified

. base is a base name which was defined by SYMDB and cannot be redefined

. the user attempts to delete an undefined base

Usage:

```
.BA $MINE,DSS+388
.SHOW B

BASE NAME          VALUE
$               00021860
$PSD            01021860
$0              00000000
$TSA            00020000
$DSS            00021800
$DSE            00028000
$PCH            00079488
$CSS            00080000
$CSE            00080000
$END            00080000
$MINE           00021B88
```

3.6 BREAK Directive

The BREAK directive transfers control from SYMDB to the user task's break receiver.

Syntax:

    BREAK

Response:

The user break receiver gets control. SYMDB regains control upon the occurrence of the next break, trap, user abort, or break receiver exit.

An error message informs the user if the user task has no break receiver.

The BREAK directive is a trap list terminator.

3.7 CC (Condition Code) Directive

The CC directive displays the four condition code bits in SYMDB base $PSD (bits 0-31 of the user PSD) or displays the old condition code of $PSD and inserts a new value.

Syntax:

    CC [cc]

cc	is a string of four binary digits that will replace the existing condition codes in $PSD. If not specified SYMDB displays the present condition codes.

Response:

An error message informs the user if the condition code is specified incorrectly.

SYMDB prompts for the next directive.

Usage:

```
.CC
OLD CC=0000
.CC 0101
OLD CC=0000
.CC
OLD CC=0101
```

3.8 CLEAR Directive

The CLEAR directive deletes all user defined bases or traps.

Syntax:

```
    CLEAR   {BASES}
            {TRAPS}
```

BASES	specifies delete all user base definitions.
TRAPS	specifies delete all traps.

Response:

An error message informs the user of any argument specification errors.

There is no output except for error messages. SYMDB prompts for the next directive.

Usage:

```
.SHOW B
BASE NAME          VALUE
$                  00037598
$PSD               01037598
$0                 00000000
$TSA               00032000
$DSS               00036000
$DSE               0003A000
$PCH               0007FB20
$CSS               00080000
$CSE               00080000
$END               00080000
$MINE              00036368


.CL B
.SHOW B
BASE NAME          VALUE
$                  00037598
$PSD               01037598
$0                 00000000
$TSA               00032000
$DSS               00036000
$DSE               0003A000
$PCH               0007FB20
$CSS               00080000
$CSE               00080000
$END               00080000
```

3.9 CM (Change Memory) Directive

The CM directive alters the contents of one or more consecutive words in the task's logical address space.

Syntax:

<u>CM</u> addr=expr$_1$ [,expr$_2$,...,expr$_n$]

addr	specifies the address of the first word to be changed (bits 30 and 31 of addr are ignored).
expr	specifies the 32-bit value to be stored at the specified address. Successive values are stored in consecutive words beginning at addr. Two consecutive commas with no intervening value can be used to skip the memory address corresponding to the missing value, leaving its contents unchanged.

Response:

Error messages inform the user if:

. addr and expr are not both present and valid

. memory changes must be stopped because addr or an address derived from it (multiple values) violates a SYMDB address restriction

. an error occurs in evaluating one of the expr values

Note: In the third case, the error message will specify which memory words, if any, were successfully changed.

A SNAP is automatically performed by SYMDB for the modified range and the new contents are displayed.

SYMDB prompts for the next directive.

When storing a double precision floating point constant into memory, two words are changed.

The special character period (.) is set at the completion of this directive (refer to Section 2.22.7 Period (.)).

3.10 CR (Change Register) Directive

The CR directive alters the contents of one or more user registers.

Syntax:

CR Rn=$expr_1$ [,$expr_2$,...,$expr_n$]

n	specifies a user register (0-7)
expr	specifies the 32-bit value to be stored in the specified register. Succeeding values, if any, are stored in consecutive user registers. Two consecutive commas with no intervening value can be used to skip the user register corresponding to the missing value, leaving its contents unchanged. If user register R7 has been altered or skipped and one or more unused values remain, they are ignored.

Response:

An error message informs the user if:

. A register specification is absent or not in the range 0-7.

. The first value is not specified.

SYMDB prompts for the next directive.

When changing a register to a double precision floating point constant, two registers are changed.

Usage:

```
.CR R1=1,2,3,4,5,6,7
PSW=01036E3A  (CC=0000)  (PC=$DSS+E3A)
REGS=01036920 00000001 00000002 00000003
     00000004 00000005 00000006 00000007
.CR R1=1,,4,,6
PSW=01036E3A  (CC=0000)  (PC=$DSS+E3A)
REGS=01036920 00000001 00000002 00000004
     00000004 00000006 00000006 00000007
```

3.11 DA (Display ASCII) Directive

The DA directive displays a memory range in ASCII format.

Syntax:

DA [low] [,high]

low	specifies the first byte address to be displayed. If not specified, the last location displayed plus one word is used as the default.
high	specifies the last byte address of the range to be displayed. If not specified, only the low address is displayed.

Response:

Memory addresses are displayed in label-field format. The contents of memory are displayed in ASCII format.

The special character period (.) is set at the completion of this directive (refer to Section 2.22.7 Period (.)).

3.12 DD (Display Double Precision) Directive

The DD directive displays a memory range in double precision floating point format.

Syntax:

DD [low] [,high]

low	specifies the first word address to be displayed. If not specified, the last location displayed plus one word is used as the default.
high	specifies the last word address of the range to be displayed. If not specified, only the low address is displayed.

Response:

Addresses specified are displayed in label-field format. The contents of the specified memory addresses plus the contents of the next word are converted to their floating point double precision equivalent and displayed.

If a range is given, the second display begins two words (8 bytes) after the first display.

The special character period (.) is set at the completion of this directive (refer to Section 2.22.7 Period (.)).

3.13 DELETE Directive

The DELETE directive deletes a specified trap and restores the user instruction to its original location.

Syntax:

<u>D</u>ELETE addr

addr	specifies a trap address.

Response:

An error message informs the user if:

. addr is not specified

. addr is not an address at which a trap has been set by the SET directive

The user instruction replaced by the trap instruction is restored to its original location.

SYMDB prompts for the next directive.

3.14 DETACH Directive

The DETACH directive detaches SYMDB from the user task and transfers control to the task at the specified address or at $ (bits 13-31 of user PSD).

Syntax:

DETACH [addr]

addr	specifies the address within the user task to which control is transferred. If not specified, defaults to $.

Response:

All traps are deleted (there is no need to enter CLEAR TRAPS to restore user instructions replaced by trap instructions).

SYMDB files and memory are deallocated.

SYMDB transfers control to the specified address.

An error message informs the user if the specified address violates SYMDB's address restriction.

DETACH is a trap list terminator.

3.15 DF (Display Floating Point) Directive

The DF directive displays a memory range in floating point format.

Syntax:

DF [low] [,high]

low	specifies the first word address to be displayed. If not specified, the last location displayed plus one word is used as the default.
high	specifies the last word address of the range to be displayed. If not specified, only the low address is displayed.

Response:

Memory addresses are displayed in label-field format. The content of the specified memory address is displayed in single precision floating point format.

If a range is given, the second display begins one word (4 bytes) after the first display.

The special character period (.) is set at the completion of this directive (refer to Section 2.22.7 Period (.)).

3.16 DI (Display Instruction) Directive

The DI directive displays a memory range as mnemonic instructions (assembly language).

Syntax:

    DI [low] [,high]

low	specifies the first word or halfword address to be displayed. If not specified, the last location displayed plus one word is used as the default.
high	specifies the last word or halfword address of the range to be displayed. If not specified, only the low address is displayed.

Response:

Memory addresses are displayed in label-field format. The contents of the memory addresses are displayed in assembly language format.

The special character period (.) is set at the completion of this directive (refer to Section 2.22.7 Period (.)).

3.17 DN (Display Numeric) Directive

The DN directive displays a memory range in decimal integer format.

Syntax:

    DN [low] [,high]

low	specifies the first word address to be displayed. If not specified, the last location displayed plus one word is used as the default.
high	specifies the last word address of the range to be displayed. If not specified, only the low address is displayed.

Response:

Memory addresses are displayed in label-field format. The contents of the memory addresses are displayed in decimal integer format.

The size is determined by the Symbol Table Entry. If there is no Symbol Table Entry, the default is one word.

The special character period (.) is set at the completion of this directive (refer to Section 2.22.7 Period (.)).

3.18 DNB (Display Numeric Byte) Directive

The DNB directive displays a memory range in decimal byte format.

Syntax:

DNB [low] [,high]

low	specifies the first byte address to be displayed. If not specified, the last location displayed plus one word is used as the default.
high	specifies the last byte address of the ranqe to be displayed. If not specified, only the low address is displayed.

Response:

Memory addresses are displayed in label-field format. The contents of the memory addresses are displayed in decimal integer byte format.

The special character period (.) is set at the completion of this directive (refer to Section 2.22.7 Period (.)).

3.19 DNH (Display Numeric Halfword) Directive

The DNH directive displays a memory range in decimal halfword format.

Syntax:

DNH [low] [,high]

low	specifies the first halfword address to be displayed. If not specified, the last location displayed plus one word is used as the default.
high	specifies the last halfword address of the range to be displayed. If not specified, only the low address is displayed.

Response:

Memory addresses are displayed in label-field format. The contents of the memory addresses are displayed in decimal integer halfword format.

The special character period (.) is set at the completion of this directive (refer to Section 2.22.7 Period (.)).

3.20 DNW (Display Numeric Word) Directive

The DNW directive displays a memory range in decimal word format.

Syntax:

DNW [low] [,high]

low	specifies the first word address to be displayed. If not specified, the last location displayed plus one word is used as the default.
high	specifies the last word address of the range to be displayed. If not specified, only the low address is displayed.

Response:

Memory addresses are displayed in label-field format. The contents of the memory addresses are displayed in decimal integer word format.

The special character period (.) is set at the completion of this directive (refer to Section 2.22.7 Period (.)).

3.21 DUMP Directive

The DUMP directive outputs the contents of a range of specified memory addresses, including the user PSD and registers. When used in interactive mode, the dump is output to the SLO file (assigned to logical file code #02). In batch mode, the dump is output to the file or assigned to logical file code #OT.

Syntax:

DUMP [low] [,high]

low and high	are expressions representing memory addresses. If high is not specified or is not greater than low, only the single word at low is displayed. If no addresses are specified, SYMDB will dump the addresses following the last address dumped. If no addresses were dumped, SYMDB will dump the contents of memory starting at absolute address zero. Bits 30 and 31 of the values of low and high are zeroed to produce word addresses.

Response:

The memory range between the specified addresses is output. The user PSD and registers are also shown.

An error message is displayed if any address in the range violates an address restriction.

3.22 E (Single Precision Floating Point) Directive

The E directive displays an expression value in single precision floating point format.

Syntax:

E expr

expr	specifies the expression to be displayed in floating point format.

Response:

The single precision floating point equivalent of the expression is displayed.

3.23 END Directive

The END directive terminates a trap list. Using a carriage return (CR) in interactive mode performs the same function.

Syntax:

END or <CR>

Response:

END is a trap list terminator.

3.24 EXIT Directive

The EXIT directive terminates debugging and returns control to TSM. Both the user task and SYMDB exit. In batch mode, if EXIT is used and/or an EOF is encountered on the file or device assigned to logical file code #IN, SYMDB terminates.

Syntax:

EXIT (or) X

Response:

SYMDB calls the M.EXIT service.

EXIT is a trap list terminator.

SYMDB also exits in response to a Control C (end-of-file).

3.25 FILE Directive

The FILE directive reads subsequent SYMDB directives from a directive file instead of from the lfc #IN. The current working directory name stored in T.CDIR in the task's TSA is used to access the directive file.

Syntax:

$\underline{\text{FI}}\text{LE}$ filename [,password]

filename	specifies a one to eight character name of a directive file.
password	is ignored

Response:

An error message informs the user if:

. filename is absent or invalid, or the file does not exist

. the FILE directive is read from a directive file

A command file is a permanent disc file containing SYMDB directives. Directives are in the form of 72-byte logical records. The Text Editor utility (EDIT) can be used to create a SYMDB command file. The STORE directive (not SAVE) should be used to create a command file.

If there are no errors, SYMDB assigns lfc #03 to the specified file and reads subsequent directives from #03 instead of #IN. When SYMDB reaches end-of-file on #03 or a break is recognized, directive input returns to #IN.

SYMDB searches for a user file by the specified filename. If a user file is not found, it then searches for a system file.

Use of the FILE directive terminates a trap list.

3.26 FORMAT Directive

The FORMAT directive sets the default input format for undesignated numeric constants and integers in expressions to hexadecimal or decimal.

Syntax:

$\underline{\text{FO}}\text{RMAT} \quad \left\{ \begin{matrix} \underline{\text{X}} \\ \underline{\text{N}} \end{matrix} \right\}$

X	sets the input radix to hexadecimal, which is the default when SYMDB is accessed.
N	sets the input radix to decimal.

Response:

An error message informs the user if the format specification is absent or invalid.

SYMDB prompts for the next directive (no output).

3.27 GO Directive

The GO directive transfers control to the user task, optionally setting a one-shot trap.

Syntax:

```
    GO [addr] [,trap]
```

addr	specifies the address within the user task to which SYMDB transfers control. If not specified, SYMDB base $ (bits 13-31) of the user PSD is used.
trap	specifies the address within the user task at which SYMDB sets a trap. The list of SYMDB directives executed when the trap occurs is as follows:

```
!* ONE-SHOT TRAP SET BY GO DIRECTIVE
!DEL $
!END
```

$ is the special SYMDB base equal to bits 13-31 of the user PSD.

If a trap address is not specified SYMDB does not set a trap before transferring control to the user task.

Response:

An error message informs the user if:

. either the transfer address or trap address violate SYMDB address restrictions

. a trap address is specified and a trap is already set there

. no trap table space remains and a trap address is specified

. the specified transfer address is not on a word boundary

. addr is an odd number

. trap is not on a word boundary

If GO is successful, SYMDB transfers control to the user task at the specified address. If the last control transfer into SYMDB was caused by a trap and control is passed to the trap address for that trap, the user instruction replaced by the trap instruction is executed first. Control is then passed to the trap address plus one word unless the replaced user instruction is any instruction which terminates the TRACE , TRACK, or WATCH commands. Such a replaced instruction may not be executed without first deleting the trap set on it.

Control remains with the user task until a trap, break, or user abort occurs, at which time SYMDB regains control and prompts for the next directive or a trap list as appropriate.

GO is a trap list terminator.

3.28 IF Directive

The IF directive makes a trap list conditional. (The trap list is executed only if specified conditions are met.) When used, this directive must be the first directive of the trap list.

Syntax:

```
IF cond
```

cond	specifies any valid SYMDB expression.

Response:

If the value of cond is nonzero, the trap is reported and remaining directives in the trap list are executed. The relational operators produce a value of 1 if the relation is true, and a value of 0 if false (refer to Section 2.13.3 for a description of relational operators).

If the value of cond is zero, no trap is reported and the program continues executing as if the user issued a GO $ command.

The trap's COUNT is incremented whether the trap is reported or not .

An error message informs the user when the IF command is entered as an immediate command or when cond is absent or invalid.

USAGE:

```
. SET S.4                  TRAP LIST EXECUTED
..IF COUNT=42              IF COUNT EQUALS 42
  .
  .
  .
..END
.
. SET S.8                  TRAP LISTED EXECUTED
..IF C(J)=18               WHEN J EQUALS 18
  .
  .
  .
..END
.
```

3.29 LIST Directive

The LIST directive displays the trap list for a specific trap.

