TOPS-20
LINK Reference Manual

AA-4183C-TM, AD-4183C-T1

March 1983

This document describes LINK-20, the linking loader for TOPS-20.

This document supersedes the document of the same name, Order No. AA-4183C-TM, published April 1982.

OPERATING SYSTEM:	TOPS-20 V5
SOFTWARE:	LINK-20 V5.1

Software and manuals should be ordered by title and order number. In the United States, send orders to the nearest distribution center. Outside the United States, orders should be directed to the nearest DIGITAL Field Sales Office or representative.

Northeast/Mid-Atlantic Region

Digital Equipment Corporation
PO Box CS2008
Nashua, New Hampshire 03061
Telephone:(603)884-6660

Central Region

Digital Equipment Corporation
Accessories and Supplies Center
1050 East Remington Road
Schaumburg, Illinois 60195
Telephone:(312)640-5612

Western Region

Digital Equipment Corporation
Accessories and Supplies Center
632 Caribbean Drive
Sunnyvale, California 94086
Telephone:(408)734-4915

digital equipment corporation • marlboro, massachusetts

First printing, January 1976
Revised, January 1978
Revised, April 1982
Updated, March 1983

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

digital™

DEC
DECmate
DECsystem–10
DECSYSTEM–20
DECUS
DECwriter
DIBOL
MASSBUS
PDP
P/OS
Professional
Rainbow
RSTS
RSX
UNIBUS
VAX
VMS
VT
Work Processor

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

CONTENTS

CONTENTS (Cont.)

FIGURES

TABLES

PREFACE

This manual is the reference document for LINK, the TOPS-20 linking loader. The manual is aimed at the intermediate to highly-experienced applications programmer, and contains complete documentation of LINK.

Chapter 1 provides a general introduction to LINK.

Chapter 2 describes automatic use of LINK through one of the system commands DEBUG, EXECUTE, or LOAD. This chapter is sufficient for most loading tasks.

Chapter 3 describes direct use of LINK. This discussion is useful for large or complicated loads. This chapter also discusses libraries and library searches.

Chapter 4 describes output from LINK: executable programs, most output files, and LINK messages. Included are descriptions of the internal format of sharable save (EXE) files.

Chapter 5 discusses overlays, including overlay structures, overlay-related output files, the overlay handler and its messages, and the FUNCT. subroutine. This chapter has an extensive example of an overlay load. Many of the elements of this example are of interest outside the context of overlays.

Appendix A gives a technical description of the output from the language translators, which is in the form of REL Blocks.

Appendix B lists all LINK messages.

Appendix C describes the job data area.

The following TOPS-20 documents are also useful:

Document	Order Number
TOPS-20 User's Guide	AA-4179C-TM
TOPS-20 Commands Reference Manual	AA-5115B-TM
MACRO Assembler Reference Manual	AA-4159C-TM
FORTRAN Language Manual	AA-4158B-TM
COBOL-68 Language Manual	AA-5057B-TK
COBOL-74 Language Manual	AA-5059B-TK
TOPS-10/TOPS-20 SPEAR Manual	AA-J833A-TK
DECsystem-10/DECSYSTEM-20 Processor Reference Manual	AA-H391A-TK

CHAPTER 1

INTRODUCTION TO LINK

LINK is the TOPS-20's linking loader. It merges independently compiled or assembled modules into a single executable program.

This merging process requires LINK to perform the following functions:

1. Perform the relocation calculations by converting relocatable addresses to virtual addresses, and by binding segments and PSECTs to addresses.

2. Resolve global symbol references by global chain fixups, Polish fixups, and library searches.

3. Produce an executable program by providing some JOBDAT information and a DDT runtime symbol table.

The virtual address space used for loading your program is not hardware memory. During loading and execution, the system simulates this virtual space by swapping code between disk and hardware memory as required. For simplicity, we will refer to the virtual address space as memory.

1.1 INPUT TO LINK

The primary input to LINK is the output from the language translators; it is a binary file containing machine language code corresponding to your program, called object modules. Other input may include your commands to LINK, and libraries containing object modules.

1.1.1 Object Modules

An object module is output from a language translator; it is part of a binary file (REL file) containing machine language code corresponding to your program. This file is formatted into blocks, called REL Blocks, that LINK recognizes and can handle appropriately. The format of each REL Block Type is described in Appendix A.

Most object modules contain relocatable code. This means that the addresses in the module are relative to the zero address. LINK loads the relocatable code at an arbitrary memory address, but adds a constant to each address referenced in the program. This resolves relative addresses to absolute addresses.

Using relocatable code simplifies your programming task and helps the system operate more efficiently. Your programming task is simpler because you need not worry about the loading addresses of your programs. System operation is more efficient because LINK can load your program at any convenient place in memory.

Besides relocating and loading your object modules, LINK resolves values for global symbols: those that are defined in one module and used in others. LINK also resolves references to entry name symbols when modules containing these symbols are loaded.

Using symbols in your programs makes your programming simpler. If you need to revise a program, it is much easier to change the value of a symbol than to change each occurrence of the value. This is especially important for global symbols. You need only change the value in the defining module; the other modules do not need retranslation.

1.1.2 Commands to Link

LINK is controlled during loading by the command strings you give. These strings consist of file specifications and switches. LINK command strings are discussed in Chapter 3.

1.1.3 Libraries

A library is a file containing object modules that may be needed to resolve references in your program. For example, the FORTRAN library FORLIB contains subroutines that may be referenced by the output from the FORTRAN compiler. When loading FORTRAN-compiled code, LINK usually searches this library to satisfy any unresolved subroutine calls. Most language translators have their own libraries.

You can construct your own libraries, and have LINK search them for necessary subroutines. Libraries and searching are discussed in Section 3.4.

1.2 OUTPUT FROM LINK

The primary output from LINK is the executable program, called the core image. In the core image, all addresses are resolved to absolute memory locations, and all symbols (including subroutine calls) are resolved to absolute values or addresses.

This core image may be executed immediately or saved as a sharable save (EXE) file. The EXE file may be created automatically by LINK. This occurs if you specify /SAVE when you run LINK, or if the program is too complex to be left in core with LINK.

You can also execute the core image under the control of a debugging program.

During its processing, LINK generates messages, which are output to your terminal or a log file. Some of these give information about LINK's operation; some warn you about possible problems; some identify errors. LINK messages are described in Appendix B.

At your option, LINK can generate three special files: the map file, the log file, and the symbol file. The map file contains information about symbols in your program modules. The log file records LINK's messages so that you can save them. The symbol file contains a symbol table for the load and has a file extension of .SYM. LINK's output files are described in Chapter 4.

1.3 LINK'S OVERLAY FACILITY

If your program is larger than your available memory, you can use LINK's overlay facility to make it fit in memory. To do this, you define a tree structure for the program's modules. Then at execution time, only part of the tree is in memory at one time. This reduces the amount of memory needed for execution. See Chapter 5 for a discussion on overlays.

1.4 USING LINK

You have two ways to use LINK:

1. You can use LINK automatically by means of the LOAD, EXECUTE, or DEBUG system commands. This is the easiest and best way to load many programs. Chapter 2 describes automatic use of LINK.

2. You can use LINK directly by means of the LINK system command. This is necessary only for very large or complicated loads, such as those involving overlays. Chapter 3 discusses direct use of LINK.

CHAPTER 2

USING LINK AUTOMATICALLY

The system commands LOAD, EXECUTE, and DEBUG invoke LINK automatically. Each of these commands uses a simple command string; the system converts the string into more complicated LINK commands.

This discussion of the LOAD, EXECUTE, and DEBUG commands does not attempt to describe them completely. Only those switches applying directly to loading will be discussed here. For a full discussion, see the TOPS-20 Commands Reference Manual.

These system commands invoke LINK:

- The LOAD command uses LINK to load your object modules into memory, but does not execute the program. Before loading, your source files are compiled, if necessary; this compilation will occur if there are no object modules for the specified source files, or if the object files are older than their source files.

- The EXECUTE command uses LINK to load your program, and then executes the loaded program. Before loading, your source files are compiled, if necessary.

- The DEBUG command works like the EXECUTE command, except that your program is executed under the control of a debugging program. The debugging program that is loaded depends on the type of program being loaded. For a COBOL program, COBDDT is loaded. For a FORTRAN program, FORDDT is loaded. For an ALGOL program, ALGDDT is loaded. For any other language, DDT is loaded. The system uses the file type to determine the language in which the program is written. Therefore, it is highly recommended that you use standard file types when naming the files of your programs. Standard file types are listed in the appropriate Commands Manual for the operating system.

2.1 COMMAND FORMATS

The formats for the LOAD, EXECUTE, and DEBUG commands are the same. Each can accept a list of input file specifications and switches. The format for these commands is:

```
@command/switches input-spec/switches, input-spec/switches,...
```

Where the command is one of the three system commands (LOAD, EXECUTE, or DEBUG), input-spec is the file specification of the program you want to load, and the switches are any of the valid switches for the command.

If you separate the input file specifications with commas, each source file will be compiled into a separate object file. If you separate the input file specifications with plus signs, they will be compiled into a single object file.

Section 2.3 shows examples of using LINK automatically.

2.2 COMMAND SWITCHES

You can use switches with the LOAD, EXECUTE, and DEBUG commands to control LINK's loading. Table 2-1 briefly describes some of the command switches that apply to LINK. Refer to the TOPS-20 Commands Reference Manual for complete descriptions of the switches for these commands.

Table 2-1
Switches for System Commands

Switch	Meaning
/COMPILE	Forces compilation of source files even if a sufficiently recent REL file exists.
/DDT	Loads DDT. This supersedes the default debugger selection, which is usually based on the file type of the first file in the command string.
/MAP	Produces a map file at the end of loading. This file shows all global symbols loaded.
/NOCOMPILE	Compiles source files only if their REL files are older than the source files. /NOCOMPILE is the default.
/NOSEARCH	Suspends the effect of an earlier global /SEARCH switch. This is the default action.
/NOSYMBOLS	Prevents loading of symbol tables with their modules.
/SEARCH	Loads only the modules from the specified library file that satisfy global references in the program.

You can use any LINK program switches with the system commands LOAD, EXECUTE, or DEBUG by using a special switch format. This format requires that you use a percent sign (%) instead of the usual slash (/), and that the entire switch specification be enclosed in double quotation marks ("). For example, you can pass the /LOG switch to LINK by using the command:

```
     @EXECUTE MYPROG %"LOG"
```

Used directly with LINK, the command strings would include:

```
     *MYPROG/LOG
```

If you give more than one switch in this format, succeeding switches within the quotation marks must have the usual slashes:

```
     @EXECUTE MYPROG%"LOG/MAP"
```

LINK program switches are described in Section 3.2.

2.3 EXAMPLE OF USING LINK AUTOMATICALLY

For this example, the following program, named MYPROG.FOR, is used:

```
          TYPE 10
     10   FORMAT (' This is written by MYPROG')
          STOP
          END
```

The following example shows an interactive execution of the program using the EXECUTE command:

```
     @EXECUTE MYPROG.FOR(RET)
     FORTRAN:MYPROG
     MAIN.
     LINK:  Loading
     [LNKXCT MYPROG Execution]


     This is written by MYPROG.

     END OF EXECUTION
     CPU TIME:  0.02 ELAPSED TIME:  0.05
     EXIT

     @
```

The following example shows how to load a program for debugging using the DEBUG command:

```
     @DEBUG MYPROG.FOR(RET)
     FORTRAN:  MYPROG
     LINK:  Loading
     [LNKDEB FORDDT Execution]

     STARTING FORTRAN DDT

     >>START


     This is written by MYPROG.

     END OF EXECUTION
     CPU TIME:  0.01 ELAPSED TIME:  0.03
     EXIT

     @
```

CHAPTER 3

USING LINK DIRECTLY

If you have a loading task that cannot be handled conveniently by the EXECUTE, LOAD, or DEBUG system commands (such as loading overlays or PSECTs), you can load your program by using LINK directly. To do this, you must already have compiled or assembled all required object modules.

To use LINK directly, type LINK to the system. LINK will respond with an asterisk:

```
@LINK(RET)
*
```

Continue typing command strings, ending each one with a carriage return. For example,

```
@LINK(RET)
*/OVERLAY(RET)
*TEST/LINK:TEST(RET)
*       /NODE:TEST SPEXP/LINK:SPEXP(RET)
*
```

A command string consists of file specifications and switches. You can continue a command string to the next line by typing a hyphen immediately before pressing carriage return; LINK continues the line by responding with a number sign (#). For example,

```
@LINK(RET)
*MYPROG,MYMAP/MAP/CONTENTS:ALL-(RET)
#/ERRORLEVEL:0/LOG/LOGLEVEL:5(RET)
*
```

The use of continuation lines is more efficient as the command scanner must be invoked for every distinct command string.

You can include a comment on a command line by beginning the comment with a semicolon; the remaining text on the line is not processed by LINK.

When LINK sees the end of the command string (a carriage return), it processes the entire string, then prints an asterisk to begin the next line. This processing continues until one of the following occurs:

1. LINK finds a /GO switch in a command string. It then completes loading and exits to system command level (if you did not specify execution), or passes control to the loaded program for execution.

2. A fatal error occurs. LINK prints an error message and exits to system command level.

3. A /RUN switch is encountered.

4. Either /EXIT or ^Z is encountered.

3.1 COMMAND STRING FORMAT

A LINK command string can contain file specifications, LINK switches, and command scanner switches. Command scanner switches are described in Section 3.2.1. LINK switches are described in Section 3.2.2.

Some LINK switches take output file specifications as arguments; some switches are suffixed to output file specifications. Other file specifications specify input files. For example, the following command string tells LINK to use an input file called MYREL.REL to generate a saved output file called MYEXE.EXE:

```
     *MYREL,MYEXE/SAVE/GO
```

LINK supplies the missing parts of the file specifications from its defaults.

DEFAULTS

For output files, the defaults are:

device	logical name (DSK:)	
filename	name of last module with start address or, if none, then nnnLNK where nnn is your job number, with leading zeros if necessary.	
type	log file	LOG
	map file	MAP
	overlay file	OVL
	plotter file	PLT
	executable file	EXE
	symbol file	SYM
directory	a PPN corresponding to a directory. Refer to Section 3.3.	

For input files, the defaults are:

device	logical name (DSK:)
type	REL
	directory a PPN corresponding to a directory. Refer to Section 3.3.

You can change these defaults by using the /DEFAULT switch (see Section 3.2.2).

You can have LINK read command strings from an indirect command file. To do this, prefix an at-sign (@) to the command file specification. For example, the following commands tell LINK to read all command strings from the file LNKPRG.CCL. (.CCL is the default file type for indirect command files):

```
     @LINK(RET)
     *@LNKPRG
```

3.2 SWITCHES TO LINK

LINK's handling of files depends on your use of LINK switches. There are two sets of switches to LINK. The first set of switches (command scanner switches) are optional switches that define your request to the system command scanner. These are described in Section 3.2.1. The second set of switches are switches to LINK that you can use to control and modify the linking and loading process. These are described in Section 3.2.2.

3.2.1 Command Scanner Switches

The system SCAN module scans command lines for various system programs, one of which is LINK. Therefore the module LNKSCN is included with LINK source files. You can include SCAN switches in your command strings for LINK, but none of these switches is required in order to run LINK.

The following SCAN switches are meaningful to LINK. The remaining SCAN switches, which are listed in LINK's HELP file, are ignored by LINK.

Like LINK switches, SCAN switches are preceded by a slash (/), and can be abbreviated up to their first unique characters.

SCAN Switches Meaningful to LINK

Switch	Meaning
/EXIT	Exits after loading, but leave LINK's core image in place. This switch is ignored if you have specified program execution or requested that the contents of memory be included with the loaded program.
/HELP:arg	Displays the LINK.HLP file. The arguments are SWITCHES, to see a list of LINK switches, and TEXT, to see the default HELP text for LINK.
/MESSAGE:keyword	Displays messages in the format specified by keyword. The keywords and their meanings are: PREFIX — Displays only the message code from SCAN, which is of the form SCNxxx. FIRST — Displays the prefix and a short message. CONTINUATION — Displays the prefix and a longer message.
/RUN:file	Runs the specified program after loading is finished. This switch is ignored if you have specified program execution or requested that the contents of memory be included with the loaded program.