Syntax:

```
LIST trap
```

trap	specifies a trap address

Response:

An error message informs the user if "trap" is not a trap address.

Usage:

```
.L 300
COUNT = 0
MSG HELLO
SNAP 300,400
END
.LOG
```

3.30 LOG Directive

The LOG directive outputs the current contents of the terminal log file to lfc #02 (SLO file only).

Note:

A log file is maintained only when the LOG option has been specified by the MODE directive. The default condition is to maintain a log file.

Syntax:

```
    LOG
```

Response:

All log file records which have not already been printed are copied to an SLO file. The LO file is then closed and deallocated. All log file records thus copied are no longer accessible (their space is released). The LOG directive is ignored in batch.

An error message is displayed if LOG is entered in response to a prompt for a deferred directive (..).

3.31 MODE Directive

The MODE directive sets the following modes for the debugging session:

. A log file is/is not maintained to log the debugging session
. Extended memory access is/is not allowed
. FORTRAN display format is/is not set

Syntax:

```
          ⎧LOG       ⎫
          ⎪NOLOG     ⎪
    MODE  ⎨EXTENDED  ⎬
          ⎪NOEXTENDED⎪
          ⎪FORTRAN   ⎪
          ⎩NOFORTRAN ⎭
```

LOG	specifies that a log file is maintained for the debugging session.
NOLOG	specifies that a log file is not maintained for the debugging session.
EXTENDED	specifies that extended addressing is allowed, thus the user has access to program locations in extended memory.
NOEXTENDED	specifies that extended addressing is not allowed, thus the user must trace through an SEA (set extended addressing) instruction to access extended memory.

FORTRAN	specifies that FORTRAN addressing format is set. The address label field is displayed as the program name and closest previous symbol name and the offset address (i.e., program \ symbol + 04).
NOFORTRAN	specifies that NOFORTRAN addressing format is set. The address label field is displayed as the program name plus the offset address (i.e., program + 04).

Response:

An error message informs the user if the mode is invalid or missing.

SYMDB prompts for the next directive (no output).

3.32 MSG Directive

The MSG directive denotes a comment in a debugging session. It is most useful in directive files and trap lists to document the directives.

Syntax:

MSG message

(or)

* message

message	specifies any character string.

Response:

The character string is displayed.

Usage:

```
.SET 300
..MSG HELLO
..END
.
```

3.33 N (Numeric) Directive

The N directive evaluates and displays the expression's value in signed decimal integer format.

Syntax:

N expr

expr	specifies the expression to be evaluated and displayed in signed decimal integer format.

Response:

The signed decimal integer equivalent of the expression is displayed.

3.34 PGM (Program) Directive

The PGM directive establishes a program name in which the Debugger will search for local symbols. This directive also sets a new relative program name (see RELATIVE command).

Syntax:

```
    PGM [progname]
```

progname	specifies a program name which may begin with the character # (the designating character # is optional). If a program name is not specified, the current program name is cleared and SYMDB defaults to the global symbol table.

Response:

An error message informs the user if the program name could not be found.

3.35 RELATIVE Directive

The RELATIVE directive sets relative mode. Subsequent addresses that do not include an explicit base, program, or symbol name are interpreted as relative to the base or program name set by this directive. The base name must have been previously defined in a BASE directive.

Syntax:

```
    RELATIVE [base    ]
             [progname]
```

base	is a base name which must begin with the character $ (refer to Section 2.16 Establishing User Bases).
progname	is a program name which may begin with the character # (the designating character # is optional).

If neither parameter is specified, the last base or program name that was set by a RELATIVE directive is used. During initialization, SYMDB sets the relative mode and establishes $DSS (DSECT start) as the default base.

Response:

Each address subsequently displayed is represented as a displacement from the nearest base or program name which is not greater than the address. If a base and a program name have the same value, SYMDB uses the program name.

An error informs the user if the specified base or program name is not defined.

3.36 REVIEW Directive

The REVIEW directive writes the log file to the lfc #OT.

Syntax:

REVIEW [screens]

screens	specifies the number of screens from the current position in the log file for SYMDB to backspace before beginning the log file display. If not specified or if specified as a number greater than the number of screens currently contained in the log file, the display begins at the first record in the log file.

Response:

SYMDB displays the log file one screen at a time.

When SYMDB reaches the end of the log file, the display is terminated and SYMDB prompts for the next directive. None of the above terminal I/O is copied to the log file.

REVIEW is treated as a comment in batch.

An error message informs the user if:

. REVIEW is entered as a deferred directive

. REVIEW is read from within a directive file

3.37 RUN Directive

The RUN directive sets the run mode. This results in TRACE or TRACK directives continuing until SYMDB reaches a full screen of output instead of prompting for input after each instruction.

Syntax:

RUN

Response:

Until a STEP directive is executed, the TRACE and TRACK directives will display a full screen of output before prompting for continuation or termination of the trace or track.

3.38 SET Directive

The SET directive sets a trap in the user task at a specified location.

Syntax:

    SET trap

trap	specifies the address at which SYMDB sets a trap.

Response:

An error message informs the user if:

. The trap address is not specified.

. The specified address is already a trap address.

. The specified address violates an address restriction.

. SYMDB's trap table is full and thus no more traps can be set until a trap is deleted. A maximum of 20 traps can be active at the same time in one task.

. "trap" is not on a word boundary.

The user instruction at the specified trap address is replaced by a trap instruction (SVC 1,X'66').

SYMDB then prompts for directives to be placed in the trap list (i.e., deferred commands).

The user can enter any SYMDB directive in a trap list. All directives placed in the trap list are checked for validity before they are actually stored in the trap list except for the directives CLEAR, FORMAT, MODE and SHOW. These directives will be validated only when they are to be executed. If a directive is invalid, SYMDB will write an error message and issue another prompt.

A nested trap list occurs if a user enters a SET directive in a trap list. This means the second trap is set only when the first trap is encountered. Nesting can continue as far as the user desires, however, there must be a terminator for each SET directive in the nested trap list. Refer to Section 2.12 for a detailed description of nested traps.

To terminate a trap list, enter one of the following directives: BREAK, END, EXIT, FILE, GO, TRACE, TRACK, or WATCH.

3.39 SHOW Directive

The SHOW directive displays current base definitions, trap addresses, option settings, or symbols.

Syntax:

```
SHOW  [BASES  ]
      [TRAPS  ]
      [OPTIONS]
      [SYMBOLS]
```

BASES	displays the current definitions of all special bases and user bases.
TRAPS	displays all trap addresses.
OPTIONS	displays the settings of the options controlled by the following directives: ABSOLUTE/RELATIVE RUN/STEP FORMAT MODE
SYMBOLS	displays all symbols defined in the default program (i.e., program name established by the most recent PGM command). If there is no default program name established, all global symbols are displayed.

If no parameters are specified, all displays are produced.

Response:

An error message informs the user if any argument but BASES, TRAPS, OPTIONS, or SYMBOLS is used.

Usage:

```
.SHOW
TRAP AT:    $DDS+300

BASE NAME         VALUE
$                 00037B34
$PSD              01037B34
$0                00000000
$TSA              00032000
$DSS              00036000
$DSE              0003A000
$PCH              0007FB20
$CSS              00080000
$CSE              00080000
$END              00080000
   FORMAT X
   RELATIVE $DSS        = 00036000
   STEP
   NON-FORTRAN
   NON-EXTENDED ADDRESSNG
   LOG FILE
```

3.40 SNAP Directive

The SNAP directive writes the contents of a range of logical addresses to the file or device assigned to lfc #OT. The format is a side-by-side hexadecimal and ASCII display.

This directive is also a default (implied) directive. Any expression entered without a directive verb performs as if it were preceded by SNAP. If a carriage return without a directive verb or expression is entered, SYMDB will snap the address following the last address snapped. If no addresses were snapped, SYMDB will snap the contents of memory starting at absolute address zero.

Syntax:

```
[SNAP] [low] [,high]
```

low	specifies the first address to snap. If not specified, the snap begins at the address following the last address snapped or at absolute zero if no previous address was snapped
high	specifies the last address to snap. If not specified, only the single word at the low address is snapped. Bits 30 and 31 are ignored and assumed to be zero.

Response:

The specified memory contents are written to lfc #OT.

The special character period (.) is set at the completion of this directive (refer to Section 3.16.7 Period (.)).

3.41 STATUS Directive

The STATUS directive displays a status report indicating the user PSD and the general purpose registers for the address contained in the program counter.

Syntax:

```
STATUS
```

Response:

SYMDB displays a status report on the terminal.

Usage:

```
.ST
PSW=01037834   (CC=0000)   (PC=$DSS+1B34)
REGS=01037590 00000000 00000000 00000000    ..u.............
     00000000 00000000 00000000 00700000    .............p..
```

3.42 STEP Directive

The STEP directive sets step mode.

This allows a single step trace or track through the execution of each instruction in the user task.

Syntax:

STEP

Reponse:

Until a RUN directive is issued, all TRACE and TRACK directives will pause after each instruction displayed so the user can inspect each instruction and its results before the next instruction is executed. Step mode is the default and is in effect until a RUN directive is executed.

STEP is ignored in batch.

3.43 TIME Directive

The TIME directive displays the current date and time of day.

Syntax:

TIME

Response:

SYMDB displays the calendar date as stored in the Communications Region (C.DATE) and the time of day as returned by the M.TDAY service.

3.44 TRACE Directive

The TRACE directive executes and displays each user instruction and its results. To trace only branching instructions, use the TRACK command.

Syntax:

TRACE [start] [,stop]

start	specifies the address of the first user instruction to be executed. If not specified, the special base $ (bits 13-31 of the user PSD) is used.
stop	specifies the address of the last user instruction to be traced. If not specified, the trace continues as described below.

Response:

The debugger executes user instructions beginning at the specified start address and displays each instruction with its results and/or operands in an Assembler-like format. The instruction results are displayed on the right-hand side of the output.

In Step mode, SYMDB pauses after each instruction is executed or simulated and waits for a 1-character response from the user. To proceed to the next instruction, enter only a carriage return. Any other response terminates TRACE. If SYMDB is in Run mode, TRACE does not pause after each instruction but proceeds immediately to the next instruction; thus the only opportunity to stop the display is at the end of each screen. Note that in batch, TRACE functions as if a RUN command were in effect.

This process continues until one of the following occurs:

. An instruction has been fetched, executed, and displayed from the specified stop address. The user context indicates that the instruction has been executed, as shown in the status report indicating trace termination.

. A user instruction is aborted. SYMDB executes most user instructions by transferring control to the user task one instruction at a time. When these instructions execute, it is as if the user had entered "GO a,b" where a is the address of an instruction and b is the address of the next instruction (logically next, not necessarily a+1W). Any abort condition caused by such instructions is reported as it would be after a GO command and the trace is terminated. The user context is reported in a status report.

. SYMDB fetches an instruction that breaks the trace (see Table 3-1). The instruction is displayed and TRACE is terminated. The user context still points to the untraceable instruction, as shown in the status report announcing trace termination.

. The address of the next instruction to be fetched would violate an address restriction. No instruction is displayed, the trace is terminated, and the user context points to the bad address as shown in the status report indicating trace termination.

If the last control transfer to SYMDB is caused by a trap, and the starting address is $ (the user PSD), the user instruction replaced by the trap instruction at $ is traced as if it were at $, and the trace is continued.

An error message informs the user if the starting address violates an address restriction, if the starting address is greater than the stop address, or if the start and/or stop address is an odd address.

TRACE is a trap list terminator.

The following Assembly Language instructions causes a trace to stop (returning control to SYMDB):

Usage:

```
.TRACE
$DSS+1598           BL    $DSS+1B2C
$DSS+1B2C           SVC   X'104C'
$DSS+1B30           TBR   R7,13                    R7=00700000
$DSS+1B32           NOP
TRACE STOPPED
```

Table 3-1
Instructions that Break a Trace

AI	HALT	TD
BEI	JWCS	TMAPR
BRI	LPSDCM	TPR
CD	LPSD	TRP
DAI	RDSTS	UEI
DI	RI	WAIT
EI	RWCS	WWCS
ECWCS	SETCPU	All undefined opcodes

3.45 TRACK Directive

The TRACK directive functions exactly like TRACE, except that it displays only instructions that result in a change in the flow of control.

Syntax:

TRACK [start] [,stop]

start	specifies the address of the first user instruction to be executed. If not specified, the special base $ (bits 13-31 of the user PSD) is used.
stop	specifies the address of the last user instruction to be executed. If not specified, the track is continued as described for TRACE.

Response:

TRACK functions exactly like TRACE, except only instructions which actually cause a branch are displayed (BCT, TRSW, LPSD, etc.).

3.46 WATCH Directive

The WATCH directive functions like TRACE, but does not display instructions. It is used to detect erroneous branches into areas such as extended address space or MPX-32.

Syntax:

WATCH [start] [,stop]

start	specifies the address of the first user instruction to be executed. If not specified, the special base $ (bits 13-31 of the user PSD) is used.
stop	specifies the address of the last user instruction to watch. If not specified, the watch continues as described below.

Response:

SYMDB performs a TRACE but inhibits the usual instruction display. When, as often happens in a new program, an erroneous branch is taken, it is often into an area completely out of the program (e.g., a branch to location 0). Especially in the case of a privileged task, many instructions may precede the inevitable disaster. While the system crumbles, many of the most useful hints as to the cause (e.g., register contents) are destroyed. WATCH provides a convenient means of detecting such branches when they happen without all the terminal output caused by TRACE or TRACK.

3.47 X (Hexadecimal) Directive

The X directive evaluates and displays the expression's value in hexadecimal format.

Syntax:

```
    X expr
```

expr	specifies the expression to be displayed in hexadecimal.

Response:

The hexadecimal equivalent of the expression is displayed.

Note: If the expression is not specified, SYMDB exits.

SECTION 4 - ERRORS AND ABORTS

4.1 SYMDB File Assignment Error Messages

The following four error messages may be written to the lfc #OT if an error occurs during SYMDB file assignments.

NO FAT/FPT SPACE AVAILABLE

If the user did not assign enough dynamic file space for the executing task via the Cataloger's FILES directive, this message is written to the lfc #OT. The user should recatalog the task specifying the correct number of dynamic file assignments. SYMDB requires three additional file assignments beyond those normally required by the task. At catalog time, the user task must specify the number of files required to execute the task. If Option 19 is not set for the Cataloger, the user task will not have symbols available for use in the debugging session.

If Option 19 is set for the Cataloger, the Cataloger adds five files to the number the user requested. These five files are needed for symbolic debugging.

NO BLOCKING BUFFER AVAILABLE

If the user did not assign enough blocking buffers for the executing task via the Cataloger's BUFFERS directive, this message is written to the lfc #OT. The user should recatalog the task specifying the correct number of blocking buffers. SYMDB requires three additional blocking buffers beyond those normally required by the task. At catalog time, the user task must specify the number of buffers required to execute the task.

If option 19 is not set for the Cataloger, the user task will not have symbolic support during debugging. If option 19 is set for the Cataloger, the Cataloger adds three buffers to the number the user requested. These three buffers are needed for symbolic debugging.

NO DISC SPACE AVAILABLE

If there is not enough disc space available for SYMDB to allocate for the SLO file (1000 lines for the lfc #OT in batch mode and 300 screens of data for the temporary log file in interactive mode), this message is written to the lfc #OT. The user must wait until disc space becomes available.