3.2.2 Link Switches

This section lists the switches that may be used to instruct LINK to take special action while loading your programs. The switches are described in this section in alphabetical order, and for each switch the following information is shown, if appropriate:

FORMAT
FUNCTION
EXAMPLES
OPTIONAL NOTATIONS
RELATED SWITCHES

Switches can be abbreviated to save typing. However, in most cases, the switch must include enough characters to make it unique from other switches. For example, the switch /NOSYMBOL cannot be abbreviated to /NOSY, because this result in a conflict with the switch /NOSYSLIB. However, /NOSYM is a unique set of characters, and thus is is a legal abbreviation for /NOSYMBOL.

Certain switches can be abbreviated to a single letter; they are:

/D for	/DEBUG
/E for	/EXECUTE
/G for	/GO
/H for	/HELP
/L for	/LOCALS
/M for	/MAP
/N for	/NOLOCAL
/S for	/SEARCH
/R for	/TEST
/U for	/UNDEFINE
/V for	/VERSION

Many switches accept a value that may be specified in decimal (which is the default) or octal. If the value can be specified in octal, this is noted in the OPTIONAL NOTATIONS section of the switch description. To specify an octal value, type a pound sign (#) before the octal number. For example, /ARSIZE:39 can be specified in octal as /ARSIZE:#47.

Some switches can be used either locally or globally (in particular, /LOCALS, /NOLOCAL, /SEARCH, and /NOSEARCH). This means that if the switch is suffixed to a file specification, it applies only to that file; if it is not suffixed to a file specification, it applies to the files that follow. For example, in the following command strings /SEARCH is used both locally and globally:

```
1.  *FILE1,FILE2/SEARCH,FILE3

2.  *FILE4,/SEARCH FILE5,FILE6
```

In the first line, /SEARCH is suffixed to the file specification FILE2; only that file is loaded in search mode. In the second line, /SEARCH is not suffixed to a file specification; all the remaining files named in the command string are to be searched.

In general, a switch used globally is disabled at the end of its command string, unless it is overridden by another switch. The second switch, if used locally, will override the first only for the local file. If the second switch is used globally, it will persist for the following files. For example, in the following command string, a globally-used switch (/SEARCH) is overridden by a locally used switch:

```
*/SEARCH FILE1,FILE2/NOSEARCH,FILE3
```

In this command string, FILE1 and FILE3 will be loaded in search mode, but FILE2 will be loaded normally.

NOTE

> The effects of a global switch on the same line as a /GO switch persist beyond the /GO switch and apply to any modules loaded during library searches.

The following pages contain the switches and their descriptions, listed in alphabetical order.

/ARSIZE

FORMAT

```
/ARSIZE:n
```

Where n is a positive decimal integer.

FUNCTION

Sets the size of the overlay handler's table of multiply-defined global symbols. Use this switch if you have received LNKARL, LNKTMA, and LNKABT messages in a previous attempt to load your program. These messages will give instructions for the argument to the /ARSIZE switch.

EXAMPLES

```
*/ARSIZE:39(RET)
*
```

Allocates 39 words for the multiply-defined global symbol table in each link of an overlay structure.

OPTIONAL NOTATIONS

You can specify the table size in octal.

/BACKSPACE

FORMAT

```
/BACKSPACE:n
```

Where n is a positive decimal integer.

FUNCTION

Backspaces over n files on the current tape device. (The switch is ignored for non-tape devices.)

EXAMPLES

```
*MTA0:/BACKSPACE:3(RET)
*
```

Backspaces magtape MTA0: by three files.

OPTIONAL NOTATIONS

If you omit n, it defaults to 1.

RELATED SWITCHES

/MTAPE, /REWIND, /SKIP, /UNLOAD

/COMMON

FORMAT

```
/COMMON:name:n
```

Where name is up to six SIXBIT-compatible ASCII characters.

n = a positive decimal integer.

FUNCTION

Allocates n words of labeled COMMON storage for FORTRAN and FORTRAN-compatible programs. The COMMON label is a name, which becomes defined as an internal symbol, and is available to other programs as an external symbol.

For unlabeled COMMON storage, use COMM. as the name, or simply omit the name.

You cannot expand a given COMMON area during loading. If your program modules define a given COMMON area to have different sizes, the module giving the largest definition must be loaded first. If the /COMMON switch gives the largest definition, it must precede the loading of the modules.

EXAMPLES

```
*/COMMON:A:1000RET
*
```

Creates a labeled COMMON area of 1000 words.

```
*/COMMON:.COMM.:1000RET
*
```

Creates an unlabeled COMMON area of 1000 words.

```
*/COMMON::1000RET
*
```

Creates an unlabeled COMMON area of 1000 words.

OPTIONAL NOTATIONS

You can specify the number of words in octal.

/CONTENTS

FORMAT /CONTENTS:(keyword,...,keyword)

FUNCTION Each keyword gives a symbol type to be included in the map file if the file is generated. To generate the map file, use the /MAP switch.

The keywords ALL, NONE, and DEFAULT reset all symbol types. Otherwise, using the /CONTENTS switch resets only those symbol types specified by keywords. In the following list of keywords, the defaults are in **boldface:**

ABSOLUTE	**Include absolute symbols.**
ALL	Include all symbols.
COMMON	**Include COMMON symbols.**
DEFAULT	Reset to LINK's defaults.
ENTRY	**Include entry-name symbols.**
GLOBAL	**Include global symbols.**
LOCALS	Include local symbols. The local symbols cannot be included in the map file unless the /LOCALS switch is also given.
NOABSOLUTE	Exclude absolute symbols.
NOCOMMON	Exclude COMMON symbols.
NOENTRY	Exclude entry-name symbols.
NOGLOBAL	Exclude global symbols.
NOLOCAL	**Exclude local symbols.**
NONE	Exclude all symbols.
NORELOCATABLE	Exclude relocatable symbols.
NOUNDEFINED	**Exclude undefined symbols.**
NOZERO	**Exclude symbols in zero-length programs.** (a zero-length program contains no code or data; it contains only symbol definitions.)
RELOCATABLE	**Include relocatable symbols.**
UNDEFINED	Include undefined symbols.
ZERO	Include symbols in zero-length programs. (a zero-length program contains no code or data; it contains only symbol definitions.)

The settings for the /CONTENTS switch operate by exclusion, not inclusion. For example, if both the NOGLOBAL and RELOCATABLE settings are in force, all global symbols are excluded regardless of their relocatability.

EXAMPLES

```
*/CONTENTS:(NOCOMMON,NOENTRY) RET
*
```

Excludes COMMON and entry-name symbols.

```
*/CONTENTS:(ALL) RET
*
```

Includes all symbols.

OPTIONAL NOTATIONS You can omit parentheses if you give only one keyword.

RELATED SWITCHES /MAP

/COUNTER

FORMAT /COUNTER

FUNCTION Requests terminal typeout of information about the relocation counters. The information that is printed gives the name, initial value, current value, and limit value of each counter.

/COUNTER may be used to determine the size of overlays when loading large programs that might be too large for the allocated memory space. Refer to Section 5.4 for more information.

EXAMPLE

```
@LINK RET
*/LIMIT:.LOW.:5000 RET
*/LIMIT:.HIGH.:402000 RET
*/COUNTER RET
[LNKRLC RELOC. CTR.      INITIAL VALUE     CURRENT VALUE     LIMIT VALUE
        .LOW.            0                 140               5000
        .HIGH.           400000            400010            402000]
```

RELATED SWITCHES /NEWPAGE, /SET

/CPU

FORMAT

```
/CPU:keyword

Keyword:     KA10
             KI10
             KL10
             KS10
```

FUNCTION

This switch is used to override LINK's handling of the processor information found in the .REL files being loaded. (See the description of the type 6 block in Appendix A). Ordinarily LINK prints a warning if all .REL files being loaded together do not have identical CPU types. This switch can be used either to make LINK flag certain modules built for a specific CPU type (by specifying all but that CPU type as keywords to /CPU) or to suppress LINK's warning message (by specifying all the CPU types associated with the .REL files being loaded).

EXAMPLE

```
*/CPU:KI10RET
*
```

Will cause LINK to issue the %LNKCCD message if any modules with the KL10 CPU type are encountered.

OPTIONAL NOTATIONS

```
/CPU:(keyword,keyword)
```

/DDEBUG

FORMAT

/DDEBUG:keyword

FUNCTION

Specifies a default debugging program to be loaded if the /DEBUG or /TEST switch appears without an argument.

The permitted keywords and the debugging programs they specify are listed below. Only those printed in **boldface** are supported by DIGITAL.

ALGDDT	**Specifies ALGDDT as the default.**
ALGOL	**Specifies ALGDDT as the default.**
COBDDT	**Specifies COBDDT as the default.**
COBOL	**Specifies COBDDT as the default.**
DDT	**Specifies DDT as the default.**
FAIL	Specifies SDDT as the default.
FORDDT	**Specifies FORDDT as the default.**
FORTRAN	**Specifies FORDDT as the default.**
MACRO	**Specifies DDT as the default.**
PASCAL	Specifies PASDDT as the default.
PASDDT	Specifies PASDDT as the default.
SAIL	Specifies the SAIL debugger as the default.
SDDT	Specifies the SAIL debugger as the default.
SIMDDT	Specifies SIMDDT as the default.
SIMULA	Specifies SIMDDT as the default.

EXAMPLES

```
*/DDEBUG:FORTRAN RET
*
```

Specifies FORDDT as the default debugging program for the /DEBUG or /TEST switch.

RELATED SWITCHES

/DEBUG, /TEST

/DEBUG

FORMAT /DEBUG:keyword

FUNCTION Requests loading of a debugging program and sets the start address for execution as the normal start address of the debugging program. The /DEBUG switch also sets the /EXECUTE switch because it is assumed that the program is to be executed. The /GO switch is still required to end loading and begin execution.

The /DEBUG switch turns on the /LOCALS switch for the remainder of the load. You can override this by using the /NOLOCAL switch, but the override lasts only during processing of the current command string.

Local symbols for the debugging program itself are never loaded.

If debugging overlaid programs, you must specify /DEBUG when loading the root node. (Refer to Section 5.4 for more information.)

The permitted keywords and the programs they load are listed below. Only those printed in **boldface** are supported by DIGITAL.

ALGDDT	**Loads ALGDDT.**
ALGOL	**Loads ALGDDT.**
COBDDT	**Loads COBDDT.**
COBOL	**Loads COBDDT.**
DDT	**Loads DDT.**
FAIL	Loads SDDT.
FORDDT	**Loads FORDDT.**
FORTRAN	**Loads FORDDT.**
MACRO	**Loads DDT.**
PASCAL	Loads PASDDT.
PASDDT	Loads PASDDT.
SAIL	Loads the SAIL debugger.
SDDT	Loads the SAIL debugger.
SIMDDT	Loads SIMDDT.
SIMULA	Loads SIMDDT.

If you give no keyword with /DEBUG, the default is either DDT or the debugging program specified by the /DDEBUG switch.

EXAMPLES

```
*/DEBUG:DDT RET
*
```

Loads DDT, sets the /EXECUTE switch, and specifies that the execution will be controlled by DDT.

OPTIONAL NOTATIONS Abbreviate /DEBUG to /D.

RELATED SWITCHES /DDEBUG, /TEST

/DEFAULT

FORMAT

```
/DEFAULT:keyword filespec
filespec/DEFAULT:keyword
```

FUNCTION

Changes default specifications for input or output files. The defaults specified remain in effect until changed by another /DEFAULT switch.

The keywords allowed are:

INPUT — Specifies the defaults for input file specifications.

OUTPUT — Specifies the defaults for output file specifications.

For input files, the initial defaults are:

device	DSK:
file type	REL
directory	User's connected directory

For output files, the initial defaults are:

device	DSK:
filename	Name of main program
directory	User's connected directory

EXAMPLES

```
*/DEFAULT:INPUT .BIN RET
*
```

Resets input file default extension to BIN.

```
*/DEFAULT:OUTPUT MTA0: RET
*
```

Resets output file default device to MTA0:.

OPTIONAL NOTATIONS

If you omit the keyword, INPUT is assumed.

/DEFINE

FORMAT

/DEFINE:(symbol:value,...,symbol:value)

FUNCTION

Assigns each symbol the decimal value following it. This causes them to be global symbols. You can use the /UNDEFINED switch to get a list of any undefined symbols, and then define them with /DEFINE.

Defining an already defined symbol with /DEFINE generates an error message.

EXAMPLES

```
*/UNDEFINED(RET)
[LNKUGS  2 UNDEFINED GLOBAL SYMBOLS]
     A       400123
     IGOR    402017

*/DEFINE:(A:591,IGOR:1)(RET)
*
```

Gives the decimal values 591 and 1 to A and IGOR, respectively.

OPTIONAL NOTATIONS

You can give the value in octal by typing # before the value.

You can omit the parentheses if you define only one symbol. Specifying /DEFINE:FOO:BAR will define FOO to have the value of BAR if BAR is already defined.

RELATED SWITCHES

/UNDEFINED, /VALUE

/ENTRY

FORMAT /ENTRY

FUNCTION Requests terminal typeout (in octal) of all entry name symbols loaded so far. Each entry name symbol will have been defined by an ENTRY statement (MACRO, FORTRAN, or BLISS), a FUNCTION statement (FORTRAN), a SUBROUTINE statement (FORTRAN, or COBOL), or a PROCEDURE declaration (ALGOL).

If you are using the overlay facility, /ENTRY requests only the entry name symbols for the current link.

EXAMPLES

```
*/ENTRY RET
[LNKLSS  LIBRARY SEARCH SYMBOLS (ENTRY POINTS)]
     SQRT.  3456
*
```

RELATED SWITCHES /NOENTRY

/ERRORLEVEL

FORMAT /ERRORLEVEL:n

FUNCTION Suppresses terminal typeout of LINK messages with message level n and less, where n is a decimal number between 0 and 30 inclusive. You cannot suppress level 31 messages. LINK's default is /ERRORLEVEL:10.

See Appendix B for the level of each LINK message.

EXAMPLES

```
*/ERRORLEVEL:10RET
*
```

Suppresses all messages of level 10 and less.

```
*/ERRORLEVEL:0RET
*
```

Permits typeout of all messages.

RELATED SWITCHES /VERBOSE

/EXCLUDE

FORMAT

```
/EXCLUDE:(subroutine,...,subroutine)
```

FUNCTION

Prevents loading of the specified modules from the current file even if they are required to resolve global symbol references. You can use the /EXCLUDE switch for any of the following purposes:

- If a library has several modules with the same search symbols, you can select the module you want by excluding the others.
- You can prevent modules from giving multiple definitions of a symbol by selectively excluding one or more of them.
- In defining an overlay structure, you can delay loading of a module until a later link by excluding it.

EXAMPLES

```
*/SEARCH LIBFIL.REL/EXCLUDE:(MOD1,MOD2)(RET)
*
```

Searches LIBFIL as a library but prevents loading of MOD1 and MOD2 even if they are referenced.

OPTIONAL NOTATIONS

You can omit the parentheses if you specify only one module.

RELATED SWITCHES

/INCLUDE

/EXECUTE

FORMAT /EXECUTE

FUNCTION Tells LINK that the loaded program is to be executed beginning at its normal start address. Loading continues until a /GO switch is found.

The /EXECUTE and /DEBUG switches are mutually exclusive.

EXAMPLES

```
*/EXECUTE RET
*
```

OPTIONAL NOTATIONS You can abbreviate /EXECUTE to /E.

RELATED SWITCHES /DEBUG, /GO, /TEST

/FRECOR

FORMAT

```
/FRECOR:nK
```

Where n is a positive decimal integer.

FUNCTION

Requires LINK to maintain a minimum amount of free memory after any expansions. LINK's default free memory is 4K. If you use the /FRECOR:nK switch, LINK computes n times 1024 words and maintains the resulting number of words of free memory, if possible.

If the modules to be loaded are quite large, a larger amount of free memory avoids some moving of areas.

LINK has nine areas that may be expanded during loading:

1. ALGOL symbol information (AS).
2. Bound global symbols (BG).
3. Dynamic area (DY).
4. Fixup area (FX).
5. Global symbol tables (GS).
6. User's high segment code (HC).
7. User's low segment code (LC).
8. Local symbol tables (LS).
9. Relocation tables (RT).

Each of these areas has a lower bound, an actual upper bound, and a maximum upper bound. LINK normally maintains space between the actual and maximum upper bounds for each area. The total of these nine spaces is at least the space given by the /FRECOR switch, if possible.