LOG FILE ALREADY ALLOCATED

If SYMDB attempts to assign the temporary log file (lfc #01) and the log file has been statically assigned by the user before accessing SYMDB, this message is written to the lfc #OT. Deallocate the user defined log file assignment (no static file assignments are allowed) to allow SYMDB to assign the log file to its default assignment.

4.2 Addressing Error Messages

The following error messages may be written to the lfc #OT if an invalid address is specified in a directive.

ADDRESS MISSING

> If the CM (change memory) directive is entered without specifying the address to be changed, the message is written to the lfc #OT. Re-enter the directive specifying the address to be changed.

NO VALUE SPECIFIED

> If the CM (change memory) directive is entered without specifying the value to be placed in the address specified, this message is written to the lfc #OT. Re-enter the directive specifying the value to replace the contents of the address specified.

NO HIGH ADDRESS

> If a directive which allows a range of addresses as its parameters is entered followed by a single address and a comma (,) and no second address, this message is written to the lfc #OT. Re-enter the directive followed by the low and high addresses separated by a comma. If only one address is desired as the parameter, no comma should be entered.

LOW > HIGH

> If a directive which allows a range of addresses as its parameters is entered followed by a low address (first address specified) which is greater than the high address (second address specified), this message is written to the lfc #OT. Re-enter the directive insuring that the first address is lower than the second address specified.

ADDRESS OUTSIDE YOUR AREA

> If a directive which allows an address as its parameter is entered (other than the CM (change memory) directive) followed by an address which is not within the user program's addressing space, this message is written to the lfc #OT. Re-enter the directive insuring that the address specified is within the allowable addressing space.

CAN'T WRITE TO THAT ADDRESS

> If the CM (change memory) directive is entered followed by an address which is not within the user program's addressing space, this message is written to the lfc #OT. Re-enter the directive insuring that the address specified is within the allowable addressing space.

MAP HOLE

If a SNAP or DUMP directive is entered followed by a range of addresses which are not within the user program's addressing space, this message is written to the lfc #OT. Re-enter the directive insuring that the range of addresses is within the allowable addressing space.

CAN'T BRANCH TO ODD ADDRESS

If the DETACH directive is entered followed by an address which is incorrectly bounded, this message is written to the lfc #OT. Re-enter the directive insuring that the address specified falls on the correct boundary.

CANNOT TRACE INSTRUCTION

If the TRACE directive is entered followed by an address which is not within the user program's addressing space, this message is written to the lfc #OT. Re-enter the directive insuring that the address is within the allowable addressing space.

REG NOT 0-7

If the CR (change register) directive is entered followed by an invalid register number, this message is written to the lfc #OT. Re-enter the directive insuring that the register number specified is zero through seven.

NO CHANGE VALUE

If the CR (change register) directive is entered followed by a valid register number and invalid or missing values to be placed in the register(s), this message is written to the lfc #OT. Re-enter the directive insuring that the value(s) to be placed in the register(s) are valid 32-bit values (refer to Section 3.10 CR (Change Register) Directive).

4.3 Trap Error Messages

The following error messages may be written to the lfc #OT if an invalid trap or trap list directive is entered.

TRAPS ON WORD BOUNDARIES ONLY

This message is written to the lfc #OT if the user enters a SET or GO directive followed by a trap address that SYMDB can't locate because the:

- trap address is not on a word boundary
- trap address is a local symbol name and the default is to global symbols
- trap address is a local symbol name that is not in the default local symbol table
- trap address is a global symbol that is not in the global symbol table

Re-enter the directive insuring that the trap address is on a word boundary, and that the symbol name specified is in the default symbol table (local or global).

NO TRAP ADDRESS SPECIFIED

If a SET directive with no trap address is entered, this message is written to the lfc #OT. Re-enter the directive specifying the address where the trap is to be set.

NOT ALLOWED IN TRAP LIST

If a LOG or REVIEW directive is entered in a trap list, this message will be written to the lfc #OT. Enter any valid trap list directive, or enter a trap list terminator before re-entering the LOG or REVIEW directive (these are the only two directives which are not allowed in a trap list).

ALREADY A TRAP THERE

If a SET directive is entered followed by the address of a previously set trap, this message will be written to the lfc #OT. Re-enter the directive followed by an address at which no trap exists.

NO TRAP THERE

If a DELETE or LIST directive is entered followed by an address at which no trap exists, this message will be written to the lfc #OT. Verify the trap addresses, and re-enter the directive followed by a valid trap address.

TRAP TABLE FULL; TRAP NOT SET

If a SET directive is entered and the trap table is full (a maximum of 20 traps are set), this message is written to the lfc #OT. Delete one or more traps before another trap can be set.

IMMEDIATE "IF" NOT ALLOWED

If the IF directive is entered outside of a trap list, this message is written to the lfc #OT. Set a trap and enter the IF directive as the first directive in the trap list if the trap is to be conditional. The IF directive may not be used at any other time.

"IF" DIRECTIVE OUT OF SEQUENCE

If the IF directive is entered in a trap list and it is not the first directive in that trap list, this message is written to the lfc #OT. Reset the trap (or set a new trap) and enter the IF directive as the first directive in the trap list if the trap list is to be conditional.

4.4 Directive Expression Error Messages

The following expression error messages may be written to the lfc #OT if an invalid expression is entered.

NO EXPRESSION

If an IF or LIST directive is entered with no expression (expression is a required parameter), this message is written to the lfc #OT. Re-enter the directive with the IF conditional expression or the expression (trap address) to be listed.

NULL SUBEXPRESSION

If a directive is entered followed by an expression parameter which contains paired parentheses with no value inside '( )' this message is written to the lfc #OT. Re-enter the directive with a value inside the parentheses, or delete the parentheses.

UNDEFINED SYMBOL

If a directive is entered followed by an expression which contains a symbol that SYMDB cannot locate within the default local symbol table (if PGM directive was entered) or within the global symbol table, this message is written to the lfc #OT. Verify the default symbol table (local to a program specified in the PGM directive or global symbols) by entering the SHOW SYMBOLS directive. If the desired symbol is a local symbol to another program name, enter the PGM directive followed by the program name which defines the desired symbol before re-entering the directive.

UNRECOGNIZABLE TERM

If a directive is entered followed by an expression which contains a term that SYMDB cannot recognize, this message is written to the lfc #OT. Re-enter the directive insuring that the expression contains valid terms (refer to Section 2.22.7 for a description of valid terms used in expressions).

MISSING OPERATOR

If a directive is entered followed by an expression which does not contain an operator, this message is written to the lfc #OT. Re-enter the directive insuring that the expression contains a valid operator (refer to Section 2.21 SYMDB Directive Expressions, for a description of valid operators).

Note: This message is also written if a decimal integer greater than ten digits is entered as a term in an expression. Decimal integers may not exceed ten digits.

DANGLING OPERATOR

If a directive is entered followed by an expression which contains an operator with only one operand, this message is written to the lfc #OT. Re-enter the directive insuring that the expression contains two operands for each operator specified.

CONSECUTIVE OPERATORS

If a directive is entered followed by an expression which contains a sequence of two or more operators that are not separated by operands, this message is written to the lfc #OT. Re-enter the directive insuring that the expression contains two operands for each operator specified.

UNMATCHED LEFT (
or
UNMATCHED RIGHT)

If a directive is entered followed by an expression which contains a left parenthesis that is not paired with a corresponding right parentheses or vice versa, one of these messages is written to the lfc #OT. Re-enter the directive insuring that the expression contains paired left and right parentheses.

ADDRESS WOULD CAUSE MAP FAULT

If a directive is entered followed by an expression which contains a memory contents term that (when evaluated) produces an indirect address which would cause a map fault, this message is written to the lfc #OT. Re-enter the directive insuring that the evaluated expression does not produce an invalid address. The contents of the memory location specified in the term can be examined through the use of the display memory directives (refer to section 2.13 Examining Memory and Registers for a summary of the display memory directives).

EFFECTIVE ADDRESS CAUSES MAP FAULT

If a directive is entered followed by an expression which produces an invalid effective address, this message is written to the lfc #OT. Re-enter the directive insuring that the evaluated expression does not produce an invalid effective address.

INVALID FLOATING POINT NUMBER
or
INVALID DECIMAL NUMBER
or
INVALID HEXADECIMAL NUMBER
or
INVALID BINARY NUMBER
or
INVALID CHARACTER STRING

If a directive is entered followed by an expression which contains an invalid constant or integer, one of the above messages is written to the lfc #OT (depending on the type of constant or integer contained in the expression).

Re-enter the directive insuring that the constant or integer value is valid (refer to Sections 2.14.1 and 2.14.2 for a description of valid integer and constant terms in expressions).

4.5 Base Error Messages

The following error messages may be written to the lfc #OT if a user base is incorrectly defined, redefined or deleted.

NO BASE NAME

If the BASE directive is entered without a base name, this message is written to the lfc #OT. Re-enter the directive followed by an existing or new user defined base name.

BAD BASE NAME

If the BASE directive is entered with an invalid base name, this message is written to the lfc #OT. Re-enter the directive insuring that a valid base name is specified. (Valid base names begin with the $ and an alphabetic character followed by one to seven alphanumeric characters).

SPECIAL BASE NOT ALLOWED

If the BASE directive is entered followed by a SYMDB base name, this message is written to the lfc #OT. There are special SYMDB defined base names (refer to section 2.22.4) which may not be redefined or deleted by the BASE directive. Re-enter the directive insuring that the base name is a new or existing user defined base.

BASE TABLE FULL

If the BASE directive is entered followed by a new base name and the base table is full, this message is written to the lfc #OT. Delete one or more bases before a new base can be defined.

4.6 Directive File Error Messages

The following error messages may be written to the lfc #OT if a directive file is incorrectly accessed.

NO FILE NAME

If the FILE directive is entered and does not specify a file name, this message is written to the lfc #OT. Re-enter the directive followed by a valid directive file name and password (if required).

NO SUCH FILE

If the FILE directive is entered followed by an invalid file name, this message is written to the lfc #OT. Re-enter the directive followed by a valid directive file name and password (if required).

FILE NAME > 8 BYTES

If the FILE directive is entered followed by a file name which is more than eight characters (eight bytes) in length, this message is written to the lfc #OT. Re-enter the directive followed by a valid file name (not exceeding eight characters) and password (if required).

4.7 Directive Argument Error Messages

The following error messages may be written to the lfc #OT if an invalid or missing argument is entered following a directive.

ARGUMENT SHOULD BE "BASES" OR "TRAPS"

If the CLEAR directive is entered with an invalid or missing argument, this message is written to the lfc #OT. Re-enter the directive specifying either bases or traps as the argument.

ARGUMENT SHOULD BE BLANK, "BASES", "OPTIONS", OR "TRAPS"

If the SHOW directive is entered with an illegal or missing argument, this message is written to the lfc #OT. Re-enter the directive specifying one of the valid arguments listed in the message.

ARGUMENT SHOULD BE "X" OR "N"

If the FORMAT directive is entered with an illegal or missing argument, this message is written to the lfc #OT. Re-enter the directive specifying either X (hexadecimal input radix) or N (decimal input radix).

ARGUMENT SHOULD BE "NOFORTRAN", "FORTRAN", "EXTENDED", "NOEXTENDED", "LOG", "NOLOG"

If the MODE directive is entered with an illegal or missing argument, this message is written to the lfc #OT. Re-enter the directive specifying one of the valid arguments listed in the message.

RELATIVE NAME NOT FOUND

If the RELATIVE directive is entered with an invalid base or program name, this message is written to the lfc #OT. Re-enter the directive insuring that a valid base or program name is specified.

CAN'T USE "$" OR "$PSD"

If the RELATIVE directive is entered with a $ or $PSD as its argument, this message is written to the lfc #OT. SYMDB special bases $ and $PSD can not be specified for relative addressing. Re-enter the directive insuring that a valid base or program name is specified.

DELETE WHAT

If the DELETE directive is entered with no argument, this message is written to the lfc #OT. Re-enter the directive insuring that the trap to be deleted is specified.

NO SUCH PROGRAM NAME

If the PGM directive is entered with an invalid program name, this message is written to the lfc #OT. Re-enter the directive insuring that a valid program name (to be established as default for local symbols) is specified.

BAD CONDITION CODE

If the CC (condition code) directive is entered with an invalid value to replace the existing condition codes, this message is written to the lfc #OT. Re-enter the directive insuring that the value to replace the existing condition codes is a 4-digit binary value. (To display existing condition codes, enter the CC directive with no argument).

4.8 Other Error Messages

These error messages may be written to the lfc #OT if one of the following errors occur.

NO USER BREAK RECEIVER

If the BREAK directive is entered and there is no break receiver in the user program, this message is written to the lfc #OT. This directive can only be used to transfer control to the user program's break receiver. If no break receiver exists, the directive is invalid.

UNRECOGNIZED DIRECTIVE

If an invalid directive is entered, this message is written to the lfc #OT. Verify the valid directive syntax, and re-enter the directive.

LOG FILE IS FULL, USE "LOG" DIRECTIVE TO OUTPUT IT

If SYMDB temporary log file is filled, this message is written to the lfc #OT. Enter the LOG directive (to write the log file to the line printer) or the REVIEW directive (to write the log file to the lfc #OT). If neither directive is entered, the contents of the log file are destroyed and a new log file started.

TRACE STOPPED

If any error occurs while in trace mode, or if a < CR > is not entered while in trace mode with step in effect this message will be written to the lfc #OT.

4.9 Abort Codes

SYMDB has abort codes which may be written to the lfc #OT if an abort occurs. These abort codes are:

. DB01 In batch mode, the end of the file assigned to lfc #OT has been encountered before the end of job (EOJ).

. DB02 A fatal I/O error has occurred on the lfc specified after the abort code in the abort message.

The following are examples of fatal I/O errors:

- The input file is not assigned and there is no default input file.
- The output file is not assigned and there is no default output file.
- The same lfc is assigned to both the input and output file.

The above abort codes usually refer to errors within the job control. Therefore, the job control should be examined for errors before the program code.

The following abort codes are internal to SYMDB, and do not apply to errors within the job control. Refer to the section 2.3 Logical File Code Assignments, for help with these errors.

. DB03 M.ASSN ERROR ON LFC #IN

. DB04 M.OPENR ERROR ON LFC #IN

. DB05 M.ASSN ERROR ON LFC #OT

. DB06 M.OPENR ERROR ON LFC #OT

SECTION 5 - SAMPLE DEBUGGING SESSIONS

5.1 Debugging Session Introduction

This section illustrates how to use SYMDB. The examples show separate subprograms or modules that can be debugged separately, and then added to the stable system.

The following sample programs and directive keys illustrate the use and results of the debugging sessions.

5.2 Example 1: Scanning Data in a Program Loop

The sample program for this debugging session searches through a table of seven values looking for the value five. If the value is contained in the table, the program successfully exits. Otherwise, the program aborts.

The debugging session shows how to set a trap at the beginning of a loop and how to build a trap list that displays the register contents for the trap address. The program then continues execution through the loop until the trap is encountered again. This cycle continues through each successive iteration of the loop. This allows the user to examine the register contents to insure correct program execution for each iteration of the loop.

The following subsections contain the sample program and the debugging session which demonstrate the procedure described above.

```
DBGTST                                                                              VER 9.3

                                   PROGRAM   DBGTST
                          *
                          *        THIS PROGRAM WILL DEMONSTRATE THE SYMBOLIC CAPABILITIES
                          *        WHICH ARE USED THROUGH THE SYMBOLIC DEBUGGER.
                          *
                          *        THIS PROGRAM WILL SIMPLY SCAN A TABLE OF DATA ITEMS
                          *        LOOKING FOR A SPECIFIC ITEM.  IF THE ITEM IS FOUND
                          *        THEN A SUCCESSFUL EXIT IS PERFORMED.  IF THE ITEM
                          *        IS NOT FOUND THEN THE PROGRAM IS ABORTED.
                          *
   00000                  R0       EQU       0
   00001                  R1       EQU       1
   00002                  R2       EQU       2
   00003                  R3       EQU       3
   00004                  R4       EQU       4
   00005                  R5       EQU       5
   00006                  R6       EQU       6
   00007                  R7       EQU       7
                          *
  P00000 00000001         BEGTABLE DATAW     X'1'
  P00004 00000002                  DATAW     X'2'
  P00008 00000003                  DATAW     X'3'
  P0000C 00000004                  DATAW     X'4'
  P00010 00000005                  DATAW     X'5'
  P00014 00000006                  DATAW     X'6'
  P00018 00000007                  DATAW     X'7'
  P0001C                  ENDTABLE EQU       $                 ! ESTABLISH ENDING ADDRESS OF TABLE
  P00020                           BOUND     1D                ! DOUBLEWORD ALIGN TASK NAME
  P00020 54535444         TASKNAME DATAB     C'TSTDBG  '       ! NAME OF THIS TASK
  P00024 42472020
                          *
                          *    START OF PROGRAM
                          *
  P00028 34800000  P00000 START    LA        R1,BEGTABLE       ! GET ADDR. OF BEGINNING OF TABLE
  P0002C C9000005                  LI        R2,5              ! PUT IN VALUE OF ITEM SEARCHED FOR
  P00030 91200000   00000 LOOP     CAMW      R2,0,R1           ! CHECK IF FOUND VALUE
  P00034 EE000051  P00050          BEQ       ENDSUCC           ! BRANCH TO END SUCCESSFULLY IF FOUND
                          *
  P00038 C8810004                  ADI       R1,1W             ! INC. TO LOOK AT NEXT ITEM IN TABLE
  P0003C 90800010  P0001C          CAMW      R1,ENDTABLE       ! CHECK IF AT END OF TABLE
  P00040 F2000031  P00030          BNE       LOOP              ! BRANCH IF NOT AT END OF TABLE
                          *
  P00044 AE800054  P00054          LW        R5,=C'ERR'        ! LOAD IN ABORT CODE
  P00048 AF000022  P00020          LD        R6,TASKNAME       ! LOAD IN ABORT TASK NAME
  P0004C C8061056                  SVC       1,X'56'           ! ABORT THIS TASK
                          *
  P00050 C8061055         ENDSUCC  SVC       1,X'55'           ! SUCCESSFUL END OF PROGRAM
  P00054 45525220
  P00058           P00028          END       START
*  0000  ERRORS IN DBGTST
```

5.2.2 Sample Debugging Session for program DBGTST

All directives in the sample debugging session are numbered to correspond to a key which describes the directives and responses. Each directive is immediately followed by its response.

Directive Key

1) SYMDB is accessed by entering the DEBUG directive in response to the TSM prompt.

   SYMDB responds with its identifying message.

2) The directive to show the symbols (SHOW SYMBOLS) is entered.

   The default is to the global symbol table, the response displays the global symbols header and indicates that there are no global symbols in the program by displaying no symbol names.

3) The directive to set the default program name and accessible symbols (PGM DBGTST) is entered.

   The default program name is set to DBGTST. The symbols local to DBGTST are accessable. There is no written response to this directive.

4) The directive to show symbols (SHOW SYMBOLS) is entered again (this time to display the local symbols).

   The response displays the local symbols header and all symbols local to the program DBGTST.

5) The directive to set a trap at the beginning of a program loop at the location whose address is specified by the symbol name LOOP (SET LOOP) is entered.

   The trap is set at the address specified by the symbol name LOOP and SYMDB responds with the trap list prompt (..).

6,7) The directive (directive 6) to snap register one (SN R1) is entered in the trap list followed by the GO directive (directive 7). These directives are deferred until the trap is encountered. When they are executed, R1 will be displayed and program execution resumed. Because the trap is set at the beginning of a loop, each time program execution is resumed the trap is encountered again and the trap list executed. This allows the user to insure correct program execution for each iteration of the loop. The GO directive in the trap list is a trap list terminator, therefore the trap list is ended and the immediate lfc #IN prompt (.) is issued.

8) The directive to begin program execution (GO) is entered. Program execution will begin at the base $ (bits 13-31 of the user PSD or the current program counter value) because no start address was specified.

   The program will begin execution. There is no written response to the GO directive.

6a) The trap is encountered and the SNAP directive (deferred in the trap list) is executed.

The response specifies the address contained in register one (DBGTST) followed by the contents of that address (00000001).

7a) The GO directive (deferred in the trap list) is executed.

Program execution is resumed. There is no written response.

6b) The trap is encountered and the SNAP directive (deferred in the trap list) is executed.

The response specifies the address contained in register one (DBGTST+4) followed by the contents of that address (00000002).

7b) The GO directive (deferred in the trap list) is executed.

Program execution is resumed. There is no written response.

6c) The trap is encountered and the SNAP directive (deferred in the trap list) is executed.

The response specifies the address contained in register one (DBGTST+8) followed by the contents of that address (00000003).

7c) The GO directive (deferred in the trap list) is executed.

Program execution is resumed. There is no written response.

6d) The trap is encountered and the SNAP directive (deferred in the trap list) is executed.

The response specifies the address contained in register one (DBGTST+C) followed by the contents of that address (00000004).

7d) The GO directive (deferred in the trap list) is executed.

Program execution is resumed. There is no written response.

6e) The trap is encountered and the SNAP directive (deferred in the trap list) is executed.

The response specifies the address contained in register one (DBGTST+10) followed by the contents of that address (00000005).

7e) The GO directive (deferred in the trap list) is executed.

Program execution is resumed and the value five encountered therefore a successful exit from the program is performed. SYMDB displays the status of the program at the exit address.

Debugging Session

```
1)   TSM> DEBUG DBGTST

     MPX-32 SYMBOLIC DEBUG V2.0  05/13/81,  13:00:00  TASK NAME = DBGTST
     PSW=01029828    (CC=0000)    (PC=DBGTST+28)
     REGS=00000000   00000000   00000000   00000000   ................
          00000000   00000000   00000000   00000000   ................
2)   .SHOW SYMBOLS
     GLOBAL SYMBOLS
3)   .PGM DBGTST
4)   .SHOW SYMBOLS
     SYMBOLS LOCAL TO PROGRAM #DBGTST
     START         TASKNAME     ENDSUCC       BEGTABLE      LOOP
     ENDTABLE
5)   .SET LOOP
6)   ..SN R1
7)   ..GO
8)   .GO
     TRAP @ DBGTST+30
     PSW=21029830    (CC=0100)    (PC=DBGTST+30)
     REGS=00000000   00029800   00000005   00000000   ................
          00000000   00000000   00000000   00000000   ................
6a)  !SN R1
     DBGTST                     00000001                            /..../
7a)  !GO
     TRAP @ DBGTST+30
     PSW=21029830    (CC=0100)    (PC=DBGTST+30)
     REGS=00000000   00029804   00000005   00000000   ................
          00000000   00000000   00000000   00000000   ................
6b)  !SN R1
     DBGTST+4                    00000002                           /..../
7b)  !GO
     TRAP @ DBGTST+30
     PSW=21029830    (CC=0100)    (PC=DBGTST=30)
     REGS=00000000   00029808   00000005   00000000   ................
          00000000   00000000   00000000   00000000   ................
6c)  !SN R1
     DBGTST+8                    00000003                           /..../
7c)  !GO
     TRAP @ DBQTST+30
     PSW=21029830    (CC=0100)    (PC=DBGTST+30)
     REGS=00000000   0002980C   00000005   00000000   ................
          00000000   00000000   00000000   00000000   ................
6d)  !SN R1
     DBGTST+C                    00000004                           /..../
7d)  !GO
     TRAP @ DBGTST+30
     PSW=21029830    (CC=0100)    (PC=DBGTST+30)
     REGS=00000000   00029810   00000005   00000000   ................
          00000000   00000000   00000000   00000000   ................
6e)  !SN R1
     DBGTST+10                   00000005                          /..../
```

5.3 Example 2: Searching Through a Linked List

The sample program for this debugging session contains a linked list with four data words in each node. If the nodes are linked correctly, the program successfully exits. Otherwise, the program aborts.

The debugging session shows how to establish a value for a period (.), which will be used to display the node address and data words of each node. The debugging session also shows how to build a conditional trap list which is executed only if the counter (CTR, which counts the number of nodes that are linked) is greater than two (a conditional trap list allows the user to execute the trap list only if a specified condition is met).

The following subsections contain the sample program and the debugging session which demonstrate the procedure described above.

```
DBGTST2                                                                                VER 9.3

00001                                             PROGRAM   DBGTST2
00002                                   *
00003                                   *         THIS PROGRAM WILL DEMONSTRATE THE SYMBOLIC CAPABILITIES
00004                                   *         WHICH ARE USED THROUGH THE SYMBOLIC DEBUGGER.
00005                                   *         THIS PROGRAM WILL ESTABLISH A LINKED LIST WHICH CAN
00006                                   *         THEN BE DISPLAYED THROUGH THE USE OF DEBUGGER COMMANDS.
00007                                   *
00008    00000                          R0        EQU       0
00009    00001                          R1        EQU       1
00010    00002                          R2        EQU       2
00011    00003                          R3        EQU       3
00012    00004                          R4        EQU       4
00013    00005                          R5        EQU       5
00014    00006                          R6        EQU       6
00015    00007                          R7        EQU       7
00016   P00000  00000014   P00014       LINKSTRT  ACW       NODE1          ! INITIALIZE BEGINNING OF LIST
00017   P00004  00000000                NODE4     DATAW     0              ! FORWARD POINTER = 0 => NO MORE NODE
00018   P00008  4E4F4445                          DATAW     C'NODE'        ! 1ST WORD OF DATA IN NODE 4
00019   P0000C  34202020                          DATAW     C'4   '        ! 2ND WORD OF DATA IN NODE 4
00020   P00010  00000004                          DATAW     4              ! 3RD WORD OF DATA IN NODE 4
00021   P00014  00000034   P00034       NODE1     ACW       NODE2          ! FORWARD POINTER IN NODE 1
00022   P00018  4E4F4445                          DATAW     C'NODE'        ! 1ST WORD OF DATA IN NODE 1
00023   P0001C  31202020                          DATAW     C'1   '        ! 2ND WORD OF DATA IN NODE 1
00024   P00020  00000001                          DATAW     1              ! 3RD WORD OF DATA IN NODE 1
00025   P00024  00000004   P00004       NODE3     ACW       NODE4          ! FORWARD POINTER IN NODE 3
00026   P00028  4E4F4445                          DATAW     C'NODE'        ! 1ST WORD OF DATA IN NODE 3
00027   P0002C  33202020                          DATAW     C'3   '        ! 2ND WORD OF DATA IN NODE 3
00028   P00030  00000003                          DATAW     3              ! 3RD WORD OF DATA IN NODE 3
00029   P00034  00000024   P00024       NODE2     ACW       NODE3          ! FORWARD POINTER IN NODE 2
00030   P00038  4E4F4445                          DATAW     C'NODE'        ! 1ST WORD OF DATA IN NODE 2
00031   P0003C  32202020                          DATAW     C'2   '        ! 2ND WORD OF DATA IN NODE 2
00032   P00040  00000002                          DATAW     2              ! 3RD WORD OF DATA IN NODE 2
00033   P00044  00000000                CTR       DATAW     0              ! INITIALIZE COUNTER OF NUMBER OF
00034   P00048                                    BOUND     1D             ! DOUBLEWORD ALIGN TASK NAME
00035   P00048  54535444                TASKNAME  DATAB     C'TSTDBG2 '    ! NAME OF THIS TASK
        P0004C  42473220
00036                                   *
00037                                   *    START OF PROGRAM
00038                                   *
00039   P00050  AC800000   P00000       START     LW        R1,LINKSTRT    ! LOAD IN POINTER TO LINKED LIST
00040   P00054  C8850000                          CI        R1,0           ! CHECK IF LINKED LIST IS NULL
00041   P00058  F2000069   P00068                 BNE       LOOP           ! BRANCH IF LINKED LIST IS NOT NULL
00042                                   *
00043   P0005C  AE80007A   P00078                 LW        R5,=C'ERR'     ! LOAD IN ABORT CODE
00044   P00060  AF000048   P00048                 LD        R6,TASKNAME    ! LOAD IN ABORT TASK NAME
00045   P00064  C8061056                          SVC       1,X'56'        ! ABORT THIS TASK
00046                                   *
00047   P00068  33880047   P00047       LOOP      ABM       31,CTR         ! INC. CTR. OF NUMBER OF NODES FOUND
00048   P0006C  AC400000    00000                 LW        R1,0,R1        ! LOAD IN POINTER TO NEXT NODE
00049   P00070  F2000069   P00068                 BNE       LOOP           ! BRANCH IF NOT AT END OF LIST
00050                                   *
00051   P00074  C8061055                ENDSUCC   SVC       1,X'55'        ! SUCCESSFUL END OF PROGRAM
        P00078  45525220
00052   P0007C             P00050                 END       START
*  0000  ERRORS IN DBGTST2
```

870025

5.3.2 Sample Debugging Session for program DBGTST2

All directives in the sample debugging session are numbered corresponding to a key which describes the directives and responses. Each directive is immediately followed by its response.

Directive Key

1) SYMDB is accessed by entering the DEBUG directive in response to the TSM prompt.

   SYMDB responds with its identifying message.

2) The directive to set the default program name and the accessible symbols (PGM #DBGTST2) is entered.

   The default program name is set to DBGTST2. The symbols local to DBGTST2 are now accessible. There is no written response to this directive.

3) The directive to snap the address which contains the address of the 1st node in the linked list (SN LINKSTRT) is entered.

   The response specifies the address to be snapped (DBSTST2) and the contents of that address (00029814). The address is displayed in the program name plus offset format. In this example, there is no offset. LINKSTRT is located at the start address, program name +0 (the +0 is not displayed). The contents of the snapped address (00029814) specifies the address of the first node in the link. The value of the special character period (.) is set to the value of LINKSTRT (period (.) can be entered in place of the last address specified in a memory related directive). The value of period (.) is reset each time a memory related directive is entered.

4) The directive to execute the directives in a directive file (FILE FILINK) is entered.

   The response to the FILE directive is to execute the directives in the directive file (FILINK) specified. The directives in directive file FILINK are

   ```
   SN C(.), C(.) + 0C
   SN . - 0C
   ```

5) The first directive in the directive file is the SNAP directive (SN C(.), C(.) + 0C). The range of addresses to be snapped specifies the contents of the special character period (.) (which is set to the address LINKSTART) through the contents of period (.) plus 12 decimal bytes. Prior to the execution of this directive the contents of period (.) is the address of the first node in the linked list.

   The response specifies the address of the first node in the linked list (DBGTST2+14) and the contents of the first node (00029834 4E4F4445 31202020 00000001 /...4NODE1 ..../). The period (.) is now set to the address of the third data word of the first node (00029820, which was the last address in the specified range).

6) The second and last directive in the directive file is also the SNAP directive (SN.-0C). The address to be snapped specifies the period (.) minus 12 decimal bytes (.-0C).

The response specifies the address of the first node in the linked list (DBGTST2+14) and the contents of that address (00029834). The contents (00029834) specifies the address of the second node. This directive is entered to reset the value of period (.) to the first word of the node just displayed which contains the address of the next node.