LINK recovers free core by concatenating these nine free areas. When all this recovered space is used, one or more of the nine areas overflows to disk, and free core is no longer maintained.

EXAMPLES

```
*/FRECOR:7K RET
*
```

Maintains 7K of free core, if possible.

OPTIONAL NOTATIONS

You can specify the free core in octal.

/GO

FORMAT

/GO

FUNCTION

Ends loading after the current module. LINK then performs any required library searches, generates any required output files, and does one of the following:

- Begins execution at the normal start address of the loaded program (if you used /EXECUTE).
- Begins execution at the start address of the debugging program (if you used /DEBUG, or both /TEST and /EXECUTE).
- Exits to the monitor (if you used no execution switch).

EXAMPLES

```
*MYPROG/EXECUTE/GO RET
```

```
[LNKXCT  MYPROG EXECUTION]
```

Begins execution of the loaded program at its normal start address.

```
*MYPROG/DEBUG/GO RET
```

```
[LNKDEB DDT EXECUTION]
```

Begins execution of the loaded program at the normal start address of DDT.

OPTIONAL NOTATIONS

Abbreviate /GO to /G.

RELATED SWITCHES

/DEBUG, /EXECUTE, /TEST

/HASHSIZE

FORMAT

/HASHSIZE:n

Where n is a positive decimal integer.

FUNCTION

Gives a minimum for the initial size of the global symbol table. LINK selects a prime number larger than n for the initial size.

If you know that you will need a large global symbol table, you can save time and space by allocating space for it with /HASHSIZE. You should give a hash size 10 percent larger than the number of global symbols in the table.

If LINK gives the message [LNKRGS Rehashing Global Symbol Table] during a load, you should use the /HASHSIZE switch for future loads of the same program. The minimum hash size for loading a program appears in the header lines of the map file.

The default hash size is a LINK assembly parameter (initially 251 decimal).

EXAMPLES

```
*/HASHSIZE:1000 RET
*
```

Sets the hash size to the prime number 1021.

/INCLUDE

FORMAT

/INCLUDE:(module,...,module)

FUNCTION

Specifies modules to be loaded regardless of any global requests for them.

In library search mode, an /INCLUDE switch requests loading of the specified modules. If the switch is associated with a file, the request is cleared after that file is searched. If not, the request persists until the modules are found.

When LINK is not in library search mode, the /INCLUDE switch associated with a file requests that only the specified modules be loaded, and the request is cleared after that file is processed. An /INCLUDE switch not associated with a file requests loading of the specified modules, and the request persists until the modules are found.

You can use /INCLUDE in an overlay load to force a module to be loaded in an ancestor link common to successor links that reference that module. This makes the module available to all links that are successors to its link.

EXAMPLES

```
*/SEARCH LIB1/INCLUDE:(MOD1,MOD2)RET
*
```

Searches LIB1 and loads MOD1 and MOD2 even if they are not referenced.

OPTIONAL NOTATIONS

You can omit the parentheses if you specify only one module.

RELATED SWITCHES

/EXCLUDE, /NOINCLUDE, /MISSING

THIS PAGE INTENTIONALLY LEFT BLANK

/LIMIT

FORMAT /LIMIT:psect:address

FUNCTION Allows you to specify an upper bound for a specific PSECT. In the format description, psect should be the PSECT name, which has been defined with either the /SET switch or in one of the modules already loaded. Address should be the upper bound address of the specified PSECT, expressed in either numeric or symbolic form. This address should be one greater than the highest location which may be loaded in the PSECT. The address can be a thirty-bit quantity, and need not be in the same section as the PSECT origin.

If the PSECT grows beyond the address specified in the /LIMIT switch, LINK will send a warning to your terminal, but will continue to process input files and to load code. The warning message will take the following form:

```
%LNKPEL PSECT <psect> exceeded limit of <address>
```

No chained references will be resolved, and LINK will suppress program execution, producing the following fatal error:

```
?LNKCFS Chained fixups have been suppressed
```

This action prevents unintended PSECT overlays. PSECT overlays can cause loops and other unpredictable behavior, because LINK uses address relocation chains in the user image that is being built.

EXAMPLE

```
*TEST1 RET
*/COUNTERS RET
[LNKRLC RELOC. CTR.       INITIAL VALUE    CURRENT VALUE    LIMIT VALUE
        .LOW.             0                140              1000000
        Q                 1000             4000             1000000
        R                 4500             10500            1000000]
*/LIMIT:Q:4000 RET
*TEST2 RET
%LNKPEL PSECT Q EXCEEDED LIMIT OF  4000
        DETECTED IN MODULE .MAIN FROM FILE DSK:TEST2REL
*/COUNTERS RET
[LNKRLC RELOC. CTR.       INITIAL VALUE    CURRENT VALUE    LIMIT VALUE
        .LOW.             0                140              1000000
        Q                 1000             5000             4000
        R                 4500             10500            1000000]
*TEST/SAVE/GO RET
%LNKPOV PSECTS R AND Q OVERLAP FROM ADDRESS 4500 TO 5000
?LNKCFS CHAINED FIXUPS HAVE BEEN SUPPRESSED
@
```

In this example, a program named TEST1, which contains two PSECTs, is loaded. The PSECTs are named Q and R. After TEST1 is loaded, the /COUNTERS switch shows that the upper bound of PSECT Q is 4000.

The /LIMIT switch is used to limit PSECT Q to 4000.

A second program, TEST2, also requires storage for PSECT Q. Therefore, when TEST2 is loaded, LINK produces a warning to the effect that the limit that was set has been exceeded. The /COUNTERS switch shows that PSECT Q now requires an upper bound of 5000.

When the programs are started (with /GO), LINK produces the POV warning message and the CFS fatal error message.

/LINK

FORMAT

```
/LINK:name
```

Where name is up to 6 RADIX-50 compatible characters.

FUNCTION

Directs LINK to give the specified name to the current core image and outputs the core image to the overlay file. /LINK is used to close an overlay link. LINK first performs any required library searches and assigns a number to the link.

For a discussion of overlay structures, see Chapter 5.

The current core image has all modules loaded since the beginning of the load or since the last /LINK switch.

EXAMPLES

```
*SPEXP/LINK:ALPHA(RET)
*
```

Loads module SPEXP and outputs the core image to the overlay file as a link called ALPHA.

OPTIONAL NOTATIONS

If you omit the link name, LINK uses only its assigned number.

RELATED SWITCH

/NODE

/LOCALS

FORMAT /LOCALS

FUNCTION Includes local symbols from a module in the symbol table. LINK does not need these tables, but you may want them for debugging.

The /LOCALS and /NOLOCAL switches may be used either locally or globally. If the switch is suffixed to a file specification, it applies only to that file; if it is not suffixed to a file specification, it applies to all following files in the same command line.

EXAMPLES

```
*/LOCALS A,B/NOLOCAL,C,/NOLOCAL DRET
*
```

Loads A with local symbols, B without local symbols, C with local symbols, and D without local symbols.

OPTIONAL NOTATIONS You can abbreviate /LOCALS to /L.

RELATED SWITCHES /NOLOCAL, /SYMSEG

/LOG

FORMAT logfilespec/LOG

FUNCTION Specifies a file specification for the log file (see Section 4.2.2). Any LINK messages output before the /LOG switch is found are not entered in the log file.

EXAMPLES

```
*LOGFIL/LOG(RET)
*
```

Specifies the file DSK:LOGFIL.LOG in the user's directory.

```
*TTY:/LOG(RET)
*
```

Directs log messages to the user's terminal.

OPTIONAL NOTATIONS You can omit all or part of the logfilespec. The defaults are:

device	DSK:
filename	name of main program
file type	LOG
directory	your connected directory

You can change the defaults using the /DEFAULT switch.

RELATED SWITCHES /LOGLEVEL

/LOGLEVEL

FORMAT /LOGLEVEL:n

FUNCTION Suppresses logging of LINK messages with level n and less, where n is a decimal number between 0 and 30 inclusive. You cannot suppress level 31 messages.

See Appendix B for the level of each LINK message.

The default is /LOGLEVEL:10.

EXAMPLES

```
*/LOGLEVEL:0RET
*
```

Logs all messages.

RELATED SWITCHES /LOG

/MAP

FORMAT mapfilespec/MAP:keyword

FUNCTION Specifies a file specification for the map output file (see Section 4.2). The contents of the file are determined by the /CONTENTS switch or its defaults.

Permitted keywords and their meanings are:

END Produces a map file at the end of the load. This is the default if you omit the keyword.

ERROR Produces a map file if a fatal error occurs. Any modules loaded after this switch will not appear in the log. To ensure that a .MAP file is generated, specify this switch before the loading of .REL files.

NOW Produces a map file immediately. Library searches will not have been performed unless forced.

EXAMPLES

```
*MAPFIL/MAP:END(RET)
*
```

Generates a map in the file DSK:MAPFIL.MAP in your disk area at the end of loading.

OPTIONAL NOTATIONS You can omit all or part of the mapfilespec. The defaults are:

device	DSK:
filename	name of main program
file type	MAP
directory	user's connected directory

You can change the defaults using the /DEFAULT switch.

You can abbreviate /MAP to /M.

RELATED SWITCHES /CONTENTS

/MAXNODE

FORMAT /MAXNODE:n

Where n is a positive decimal integer.

FUNCTION Specifies the number of links to be defined when the overlayed program requires more than 256 links. LINK will allocate extra space in the OVL file for certain fixed-length tables based on the number of links specified with this switch.

Note that this switch must be placed after the /OVERLAY switch and it must precede the first /NODE switch in the set of commands to LINK.

EXAMPLES

```
*TEST/OVERLAY/MAXNODE:500(RET)
*
```

Reserves space for 500 defined links. See Chapter 5 for a discussion on overlays.

RELATED SWITCHES /OVERLAY

/MISSING

FORMAT

/MISSING

FUNCTION

Requests terminal typeout of modules requested with the /INCLUDE switch that have not yet been loaded.

EXAMPLES

```
*MYPROG(RET)
*/SEARCH/INCLUDE:(MOD1,MOD2) LIB1(RET)
*/MISSING(RET)
[LNKIMM  1 INCLUDED MODULE MISSING]
*LIB2/INCLUDE:(MOD2)(RET)
*/MISSING(RET)
[LNKIMM  NO INCLUDED MODULES MISSING]
*
```

This example shows the use of /MISSING to see if all the required modules have been loaded. The module MOD2 was not yet loaded, and it was in LIB2.

In response to the first use of the switch, LINK indicated that one necessary module was missing. After the missing module was included (module named LIB2), the switch is used again. LINK responded to the second use of the switch by indicating that all necessary modules were present.

RELATED SWITCHES

/INCLUDE

/MTAPE

FORMAT /MTAPE:keyword

FUNCTION Specifies tape operations to be performed on the current device. (A tape device remains current only until end-of-line or until another device is specified, whichever is earlier.) The switch is ignored if the current device is not a tape.

The operation is performed immediately if /MTAPE is given with an input file or with an already initialized output file. Otherwise, the operation is performed when the output file is initialized.

The valid keywords and the operations they specify are:

MTBLK	Writes 3 inches of blank tape.
MTBSF	Backspaces one file.
MTBSR	Backspaces one record.
MTDEC	Initializes DIGITAL-compatible 9-channel tape.
MTEOF	Writes an end-of-file mark.
MTEOT	Spaces to logical end-of-tape.
MTIND	Initializes industry-compatible 9-channel tape.
MTREW	Rewinds tape to the load point (BOT).
MTSKF	Skips one file.
MTSKR	Skips one record.
MTUNL	Rewinds and unloads tape.
MTWAT	Waits for tape I/O to finish.

EXAMPLES

```
*MTA0:/MTAPE:MTEOT RET
*MTA0:/MAP:NOW RET
*
```

Spaces to logical end-of-tape on MTA0: and writes a map file.

RELATED SWITCHES /BACKSPACE, /REWIND, /SKIP, /UNLOAD

/NEWPAGE

FORMAT /NEWPAGE:keyword

FUNCTION Sets the relocation counter to the first word of the next page. If the counter is already at a new page, this switch is ignored.

The permitted keywords and their relocation counters are:

LOW	Resets the low-segment counter to new page. If you omit the keyword, this is the default.
HIGH	Resets the high-segment counter to new page.

EXAMPLES

```
*/NEWPAGE:HIGH(RET)
*SUBR1(RET)
*/NEWPAGE:LOW(RET)
*SUBR2(RET)
*
```

Sets the high-segment counter to a new page, loads SUBR1, sets the low-segment counter to a new page, and loads SUBR2. Note that SUBR1 and SUBR2 are not necessarily loaded into the high and low segments respectively; the /NEWPAGE switch sets a counter, but does not force the next loaded module into the specified segment.

RELATED SWITCHES /SET

/NODE

FORMAT /NODE:argument

FUNCTION Opens an overlay link. /NODE places LINK's relocation counter at the end of a previously defined link in an overlay structure, which becomes the immediate ancestor to the next link defined. (For a discussion of overlay structures, see Chapter 5.)

The /NODE switch must precede any modules to be placed in the new link.

Three kinds of arguments are permitted:

- A name given with a previous /LINK switch. LINK will place the relocation counter at the end of the specified link.
- A negative number (-n). LINK backs up n links along the current path.
- A positive number n or 0. LINK begins further loading at the end of link number n. You can use 0 to begin loading at the root link.

NOTE

It is recommended that you use a link name (or 0 for the root link) rather than a nonzero number. This is because a change in commands defining an overlay may change some of the link numbers.

EXAMPLES For examples defining overlay structures, see Chapter 5.

RELATED SWITCHES /LINK, /OVERLAY, /PLOT

/NOENTRY

FORMAT /NOENTRY:(symbol,symbol,...)

FUNCTION Deletes entry name symbols from LINK's overhead tables when loading overlays, thereby saving space at run time. If you know that execution of the current load will not reference certain entry points, you can use /NOENTRY to delete them.

/NOENTRY differs from /NOREQUEST in that /NOREQUEST deletes requests for symbols, while /NOENTRY deletes symbols that might be requested.

EXAMPLES

```
*/ENTRY RET
[LNKLSS  LIBRARY SEARCH SYMBOLS (ENTRY POINTS)]
     SQRT.   3456
*/NOENTRY:(SQRT.) RET
*/ENTRY RET
*
```

Deletes SQRT. so that it cannot be used to fulfill a symbol request.

OPTIONAL NOTATIONS You can omit the parentheses if only one symbol is given.

RELATED SWITCHES /ENTRY, /EXCLUDE, /NOEXCLUDE, /INCLUDE, /NOINCLUDE, /MISSING, /REQUEST, /NOREQUEST

/NOINCLUDE

FORMAT /NOINCLUDE

FUNCTION Clears requests for modules that were specified in a previous /INCLUDE.

EXAMPLE

```
*LIB1/INCLUDE:(MOD1,MOD3) RET
*/NOINCLUDE RET
*
```

Loads MOD1 and MOD3 from LIB1. However, if the modules are not found immediately, stop searching.

RELATED SWITCHES /INCLUDE, /EXCLUDE, /MISSING

/NOINITIAL

FORMAT /NOINITIAL

FUNCTION Prevents loading of LINK's initial global symbol table (JOBDAT). The /NOINITIAL switch cannot operate after the first file specification because JOBDAT will be already loaded. The initial global symbol table contains the JBxxx symbols described in Appendix C.

The /NOINITIAL switch is commonly used for:

- Loading LINK itself (to get the latest copy of JOBDAT).
- Loading a private copy of JOBDAT (to alter if necessary).
- Building an EXE file that will eventually run in executive mode (for example, a monitor or bootstrap loader).
- Building a TOPS-20 native program which does not use a JOBDAT area.

EXAMPLES

```
*/NOINITIAL RET
*
```

/NOLOCAL

FORMAT

```
/NOLOCAL
```

FUNCTION

Suspends the effect of a preceding /LOCALS switch so that local symbol tables will not be loaded with their modules.

The /LOCALS and /NOLOCAL switches may be used either locally or globally. If the switch is suffixed to a file specification, it applies only to that file; if it is not suffixed to a file specification, it applies to all following files in the same command string.

This switch is useful if you need to conserve memory space, because local symbols are loaded into the low segment by default.