7) The FILE directive is entered.

The directives in the directive file (FILINK) are accessed.

8) The SNAP directive (specified in the directive file) is executed. The range of addresses to be snapped specifies the contents of period (.) (which is set to the address of the first word of the first node in the linked list) through the contents of period (.) plus 12 decimal bytes. Prior to the execution of this directive, the contents of period (.) is the address of the second node.

The response specifies the address of the second node (DBGTST2+34) and the contents of the second node (00029824 4E4F4445 32202020 00000002 /...$ NODE 2 ..../). The period (.) is now set to the address of the third data word of the second node (00029840).

9) The second SNAP directive (specified in the directive file) is executed. The address to be snapped specifies the period (.) minus 12 decimal bytes (.-0C).

The response specifies the address of the second node in the linked list (DBGTST2+34) and the contents of that address (00029824). The contents (00029824) specifies the address of the third node in the linked list. The special character period (.) is now set to the address of the second node (00029834).

10) The FILE Directive is entered again.

The directives in the directive file (FILINK) are accessed.

11) The SNAP directive (specified in the directive file) is executed. The range of addresses to be snapped specifies the contents of period (.) (which is set to the address of the first word of the second node in the linked list) through the contents of period (.) plus 12 decimal bytes. Prior to the execution of this directive the contents of period (.) is the address of the third node.

The response specifies the address of the third node (DBGTST+24) and the contents of the third node (00029804 4E4F4445 33202020 00000003 /.... NODE 3 ..../). The period has the value of the address of the third data word of the third node (00029830).

12) The second SNAP directive (specified in the directive file) is executed. The address to be snapped specifies the period (.) minus 12 decimal bytes (.-0C).

The response specifies the address of the third node in the linked list (DBGTST2+24) and the contents of that address (00029804). The contents (00029804) specify the address of the fourth node in the linked list. The period (.) is now set to the address of the third node (00029824).

13) The FILE directive is entered again.

The directives in the directive file (FILINK) are accessed.

14) The SNAP directive (specified in the directive file) is executed. The range of addresses to be snapped specifies the contents of period (.) (which is set to the address of the first word of the third node in the linked list) through the contents of period (.) plus 12 decimal bytes. Prior to the execution of this directive the contents of period (.) is the address of the fourth node.

The response specifies the address of the fourth node (DBGTST+4) and the contents of the fourth node (00000000 4E4F4445 34202020 00000004 /.... NODE 4 ..../). The period (.) is now set to the address of the third data word of the fourth node (00029810).

15) The second SNAP directive (specified in the directive file) is executed. The address to be snapped specifies the period (.) minus 12 decimal bytes (.-0C).

The response specifies the address of the fourth node in the linked list (DBGTST2+4) and the contents of the address (00000000). The contents (00000000) specifies the start address of the program (this indicates that there are no other nodes). The period (.) is now set to the address of the fourth node (00029804). All nodes and their data words have been examined.

16) The directive to set a trap at the beginning of a program loop (SET LOOP) is entered.

The trap is set at the address specified by the symbol name LOOP and SYMDB responds with the trap list prompt (..).

17) The directive to establish a conditional trap list (IF C(CTR) 2) is entered. The argument specifies that the trap list at location LOOP will be executed only if the contents of the local symbol CTR is greater than two. The conditional directive (IF) is deferred (along with directives 18 and 19) until the trap at LOOP is encountered.

18) The directive to snap the start address of the program (SNAP LINKSTRT) is entered. This directive is deferred until the trap is encountered.

19) The directive to execute the directives in a directive file (FILE FILINK) is entered. The directives in the directive file (FILINK) specified will be executed when the trap is encountered. The FILE directive is a trap list terminator, therefore the trap list is ended.

20) The directive to begin program execution (GO) is entered. Program execution will begin at the base $ (bits 13-31 of the user PSD or the current program counter value) because no start address was specified.

The program will begin execution. There is no written response to the GO directive.

17a) The IF directive (deferred in the trap list) is executed.

The response specifies the IF conditional statement and the current status at the address where the condition became true. The relational value of C(CTR) 2 is equal to one (the condition is true).

18a) The SNAP directive (deferred in the trap list) is executed.

The response specifies the address to be snapped (DGBTST2), which is the same address as the symbol name (LINKSTRT), and the contents of that address (00029814).

19a) The FILE directive (deferred in the trap list) is executed.

The directives in the directive file (FILINK) are accessed.

The SNAP directive (specified in the directive file) is executed. The range of addresses to be snapped specifies the contents of the period (.) (which is set to the start address of the program) through the contents of period (.) plus 12 decimal bytes. The contents of period (.) is the address of the first node in the linked list.

The response specifies the address of the first node in the linked list (DBGTST2+14) and the contents of the first node (00029834 4E4F4445 31202020 00000001 /...4NODE1 ..../). The period (.) is now set to the address of the third data word of the first node (00029820).

The second SNAP directive (specified in the directive file) is executed. The address to be snapped specifies the period (.) minus 12 decimal bytes (.-0C).

The response specifies the address of the first node in the linked list (DBGTST2+14) and the contents of that address (00029834). The contents (00029834) specifies the address of the second node. The period is now set to the address of the first node in the linked list (00029814).

Debugging Session

```
1)     TSM> DEBUG DBGTST2

       MPX-32 SYMBOLIC DEBUG V2.0 05/13/81, 13:30:00 TASK NAME = DBGTST2
       PSW=01029850   (CC=0000)   (PC=DBGTST2+50)
       REGS=00000000  00000000  00000000  00000000 ................
            00000000  00000000  00000000  00000000 ................

2)     .PGM #DBGTST2
3)     .SN LINKSTRT
       DBGTST2        00029814                                      /..../
4)     .FILE FILINK
5)     SN  C(.),C(.)+0C
       DBGTST2+14     00029834  4E4F4445  31202020  00000001 /...4NODE1..../
6)     SN . -0C
       DBGTST2+14     00029834                                      /...4/
7)     .FILE FILINK
8)     SN C(.),C(.)+0C
       DBGTST2+34     00029824  4E4F4445  32202020  00000002 /...$NODE2..../
9)     SN .-0C
       DBGTST2+34     00029824                                      /...$/
10)    .FILE FILINK
11)    SN C(.),C(.)+0C
       DBGTST2+24     00029804  4E4F4445  33202020  00000003 /....NODE3..../
12)    SN .-0C
       DBGTST2+24     00029804                                      /..../
13)    .FILE FILINK
14)    SN C(.),C(.)+0C
       DBGTST2+4      00000000  4E4F4445  34202020  00000004 /....NODE4..../
15)    SN .-0C
       DBGTST2+4      00000000                                      /..../
16)    SET LOOP
17)    ..IF C(CTR) 2
18)    ..SN LINKSTRT
19)    ..FILE FILINK
20)    .GO
       TRAP @ DBGTST2+68
17a)   IF C(CTR)    2
       'IF' VALUE =   00000001
       PSW=21029868   (CC=0100)  (PC=DBGTST2+68)
       REGS=00000000  00029804  00000000  00000000  ................
       00000000       00000000  00000000  00000000  ................
18a)   !SN LINKSTRT
       DBGTST2        00029814                                      /..../
19a)   !FILE FILINK
       SN C(.),C(.)+0C
       DBGTST2+14     00029834  4E4F4445  31202020  00000001 /...4NODE1..../
       SN .-0C
       DBGTST2+14     00029834                                      /...4/
```

Text Editor (EDIT)

MPX-32 Utilities

CONTENTS

TEXT EDITOR (EDIT)

SECTION 1 - OVERVIEW

1.1 General Description

The Text Editor (EDIT) utility builds and edits text files, merges files or parts of files into one file, copies existing text from one location to another, and performs general editing functions.

EDIT is typically used to create source files and build job control and general text files. A job control file built by EDIT can be copied directly into the batch stream using the EDIT BATCH directive.

EDIT is based on the concept of a work file in which editing directives are used. Source text may be transferred between permanent disc files and the work file. Access to source text is based on line numbers contained within text lines.

To ensure only printable characters are contained within the text, control characters embedded within the text (i.e., characters whose values are X'00' through X'1F') are replaced by blanks (X'20').

The Text Editor is cataloged with TSM OPTION LOWER (option 22) set. Files that are built with upper and lower case text must be edited with the same character conventions in effect. File names and work file codes can be entered in upper and/or lower case. They are automatically converted to upper case by EDIT. OPTION LOWER may be reset by recataloging the Text Editor utility.

EDIT recognizes 1 to 16 character file names. EDIT keywords cannot be used as file names. If a complete pathname is specified, any valid file name can be used. If only a file name is specified, the file name cannot begin with a dot or a string of digits (0 through 9) followed by a dot.

1.1.1 Accessing Files

The USE directive retrieves files from outside EDIT so editing functions can be performed. If the records of an accessed file are longer than 80 bytes, the records are truncated to 80 bytes when the file is copied into the work file. In addition, if valid line numbers do not exist in record bytes 73 through 80, data in bytes 73 through 80 are replaced by EDIT line numbers. The generated line numbers reflect the current value of DELTA, which determines line increments. If DELTA is set at the default, numbering starts at 1.0 and increments by 1, up to 9999.

If an unnumbered file is accessed that has more than 9999 records, records 10,000 and beyond are ignored. DELTA can be reset so that up to 10,000,000 records are copied into the work file. See the SET DELTA directive.

To move a file from media other than disc into EDIT, first copy the file to disc using another utility, e.g., MEDIA, then issue the EDIT USE directive.

1.2 Directive Summary

EDIT directives are summarized below and described in detail in the Directives section. Most EDIT directives can be abbreviated to three characters. Valid abbreviations are indicated by underlining. EDIT directives are keywords and cannot be used as file names. Back slashes, forward slashes, and commas are special characters used in EDIT directives and cannot be used in file names.

The break key can be used to interrupt operations in progress. Use of the break key is not recommended during SAVE and STORE operations.

Most EDIT directive parameters can be entered in any order. If directive parameter input is order-dependent, the restriction is noted in the directive description.

Directive	Function
APPEND	Appends text to end of a line or group of lines
BATCH	Copies work file or specified file into batch stream
CHANGE	Replaces a character string with another character string
CLEAR	Clears work file
COLLECT	Adds lines of text
COMMAND	Displays the last four directives performed
COPY	Copies existing text to work file
DELETE	Deletes lines
EXIT	Ends current EDIT session
INSERT	Inserts lines of text
LIST	Lists text on terminal screen
MODIFY	Allows a one-for-one replacement of characters in an existing line
MOVE	Moves lines of text within the work file
NUMBER	Renumbers lines in work file
PREFACE	Inserts characters at beginning of lines
PRINT	Copies work file or specified permanent file to SLO file
PUNCH	Copies work file or specified permanent file to SBO file
REPLACE	Replaces existing lines with new lines

RUN	Copies work file or specified file into batch stream (same as the BATCH directive)
SAVE	Saves work file compressed in permanent file
SCRATCH	Deletes a permanent file
SET DELTA	Sets an increment for line numbering
SET TABS	Modifies tab settings
SET VFA	Sets automatic verification
SET VFN	Inhibits automatic verification
SHOW	Shows current line increment, files, or tab settings
STORE	Stores work file uncompressed in permanent disc file
USE	Copies a permanent file into a cleared work file
WORKFILE	Accesses a different work file

SECTION 2 - USAGE

2.1 Accessing EDIT

EDIT can be accessed from TSM in one of three ways:

```
$EDIT
$RUN EDIT
$EXECUTE EDIT
```

$RUN EDIT is valid only from the system directory.

After being accessed, EDIT prompts for a work file code:

```
ENTER WORK FILE CODE OR CR TO TERMINATE:
```

Enter the file code. A file code is any two printable alphanumeric characters, the first of which must be alphabetic, to be associated with the work file. Lower case letters are converted to upper case for work file codes and file names. If the code supplied has been used previously in accessing EDIT, the work file with that code is retrieved. A message indicates whether the work file was cleared, saved, or changed (without a save) at the end of the last EDIT session. The EDT> prompt is then displayed.

If the code supplied has not been used before, EDIT creates a work file with that prefix.

The prompt for a work file code can be bypassed by entering the code when EDIT is accessed.

```
TSM> $EDIT filecode
EDT>
```

2.2 Logical File Code Assignments

There are two logical file codes (LFC) associated with EDIT: Input (TIN) and Output (TOT).

2.2.1 Source Input File (TIN)

The source input file contains EDIT directives. The source input file is assigned to logical file code TIN.

TIN Default and Optional Assignments

The default assignment for TIN is to logical file code UT:

```
$AS TIN TO LFC=UT
```

In the interactive mode, input is entered on the user terminal. In the batch mode, input is entered from the SYC file.

There are two optional assignments for TIN:

$$\text{\$AS TIN TO} \begin{Bmatrix} \text{pathname} \\ \text{DEV=devmnc} \end{Bmatrix}$$

pathname	is the pathname of a file containing EDIT directives
devmnc	is the device mnemonic of a device containing EDIT directives

2.2.2 Output File (TOT)

The output file contains EDIT output. The output file is assigned to logical file code TOT.

TOT Default and Optional Assignments

The default assignment for TOT is to logical file code UT:

```
$AS TOT TO LFC=UT
```

In the interactive mode, output is generated on the user terminal. In the batch mode, output is generated on the SLO device.

There are two optional assignments for TOT:

$$\text{\$AS TOT TO} \begin{Bmatrix} \text{pathname} \\ \text{DEV=devmnc} \end{Bmatrix}$$

pathname	is the pathname of a file to contain EDIT output
devmnc	is the device mnemonic of a device to contain EDIT output

2.3 Exiting EDIT

To exit EDIT from the batch and interactive modes, specify the EXIT directive. Pressing CNTRL C can also be used to exit EDIT from the interactive mode.

2.4 Lines and Line Numbers

Line numbers are decimal numbers in the range 0 to 9999.999, with a maximum of three digits after the decimal point.

The work file is limited to 10,000,000 lines if the lines are numbered 0.000, 0.001,...,9999.999, inclusive. Otherwise, the work file is limited by the highest line number, 9999.

2.4.1 Line Numbers Generated by the Editor

EDIT generates line numbers with the COLLECT, MOVE, COPY, and INSERT directives. These line numbers are generated according to the specified beginning line number (base) and optional increment. The following rules apply:

- The least significant decimal position specified for the base (or increment) is used. A specification in tenths implies lines with decimal fractions from .1 to .9. A specification in hundredths implies lines with decimal fractions from .01 to .99. A specification in thousandths implies lines with decimal fractions from .001 to .999.

- EDIT stops the current operation if the next line number to be generated already exists. As long as no existing lines are encountered, EDIT generates line numbers until it reaches an existing line number or the operation completes.

- An increment is an absolute number to add to the previous line number to obtain the next line number. It is not automatically relative to the base. For instance, to specify a base line number in tenths and increment in tenths, the increment must reflect the base decimal position. If line 2.2 is specified as the starting line and .2 is specified as an increment, EDIT generates lines 2.2, 2.4, 2.6, and 2.8. If 2 is specified as an increment, the next line number generated after 2.2 is 4.2. The default increment used in generating line numbers is 1, .1, .01, or .001 depending on the least significant position of the specified base number.

- DELTA increments apply to lines following the last line of the work file.

- A special DELTA increment (1/10 of DELTA value) is used by EDIT to generate line numbers for COLLECT and INSERT directives when the line number supplied (or implied) already exists.

2.4.2 Line Numbers at the Beginning and End of the Work File

Unless otherwise specified at the beginning of a work file, EDIT defaults to line one for the beginning of a file. Lines 0 to 0.999 are, however, valid line numbers and can be used to insert lines before line one.

The special character E specifies the last line in the file plus an increment. The value of E depends on the setting of DELTA. DELTA is normally an increment of one more than the last line in the file. DELTA can be overridden to a different significant digit and increment (e.g., .02) by using the SET DELTA directive.

2.4.3 Physical Position of Line Numbers

EDIT displays line numbers at the beginning of the line, but does not store, save, or write them there. EDIT physically writes the line numbers in columns 73 to 80 of each line of text so that when another program reads the file, it can ignore the line numbers, if desired. In addition, a directive, label, or other significant symbol then falls in the first byte as required for processing.

2.4.4 Text Listed Without Line Numbers

The work file can be listed or written without line numbers. Specifying the unnumbered option, UNN, on the appropriate directive line replaces EDIT line numbers in bytes 73 to 80 with blanks.

2.5 Addressing Techniques

There are several ways of specifying lines to be edited. These include specifying one line, specifying a group of lines, and specifying ranges of lines. The following sections describe various addressing techniques.

2.5.1 Special Characters

Several characters can be used in directives in place of specifying a line number. These characters apply to work file line numbers only:

F or FIRST	first line in the file
L or LAST	last line in the file
E or END	last line in the file (L) plus the current increment (DELTA) for adding lines at the end of the file
A or ALL	all lines in the file (F through L)
C or CURRENT	last line currently displayed on the terminal
N or NEXT	line following the current line

Because these characters and words are EDIT keywords, do not use them as file names.

When referring to files other than the current work file, specific line numbers should be stated to avoid erroneous results.

2.5.2 Line and Range Addressing

To access a specific line, enter the line number at the EDT> prompt. To access a contiguous set of lines (range), type the first line number, a forward slash, and the last line number in the range:

```
lineno

or

lineno/lineno
```

The characters F, L, and E can be used in place of a specific line number when referring to the current work file. The range implied by ALL is F/L.

The characters C and N display a single line only. Do not use them as part of a range, as they will cause incorrect results.

When only part of a range is specified with an EDIT directive, the rest of the range is implied. If a beginning line number is followed only by a forward slash:

```
lineno/
```

the last line in the range defaults to the end of the range specified in the previous directive. Likewise, if a forward slash is followed by the last line number of a range:

```
/lineno
```

the first line number defaults to the first line number from the range specified with the previous directive.

2.5.3 Groups

A group is any combination of line numbers and ranges, where each specification is separated from the next by a comma:

```
lineno/lineno,lineno,lineno/lineno
```

There can be up to 24 specifications in a group. The above example shows three specifications.

A single line number is acceptable as a group.

2.5.4 Content Identifiers

Access can be limited within a group to lines containing a specific string. To do this, enter the string enclosed in backslashes:

```
\string\
```

Only the lines containing the specified string are accessed.

A content identifier within a group applies to the entire group:

```
1/60 \TRAP\ ,75, 209/L
```

specifies that all lines within the group containing the content identifier TRAP are to be accessed. This applies to lines 1 through 60, 75, and 209 through the last line of the file.

The text specified in a content identifier must use the same upper and lower case conventions as the text was originally entered. A content identifier in upper case will not match text with the same characters if the text characters are lower case.

2.5.5 Defaults

When no lines are specified with an EDIT directive, the lines specified with the previous EDIT directive are used by default, with the following exceptions:

- If a group was specified previously, only the last line or range in the group (the specification to the right of the last comma) is taken for the current directive.

- Content identifiers do not carry over. If lines were selected within a range in the previous directive by using a content identifier, the entire last range in the group is accessed without regard to the content identifier.

2.6 Using the Break Key

EDIT responds to the break key, providing a way to end a long display, global change, or large set of deletions in progress. Response to the break is not guaranteed in a specific time frame. When the break interrupt is received, the directive being processed is terminated at the earliest safe termination point.

SECTION 3 - DIRECTIVES

3.1 Introduction

EDIT directives are detailed alphabetically in this section. Most EDIT directives can be abbreviated to three characters. Valid abbreviations are indicated by underlining. EDIT directives are keywords and cannot be used as file names. Back slashes, forward slashes, and commas are EDIT special characters and cannot be used in file names.

Most EDIT directive parameters can be entered in any order. If directive parameter input is order-dependent, the restriction is noted in the directive description.

3.2 APPEND Directive

The APPEND directive adds text to the end of an existing line. If more than one line is specified, each line is processed individually.

When an APPEND directive is entered, the first line in the group is displayed, with the cursor positioned at the end of the line. All characters typed before a carriage return are appended to the line. This continues with the next line in the group, and so on until the directive is terminated.

A tab can be produced by using the tab key or entering the EDIT tab character (a backslash). If the tab key is used, EDIT skips over the number of spaces from column one to the first tab. Using the tab character spaces to the next set tab position.

Syntax:

APPEND [group]

group	specifies the lines to be appended. If not specified, initially defaults to the first line of the file. After that, defaults to the last range specified in the previous directive.

APPEND terminates when one of the following occurs:

. The last line number in the group is processed.
. A carriage return is the first character entered on a line.

Usage:

```
EDT> LIST 1/3
   1.   THIS IS THE FIRST LINE
   2.   THIS IS THE SECOND LINE
   3.   THIS IS THE THIRD LINE
EDT> APP 1/3
   1.   THIS IS THE FIRST LINE THIS IS APPENDED TO THE FIRST LINE
   2.   THIS IS THE SECOND LINE\THE BACKSLASH IS A TAB
   3.   THIS IS THE THIRD LINE THIS IS THE END
```

```
EDT> LIST 1/3
  1. THIS IS THE FIRST LINE THIS IS APPENDED TO THE FIRST LINE
  2. THIS IS THE SECOND LINE        THE BACKSLASH IS A TAB
  3. THIS IS THE THIRD LINE THIS IS THE END
EDT> APP 1/3
  1. THIS IS THE FIRST LINE THIS IS APPENDED TO THE FIRST LINE<CR>
VOID RANGE
EDT>
```

3.3 BATCH or RUN Directive

The BATCH directive or the RUN directive copies a job file into the batch stream. The file can be the current work file or another job file in the user's directory. In either case, the file must contain job control statements for a complete job.

The file is entered into the batch stream and the EDT > prompt is returned. The OPCOM LIST directive can be used to check on the status of the job.

Syntax:

BATCH [pathname] [UNN]

pathname	specifies the pathname of a file. If not specified, defaults to the pathname of the current work file. To copy a noncurrent work file into the batch stream, use the WORK FILE directive followed by the BATCH directive.
UNN	replaces the line numbers in physical line positions 73 to 80 with blanks

Usage:

```
EDT> LIST A
    1.    $JOB TEST OWNER
    2.    $OPTION 2 3 4 5
    3.    $FORTRAN
    4.          PROGRAM MAIN
    5.               .
    6.               .
    7.               .
    8.          END
    9.    $CATALOG
   10.    AS 5 TO DATA
   11.    AS 6 TO SLO
   12.    BUILD LM.TEST NOM
   13.    $EOJ
   14.    $$
EDT> BATCH
EDT> X
TSM> !LIST
          .
          .
          .
2F000193     FORTRAN     OWNER        JOB.0022    62    SWIO     IN
TSM>
```

3.4 CHANGE Directive

The CHANGE directive replaces an existing string with another string. Existing characters following the replacement string are adjusted left or right to compensate for replacing a string of one length with one of a different length.

Lines containing tabs are also shifted to the left or right. To maintain the original alignment of tabbed text when the existing string and replacement string are different lengths, blanks can be included in the smaller string.

As each line is changed, the resulting line is displayed. Changed lines scroll up on the screen until the screen is full. EDIT then pauses until a carriage return is entered to continue the changes, or another character is entered to terminate the directive.

Syntax:

```
CHANGE [group]\string\\[newstring]\[NOLIST]
```

group	specifies the lines to be modified. If not specified, defaults to the last range specified in the previous directive.
string	specifies the string to replace. The string must be enclosed in backslashes.
newstring	specifies the string to replace the existing string. The new string must also be enclosed in backslashes. To delete the existing string, enter three consecutive backslashes instead of specifying a new string.
NOLIST	inhibits the automatic display of lines as they are changed.

CHANGE terminates when one of the following occurs:

. All lines in the range are processed.
. The break key is pressed.
. The response to an error message is N.
. A character other than a carriage return is entered in response to the ENTER <CR> FOR MORE message.

Usage:

```
EDT> LIST 1/4
   1.   THIS IS A TEST
   2.   TO SHOW HOW
   3.   THE CHANGE DIRECTIVE
   4.   WORKS
EDT> CH \S\\SS\1/4
   1.   THISS ISS A TESST
   2.   TO SSHOW HOW
   3.   THE CHANGE DIRECTIVE
   4.   WORKSS
EDT> LIST 6
   6.   THIS LINE ISN'T                TABBED
EDT> CHA 6\ISN'T\\IS\
EDT> LIST 6
   6.   THIS LINE IS                TABBED
```

3.5 CLEAR Directive

The CLEAR directive clears the contents of the work file. This directive is used before the USE directive when accessing a disc file or before building text into a new file interactively with the COLLECT directive.

The contents of the work file are not saved unless a SAVE or STORE directive is used before CLEAR.

Syntax:

CLEAR

Usage:

```
EDT> LIST A
   1.    THIS IS A TEST
   2.    TO SHOW HOW
   3.    THE CLEAR
   4.    DIRECTIVE WORKS
EDT> SAVE MYFILE
EDT> CLE
EDT> LIST A
VOID RANGE
EDT> USE NEWFILE
EDT> LIST A
   1.    NEWFILE IS
   2.    NOW IN THE
   3.    WORK FILE
EDT> USE BIGFILE CLEAR
EDT> LIST 1/3
   1.    THIS IS THE FIRST LINE
   2.    OF A FILE
   3.    NAMED BIGFILE
```

3.6 COLLECT Directive

The COLLECT directive enters new lines of text in the work file.

When a COLLECT directive is entered, the line number where collection begins is displayed as a prompt. Enter the text followed by a carriage return. From this point on, line numbers are generated automatically. The next sequential line number is displayed. Continue entering lines. A blank followed by a carriage return creates a blank line.

Syntax:

```
  COLLECT [lineno [/lineno]] [BY increment]
```

lineno	specifies the line number where collection begins. If the line number exists, EDIT adds .1 of DELTA, and begins collecting at that line number. If not specified, the default is the line number following the last line number collected or inserted. If this is the first COLLECT of the editing session and no INSERT directive has been entered, the lines are collected at the end of the file. The default increment is used unless overridden. To add lines to the beginning of a file, specify a starting line number between 0.0 and 0.999. To add lines to the end of a file, specify E or L.
/lineno	specifies the last line number to be added. If this line number exists, text is collected up to this line number and an error message is displayed.
increment	specifies an absolute number, .001 to 9999.999, to add to each line number to generate the next line number. If not specified, the default increment is DELTA or .1 of DELTA depending on the least significant digit used in the line number specification.

COLLECT terminates when one of the following occurs:

. The line number at the end of a specified range is entered.
. The line number in a collection sequence already exists.
. The first character entered on a line is a carriage return.
. The break key is pressed.

Usage:

```
EDT> LIST
VOID RANGE
EDT> COL
   1.    THIS IS THE FIRST LINE
   2.    THIS IS THE SECOND AND LAST
   3.    <CR>
EDT> COL 0 BY .1
   0.    THIS IS THE FIRST LINE NOW
   0.1   AND THIS IS THE SECOND
   0.2   <CR>
EDT> COL E
   3.    THIS IS THE LAST LINE NOW
   4.    <CR>
EDT> A
   0.    THIS IS THE FIRST LINE NOW
   0.1   AND THIS IS THE SECOND
   1.    THIS IS THE FIRST LINE
   2.    THIS IS THE SECOND AND LAST
   3.    THIS IS THE LAST LINE NOW
EDT> COL L
   3.    THIS IS THE LAST LINE NOW
   4.    USE OF L DISPLAYS LAST LINE BEFORE COLLECTION BEGINS
   5.    <CR>
EDT>
```

3.7 COMMAND Directive

The COMMAND directive displays the last four EDIT directives issued to EDIT. If less than four directives were issued to EDIT, only those issued are displayed.

Syntax:

COMMAND

Usage:

```
EDT> LIST 8
   8.    THE LAST LINE
EDT> DEL 3/7
EDT> APP 1
   1.    FIRST LINE OF TEST
EDT> CHA 2 \I\\O\
   2.    SECOND LONE
EDT> COM
LIST 8
DEL 3/7
APP 1
CHA 2 \I\\O\
EDT>
```

3.8 COPY Directive

The COPY directive copies existing lines of text to the work file. The lines to copy can be in the work file or in a file saved or stored previously.

To copy specific lines from a file, the file must be saved or stored with line numbers. Line numbers are not necessary if the whole file is to be copied. If lines are coming from a file other than the work file, the name of the file and the message *FILE* are displayed.

The original lines are not deleted. To delete lines in one part of a work file and copy them to a different part, use the MOVE directive.

Files can be copied that were not built by EDIT or accessed and saved by EDIT USE and SAVE or STORE directives. To copy such files, access the file with the USE directive. This directive attaches line numbers in a form acceptable to EDIT. Save or store the file on disc.

Syntax:

COPY [group] [[FROM] pathname] [TO lineno] [BY increment] [LIST]

group	specifies the numbers of the lines to be copied into the work file. If not specified, the default is all lines in the file.
pathname	specifies the pathname of a file from which lines are to be copied. A pathname must be specified to copy lines from a file other than the current work file. If not specified, the default is the current work file. Using the keyword FROM is optional.
lineno	specifies the work file line number where copying begins. The least significant digit in the specified line number determines the line numbers EDIT generates for the text being copied. The line number must not currently exist. If a beginning line number is not specified, EDIT defaults to E, the line following the last line in the work file. If collecting at the end of the file, line numbers for copied text will reflect the least significant position and increment of the DELTA value for line numbers. See the SET DELTA directive.
increment	specifies an absolute number, .001 to 9999.999, to add to each line number to generate the next line number. If not specified, the default increment is DELTA or .1 of DELTA depending on the least significant digit used in the line number specification.
LIST	specifies that lines are to be displayed as they are copied. If not specified, the lines are not displayed.

COPY terminates when one of the following occurs:

. A line number is generated which already exists.
. The break key is pressed.
. The specified lines are copied.

Usage:

```
EDT> LIST 1/4
    1.    LINE ONE
    2.    LINE TWO
    3.    LINE THREE
    4.    LINE FOUR
EDT> COP 2/3 TO 5
EDT> LIST 1/6
    1.    LINE ONE
    2.    LINE TWO
    3.    LINE THREE
    4.    LINE FOUR
    5.    LINE TWO
    6.    LINE THREE
EDT> COP 1 TO 4.1
EDT> LIST 4/5
    4.    LINE FOUR
    4.1   LINE ONE
    5.    LINE TWO
```

3.9 DELETE Directive

The DELETE directive deletes lines of text from the work file.

Syntax:

DELETE [group] [LIST]

group	specifies the numbers of the lines to be deleted. Content identifiers can be used to identify lines to be deleted. If not specified, initially defaults to the first line of the file. After that, defaults to the last range specified in the previous directive.
LIST	specifies the lines are to be displayed as they are deleted. If not specified, the lines are not displayed.

DELETE terminates when one of the following occurs:

- The specified line numbers are deleted.
- The break key is pressed.