EXAMPLES

```
*/LOCALS A,B/NOLOCAL,C,/NOLOCAL D<RET>
*
```

Loads A with local symbols, B without local symbols, C with local symbols, and D without local symbols.

OPTIONAL NOTATIONS

Abbreviate /NOLOCAL to /N.

RELATED SWITCHES

/LOCALS

/NOREQUEST

FORMAT

```
/NOREQUEST:(symbol,symbol,...)
```

FUNCTION

Deletes references to links from LINK's overhead tables when loading overlay programs. If you know that the execution of the current load will not require certain links, you can use /NOREQUEST to delete references to them.

/NOREQUEST differs from /NOENTRY in that /NOENTRY deletes symbols that might be requested, while /NOREQUEST deletes the requests for them.

EXAMPLES

```
*/REQUEST RET
[LNKRER  REQUEST EXTERNAL REFERENCES]
     ROUTN.
     SQRT.
*/NOREQUEST:(ROUTN.,SQRT.) RET
*
```

Deletes references to ROUTN. and SQRT.

OPTIONAL NOTATIONS

You can omit the parentheses if only one symbol is given.

RELATED SWITCH

/NOENTRY

/NOSEARCH

FORMAT /NOSEARCH

FUNCTION Suspends the effect of a previous /SEARCH switch. Files named between a /SEARCH and the next /NOSEARCH are searched as libraries, so that modules are loaded only to resolve global references.

The /SEARCH and /NOSEARCH switches may be used either locally or globally. If the switch is suffixed to a file specification, it applies only to that file; if it is not suffixed to a file specification, it applies to all following files in the same command string.

EXAMPLES

```
*FILE1(RET)
*/SEARCH A,B/NOSEARCH,C,/NOSEARCH D(RET)
*
```

Searches A, loads B, searches C, and loads D.

RELATED SWITCHES /SEARCH

/NOSTART

FORMAT /NOSTART

FUNCTION Directs LINK to disregard any start addresses found after the /NOSTART switch. Normally LINK keeps the most recent start address found, overwriting any previously found. The /NOSTART switch prevents this replacement.

EXAMPLES

```
*MAIN1,/NOSTART MAIN2,MAIN3(RET)
*
```

Directs LINK to save the start address from MAIN1 instead of replacing it with other start addresses from MAIN2 and MAIN3.

RELATED SWITCHES /START

/NOSYMBOL

FORMAT /NOSYMBOL

FUNCTION Prevents construction of user symbol tables. Symbols are then not available for the map file, but the header for the file can still be generated by the /MAP switch.

The /NOSYMBOL switch prevents writing an ALGOL SYM file if it would otherwise have been written.

If you do not need the map file or symbols, you can speed loading by using the /NOSYMBOL switch.

EXAMPLES

```
*/NOSYMBOLRET
*
```

/NOSYSLIB

FORMAT

```
/NOSYSLIB:(keyword,...,keyword)
```

FUNCTION

Prevents automatic search of the system libraries named as keywords. LINK usually searches system libraries at the end of loading to satisfy unresolved global references. The /NOSYSLIB switch prevents this search.

The /NOSYSLIB switch can also be used to terminate searching of libraries that were specified in a previous /SYSLIB switch. When you specify searching of a library with /SYSLIB, that library will continue to be searched for every module you load. You can use /NOSYSLIB to specify libraries that should not be searched. Refer to /SYSLIB for more information.

The permitted keywords and the libraries they specify are listed below. Only those printed in **boldface** specify libraries supported by DIGITAL.

ANY	**Prevents all library searches.**
ALGOL	**Prevents search of ALGLIB.**
BCPL	Prevents search of BCPLIB.
COBOL	**Prevents search of LIBOL or C74LIB.**
F40	Prevents search of LIB40.
FORTRAN	**Prevents search of FORLIB.**
NELIAC	Prevents search of LIBNEL.
PASCAL	Prevents search of PASLIB.
SAIL	Prevents search of SAILIB.
SIMULA	Prevents search of SIMLIB.

EXAMPLES

```
*/NOSYSLIB:(ALGOL,COBOL) RET
*
```

Prevents search of the system libraries ALGLIB and LIBOL.

OPTIONAL NOTATIONS

If you omit keyword it defaults to ANY.
You can omit parentheses if only one keyword is given.

/NOUSERLIB

FORMAT

filespec/NOUSERLIB

FUNCTION

Discontinues automatic searching of the specified file at each /LINK or /GO switch. If you need a file searched for some links but not others, you can use the /USERLIB and /NOUSERLIB switches to enable and disable automatic search of the file.

EXAMPLES

```
*/OVERLAY RET
*MYFORL/USERLIB:FORTRAN RET
*MOD1/LINK:MOD1 RET
*/NODE:MOD1 MOD2/LINK:MOD2 RET
*MYFORL/NOUSERLIB RET
*
```

Loads the overlay handler; requests search of MYFORL as a FORTRAN library; loads MOD1 and MOD2 as links; discontinues search of MYFORL.

OPTIONAL NOTATIONS

If you omit the filespec, LINK discontinues search of all user libraries.

RELATED SWITCHES

/USERLIB

/ONLY

FORMAT /ONLY:keyword

FUNCTION Directs LINK to load only the specified segment of two-segment modules. The permitted keywords are:

HIGH	Loads only high segments.
LOW	Loads only low segments.
BOTH	Loads both segments.

The /ONLY switch is ignored for one-segment modules and for PSECTed modules.

EXAMPLES

```
*/ONLY:HIGH MOD1,MOD2RET
*MOD3/ONLY:BOTHRET
*
```

Loads high segment for MOD1 and MOD2; loads both segments for MOD3.

/OTSEGMENT

FORMAT /OTSEGMENT:keyword

FUNCTION Specifies the time and manner of loading the object-time system.

The permitted keywords are:

DEFAULT Suspends the effect of a previous /OTSEGMENT:SHAR or /OTSEGMENT:NONSHAR switch.

HIGH Loads the object-time system into the high segment.

LOW Loads the object-time system into the low segment.

NONSHARABLE

Loads the object-time system into user core image at load time. The user program may have code in both segments. The object-time system may have code in both segments.

SHARABLE

Binds the object-time system at execution time. The user program is in the low segment and the object-time system is in the high segment.

LINK's default action is to bind the object-time system at execution time. This normal action occurs if **none** of the following are true.

- You specify /OTSEGMENT:NONSHARABLE.
- You have loaded any code into the high segment.
- You have specified /SEGMENT:HIGH for some modules.
- You have specified /SYMSEG:HIGH.
- Your low segment is too big for sharable object-time systems to fit.

If **any** of these is true, a non-sharable object-time system is loaded as part of your program.

EXAMPLES

```
*MYPROG/SYSLIB/OTSEGMENT:NONSHAR RET
*
```

Loads a non-sharable copy of the object-time system as part of your program.

RELATED SWITCHES /SEGMENT

/OVERLAY

FORMAT filespec/OVERLAY:(keyword,...,keyword)

FUNCTION Initiates construction of an overlay structure. For a discussion of overlay structures, see Chapter 5.

The permitted keywords and their meanings are listed below. The default settings are printed in **boldface**.

ABSOLUTE Specifies that links are absolute. This is the default situation when overlays are loaded. The inverse situation is to use /OVERLAY:RELOCATABLE. Relocatable overlays are described in Chapter 5.

LOGFILE Outputs runtime overlay messages to your terminal.

NOLOGFILE Suppresses output of runtime overlay messages.

NONWRITABLE Specifies that links are nonwritable.

NOWARNING Suppresses overlay warning messages.

PATH Specifies that each link path will be loaded with its link.

RELOCATABLE Specifies that links are relocatable.

TREE Specifies that the overlay will have a tree structure.

WARNING Outputs overlay warning messages to user terminal.

WRITABLE Specifies that the links are writable. Refer to Chapter 5 for more information.

EXAMPLES See Chapter 5.

OPTIONAL NOTATIONS You can omit the parentheses if only one keyword is given.

/PATCHSIZE

FORMAT

/PATCHSIZE:n

Where n is a positive decimal integer.

FUNCTION

Allocates n words of storage to precede the symbol table. The allocated storage is in the same segment (high or low) as the symbol table. The default is /PATCHSIZE:64.

The storage allocated is available for patching or for defining new symbols with DDT, and is identified by the global symbol "PAT.."

EXAMPLES

```
*/SYMSEG:HIGH/PATCHSIZE:200(RET)
*
```

Loads the symbol table in the high segment after allocating 200 words between the last loaded module and the symbol table.

OPTIONAL NOTATIONS

You can specify the patchsize in octal.

RELATED SWITCHES

/SYMSEG

/PLOT

FORMAT filespec/PLOT

FUNCTION Directs LINK to output a tree diagram of your overlay structure. You can have the diagram formatted for a plotter (by default) or for a line printer (by giving the device as LPT:).

Each box in the diagram shows a link number, its name (if you gave one with the /LINK switch), and its relationship to other links (as defined by your commands).

The /PLOT switch cannot precede the /OVERLAY switch.

EXAMPLES See Chapter 5.

OPTIONAL NOTATIONS LINK has default settings for the size of the overlay diagram and the increment for drawing lines. You can override these by giving the /PLOT switch in the form:

```
filespec/PLOT:(LEAVES:value,INCHES:value,STEPS:value)
```

Where the values for each parameter define:

INCHES Width of diagram in inches. The defaults are INCHES:29 for plotter and INCHES:12 for line printer.

LEAVES Number of links without successors that can appear in one row. The defaults are LEAVES:16 for plotter and LEAVES:8 for line printer.

STEPS Increments per inch for drawing lines. The defaults are STEPS:100 for plotter and STEPS:20 for line printer.

For line printer diagrams, you cannot give INCHES or LEAVES different from the defaults. The STEPS parameter should be between 10 and 25.

For plotter diagrams, you should give INCHES and LEAVES in a ratio of about 2 to 1. For example, INCHES:40 and LEAVES:20.

If LINK cannot design the diagram on one page, it will automatically design subtrees for diagrams on more pages.

RELATED SWITCHES /LINK, /NODE, /OVERLAY

/PLTTYP

FORMAT /PLTTYP:keyword

FUNCTION Allows a user to specify the type of plot file to be generated by the /PLOT switch.

KEYWORDS

DEFAULT Generate output for a printer only if the device is a printer or terminal.

PLOTTER Generate output for a plotter.

PRINTER Generate output for a printer.

EXAMPLES

```
.
.R LINK(RET)

*TEST/OVERLAY(RET)
*DSK:TEST/PLOT/PLTTYP:PRINTER(RET)
*OVL0,OVL1/LINK:TEST(RET)
*/NODE:TEST  OVL2 /LINK:LEFT(RET)
*/NODE:LEFT  OVL5 /LINK:LEFT1(RET)
*/NODE:LEFT  OVL6 /LINK:LEFT2(RET)
*/NODE:TEST  OVL3,OVL4 /LINK:RIGHT(RET)
*TEST /SAVE /GO(RET)

EXIT

.
```

Causes all output from the /PLOT switch to be in line printer format.

RELATED SWITCHES /PLOT

/PVBLOCK

FORMAT /PVBLOCK:keyword

FUNCTION Requests a program data vector from LINK and gives the user control over where the vector goes. The vector specified with this switch is the primary data vector, and it therefore supercedes any vectors specified in Type 1100-1107 blocks. Refer to Section 4.2.1.

NOTE

> This switch functions only under LINK version 5 and TOPS-20 version 5. Refer to Section 3.5 for more information about the use of extended addressing.

Keywords and their meanings are:

DEFAULT disables the previous /PVBLOCK:HIGH or /PVBLOCK:LOW, restoring LINK's default action. The default for one-segment programs is to position the program data vector after the the program, before the patch area and the symbol table. The default for two-segment programs is to position the program data vector at the end of the high segment.

HIGH places the program data vector at the end of the high segment.

LOW places the program data vector at the end of the low segment.

NONE prevents the loading of the program data vector.

PSECT:name places the program data vector for the named PSECT at the end of the PSECT (after allocating any space required by the /PATCHSIZE switch).

EXAMPLE

```
*TEST1 /PVBLOCK:DEFAULT(RET)
*TEST1 /SAVE /GO(RET)
@GET TEST1(RET)
@INFORMATION VERSION(RET)
 2102 TOPS-20 DEVELOPMENT SYSTEM, TOPS-20 MONITOR 5(4451)
 TOPS-20 COMMAND PROCESSOR 5(706)
 PROGRAM IS TEST1
PDVS:    PROGRAM NAME TEST1, VERSION
@
```

THIS PAGE INTENTIONALLY LEFT BLANK

This example loads a program (TEST1) that allocates memory words. The /PVBLOCK switch is used when the program is loaded, to request a program data vector. After the program is saved and loaded (using the GET system command), the INFORMATION VERSION command shows that a PDV has indeed been allocated to the program.

OPTIONAL NOTATION If you specify /PVBLOCK with no keyword, DEFAULT is the default.

/PVDATA

FORMAT /PVDATA:keyword:value

FUNCTION changes the contents of a program data vector block specified with the /PVBLOCK switch. The /PVDATA switch also allocates storage for the program data vector. If the storage to be allocated conflicts with any PSECT, LINK issues a message with the severity level of 16, and does not write the program data vector information into the image or EXE file. Refer to Section 4.2.1.

NOTE

> This switch functions only under LINK version 5 and TOPS-20 version 5. Refer to Section 3.5 for more information about extended addressing.

Keywords and their intended values are:

NAME	program name
VERSION	a global symbol or numeric value.
MEMORY	address (absolute or symbolic) of a user supplied memory map, suppressing the map generated by LINK
PROGRAM	address (absolute or symbolic) of a program-specific data block
CBLOCK	address (absolute or symbolic) of a customer-defined block
START	start address (absolute or symbolic)

EXAMPLE

```
*TEST1 /PVDATA:NAME:STORAG<RET>
*TEST1 /SAVE /GO<RET>
@GET TEST1<RET>
@INFORMATION VERSION<RET>
 2102 TOPS-20 DEVELOPMENT SYSTEM, TOPS-20 MONITOR 5(4451)
 TOPS-20 COMMAND PROCESSOR 5(706)
 PROGRAM IS TEST1
PDVS: PROGRAM NAME STORAG, VERSION
```

This example shows a program (TEST1) being loaded. Using the /PVDATA switch, the program is named STORAG. After the program is saved and again loaded, the INFORMATION VERSION command shows that TEST1 has a program data vector named STORAG.

/REQUEST

FORMAT /REQUEST

FUNCTION Requests terminal typeout of all external references to other links. (LINK recognizes only those that use the standard calling sequence.)

If you use /REQUEST to get the names of external references, you can then either delete the references with the /NOREQUEST switch, or load the referenced modules.

EXAMPLES

```
*/REQUEST(RET)
[LNKRER  REQUEST EXTERNAL REFERENCES]
     ROUTN.
     SQRT.
*/NOREQUEST:ROUTN.(RET)
*/SEARCH LIB1(RET)
*
```

Obtains the external references ROUTN. and SQRT.; deletes the request for ROUTN.; searches the file LIB1 for a module containing the entry point SQRT.

RELATED SWITCHES /NOREQUEST

/REQUIRE

FORMAT

/REQUIRE:(symbol,...,symbol)

FUNCTION

Generates global requests for the specified symbols. LINK uses these symbols as library search symbols (entry points).

/REQUIRE differs from /INCLUDE in that /INCLUDE requests a module by name, while /REQUIRE requests an entry name symbol. Thus you can use /REQUIRE to specify a function (for example, SQRT.) even if you do not know the module name.

You can use /REQUIRE to load a module into a link common to all links that reference the module.

Note that the global requests generated by the /REQUIRE switch do not use the standard calling sequence, and are therefore not visible to the /REQUEST switch.

EXAMPLES

```
*/UNDEFINED(RET)
[LNKUGS  NO UNDEFINED GLOBAL SYMBOLS]
*/REQUIRE:(ROUTN.,SQRT.)(RET)
*/UNDEFINED(RET)
[LNKUGS  2 UNDEFINED GLOBAL SYMBOLS]
     ROUTN.
     SQRT.
*
```

OPTIONAL NOTATIONS

You can omit the parentheses if only one symbol is given.

/REWIND

FORMAT /REWIND

FUNCTION Rewinds the current input or output device if the device is a tape. If not, the switch is ignored.

EXAMPLES

```
*MTA0:/REWINDRET
*
```

Rewinds tape on MTA0:.

/RUNAME

FORMAT /RUNAME:name

FUNCTION Assigns a job name for execution of your program. This name is used only by the SYSTAT and INFORMATION modules of the system monitor.

EXAMPLES

```
*/RUNAME:LNKDEV(RET)
*
```

Assigns the name LNKDEV for job execution.

/SAVE

FORMAT filespec/SAVE

FUNCTION Directs LINK to create an EXE file with the specified filespec. Unless you have specified otherwise, the file type will be EXE.

Note that if you want to run the saved file with the system command, the file extension must be .EXE.

EXAMPLES

```
*MYPROG RET
*DSKZ:GOODIE.EXE/SAVE/GO RET
*
```

Directs LINK to save the linked version of MYPROG as GOODIE.EXE on DSKZ:.

/SEARCH

FORMAT /SEARCH

FUNCTION Directs LINK to load selectively from all following files up to the next /NOSEARCH or /GO. These files are searched as libraries, and only modules whose entry point name resolves a global request are loaded.

Using /NOSEARCH discontinues the library search mode, but for each link the system libraries are still searched (unless you used the /NOSYSLIB switch), and user libraries are still searched (if you used the /USERLIB switch).

The /SEARCH and /NOSEARCH switches may be used either locally or globally.

EXAMPLES

```
*/SEARCH A,B/NOSEARCH,C,/NOSEARCH D RET
*
```

Searches A, loads B, searches C, and loads D.

RELATED SWITCHES /NOSEARCH

/SEGMENT

FORMAT

```
/SEGMENT:keyword
```

FUNCTION

Specifies which segment is to be used for loading following modules. FORTRAN object code is an exception; both segments are loaded into the low segment unless one or more of the following is true:

- You used the /OTSEGMENT:NONSHARABLE switch.
- You used the /SEGMENT:HIGH switch to load code into the high segment.
- You used the /SEGMENT:DEFAULT switch to load code into both segments.
- Some code is already loaded into the high segment.

The keywords for the /SEGMENT switch are:

DEFAULT	Suspends effect of /SEGMENT:LOW or /SEGMENT:HIGH.
HIGH	Load into high segment, even if impure code.
LOW	Loads into low segment.
NONE	Same as DEFAULT.

If the switch is suffixed to a file specification, it applies only to that file; if it is not suffixed to a file specification, it applies to all following files in the same command string.

EXAMPLES

```
*/SEGMENT:LOW MOD1,MOD2,/SEGMENT:HIGH MOD3 RET
*
```

Loads MOD1 and MOD2 into the low segment; loads MOD3 into the high segment even if its code is impure.

RELATED SWITCHES

/OTSEGMENT

/SET

FORMAT

/SET:name:address

Where name is .HIGH., .LOW., or a PSECT name, and address is an octal address or a defined symbol.

FUNCTION

Sets the loading position of a PSECT, or sets the .HIGH. or .LOW. relocation counter.

For setting the loading position of a PSECT, name is the name of the PSECT, and address is a virtual memory address. The /SET switch must precede the modules that will make up the specified PSECT. The /SET switch is not needed if the REL files already contain origin information.

NOTE

> If you load PSECTs so that the resulting core image contains gaps, you must generate an EXE file and execute that file (rather than executing the loaded core image). It is good practice to generate an .EXE file for all PSECTed programs.

If you do not ask for an .EXE file and you need one, LINK will generate one for you.

EXAMPLES

```
*/SET:A:200000 RET
*
```

Specifies that the PSECT named A is to be loaded with its origin at address 200000.

```
*/SET:.HIGH.:400000 RET
*
```

Sets the high segment relocation counter .HIGH. to the address 400000. Note that saying /SET:.HIGH. causes a high segment to appear and a vestigial JOBDAT area to be built.

RELATED SWITCHES

/COUNTER, /LIMIT

/SEVERITY

FORMAT /SEVERITY:n

FUNCTION Specifies that messages of severity level greater than n will terminate the job, where n is a decimal number between 0 and 30 inclusive. Level 31 messages always terminate the job.

The defaults are /SEVERITY:24 for timesharing jobs, and /SEVERITY:16 for batch jobs.

EXAMPLES

```
*/SEVERITY:30RET
*
```

Specifies that only level 31 messages are fatal.

/SKIP

FORMAT

```
/SKIP:n
```

Where n is a positive decimal integer.

FUNCTION

Skips forward over n files on the current tape device. (A tape device remains current only until end-of-line or until another device is specified, whichever occurs first.) If the device is not a tape, the switch is ignored.

EXAMPLES

```
*MTA0:/SKIP:4RET
*
```

Skips forward over 4 files on MTA0:.

RELATED SWITCHES

/BACKSPACE, /MTAPE, /REWIND, /UNLOAD

/SPACE

FORMAT /SPACE:n

Where n is a positive decimal integer.

FUNCTION Specifies that n words of memory will follow the current link at execution time. This memory allocation will not increase the size of the overlay file, but it will increase the size of the program at run time.

The /SPACE switch is used to allocate space for use by the object time system. The OTS uses this space for I/O buffers, and as scratch space in FORTRAN and heap space in ALGOL.

You should place the /SPACE switch before the first /LINK switch, to ensure allocation for the root link. It is possible to allocate space after one or more overlays are linked. This might be useful if an overlay has unusual storage requirements: buffers for a file which is open only while that overlay is resident, or a large local matrix. To allocate space between overlays, use /SPACE when loading the overlay that will be using this file or matrix. LINK allows one /SPACE switch for the root node, and one for each overlay.

The default amount of memory allocated, if you do not specify /SPACE, is 2000 for the root link and 0 (zero) for other links.

If the space allocated for a relocatable link is too small, the overlay handler can relocate it. If the space allocated for an absolute link is too small, a fatal error occurs.

EXAMPLES

```
*/OVERLAY(RET)
*TEST/SPACE:90/LINK:MAIN(RET)
*       /NODE:MAIN SUB1/LINK:SUB1(RET)
*       /NODE:MAIN SUB2/LINK:SUB2(RET)
*
```

Allocates 90 words of memory to follow the root link for the program. See Chapter 5 for a discussion on overlay.

OPTIONAL NOTATIONS You can specify the number of words in octal.

/START

FORMAT

```
/START:symbol
/START:address
/START
```

Where symbol is a defined global symbol.

FUNCTION

Specifies the start address for the loaded program, and prevents replacement by any start addresses found later. You can use the /START switch with no argument to disable a previously given /NOSTART switch.

EXAMPLES

```
*MAIN1/START:ENTRY1,MAIN2,MAIN3(RET)
*
```

Defines the start address as ENTRY1 in MAIN1, and prevents replacement of this start address by any others found in MAIN2 or MAIN3.

OPTIONAL NOTATIONS

You can specify the start address in octal.

RELATED SWITCHES

/NOSTART

/SUPPRESS

FORMAT

```
/SUPPRESS:symbol
```

Where symbol is a previously defined global symbol.

FUNCTION

Used to suppress a previously defined global symbol. If the symbol is unknown, this switch has no effect. You should use this switch when two modules define a global symbol and you wish to suppress the unwanted definition. The symbol is removed from LINK's internal tables; it will not appear in map and SYM files, nor in the symbol table supplied to DDT.

This switch is used to suppress errors from multiply-defined symbols. When LINK encounters a new definition for a previously defined symbol, the new definition will supersede the old definition.

EXAMPLES

```
*/SUPPRESS:ENTPTR(RET)
*
```

This switch would suppress any definition attached to ENTPTR.

/SYFILE

FORMAT filespec/SYFILE:keyword

FUNCTION Requests LINK to output a symbol file to the given filespec, and sets the /SYMSEG:DEFAULT switch. If you previously specified /NOSYM, the /SYFILE switch has no effect.

The symbol file contains global symbols sorted for DDT. If you used the /LOCALS switch, it also contains local symbols, module names, and module lengths.

The permitted keywords and their meanings are:

ALGOL Requests symbols in ALGOL's format. The first word of the table is 1044,,count. The remaining words are copied out of Type 1044 REL blocks. If an ALGOL main program has been loaded, then /SYFILE:ALGOL becomes the default.

RADIX-50 Requests symbols in Radix-50 format. The first word of the table is negative. Each symbol requires two words in the table: the first is the symbol name in Radix-50 format; the second is the symbol value.

TRIPLET Requests symbols in triplet format. The first word of the table is zero. Each symbol requires three words in the table: the first word contains flags; the second is the symbol name in SIXBIT; the third is the symbol value.

EXAMPLES

```
*SYMBOL/SYFILE(RET)
*
```

Creates a symbol file called SYMBOL with the symbols in Radix-50 format.

OPTIONAL NOTATIONS If you omit the keyword, RADIX-50 is assumed.

/SYMSEG

FORMAT /SYMSEG:keyword

FUNCTION Places the symbol table so that it will not be overwritten during execution or debugging.

Keywords and their meanings are:

DEFAULT Places the symbol table in the low segment, except for overlayed programs, in which case symbols are not loaded by default.

HIGH Places the symbol table in the high segment.

LOW Places the symbol table in the low segment.

NONE Prevents loading of the symbol table.

PSECT:name Places the symbol table for the named PSECT at the end of the PSECT (after allocating any space required by the /PATCHSIZE switch).

EXAMPLES

```
*/SYMSEG:LOW RET
*
```

Places the symbol table in the program low segment.

RELATED SWITCHES /LOCALS, /NOLOCALS

/SYSLIB

FORMAT

```
/SYSLIB:keyword
```

FUNCTION

Forces searching of one or more system libraries, immediately after you end the command line. LINK will also automatically search a system library if code from the corresponding compiler has been loaded. By default, LINK searches the system libraries that are appropriate for the language compiler, after all the modules of the program are loaded. /SYSLIB forces the search to take place immediately.

After you specify a library with /SYSLIB, the library you specified will be searched every time you load a module, until you use /NOSYSLIB to end searching of that library.

The permitted keywords and the libraries they specify are listed below. Those printed in **boldface** specify libraries supported by DIGITAL.

ANY	**Forces search of all system libraries.**
ALGOL	**Forces search of ALGLIB.**
BCP	Forces search of BCPLIB.
COBOL	**Forces search of LIBOL or C74LIB.**
F40	Forces search of LIB40.
FORTRAN	**Forces search of FORLIB.**
NELIAC	Forces search of LIBNEL.
PASCAL	Forces search of PASLIB.
SAIL	Forces search of SAILIB.
SIMULA	Prevents search of SIMLIB.

EXAMPLES

```
*TEST1/SYSLIB:ALGOL RET
*TEST2/NOSYSLIB:ALGOL RET
*
```

Where TEST1 is a FORTRAN module, LINK will search both FORLIB and ALGLIB for TEST1. Where TEST2 is a FORTRAN module, LINK will search only FORLIB when TEST2 is loaded.

OPTIONAL NOTATIONS

You can omit the keyword. LINK will search all libraries for which corresponding code has been loaded.

RELATED SWITCHES

/NOSYSLIB

/TEST

FORMAT /TEST:keyword

FUNCTION Loads the debugging program indicated by keyword. Unlike the /DEBUG switch, /TEST causes execution to begin in the loaded program (not in the debugging module). This switch is useful if you expect the program to run successfully, but want the debugger available in case the program has errors.

The /TEST switch turns on the /LOCALS switch for the remainder of the load. You can override this by using the /NOLOCAL switch, but the override lasts only during processing of the current command string.

Local symbols for the debugging module itself are never loaded.

The permitted keywords and the programs they load are listed below. Only those printed in **boldface** are supported by DIGITAL.

ALGDDT	**Loads ALGDDT.**
ALGOL	**Loads ALGDDT.**
COBDDT	**Loads COBDDT.**
COBOL	**Loads COBDDT.**
DDT	**Loads DDT.**
FAIL	Loads SDDT.
FORDDT	**Loads FORDDT.**
FORTRAN	**Loads FORDDT.**
MACRO	**Loads DDT.**
PASCAL	Loads PASDDT.
PASDDT	Loads PASDDT
SAIL	Loads the SAIL debugger.
SDDT	Loads the SAIL debugger.
SIMDDT	Loads SIMDDT.
SIMULA	Loads SIMDDT.

EXAMPLES

```
*MYPROG/TEST:FORTRAN RET
*
```

Loads MYPROG and FORDDT.

OPTIONAL NOTATIONS If you give no keyword with /TEST, the default is either DDT or the debugging program specified by the /DDEBUG switch.

RELATED SWITCHES /DDEBUG, /DEBUG

/UNDEFINED

FORMAT /UNDEFINED

FUNCTION Requests terminal typeout (in octal) of undefined global symbols. You can use /UNDEFINED to get a list of undefined symbols, and then define them with the /DEFINE switch.

EXAMPLES

```
*/UNDEFINED(RET)
[LNKUGS  2 UNDEFINED GLOBAL SYMBOLS]
     A        400123
     IGOR     402017
*/DEFINE:(A:591,IGOR:1)(RET)
*
```

Gives the decimal values 591 and 1 to A and IGOR, respectively.

OPTIONAL NOTATIONS You can abbreviate /UNDEFINE to /U.

RELATED SWITCHES /DEFINE, /VALUE

/UNLOAD

FORMAT device/UNLOAD

FUNCTION Rewinds and unloads the specified tape device. (This switch is ignored if the current device is not a tape device.) The /UNLOAD is not performed until the current file processing is completed.

EXAMPLES

```
*MTA0:/UNLOAD RET
*
```

Rewinds and unloads MTA0.

RELATED SWITCHES /BACKSPACE, /MTAPE, /REWIND, /SKIP

/UPTO

FORMAT /UPTO:addr

Where addr is the upper limit to which the symbol table can grow. The addr value can be replaced by a symbol.

FUNCTION Sets an upper limit to which the symbol table can expand.

EXAMPLE

```
*/UPTO:550000RET
*
```

Included in a FORTRAN load, this switch would override the default upper bound for the symbol table. This might be used if FOROTS begins above 400000.

RELATED SWITCH /SYMSEG

/USERLIB

FORMAT

```
filespec/USERLIB:(keyword,...,keyword)
```

FUNCTION

Directs LINK to search the user library given by filespec before searching system libraries. The keyword indicates that the given library is to be searched only if code from the corresponding compiler was loaded.

Keywords and their meanings are given below. Only those printed in **boldface** indicate compilers and libraries supported by DIGITAL.

ALGOL	**Search as an ALGOL library.**
ANY	**Always search this library.**
BCPL	Search as a BCPL library.
COBOL	**Search as a COBOL library.**
FORTRAN	**Search as a FORTRAN library.**
NELIAC	Search as a NELIAC library.
PASCAL	Search as a PASCAL library.
SAIL	Search as a SAIL library.
SIMULA	Search as a SIMULA library.

EXAMPLES

```
*MYFORL/USERLIB:FORTRAN RET
*
```

Directs LINK to search the user library MYFORL (before searching FORLIB) if any FORTRAN-compiled code is loaded.

OPTIONAL NOTATIONS

You can omit the parentheses if only one keyword is given.

RELATED SWITCHES

/NOUSERLIB

/VALUE

FORMAT

/VALUE:(symbol,symbol,...)

FUNCTION

Requests terminal typeout of the values of each specified global symbol. LINK will type out its LNKVAL message, giving the symbol, its current value, and its status. The status is one of the following:

defined — The symbol and its value are known.
undefined — The symbol is known, but has no value.
common — The symbol is known and is defined as COMMON.
unknown — The symbol is not in the symbol table.

EXAMPLES

```
*TEST(RET)
*SPEXP(RET)
*SPEX2(RET)
*/VALUE:(SPEX2,DPEXP,X2,X)(RET)
[LNKVAL  SPEX2    460    DEFINED]
[LNKVAL  DPEXP    221    UNDEFINED]
[LNKVAL  X2       324    COMMON, LENGTH 1 (DECIMAL)]
[LNKVAL  X        UNKNOWN]
*
```

For DPEXP, 221 is the last location at which the symbol was referenced, and marks the beginning of a fixup chain.

OPTIONAL NOTATIONS

You can omit the parentheses if only one symbol is given.

/VERBOSITY

FORMAT /VERBOSITY:keyword

FUNCTION Specifies the length of LINK messages.

The permitted keywords and their meanings are:

SHORT	Output only the 6-letter code.
MEDIUM	Output the 6-letter code and the medium-length message (usually one line or less).
LONG	Output the 6-letter code, the medium-length message, and the long message (usually several lines).