Usage:

```
EDT> LIST 1/5
    1.    LINE ONE
    2.    LINE TWO
    3.    LINE THREE
    4.    LINE FOUR
    5.    LINE FIVE
EDT> DEL 2/4
EDT> LIST 1/5
    1.    LINE ONE
    5.    LINE FIVE
```

3.10 EXIT Directive

The EXIT directive exits EDIT and returns control to TSM.

Syntax:

EXIT

3.11 INSERT Directive

The INSERT directive adds one or more new lines of text to a file. INSERT is similar to COLLECT, except INSERT adds one line at a time. COLLECT adds successive lines.

When an INSERT directive is entered, the line number where insertion begins is displayed as a prompt. Enter the text of the line followed by a carriage return.

Syntax:

INSERT [group] [BY increment]

group	specifies the numbers of the lines to be inserted. A range cannot include an existing line number. If not specified, the default is the line number following the last line number entered. If this is the first COLLECT or INSERT of the editing session, lines are added to the end of the file. The default increment will be used unless overridden. To add lines to the beginning of a file, specify a starting line number between 0.0 and 0.999. To add lines to the end of a file specify E.
increment	specifies an absolute number, .001 to 9999.999, to add to each line number to generate the next line number. If not specified, the default increment is DELTA or .1 of DELTA depending on the least significant digit used in the line number specification.

INSERT terminates when one of the following occurs:

- A carriage return is the first character entered on a line.
- The last line number specified is processed.
- The line number in the insertion sequence already exists.
- The break key is pressed.

Usage:

```
EDT> LIST 1/3
   1.    LINE ONE
   2.    LINE TWO
   3.    LINE THREE
EDT> INS 1 BY .5
   1.5   THIS LINE HAS BEEN INSERTED
EDT> LIST 1/3
   1.    LINE ONE
   1.5   THIS LINE HAS BEEN INSERTED
   2.    LINE TWO
   3.    LINE THREE
```

3.12 LIST Directive

The LIST directive displays lines from either the current work file or a permanent file. The LIST directive is the default when no directive is supplied in response to the EDT prompt.

Syntax:

```
    [LIST]  [group]  [ [FROM] pathname]  [UNN]  [SYS]
```

group	specifies the numbers of the lines to be listed. Content identifiers can be used to identify lines to be listed. If not specified, initially defaults to the first line of a file. After that, defaults to the last range specified in the previous directive. To list an entire work file, use LIST ALL, ALL, or A.
pathname	specifies the pathname of a file. If not specified, defaults to the current work file.
UNN	replaces the line numbers in physical line positions 73 to 80 with blanks
SYS	specifies the system volume and system directory. This specification overrides any pathname component specified.

LIST terminates when one of the following occurs:

. All specified lines have been listed.
. The break key is pressed.

Usage:

```
EDT> LIST L
  30.    LAST LINE
EDT> LIST 25/30
  25.    LINE 25
                .
                .
                .
  30.    LAST LINE
EDT>
```

3.13 MODIFY Directive

The MODIFY directive changes an existing line by spacing past wanted characters and replacing unwanted characters. A circumflex (^) can be used to replace a character with a blank. When MODIFY is entered, one line at a time from the specified range is displayed for modifications.

Replacement strings must be equal to or less than the number of characters being changed. For replacement strings larger than the original string, global modifications, or circumflex insertion, use the CHANGE directive.

Syntax:

MODIFY [group]

group	specifies the numbers of the lines to be modified. Content identifiers can be used to identify lines to be modified. If not specified, defaults to the last range specified in the previous directive.

MODIFY terminates when one of the following occurs:

. The last line specified is processed.
. A carriage return is the first character entered on a line.
. The break key is pressed.

Usage:

```
        EDT> MOD 1/3
            1.    THERE WILL BE A SHORTTMEETING ON TUESDAY.
1           1.                     ^<CR>
            1.    THERE WILL BE A SHORT MEETING ON TUESDAY.
            2.    PLEASE PLAN OT ATTEND.
2           2.               TO<CR>
            2.    PLEASE PLAN TO ATTEND.
            3.    ATTENDANCE WILL NOT BE NOTED.
3           3.    <CR>
        EDT>
```

Comments:

1	Space past good characters, enter circumflex (^) to replace T with a space.
2	Space past good characters, enter correction.
3	Immediate carriage return leaves line as is and returns the EDT > prompt.

3.14 MOVE Directive

The MOVE directive moves existing lines of text from one part of the work file to another. Each line is deleted from its original position after it has been moved successfully to the new position.

Syntax:

```
MOVE [group] [TO lineno] [BY increment] [LIST]
```

group	specifies the numbers of the lines to be moved. Content identifiers can be used to identify lines to be moved. If not specified, defaults to the last range specified in the previous directive.
lineno	specifies the work file line number where the first line of text is moved. The least significant digit in the specified line number determines the line numbers EDIT generates for the text being moved. The line number must not currently exist. If a beginning line number is not specified, EDIT defaults to E, the line following the last line in the work file. Line numbers for text that is moved reflect the least significant position and increment of the DELTA value for line numbering unless an increment is specified. See the SET DELTA directive.
increment	specifies an absolute number, .001 to 9999.999, to add to each line number to generate the next line number. If not specified, the default increment is DELTA or .1 of DELTA depending on the least significant digit used in the line number specification.
LIST	specifies lines are to be displayed as they are moved. If not specified, the lines are not displayed.

Usage:

```
EDT> LIST 1/5
    1.    LINE ONE
    2.    LINE TWO
    3.    LINE THREE
    4.    LINE FOUR
    5.    LINE FIVE
EDT> MOVE 1,2 TO 6
EDT> LIST 1/6
    3.    LINE THREE
    4.    LINE FOUR
    5.    LINE FIVE
    6.    LINE ONE
    7.    LINE TWO
EDT> MOV 3/5 TO 5.1 BY .1
    5.1   LINE THREE
    5.2   LINE FOUR
    5.3   LINE FIVE
    6.    LINE ONE
    7.    LINE TWO
```

3.15 NUMBER Directive

The NUMBER directive renumbers all lines in the work file using the specified decimal position and increment.

Syntax:

```
NUMBER [lineno] [BY increment]
```

lineno	specifies the line number for the first line in the work file. If not specified, defaults to line one. Work file line numbers reflect the increment of the DELTA value for line numbers unless overridden.
increment	specifies an absolute number, .001 to 9999.999, to add to each line number to generate the next line number. If not specified, the default increment is DELTA.

Usage:

```
EDT> LIS 3/4
   3.    THIS IS LINE NUMBER THREE
   3.01  THIS LINE HAS BEEN INSERTED
   4.    THIS IS LINE NUMBER FOUR
EDT> NUM
EDT> LIS 3/L
   3.    THIS IS LINE NUMBER THREE
   4.    THIS LINE HAS BEEN INSERTED
   5.    THIS IS LINE NUMBER FOUR
     <break>
EDT>
```

3.16 PREFACE Directive

The PREFACE directive inserts one or more characters at the beginning of an existing line. Additions are made character-by-character, resulting in a right shift in the existing line.

If more than one line is to be prefaced, the first line in the specified group is displayed. EDIT reissues the line number as a prompt. Type the new string, followed by a carriage return. Existing characters in the line are right shifted with the new string at the beginning of the line. The rest of the line remains unchanged. The next line in the group is displayed for modification. This continues until the last line in the group is processed.

Syntax:

```
PREFACE [group]
```

group	specifies the numbers of the lines to be modified. Content identifiers can be used to identify lines to be prefaced. If not specified, defaults to the last range specified in the previous directive.

PREFACE terminates when one of the following occurs:

. The last number specified is processed.
. A carriage return is the first character entered on a line.
. The break key is pressed.

Usage:

```
EDT> PRE 1,15,20
   1.    ON MODIFYING A LINE
   1.    UP<CR>
   1.    UPON MODIFYING A LINE
  15.    MODIFIED LINE SHIFTS
  15.    A <CR>
  15.    A MODIFIED LINE SHIFTS
  20.    TO BYPASS MODIFICATION
  20.    <CR>
VOID RANGE
EDT>
```

3.17 PRINT Directive

The PRINT directive prints the current work file or another file on the device assigned for system SLO files.

Syntax:

```
PRINT [pathname] [UNN]
```

pathname	specifies the pathname of a file. If not specified, defaults to the current work file.
UNN	replaces the line numbers in physical positions 73 to 80 with blanks

3.18 PUNCH Directive

The PUNCH directive sends the current work file or another file to the device assigned for system SBO files.

Syntax:

PUNCH [pathname] [UNN]

pathname	specifies the pathname of a file. If not specified, defaults to the current work file.
UNN	replaces the line numbers in physical positions 73 to 80 with blanks

3.19 REPLACE Directive

The REPLACE directive replaces existing lines in the work file with different lines of text.

If more than one line is to be replaced, the first line in the specified range is displayed. EDIT reissues the line number as a prompt. Type the replacement line, followed by a carriage return. The next line in the range is displayed for replacement. This continues until the last line in the group is processed.

To replace an existing line with all blanks, type a blank space followed by a carriage return.

Syntax:

REPLACE [group]

group	specifies the numbers of the lines to be replaced. Content identifiers can be used to identify lines to be replaced. If not specified, defaults to the last range specified in the previous directive.

REPLACE terminates when one of the following occurs:

. The last line specified is processed.
. A carriage return is the first character entered on a line.
. The break key is pressed.

Usage:

```
EDT>  REP 24
  24.    The replace directive replaces existing lines.
  24.    The REPLACE directive is used to replace existing lines.
EDT>
```

3.20 RUN Directive

The RUN directive copies a file into the batch stream. Refer to the BATCH directive.

3.21 SAVE Directive

The SAVE directive writes a copy of the current work file on disc as a blocked permanent file. The text is compressed; consecutive blanks are replaced with a string indicating the number of blanks compressed. The work file remains intact.

To save a work file under a file name that already exists in your directory, delete the existing contents of the permanent file by either specifying the SCRATCH parameter or responding affirmatively to the prompt.

Syntax:

SAVE [pathname] [SYS] [SCRATCH]

pathname	specifies the pathname of a file. If not specified, defaults to the file name used in the most recent SAVE, STORE, or USE directive, unless that file was in a different directory. If the file was in a different directory, the work file is copied to a file of the same name in the current directory.
SYS	specifies the system volume and system directory. This parameter overrides any pathname component specified.
SCRATCH	clears the contents of the file named with the SAVE directive before the work file is copied to that file space. If not specified and the file already exists, a prompt for the scratch function is displayed.

When the save is complete, the following message is displayed:

```
    volume (directory) filename  ownername  size  type
    volume (directory) ii*WRKFL  ownername  size  type
    ii*WRKFL SAVED xx LINES
```

volume	is the name of the current working volume
directory	is the name of the current working directory
filename	is the name of the file saved
ownername	is the owner name associated with the file
size	is the file size
type	is the file type
ii	is the work file code
xx	is the number of lines in the work file

Usage:

```
EDT> SAVE TEST
TB00            (SMITH          )TEST            SMITH          4 ED
TB00            (SMITH          )SS*WRKFL        SMITH         80 FE
SS*WRKFL SAVED          8 LINES
EDT>
```

Work file SS is saved in the permanent file TEST in the current working volume and directory.

3.22 SCRATCH Directive

The SCRATCH directive deletes a file. The file name and its contents are removed from the disc. The file name is removed from the disc directory.

Syntax:

```
SCRATCH [pathname] [SYS]
```

pathname	specifies the pathname of a file. If not specified, defaults to the file name used in the most recent SAVE, STORE, or USE directive.
SYS	specifies the system volume and system directory. This parameter overrides any pathname component specified.

If no user file is found, but a system file is found with the specified parameter and SYS was not specified, the following message is displayed:

```
SYSTEM FILE FOUND BY THAT NAME. CORRECT FILE (Y OR N)?
```

A Y (Yes) response scratches the file. A N (No) response terminates the scratch operation.

Usage:

```
EDT> SCR AAA
EDT> SCR BAR SYS
EDT>
```

User file AAA in the current working volume and directory is scratched. File BAR in the system volume and system directory is scratched.

```
EDT> SCR @TB00(TRAINING)INDEX
```

User file INDEX in volume TB00 and directory TRAINING is scratched.

3.23 SET DELTA Directive

The SET DELTA directive specifies an increment other than 1.0 to be used for line numbering. The specified DELTA increment remains in effect only for the current editing session. DELTA is reset to 1.0 after the current editing session.

Syntax:

```
SET DELTA [increment]
```

increment — specifies an absolute number, .001 to 9999.999, to add to each line number to generate the next line number.

If 0 is entered as the increment value, the following message is displayed:

```
ERROR         0.
```

3.24 SET TABS Directive

The SET TABS directive modifies tab positions. In MPX-32, the M.KEY file can contain tab settings for each logon owner name. TSM defaults to these tabs if they are set. If no tabs are set in M.KEY, TSM sets system tabs when EDIT is entered.

Default tabs are set in columns 10, 20, 36, 41, 46, 51, 61, and 71.

The most recent tabs set with SET TABS remain in effect until TSM is exited. When TSM is exited, the SET TABS are not saved. When a logon is performed, the default tabs are set.

When typing text with tabs, a tab character can be produced by a CNTRL I or the tab key. This is interpreted by the TSM device handlers and replaced by the appropriate number of blanks. The cursor is adjusted by echoing the spaces to the terminal. This allows the tabbed spacing to be seen on the screen as text is entered.

Tabs can also be entered by using the tab character. The default tab character is a backslash (\). The character used to define tab positions in the SET TABS directive can be either a backslash or a user-defined character. The last tab character defined in SET TABS is the one to use when entering a tabbed record, unless CNTRL I is used. The tab character defined remains in effect until EDIT is exited. If EDIT is entered again before exiting TSM, the tab positions defined in the SET TABS directive remain in effect and the tab character defaults to a backslash. If EDIT and TSM are both exited, the default tab character and positions are set at the next logon.

The tab character is removed from the text when it is interpreted, so no special treatment of tab characters is required from subsequent processors such as the Macro Assembler. If a tab character is entered when all tabs have been used, a blank is inserted in place of a tab. If CNTRL I is entered when all tabs have been used, no data is inserted.

Syntax:

```
SET TABS
```

EDIT displays the current tab settings, then prompts for new tab positions by locating the cursor in column two. Type blanks for nontab positions and any character desired in tab positions (a maximum of eight tab positions can be set at one time). The tab character supplied overrides the backslash. If different characters are used for tab specifications, the last character typed is the tab character. Any tab that is not specified explicitly is not set. If more than eight tab positions are entered, only the first eight are recognized and set.

To change the tab delimiter but keep tab settings in their current positions, enter the following directive:

```
SET TC
```

EDIT responds:

```
CHAR=
```

Enter the desired tab delimiter character. To return to the default tab delimiter, enter the following directive:

```
SET OTC
```

To prohibit the use of a tab character, enter the following directive:

```
SET NTC
```

This is useful when submitting directives in batch.

Usage:

```
EDT> SET TABS

TABS  =12345678 \ 2345678 \ 2345678 3 2345\78 4\2345\78 5\2345...
SET   =         A       B  #                 #        B       #...
TABS  =12345678 1#234567# 2 2#45678 3 234#678 4 #345678 5 234#...
```

The tab character becomes the symbol #.

To return to the EDIT default tab settings, enter one of the following directives:

```
SET   OLD
      OLDTAB
      OLDTABS
```

3.25 SET VERIFICATION Directive

The SET VERIFICATION directive sets or inhibits the automatic verification of a file before a NUMBER, SAVE, or STORE operation is performed. If not specified, the default is to inhibit automatic verification.