For a few messages no long message exists; in these cases the LONG specification is ignored.

EXAMPLES

```
@LINK RET
*FOO/VERBOSITY:SHORT RET

%LNKFLE

[ PLEASE RETYPE THE INCORRECT PARTS OF THE FILE SPECIFICATION]

*FOO/VERBOSITY:MEDIUM RET

%LNKFLE LOOKUP ERROR (0) FILE WAS NOT FOUND DSK:FOO.REL

[ PLEASE RETYPE THE INCORRECT PARTS OF THE FILE SPECIFICATION]

*FOO/VERBOSITY:LONG RET

%LNKFLE LOOKUP ERROR (0) FILE WAS NOT FOUND DSK:FOO.REL
 THE NAMED FILE WAS NOT FOUND.  SPECIFY AN EXISTING FILE.
[ PLEASE RETYPE THE INCORRECT PARTS OF THE FILE SPECIFICATION]
```

RELATED SWITCHES /MESSAGE

/VERSION

FORMAT /VERSION:ic(j)-k

Where:

i = an octal number between 0 and 777 inclusive.

c = one or two alphabetic characters.

j = an octal number between 0 and 777777 inclusive.

k = an octal number between 0 and 7 inclusive.

FUNCTION Changes the value of .JBVER (location 137 in JOBDAT) and .JBHVR in the vestigial job data area.

If the switch is associated with an input specification, or with no specification, the version number is entered in .JBVER and .JBHVR (location 4 in the vestigial job data area).

There are four parts to the version arguments, given as i, c, j, and k above. The first number (i) gives the major version number. The character (c) gives the minor version. The second number (j) gives the edit number. The last number (k), which must be preceded by a hyphen (-), shows which group last modified the file (0 = DIGITAL development, 1 = other DIGITAL personnel, 2-7 = customer use).

EXAMPLES

```
*/VERSION:3A(461)-0 RET
*
```

Sets the version so that the major version is 3, the minor version is A, the edit number is 461, and the last group to modify the file was DIGITAL development.

3.3 ACCESSING ANOTHER USER'S FILE

LINK allows you to access another user's file in two ways. The first is to give a logical name in place of the device name; the second is to give a project-programmer number instead of a directory name. You can give either of these in a LINK command string.

For more information about referencing other users' files, refer to the TOPS-20 User's Guide.

3.3.1 Using Logical Names

To use a logical name in accessing another user's file, you must:

1. Give the DEFINE command to define a logical name (of no more than six characters) as the other user's directory name.

2. Use the logical name as the device name whenever giving the file specification.

3.3.2 Giving the DEFINE Command

To give the DEFINE command:

1. Type DEF and press the ESCAPE key; the system prints INE (LOGICAL NAME).

   ```
   @DEFINE (LOGICAL NAME)
   ```

2. Type the logical name, ending it with a colon; then type the directory name in angle brackets and RETURN:

   ```
   @DEFINE (LOGICAL NAME) BAK: <BAKER>RET
   @
   ```

   To check the logical name, give the INFORMATION (ABOUT) LOGICAL-NAMES command.

   ```
   @INFORMATION (ABOUT) LOGICAL-NAMES (OF) JOB RET
   BAK:  => <BAKER>
   @
   ```

3.3.3 Using the Logical Name

You can include the logical name in a command string as part of a file specification. To do this, type the logical name in place of a device name.

The following example shows how to load the file <BAKER>SPEC.REL. You must have already defined the logical name BAK: as <BAKER>.

```
@LINK RET
*BAK:SPEC.REL
```

3.3.4 Using Project-Programmer Numbers

To use a project-programmer number in accessing another user's file, you must:

1. Use the TRANSLATE command to find the corresponding project-programmer number for the given directory name.

2. Include the project-programmer number after the filename.

You do not have to define a logical name if you use a project-programmer number. However, project-programmer numbers sometimes change; therefore it is better to use logical names wherever possible.

3.3.4.1 **Using the TRANSLATE Command** - To use the TRANSLATE command, you must:

1. Type TRANSLATE and press ESC. The system prints (DIRECTORY).

   ```
   @TRANSLATE (DIRECTORY)
   ```

2. Type the appropriate directory name and press RET. The system prints the appropriate project-programmer number.

   ```
   TRANSLATE (DIRECTORY) <BAKER>RET

   <BAKER> IS [4,204]
   ```

You can also use the TRANSL program to make sure a project-programmer number is correct. Simply replace the directory name with the project-programmer number.

```
@TRANSLATE RET
TRANSLATE (DIRECTORY)[4,204] RET
[4,204] IS <BAKER>
```

3.3.4.2 **Using the Project-Programmer Number** - Because project-programmer numbers can change, you should use a logical name. You may include the project-programmer number in a LINK command string. To do this, type the project-programmer number after the file specification.

The following example shows how to load the file <BAKER>SPEC.REL by using a project-programmer number.

```
@LINK RET
*SPEC.REL[4,204]
```

3.4 LIBRARIES AND SEARCHES

A library is a file having one or more object modules; when a library is searched, a module is loaded from the file only if it satisfies an unresolved global reference.

System libraries are available to all users for searching. Most language translators also have libraries associated with them. The translators generate calls for subroutines or functions in their corresponding libraries, and library searches find and load the necessary modules.

LINK normally searches system libraries before completing its loading. The kinds of programs you load determine which libraries are searched. An object module usually has data telling LINK which translator generated the module. When LINK finds a /GO or /LINK switch in a command string, it searches a system library if a module from the corresponding translator was loaded.

For example, if you load a FORTRAN-compiled module, LINK will search the system FORTRAN library SYS:FORLIB.REL when a /GO or /LINK switch is processed. This search will resolve requests for FORTRAN-defined subroutines and functions.

You can change this normal search procedure by using LINK switches. The /SYSLIB switch requires LINK to search specified system libraries no matter what kind of modules were loaded. The /NOSYSLIB switch forbids search of specified system libraries. Using these two switches, you can select the time for searching system libraries.

The /USERLIB switch specifies that, for modules from a specified translator, a user library must be searched before the corresponding system library. For example, using the switch MYFORL/USERLIB:FORTRAN requires LINK to search MYFORL.REL before searching FORLIB. The /NOUSERLIB switch can suspend the effect of a /USERLIB switch.

The /SEARCH and /NOSEARCH switches, respectively turn on and off LINK's library search mode. When the library search mode is off (the initial default), LINK loads all modules from each input file you specify. When the library search mode is on (between a global /SEARCH switch and the next global /NOSEARCH switch at the end of the command line), LINK searches each specified input file as a library.

Using combinations of these search-related switches gives you precise control of library searches.

3.5 USING EXTENDED ADDRESSING WITH LINK

TOPS-20 version 5 provides the extended addressing feature for programs that use more than 512 pages of address space. This section of the LINK Manual refers to MACRO programming only. Before attempting to use extended addressing with another language, consult the documentation for that language.

When you load a program into a nonzero section, you must use the /PVDATA switch to set the start address. The start address must be a global (30-bit) address. The /PVDATA switch is described in Section 3.2.

If your program is using extended addressing, you must be sure to pay particular attention to your use of local and global symbols. LINK flags local (halfword) references to global (30-bit) symbols, and truncates the global symbols. When this truncation occurs, LINK notifies you by producing the %LNKFTH message. Refer to Appendix B for more information about the %LNKFTH message.

While you are writing programs that use extended addressing, you should keep the following restrictions in mind.

- Overlay programs cannot use nonzero sections.
- Only PSECTed programs can use nonzero sections.
- LINK cannot load the code or data of a single program into both section 0 and a nonzero section.
- LINK will not set up JOBDAT for a program loaded entirely in a nonzero section. Therefore, programs loaded into non-zero sections should use program data vectors. JOBDAT areas are described in Appendix C.
- LINK uses the SSAVE% JSYS to produce nonzero-section EXE files. LINK writes the EXE file itself to produce zero section EXE files.
- Programs should not put code into locations 0-20 of nonzero sections.

3.6 EXAMPLES USING LINK DIRECTLY

For the following examples, the loaded program is a FORTRAN program called MYPROG that writes the following:

```
This is written by MYPROG.
```

The following example shows an interactive execution of the program using a LINK command string. After running LINK, the command string calls for MYPROG to be loaded. Then the string MYLIB/USERLIB requests searching of the library DSK:MYLIB.REL at the end of loading. The /NOSYSLIB switch prevents searching the default system library (SYS:FORLIB.REL for FORTRAN programs). Finally the /EXECUTE switch directs LINK to execute the loaded program, and the /GO switch tells LINK that there are no more command strings.

```
@LINK(RET)
*MYPROG,MYLIB/USERLIB/NOSYSLIB/EXECUTE/GO(RET)
[LNKXCT MYPROG Execution]


This is written by MYPROG.

END OF EXECUTION
CPU TIME:  0.02 ELAPSED TIME:  0.05
EXIT
@
```

USING LINK DIRECTLY

The example below shows how to use LINK to load the program exactly as above, except that the program will be executed under the control of a debugging program (FORDDT for FORTRAN programs):

```
@LINK(RET)
*/DEBUG:FORDDT MYPROG,MYLIB/USERLIB/NOSYSLIB/GO(RET)
[LNKDEB FORDDT Execution]
STARTING FORTRAN DDT
>>START


This is written by MYPROG.

END EXECUTION
CPU TIME:  0.02 ELAPSED TIME:  0.08
EXIT
@
```

CHAPTER 4

OUTPUT FROM LINK

The primary output from LINK is the executable program formed from your input modules and switches. During its processing, LINK gives errors, warnings, and informational messages. At your option, LINK can generate any of several files.

4.1 THE EXECUTABLE PROGRAM

The executable program that LINK generates (called the core image) consists mostly of data and machine instructions from your object modules. In the core image, all relocatable addresses have been resolved to absolute addresses, and the values of all global references have been resolved.

You have several options for loading the program, depending on the purpose of the load. Those options are:

- Execute the program. To do this, include the /EXECUTE switch any place before the /GO switch. LINK will pass control to your program for execution.

- Execute the program under the control of DDT. To do this, use the /DEBUG switch before the first input file specification.

- Execute the program and debug it after execution. To do this, use the /TEST and /EXECUTE switches before the first input file specification. After execution, type DDT to the system to enter the debugging program.

- Save the core image as an EXE file. To do this, use the /SAVE switch. See Section 4.2.

4.2 OUTPUT FILES

At your option, LINK can produce any of the following output files:

- Saved (executable) file.

- Log file.

- map file.

- Symbol file.
- Plotter file (see Section 5.1).
- Overlay file (see Section 5.1).

4.2.1 Executable Files

The executable file, sometimes called the saved or EXE file, is a copy of the completed core image generated by LINK. You can create an executable file by supplying the /SAVE switch before the /GO switch when you are loading the program with direct commands to LINK. The executable file will retain the same file name as the source program, with a file type EXE.

Alternatively, you can type the file specification, followed by /SAVE, and the executable file will be written to the file you specified. If you load the program with the system LOAD command, you may then save the executable file by typing the system SAVE command.

You can run the executable file later, without running LINK, by using the system command RUN, or the two system commands GET and START. The following section describes the internal format of the executable file.

4.2.1.1 **Format of Sharable Save Files** - A sharable save file is divided into two main areas: the directory area, which contains information about the structure of the file, and the data area, which contains the data of the file.

The following diagram illustrates the general format of a sharable save file:

```
Directory         ========================
Area:             !  Directory Section   !
                  !                      !
                  !                      !
                  ------------------------
                  ! Entry Vector Section !
                  ------------------------
                  ! Program Data Vector  !
                  !       Section        !
                  ------------------------
                  ! Terminating Section  !
                  ========================
Data Area:        !    Data Section      !
                  !                      !
                  !                      !
                  !                      !
                  !                      !
                  !                      !
                  !                      !
                  !                      !
                  !                      !
                  ========================
```

NOTE

> The Program Data Vector area is useful only with TOPS-20 version 5 and later monitors. Earlier monitors ignore this area.

The directory area of the sharable save file has four distinct sections: the directory section, the entry vector section, the program data vector section, and the terminating section. The size of the directory area depends on the access characteristics of the pages in the data area of the save file. The directory area of the save file has three distinct sections: the directory section, the terminating section, and the data section.

Each of the sections in the directory area begins with a header word containing its identifier code in the left half and its length in the right half. Each section is described in the following paragraphs.

The directory section is the first of the three sections and describes groups of contiguous pages that have identical access. The length of this section varies according to the number of groups that can be generated from the data portion of the save file. The more data pages that can be combined into a single group, the fewer groups required, and the smaller the directory section.

The format of the directory section is as follows:

```
 0            8 9          17 18                       35
!=======================================================!
!      Identifier code       !      Number of words     !
!           1776             !   (including this word)  !
!                            !   in directory section   !
!=======================================================!
!    Access    !   Page number in file, or 0 if group   !
!     bits     !          of pages is all zero          !
!=======================================================!
!    Repeat    !        Page number in the process      !
!    count     !                                        !
!=======================================================!
                             .
                             .
                             .
!=======================================================!
! Access bits  !          Page number in the file       !
!=======================================================!
! Repeat count !        Page number in the process      !
!=======================================================!
```

PSECT attributes are used to set the access bits. Refer to the description of Block Type 24 in Appendix A. The bits currently defined in the directory section are:

B1   The process pages in this group are sharable

B2   The process pages in this group are writable

The remaining access bits in the directory section are zero.

The repeat count is the number (minus 1) of consecutive pages in the group described by the word pair. Pages are considered to be in a group when the following three conditions are met:

1. The pages are contiguous.

2. The pages have the same access.

3. The pages are allocated but not loaded.

A group of all zero pages is indicated by a file page number of 0.

The word pairs are repeated for each group of pages in the address space.

The entry vector section follows the directory section and points to the first word of the entry vector and gives the length of the vector.

```
 0                        17 18                        35
!=======================================================!
!      Identifier code      !      Number of words      !
!           1775            !   (including this word)   !
!                           !  in entry vector section  !
!=======================================================!
!                          254000                       !
!=======================================================!
!                     Starting Address                  !
!=======================================================!
```

This format is the default. However, if you make special provisions in your program, the format becomes the following. (Refer to the description of Block Type 7 in Appendix A and the description of the SFRKV JSYS in the Monitor Calls Manual for further information.)

```
 0                        17 18                        35
!=======================================================!
!      Identifier code      !      Number of words      !
!           1775            !   (including this word)   !
!                           !  in entry vector section  !
!=======================================================!
!             Number of words in entry vector           !
!=======================================================!
!                 Address of entry vector               !
!=======================================================!
```

The data for this section is the address of the entry vector.

The program data vector section may follow the entry vector section and contains the addresses at which the program data vectors begin (PDVAs). The format of the program data vector section is as follows:

```
 0                              17 18                          35
!==============================================================!
!        Identifier code          !        Number of words     !
!              1774               !   (including this word)    !
!                                 !  in data vector section    !
!==============================================================!
!                    Address of data vector 1                  !
!==============================================================!
!                    Address of data vector 2                  !
!==============================================================!
                                  .
                                  .
                                  .
!==============================================================!
!                    Address of data vector n                  !
!==============================================================!
```

The terminating section, called the end section, always immediately precedes the data section. The format of the terminating section is the following:

```
!==============================================================!
!        Identifier code          !                            !
!              1777               !             1              !
!==============================================================!
```

The data area follows the terminating section, beginning at the next page boundary.

4.2.1.2 **Program Data Vector** - A program data vector (PDV) is a block of data that LINK can write into memory when loading and linking a program. Refer to the /PVBLOCK and /PVDATA switches in Section 3.2.

The PDV resides in memory as a part of the program, and starts at a program data vector address (PDVA). User programs can use this data. Although TOPS-20 currently does not use the data in the PDV, words 13 and 14 of the PDV are provided for possible future system use.

The format of the program data vector is as follows:

Word	Symbol	Meaning
0	.PVCNT	Length of the PDV (including this word).
1	.PVNAM	Address of an ASCIZ string which is the program name.
2	.PVSTR	Program starting address.
3	.PVREE	Program reenter address.
4	.PVVER	Program version number.
5	.PVMEM	Pointer to a block describing the memory layout of the program. The first word of this block specifies the block length.
6	.PVSYM	Address of the program symbol table.
7	.PVCTM	Time at which the user program was compiled.

Word	Symbol	Meaning
10	.PVCVR	Version number of the compiler of main program.
11	.PVLTM	Time at which the program was loaded.
12	.PVLVR	Version number of LINK. (See /VERSION in Section 3.2.)
13	.PVMON	Address of a monitor data block. (Not currently used.)
14	.PVPRG	Address of a program data block. (Not currently used.)
15	.PVCST	Address of a customer-defined data block.

For more information about PDVs, refer to the PDVOP% JSYS in the Monitor Calls Manual.

4.2.2 LOG Files

A LOG file is generated if you use the /LOG switch. LINK then writes most of its messages into the specified file. You can control the kinds of messages entered in the LOG file by using the /LOGLEVEL switch. For an example of a LOG file, see Section 5.1.

4.2.3 Map files

The map file is generated if you use the /MAP switch. LINK constructs a symbol map in this file. The kinds of symbols included depends on your use of the /CONTENTS, /LOCALS, /NOLOCALS, and /NOINITIAL switches. For an example of a map file, see Section 5.1. For a list of /MAP options, refer to Section 3.2.2.

4.2.4 Symbol Files

The symbol file (or SYM file) is generated if you use the /SYFILE switch. This file contains all global symbols, module names, and module lengths, and, if you used the /LOCALS switch, all local symbols.

4.3 MESSAGES

During its processing, LINK issues messages about what it is doing, and about errors or possible errors it finds. LINK also responds to query switches such as /COUNTER, /ENTRY, /MISSING, /REQUEST, and /UNDEFINED.

Each LINK message has an assigned level and an assigned severity. (See Appendix B for the level and severity of each message.)

The level of a message determines whether it will be output to your terminal, the log file, or both. You can control this output by using the /ERRORLEVEL switch for the terminal and the /LOGLEVEL switch for the log file. LINK's defaults are /ERRORLEVEL:10 and /LOGLEVEL:10.

Responses to query switches and messages that require you to do something immediately are never output to the LOG file. For example, if you use the /UNDEFINE switch, LINK responds with the LNKUGS message; this message is output to the terminal but not to the log file.

The severity of a message determines whether LINK considers the message fatal (that is, whether the job is terminated). You can set the fatal severity with the /SEVERITY switch. The default severities are 24 for interactive jobs and 16 for batch jobs.

For both terminal messages and log file entries, LINK can issue short, medium, or long messages, depending on your use of the /VERBOSITY switch. For /VERBOSITY:SHORT, LINK gives only a 6-letter code; for /VERBOSITY:MEDIUM, LINK gives the code and a medium-length message; for /VERBOSITY:LONG, LINK gives the code, a medium-length message, and a long message.

Appendix B gives each 6-letter message code, its medium-length and long messages, and its level and severity.

CHAPTER 5

OVERLAYS

If your loaded program is too large to execute in one piece, you may be able to define an overlay structure for it. This permits the system to execute the program with only some parts at a time in your virtual address space. The overlay handler removes and reads in parts of the program, according to the overlay structure.

> NOTE
>
> You only need an overlay structure if your program is too large for your virtual address space. If the program can fit in your virtual space, you should not define an overlay structure for it; the monitor's page swapping facility is faster than overlay execution.

5.1 OVERLAY STRUCTURES

An overlay program has a tree structure. (The tree is usually pictured upside down.) The tree is made up of links, each containing one or more program modules. These links are connected by paths. Using LINK switches, you define each link and each path.

At the top of the (upside down) tree is the root link, which must contain the main program. First-level links are below the root link; each first-level link is connected to the root link by one path.

Second-level links are below the first-level links, and each is connected by a path to exactly one first-level link. A link at level n is connected by a path to exactly one link at level n-1.

Notice that a link can have more than one downward path (to successor links), but only one upward path (to predecessor links).

Figure 5-1 shows a diagram of an overlay structure with 5 links. The root link is TEST; the first-level links are LEFT and RIGHT; the second-level links are LEFT1 and LEFT2.

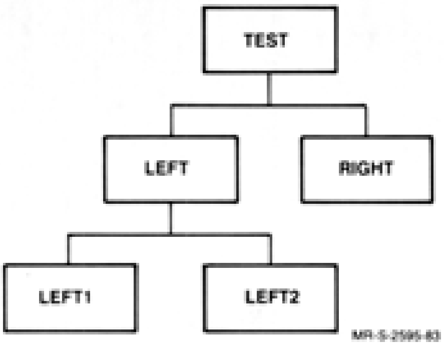


Figure 5-1 Example of an Overlay Structure

Defining an overlay structure allows your program to execute in a smaller space. This is because the code in a given link is allowed to make reference to memory only in links along a direct upward or downward path.

In the structure in Figure 5-1, the link LEFT can reference memory in itself, in the root link (TEST), or in its successor links LEFT1 and LEFT2. More generally, a link can reference memory in any link that is vertically connected to it.

Referencing memory in any other link is not allowed. For example, a path from LEFT1 to LEFT2 is not a direct upward or downward path.

Because of this restriction on memory references, only one complete vertical path (at most) is required in the virtual address space at any one time. The remaining links can be stored on disk while they are not needed.

5.1.1 Defining Overlay Structures

LINK has a family of overlay-related switches. These switches are described in detail in Section 3.2.2. The following example shows command strings for defining the overlay diagrammed in Figure 5-1. (Some of the command lines in this example are indented for clarity.)

```
*TEST/LOG/LOGLEVEL:2                     ;Define TEST.LOG
*/ERRORLEVEL:5                           ;Important messages
*TEST/OVERLAY                            ;Define TEST.OVL
*TEST/MAP                                ;Define TEST.MAP
*LPT:TEST/PLOT                           ;Request diagram
*OVL0,OVL1/LINK:TEST                     ;Root link
*      /NODE:TEST  OVL2/LINK:LEFT        ;Left branch
*            /NODE:LEFT  OVL5/LINK:LEFT1 ;Left-left branch
*           /NODE:LEFT  OVL6/LINK:LEFT2  ;Left-right branch
*       /NODE:TEST  OVL3,OVL4/LINK:RIGHT ;Right branch
*TEST/SAVE                               ;Define TEST.EXE
*/EXECUTE/GO
```

The first command string above defines the log file for the overlay. TEST/LOG specifies that the file is named TEST.LOG. The /LOGLEVEL:2 switch directs that messages of level 2 and above be entered in the log file.

In the second command string, the /ERRORLEVEL:5 switch directs that messages of level 5 and above be typed out on the terminal. The third command string, TEST/OVERLAY, tells LINK that an overlay structure is to be defined, and that the file for the overlay is to be TESTOVL.

The fourth command string, TEST/MAP, defines the file TEST.MAP, which will contain symbol maps for each link.

The next command string, LPT:TEST/PLOT, directs that a tree diagram of the overlay links be printed on the line printer.

The next command string, OVL0,OVL1/LINK:TEST, loads the files OVL0.REL and OVL1.REL into the root link. The /LINK:TEST switch tells LINK that no more modules are to be in the root link, and that the link name is TEST.

Each of the next four lines defines one link with a string of the form:

```
     /NODE:linkname filename/LINK:linkname
```

The /NODE:linkname switch specifies the previously defined link to which the present link is an immediate successor. The filenames/LINK:linkname part of the line names the files containing modules to be included in the current link and specifies the name of the link.

The first of these four lines begins with /NODE:TEST, which tells LINK that the link being defined is to be an immediate successor to TEST, the root link. Then (on the same line), the string OVL2/LINK:LEFT loads the file OVL2.REL, ends the link, and names it LEFT.

The next line, /NODE:LEFT OVL5/LINK:LEFT1, defines a link named LEFT1 containing the file OVL5.REL, and this link is an immediate successor to the link LEFT.

The next line, /NODE:LEFT OVL6/LINK:LEFT2, defines another immediate successor to LEFT, this time containing the file OVL6.REL and called LEFT2.

The last link is defined in the next line, /NODE:TEST OVL3,OVL4/LINK:RIGHT. This string defines the link RIGHT, which is an immediate successor to TEST and contains the files OVL3.REL and OVL4.REL.

The next-to-last line, TEST/SSAVE, directs LINK to create the saved file TEST.EXE. The last line, /EXECUTE/GO, specifies that the loaded program is to be executed, and that all commands to LINK are completed.

The process also produced an executable file TEST.EXE, which can be run using the RUN system command. However, to run the program, the file TEST.OVL must be present, because it provides the code for the links.

5.1.2 An Overlay Example

The following pages show terminal listings of the files associated with the example above. These pages are:

1. Terminal copy of the FORTRAN source files used in the overlay.
2. Terminal copy of the compilation of the source files.
3. Terminal copy of the interactive use of LINK to define and execute the overlay.
4. The file TEST.LOG generated by LINK, which shows the log messages issued during the load.
5. The file TEST.MAP generated by LINK, which shows symbol maps for the overlay.
6. The tree diagram requested by the LPT:/PLOT switch.

OVERLAYS

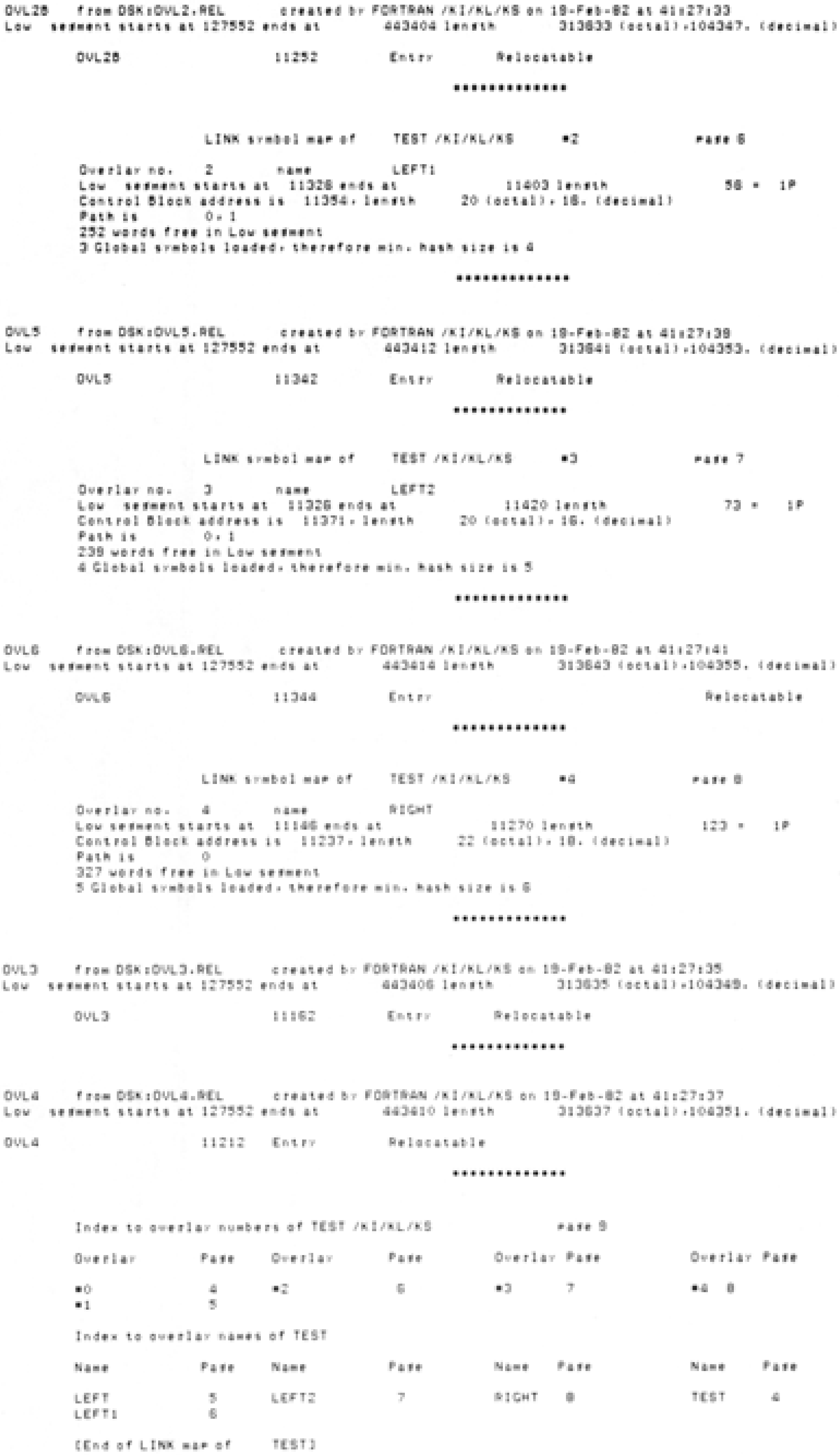
```
@type ovl0.for
        TYPE 1
1       FORMAT('1','Execution begins in main program OVL0')
        TYPE 11
11      FORMAT(1X,'OVL0 calls OVL2A')
        CALL OVL2A
        TYPE 2
2       FORMAT(/1X,'Return to OVL0')
        TYPE 21
21      FORMAT(1X,'OVL0 calls OVL4')
        CALL OVL4
        TYPE 2
        TYPE 3
3       FORMAT(/1X,'Execution ends in main program OVL0'//)
        STOP
        END

@type ovl1.for
        SUBROUTINE OVL1
        TYPE 1
1       FORMAT(/1X,'    OVL1 calls OVL3')
        CALL OVL3
        TYPE 2
2       FORMAT(/1X,'    Return to OVL1')
        RETURN
        END

@type ovl2.for
        SUBROUTINE OVL2A
        TYPE 1
1       FORMAT(/1X,'    'OVL2A calls OVL5')
        CALL OVL5
        TYPE 2
2       FORMAT(/1X,'    Return to OVL2A')
        TYPE 3
3       FORMAT(1X,'     OVL2A calls OVL6')
        CALL OVL6
        TYPE 2
        RETURN
        END
        SUBROUTINE OVL2B
        TYPE 1
1       FORMAT(/1X,'             OVL2B doesn't call anything')
        RETURN
        END

@type ovl3.for
        SUBROUTINE OVL3
        TYPE 1
1       FORMAT(/1X,'             OVL3 doesn't call anything')
        RETURN
        END

@type ovl4.for
        SUBROUTINE OVL4
        TYPE 1
1       FORMAT(/1X,'    OVL4 calls OVL1')
        CALL OVL1
        TYPE 2
2       FORMAT(/1X,'    Return to OVL4')
        RETURN
        END

@type ovl5.for
        SUBROUTINE OVL5
        TYPE 1
1       FORMAT(/1X,'             OVL5 doesn't call anything')
        RETURN
        END

@type ovl6.for
        SUBROUTINE OVL6
        TYPE 1
1       FORMAT(/1X,'    OVL6 calls OVL2B')
        CALL OVL2B
        TYPE 2
2       FORMAT(/1X,'    Return to OVL6')
        RETURN
        END
```

```
@COMPILE OVL0,OVL1,OVL2,OVL3,OVL4,OVL5,OVL6<RET>
FORTRAN: OVL0
OVL0
FORTRAN: OVL1
OVL1
FORTRAN: OVL2
OVL2A
OVL2B
FORTRAN: OVL3
OVL3
FORTRAN: OVL4
OVL4
FORTRAN: OVL5
OVL5
FORTRAN: OVL6
OVL6


@LINK
*TEST/LOG/LOGLEVEL:5
*/ERRORLEVEL:5/NOINITIAL
*TEST/OVERLAY
*TEST/MAP
*LPT:TEST/PLOT
*OVL0,OVL1/LINK:TEST
[LNKLMN Loading module MAIN. from file DSK:OVL0.REL]
[LNKLMN Loading module OVL1 from file DSK:OVL1.REL]
[LNKLMN Loading module OVRLAY from file SYS:OVRLAY.REL]
[LNKLMN Loading module JOBDAT from file SYS:JOBDAT.REL]
[LNKLMN Loading module FORINI from file SYS:FORLIB.REL]
[LNKLMN Loading module FORPSE from file SYS:FORLIB.REL]
[LNKELN End of link number 0 name TEST]
*/NODE:TEST OVL2/LINK:LEFT
[LNKLMN Loading module OVL2A from file DSK:OVL2.REL]
[LNKLMN Loading module OVL2B from file DSK:OVL2.REL]
[LNKELN End of link number 1 name LEFT]
*/NODE:LEFT OVL5/LINK:LEFT1
[LNKLMN Loading module OVL5 from file DSK:OVL5.REL]
[LNKELN End of link number 2 name LEFT1]
*/NODE:LEFT OVL6/LINK:LEFT2
[LNKLMN Loading module OVL6 from file DSK:OVL6.REL]
[LNKELN End of link number 3 name LEFT2]
*/NODE:TEST OVL3,OVL4/LINK:RIGHT
[LNKLMN Loading module OVL3 from file DSK:OVL3.REL]
[LNKLMN Loading module OVL4 from file DSK:OVL4.REL]
[LNKELN End of link number 4 name RIGHT]
*TEST/SAVE
*/EXECUTE/GO
[LNKXCT OVL0 execution]


Execution begins in main program OVL0
OVL0 calls OVL2A

         OVL2A calls OVL5

                 OVL5 doesn't call anything

         Return to OVL2A
         OVL2A calls OVL6

         OVL6 calls OVL2B

                         OVL2B doesn't call anything

         Return to OVL6

         Return to OVL2A

Return to OVL0
OVL0 calls OVL4

         OVL4 calls OVL1

                 OVL1 calls OVL3

                         OVL3 doesn't call anything

                 Return to OVL1

         Return to OVL4

Return to OVL0

Execution ends in main program OVL0

STOP

END OF EXECUTION
CPU TIME: 0.4    ELAPSED TIME: 4.1
EXIT
```

```
@type test.log
8:44:58   6   1   LMN   Loading module MAIN. from file DSK:OVL0.REL
8:44:59   6   1   LMN   Loading module OVL1 from file DSK:OVL1.REL
8:45:00   6   1   LMN   Loading module OVRLAY from file SYS:OVRLAY.REL
8:45:01   6   1   LMN   Loading module JOBDAT from file SYS:JOBDAT.REL
8:45:02   6   1   LMN   Loading module FORINI from file SYS:FORLIB.REL
8:45:02   6   1   LMN   Loading module FORPSE from file SYS:FORLIB.REL
8:45:13   7   1   ELN   End of link number 0 name TEST

8:45:13   6   1   LMN   Loading module OVL2A from file DSK:OVL2.REL
8:45:14   6   1   LMN   Loading module OVL2B from file DSK:OVL2.REL
8:45:15   7   1   ELN   End of link number 1 name LEFT

8:45:16   6   1   LMN   Loading module OVL5 from file DSK:OVL5.REL
8:45:16   7   1   ELN   End of link number 2 name LEFT1

8:45:17   6   1   LMN   Loading module OVL6 from file DSK:OVL6.REL
8:45:20   7   1   ELN   End of link number 3 name LEFT2

8:45:23   6   1   LMN   Loading module OVL3 from file DSK:OVL3.REL
8:45:24   6   1   LMN   Loading module OVL4 from file DSK:OVL4.REL
8:45:24   7   1   ELN   End of link number 4 name RIGHT


@type test.map
LINK symbol map of      TEST /KI/KL/KS                      page 1
Produced by LINK version 5(1442) on 29-Oct-81 at 8:45:35

Overlay no.             0        name          TEST
Low segment starts at       0 ends at          5145 length          5146 =    6P
Control Block address is    5104, length      32 (octal), 26. (decimal)
410 words free in Low segment
138 Global symbols loaded, therefore min. hash size is 154
Start address is 202, located in program MAIN.

                                                      *************


MAIN.    from DSK:OVL0.REL       created by FORTRAN /KI/KL/KS on 19-Feb-82 at 41:27:28
Low segment starts at 127552 ends at        443377 length       313626 (octal),104342. (decimal)

MAIN.             202    Entry    Relocatable

                                                      *************


OVL1     from DSK:OVL1.REL       created by FORTRAN /KI/KL/KS on 19-Feb-82 at 41:27:31
Low segment starts at 127552 ends at        443402 length       313631 (octal),104345. (decimal)

OVL1             307   Entry    Relocatable                 FOROTI          400010  Global  Absolute

                                                      *************


OVRLAY   from SYS:OVRLAY.REL     created by MACRO on 18-Feb-82 at 61:39:23
Low segment starts at 127551 ends at        661412 length        531642 (octal),177058. (decimal)

BOUTI    104000000051  Global  Absolute                CLOSFI   104000000022  Global  Absolute
ERJMP    320700000000  Global  Absolute                ERSTRI   104000000011  Global  Absolute
IOVRLA   401000050     Global  Absolute    Suppressed  .FHJOB   777773        Global  Absolute    Suppressed
.FHSLF   400000        Global  Absolute    Suppressed  .JSAOF   1             Global  Absolute    Suppressed
.NULIO   377777        Global  Absolute    Suppressed  .OVRLA   3503          Entry   Relocatable
.OVRLO   3510          Global  Relocatable             .OVRLU   2274          Entry   Relocatable
GCVECI   104000000300  Global  Absolute                GETOV.   1751          Entry   Relocatable
GJIOLD   100000000000  Global  Absolute    Suppressed  GTJFNI   104000000020  Global  Absolute
HALTFI   104000000170  Global  Absolute                INIOV.   1740          Entry   Relocatable
JFNSI    104000000030  Global  Absolute                JSIDIR   70000000000   Global  Absolute    Suppressed
JSINAM   7000000000    Global  Absolute    Suppressed  JSIPAF   1             Global  Absolute    Suppressed
LOGOV.   2537          Entry   Relocatable             OFIBSZ   770000000000  Global  Absolute    Suppressed
OFIRD    200000        Global  Absolute    Suppressed  OFIWR    100000        Global  Absolute    Suppressed
OPENFI   104000000021  Global  Absolute                PAIEX    20000000000   Global  Absolute    Suppressed
PAIPRV   200000000     Global  Absolute    Suppressed  PBOUTI   104000000074  Global  Absolute
PSOUTI   104000000076  Global  Absolute                REMOV.   1772          Entry   Relocatable
RMAPI    104000000061  Global  Absolute                RPACSI   104000000057  Global  Absolute
RUNOV.   2010          Entry   Relocatable             RUNTMI   104000000015  Global  Absolute
SFPTRI   104000000027  Global  Absolute                SINI     104000000052  Global  Absolute
SOUTI    104000000053  Global  Absolute                TIME     104000000014  Global  Absolute
.OVRWA   3507          Global  Relocatable

                                                      *************
```