Syntax:

$$\text{SET} \begin{Bmatrix} \text{VFA} \\ \text{VFN} \end{Bmatrix}$$

VFA	enables the automatic verification of a file before any NUMBER, SAVE, or STORE operation is performed. VFA remains in effect until a SET VFN is entered or EDIT is exited.
VFN	inhibits the automatic verification of a file before any NUMBER, SAVE, or STORE operation is performed.

Response:

When a SET VFA directive is used, the following message is displayed when a NUMBER, SAVE, or STORE directive is entered:

```
     VERIFYING BEFORE NUM/SAV/STO.  PLEASE WAIT.
```

The number, save, or store operation is performed after verification is complete.

Usage:

```
EDT> SET VFA
EDT> NUM
VERIFYING BEFORE NUM/SAV/STO.  PLEASE WAIT
```

3.26 SHOW Directive

The SHOW directive displays the following status information:

. Current increment setting of DELTA.
. Current tab settings.
. Current files in user directory.

Syntax:

$$\underline{\text{SHOW}} \left[\begin{Bmatrix} \text{pathname} \\ \underline{\text{DELTA}} \\ \underline{\text{FILES}} \\ \underline{\text{TABS}} \end{Bmatrix} \right]$$

If no parameters are specified, EDIT displays the name of the file accessed in the current editing session if the file is in the current directory, plus the status of the current work file.

pathname	is the pathname of a file
DELTA	displays DELTA
FILES	displays names of permanent disc files in the user directory
TABS	displays tab settings

For SHOW, the following message is displayed:

```
volume (directory) filename  ownername  size  type
volume (directory) ii*WRKFL  ownername  size  type
ii*WRKFL status xx LINES
```

volume	is the name of the current working volume
directory	is the name of the current working directory
filename	is the name of the file in the work file (if any)
ownername	is the owner name associated with the file
size	is the file size
type	is the file type
ii	is the work file code
status	is one of the following: CHANGED - the file has been edited but the edited version has not been saved. CLEAR - the file has been cleared from the work file and another disc file has not been copied into the work file. SAVED - that the work file has been stored or saved. No editing has been performed on the file since the last STORE or SAVE.
xx	is the number of lines in the work file

When the FILES parameter is specified, EDIT lists each file in the current working directory. For each file, the following information is displayed:

. volume
. directory
. file name
. owner name associated with the file
. file size
. file type

When a full screen of files is displayed, enter a carriage return to continue the listing or any other key to terminate the directive. The display of file names can also be terminated with the break key.

When the TABS parameter is specified, positions 1 through 72 of the tab line are displayed. The current tab character is displayed in the positions corresponding to the current tab stops.

When the DELTA parameter is specified, the DELTA increment value used to generate lined numbers is displayed.

Usage:

```
     EDT> SHOW FILES
     TB00           (SMITH      )TEST                  SMITH   4 ED
     TB00           (SMITH      )SS*WRKFL              SMITH  80 FE
1      <break>
     EDT> SHOW
     TB00           (SMITH      )SS*WRKFL              SMITH   8 FE
2    SS*WRKFL CHANGED        4 LINES
3    EDT> SHO DELTA
        0.001(DELTA)
     EDT>
```

Comments:

1 The break key is used to terminate the listing of files in midscreen. FE designates a work file, ED designates a saved file, and EE designates a stored file.

2 No files have been saved or stored in the current session. Only the status of the work file is displayed.

3 The increment used to generate line numbers is .001.

3.27 STORE Directive

The STORE directive writes a copy of the current work file on disc as a blocked, uncompressed, permanent file. The work file remains intact.

To store a work file under a file name that already exists in your directory, delete the existing contents of the permanent file by either specifying the SCRATCH parameter or responding affirmatively to the prompt. This prevents inadvertent writes over existing files.

Syntax:

```
    STORE [pathname] [SYS] [UNN] [SCRATCH]
```

pathname	specifies the pathname of a file. If not specified, defaults to the file name used in the most recent SAVE, STORE, or USE directive, unless that file was in a different directory. If the file was in a different directory, the work file is copied to a file of the same name in the current directory.
SYS	specifies the system volume and system directory. This parameter overrides any pathname component specified.
UNN	replaces the line numbers in physical positions 73 to 80 with blanks
SCRATCH	clears the contents of the file named with the STORE directive before the work file is copied to that file space. If not specified and the file already exists, a prompt for the scratch function is displayed.

When the store is complete, the following message is displayed:

```
    volume (directory) filename  ownername  size  type
    volume (directory) ii*WRKFL  ownername  size  type
    ii*WRKFL STORED xx LINES
```

volume	is the name of the current working volume
directory	is the name of the current working directory
filename	is the name of the file stored
ownername	is the owner name associated with the file
size	is the file size
type	is the file type
ii	is the work file code
xx	is the number of lines in the work file

Usage:

```
EDT> STO RRR
TB00         (SMITH        )RRR                     SMITH    4 ED
TB00         (SMITH        )SS*WRKFL                SMITH   80 FE
SS*WRKFL STORED         8 LINES
EDT>
```

Work file SS is stored as a numbered file in the permanent file RRR in the current working volume and directory.

3.28 USE Directive

The USE directive copies a permanent disc file into the current work file.

The file can be one that has been created or edited previously using EDIT, or one not edited previously. To use a file not edited previously that contains more than 9999 physical records, reset DELTA to an increment less than 1.0. If DELTA is not reset, records 10,000 and up will not be brought into the work file.

Syntax:

```
    USE pathname  [CLEAR  ] [SYS]
                  [SCRATCH]
```

pathname	specifies the pathname of a file. The records on the file must be blocked.
CLEAR	clears the current work file before copying the contents of the specified file into the work file. The function of this parameter is identical to that of the CLEAR directive.
SCRATCH	is the equivalent of CLEAR
SYS	specifies the system volume and system directory. This parameter overrides any pathname component specified.

If a user file and a system file have the same pathname and SYS is not specified, the user file is brought into the work file. If SYS is not specified and no user file is found, but a system file is found with the pathname specified, the following message is displayed:

```
SYSTEM FILE FOUND BY THAT NAME. CORRECT FILE (Y OR N)?
```

If the specified file is not a normal source/text file, that is, it does not have a file code type of EE, ED, or C0, the following message is displayed:

```
FILE TYPE NOT ED, EE, OR C0, PROCESS IT (Y OR N)?
```

If the work file has been changed, but not saved or stored, and CLEAR has not been specified, EDIT prompts:

```
CLEAR = N
```

Enter N (No) or a carriage return to terminate the directive. Enter Y (Yes) to continue. The prompt defaults to No to protect against the inadvertent clearing of unsaved edited text. To determine the status of the work file, terminate the USE directive and use the SHOW directive.

Usage:

```
EDT> USE RRR
EDT> USE XXX SYS CLE
EDT>
```

User file RRR is brought into the work file, then system file XXX is brought into the work file.

3.29 WORKFILE Directive

The WORKFILE directive accesses a work file other than the one currently in use. The current work file is written back on disc as is. The status message is also retained. The specified work file is created or retrieved. Work files are named:

ii*WRKFL

ii	is the specified work file code. The work file code is any two alphanumeric characters, the first of which must be alphabetic, to be associated with the work file.

Syntax:

WORKFILE [filecode]

filecode	specifies a work file code

If WOR is entered without a file code, EDIT prompts:

```
ENTER WORK FILE CODE OR CR TO TERMINATE
```

Enter a valid work file code or terminate the directive with a carriage return.

If the file code entered identifies an existing work file, EDIT writes the current work file on disc and retrieves the new specified work file. The status of the new work file is displayed.

If the file code entered does not identify an existing work file, EDIT creates a new work file.

Usage:

```
     EDT>  WOR SS
     TB00           (SMITH        )SS*WRKFL                SMITH 80 FE
1    SS*WRKFL SAVED          4 LINES
     EDT>
```

Comments

1 The status of work file SS is SAVED.

SECTION 4 - ERRORS AND ABORTS

4.1 Abort Codes

The following are EDIT abort codes and their descriptions. Only the codes are displayed when an abort occurs.

Code	Description
ED01	User terminal I/O hardware error
ED02	Internal line linkage invalid
ED03	Reserved
ED04	Internal logic error

4.2 Error Messages

The following are EDIT error messages and their solutions.

ILLEGAL USE OF AN EDIT RESERVED KEYWORD

ILLEGAL PARAMETER

MISSING PARAMETER

ILLEGAL USE OF /

ILLEGAL USE OF TO

ZERO NUMBER DETECTED ON BY COMMAND

> The increment specified by the BY parameter is zero. This is not allowed.

TOO LONG, GO = Y

> The line exceeds 72 characters. Enter Y or press the carriage return to keep the first 72 characters of the changed line. Enter N to keep the line as it was before the directive was attempted.

lineno NOT PROCESSED

> Processing of the last directive terminated at the line number indicated. To process the remaining lines, reenter the directive, specifying a smaller increment.

LAST LINE OVERFLOW TRY SMALLER DELTA

EDITOR FOUND UNPRINTABLE CHARS???

???NOTHING IN WORK FILE TO BE SAVED!

EXPANDING WORK FILE. PLEASE WAIT

COULD NOT SCRATCH WORK FILE

> There is a possible problem with the MPX-32 M.DELR system service. Otherwise, check the file's access rights.

INCORRECT POINTERS IN WORKING FILE

> Clear the work file and reuse the file to be changed. Any changes that were made and not saved are lost.

COULD NOT ACCESS WORK FILE

> Another user may be using the specified work file.

IS NOT A VALID SOURCE FILE

FILE IS IN USE BY ANOTHER

INVALID FILENAME OR WORK FILE NAME CANNOT BE SPECIFIED HERE

FILE IS TOO BIG FOR EDITOR TO HANDLE

UNABLE TO ALLOCATE FILE

NO CURRENT DEFAULT FILENAME - PLEASE SPECIFY A FILENAME

COULD NOT SCRATCH FILE SPECIFIED

> Check the file's access rights.

FILE TYPE NOT ED, EE, OR C0. PROCESS IT (Y,N)?

> The specified file is not an EDIT saved, EDIT stored, or spooled output file.

SCRATCH FAILED

> There is a possible problem with the MPX-32 M.DELR system service. Otherwise, check the file's access rights.

FILE ACCESS WAS DENIED

> The file is either sharable and exclusively locked, or nonsharable and allocated.

DISC FILE SPACE UNAVAILABLE

NO DISC SPACE AVAILABLE FOR WORK FILE

CREATION FAILED - REASON xx

COMMAND IGNORED - JOB QUEUE FULL

BAD COMPRESSED RECORD DETECTED

Try to reuse the file.

IS RESTRICTED FROM BATCH BY M.KEY

READ ERROR

There is a possible hardware or I/O problem outside EDIT.

WRITE ERROR

There is a possible hardware or I/O problem outside EDIT.

I/O ERROR

SEQUENCE ERROR - LINE NOT INCLUDED

There is a possible problem with the file being used. Try to reuse the file.

BAD LINE COUNT IN WORK FILE - RECOUNTING

Verification of a file being used failed. The line count is being reverified.

SEQ ERR - BAD SECTOR LINKAGE - INTEGRITY UNCERTAIN

Verification of a file being used failed. Clear the work file and reuse the file.

INVALID SECTOR NUMBER - INTEGRITY UNCERTAIN

Verification of a file being used failed. Clear the work file and reuse the file.

HEADER SEQ INVALID - INTEGRITY UNCERTAIN

Verification of a file being used failed. Clear the work file and reuse the file.

LINE COUNT ERROR - INTEGRITY UNCERTAIN

Verification of a file being used failed. Clear the work file and reuse the file.

POINTER INVALID - INTEGRITY UNCERTAIN

Verification of a file being used failed. Clear the work file and reuse the file.

RECOUNT FAILED - WORK FILE INTEGRITY UNCERTAIN

Verification of a file being used failed. Clear the work file and reuse the file.

NEW LINE COUNT DOES NOT MATCH OLD

Verification of a file being used failed. Clear the work file and reuse the file.

FORCE THIS TO BE A STORE

The verification process forces a store to salvage a file.

PROBABLE DATA LOSS IF SAV/STO CONTINUES. RISK FOR SALVAGE ATTEMPT (Y,N)?

UNEXP EOM/EOF. DATA LOST. PROCESS AS EOF.

There is a possible problem with the file. Retry the last directive entered.

NEXT FREE SECTOR IN HEADER AND FREEPAGE DO NOT MATCH

Save or store the work file. Change to a different work file. Reuse the stored or saved file.

FREEPAGE IS IN HEADER AS AN ACTIVE SECTOR

Save or store the work file. Change to a different work file. Reuse the stored or saved file.

DETECTED EOM ON WRITE - LOGIC ERROR

There is a problem in the save size computation.

CONTINUING MAY RESULT IN LOST DATA

Issued in conjunction with an error message that provides the reason.

XXXX NOT FOUND

An invalid directory, volume, or file (XXXX) was specified in a USE directive.

Gould Inc., Computer Systems Division
6901 W. Sunrise Blvd.
P. O. Box 409148
Fort Lauderdale, FL 33340-9148
Telephone (305) 587-2900

GOULD
Electronics

Users Group Membership Application

USER ORGANIZATION: ____________________

REPRESENTATIVE(S): ____________________

____

____

ADDRESS: ____________________

____

TELEX NUMBER: ____________________ PHONE NUMBER: ____________________

NUMBER AND TYPE OF GOULD CSD COMPUTERS: ____________________

____

OPERATING SYSTEM AND REV. LEVEL: ____________________

____

APPLICATIONS (Please Indicate)

1. EDP
   - A. Inventory Control
   - B. Engineering & Production Data Control
   - C. Large Machine Off-Load
   - D. Remote Batch Terminal
   - E. Other

2. Communications
   - A. Telephone System Monitoring
   - B. Front End Processors
   - C. Message Switching
   - D. Other

3. Design & Drafting
   - A. Electrical
   - B. Mechanical
   - C. Architectural
   - D. Cartography
   - E. Image Processing
   - F. Other

4. Industrial Automation
   - A. Continuous Process Control Op.
   - B. Production Scheduling & Control
   - C. Process Planning
   - D. Numerical Control
   - E. Other

5. Laboratory and Computational
   - A. Seismic
   - B. Scientific Calculation
   - C. Experiment Monitoring
   - D. Mathematical Modeling
   - E. Signal Processing
   - F. Other

6. Energy Monitoring & Control
   - A. Power Generation
   - B. Power Distribution
   - C. Environmental Control
   - D. Meter Monitoring
   - E. Other

7. Simulation
   - A. Flight Simulators
   - B. Power Plant Simulators
   - C. Electronic Warfare
   - D. Other

8. Other

Please return to:

Users Group Representative

Date: ____________

Gould Inc., Computer Systems Division Users Group. . .

The purpose of the Gould CSD Users Group is to help create better User/User and User/Gould CSD communications.

There is no fee to join the Users Group. Simply complete the Membership Application on the reverse side and mail to the Users Group Representative. You will automatically receive Users Group Newsletters, Referral Guide and other pertinent Users Group activity information.

Fold and Staple for Mailing

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 947 FT. LAUDERDALE, FL

POSTAGE WILL BE PAID BY ADDRESSEE

GOULD INC., COMPUTER SYSTEMS DIVISION
ATTENTION: USERS GROUP REPRESENTATIVE
6901 W. SUNRISE BLVD.
P. O. BOX 409148
FT. LAUDERDALE FL 33340-9970

(Detach Here)

Fold and Staple for Mailing

GOULD
Electronics