OVERLAYS

```
JOBDAT   from SYS:JOBDAT.REL     created by MACRO on 18-Feb-82 at 02:38:14
Low segment starts at   127551 ends at        670325 length        540555 (octal),180589. (decimal)

%JOBDT  43100000447    Global  Absolute      Suppressed  .JB41   41            Entry   Absolute
.JBAPR  125            Entry   Absolute                  .JBBLT  45            Entry   Absolute
.JBCHN  131            Entry   Absolute                  .JBCNI  26            Entry   Absolute
.JBCOR  133            Entry   Absolute                  .JBCST  36            Entry   Absolute
                       LINK symbol map of    TEST /KI/KL/KS                    page 2
JOBDAT
.JBDA   140            Entry   Absolute                  .JBDDT  74            Entry   Absolute
.JBEDV  112            Entry   Absolute                  .JBERR  42            Entry   Absolute
.JBFF   121            Entry   Absolute                  .JBH41  1             Global  Absolute    Suppressed
.JBHCR                 Global  Absolute      Suppressed  .JBHDA  10            Entry   Absolute    Suppressed
.JBHGA  7              Global  Absolute      Suppressed  .JBHGH  400000        Entry   Absolute    Suppressed
.JBHNM  5              Entry   Absolute      Suppressed  .JBHRL  115           Entry   Absolute
.JBHRN  3              Global  Absolute      Suppressed  .JBHSA  0             Global  Absolute    Suppressed
.JBHSM  6              Entry   Absolute      Suppressed  .JBHVR  4             Entry   Absolute    Suppressed
.JBINT  134            Entry   Absolute                  .JBOPC  130           Entry   Absolute
.JBOPS  135            Entry   Absolute                  .JBOVL  131           Entry   Absolute
.JBPFH  123            Entry   Absolute                  .JBPFI  74            Entry   Absolute
.JBREL  44             Entry   Absolute                  .JBREN  124           Entry   Absolute
.JBSA   120            Entry   Absolute                  .JBSYM  116           Entry   Absolute
.JBTPC  127            Entry   Absolute                  .JBUSY  117           Entry   Absolute
.JBUUO  40             Entry   Absolute                  .JBVER  137           Entry   Absolute

                                           **************


FORINI   from SYS:FORLIB.REL     created by MACRO on 18-Feb-82 at 01:51:51
Low segment starts at   127551 ends at        662766 length        533216 (octal),177806. (decimal)

ALCHN.  4377           Entry   Relocatable               ALCOR.  4375          Entry   Relocatable
CLOSE.  4356           Entry   Relocatable               DBMS.   4403          Entry   Relocatable
DEC.    4365           Entry   Relocatable               DECHN.  4400          Entry   Relocatable
DECOR.  4376           Entry   Relocatable               ENC.    4364          Entry   Relocatable
ESOUT%  104000000313   Global  Absolute                  EXIT.   4374          Entry   Relocatable
FIN.    4371           Entry   Relocatable               FIND.   4373          Entry   Relocatable
FORER.  4354           Entry   Relocatable               FOROP.  4405          Entry   Relocatable
FUNCT.  4402           Entry   Relocatable               GET%    104000000200  Global  Absolute
GEVEC%  104000000205   Global  Absolute                  GJ%PHY  10000000      Global  Absolute    Suppressed
GJ%SHT  1000000        Global  Absolute      Suppressed  GTINDV  40000         Global  Absolute    Suppressed
IN.     4360           Entry   Relocatable               INIT.   4353          Entry   Relocatable
INQ.    4404           Entry   Relocatable               IOLST.  4370          Entry   Relocatable
MTOP.   4372           Entry   Relocatable               NLI.    4366          Entry   Relocatable
NLO.    4367           Entry   Relocatable               OPEN.   4355          Entry   Relocatable
OUT.    4361           Entry   Relocatable               PM%CNT  400000000000  Global  Absolute    Suppressed
PM%RD   100000000000   Global  Absolute      Suppressed  PM%RWX  160000000000  Global  Absolute    Suppressed
PM%WR   40000000000    Global  Absolute      Suppressed  PMAP%   104000000056  Global  Absolute
PORTAL  254040000000   Global  Absolute                  RELEA.  4357          Entry   Relocatable
RESET$  4307           Entry   Relocatable               RESET.  4206          Entry   Relocatable
RF%LNG  400000000000   Global  Absolute      Suppressed  RFSTS%  104000000156  Global  Absolute
RTB.    4362           Entry   Relocatable               SEVEC%  104000000204  Global  Absolute
SMAP%   104000000767   Global  Absolute                  TRACE.  4401          Entry   Relocatable
WTB.    4363           Entry   Relocatable               XJRSTF  254240000000  Global  Absolute
.RFSFL  4              Global  Absolute      Suppressed

                                           **************


FORPSE   from SYS:FORLIB.REL     created by MACRO on 18-Feb-82 at 01:51:51
Low segment starts at   127551 ends at        662766 length        533216 (octal),177806. (decimal)

PAUS.   4601           Entry   Relocatable               RFMOD%  104000000107  Global  Absolute
SFMOD%  104000000110   Global  Absolute                  STOP.   4604          Entry   Relocatable
TTIOSP  400000000000   Global  Absolute      Suppressed  .PRIOU  101           Global  Absolute    Suppressed

                                           **************


                       LINK symbol map of    TEST /KI/KL/KS                    page 3

          Index to LINK symbol map of        TEST /KI/KL/KS                    page 4

          Name           Page   Name         Page       Name     Page          Name     Page

          FORINI          2     JOBDAT        1         OVL1      1            OVRLAY    1
          FORPSE          2     MAIN.         1

                       LINK symbol map of    TEST /KI/KL/KS        #1          page 5

          Overlay no.    1       name        LEFT
          Low   segment starts at  11140 ends at         11325 length          166 = 1P
          Control Block address is 11264, length     30 (octal), 24. (decimal)
          Path is        0
          298 words free in Low segment
          6 Global symbols loaded, therefore min. hash size is 7

                                           **************


OVL2A    from DSK:OVL2.REL        created by FORTRAN /KI/KL/KS on 18-Feb-82 at 11:27:33
Low   segment starts at 127552 ends at       443404 length        313633 (octal),104347. (decimal)

          OVL2A                   11171        Entry        Relocatable

                                           **************
```



```
OVL2B    from DSK:OVL2.REL       created by FORTRAN /KI/KL/KS on 19-Feb-82 at 41:27:33
Low   segment starts at 127552 ends at        443404 length        313633 (octal),104347. (decimal)

         OVL2B                    11252         Entry        Relocatable

                                                    **************


                    LINK symbol map of     TEST /KI/KL/KS      #2                page 6

         Overlay no.    2         name          LEFT1
         Low   segment starts at  11326 ends at              11403 length               56 +   1P
         Control Block address is  11354, length      20 (octal), 16. (decimal)
         Path is          0, 1
         252 words free in Low segment
         3 Global symbols loaded, therefore min. hash size is 4

                                                    **************


OVL5     from DSK:OVL5.REL       created by FORTRAN /KI/KL/KS on 19-Feb-82 at 41:27:39
Low   segment starts at 127552 ends at        443412 length        313641 (octal),104353. (decimal)

         OVL5                     11342         Entry        Relocatable

                                                    **************


                    LINK symbol map of     TEST /KI/KL/KS      #3                page 7

         Overlay no.    3         name          LEFT2
         Low   segment starts at  11326 ends at              11420 length               73 +    1P
         Control Block address is  11371, length      20 (octal), 16. (decimal)
         Path is          0, 1
         239 words free in Low segment
         4 Global symbols loaded, therefore min. hash size is 5

                                                    **************


OVL6     from DSK:OVL6.REL       created by FORTRAN /KI/KL/KS on 19-Feb-82 at 41:27:41
Low   segment starts at 127552 ends at        443414 length        313643 (octal),104355. (decimal)

         OVL6                     11344         Entry                                   Relocatable

                                                    **************


                    LINK symbol map of     TEST /KI/KL/KS      #4                page 8

         Overlay no.    4         name          RIGHT
         Low segment starts at  11146 ends at              11270 length             123 +    1P
         Control Block address is  11237, length      22 (octal), 18. (decimal)
         Path is          0
         327 words free in Low segment
         5 Global symbols loaded, therefore min. hash size is 6

                                                    **************


OVL3     from DSK:OVL3.REL       created by FORTRAN /KI/KL/KS on 19-Feb-82 at 41:27:35
Low   segment starts at 127552 ends at        443406 length        313635 (octal),104349. (decimal)

         OVL3                     11162         Entry        Relocatable

                                                    **************


OVL4     from DSK:OVL4.REL       created by FORTRAN /KI/KL/KS on 19-Feb-82 at 41:27:37
Low   segment starts at 127552 ends at        443410 length        313637 (octal),104351. (decimal)

OVL4                    11212    Entry          Relocatable

                                                    **************


         Index to overlay numbers of TEST /KI/KL/KS                page 9

         Overlay        Page     Overlay        Page          Overlay Page          Overlay Page

         #0              4       #2              6            #3       7            #4  8
         #1              5

         Index to overlay names of TEST

         Name           Page     Name           Page          Name    Page          Name      Page

         LEFT            5       LEFT2           7            RIGHT    8            TEST       4
         LEFT1           6

         [End of LINK map of      TEST]
```

The listing file TEST.OVL will look similar to the following:

```
                            *************
                            *           *
                            * 0         *
                            *           *
                            *           *
                            * TEST      *
                            *************
                                   =
                                   =
                     ===============================
                     =                             =
                     =                             =
                     =                             =
                     =                             =
             *************                 *************
             *           *                 *           *
             *           *                 *           *
             * 1         *                 * 4         *
             *           *                 *           *
             * LEFT      *                 * RIGHT     *
             *************                 *************
                     =
                     =
       ===============================
       =                             =
       =                             =
       =                             =
       =                             =
*************                 *************
*           *                 *           *
* 2         *                 * 3         *
*           *                 *           *
*           *                 *           *
* LEFT1     *                 * LEFT2     *
*************                 *************
```

The process also produced an executable file TEST.EXE, which can be run using the RUN system command. However, to run the program, the file TEST.OVL must be present, because it provides the code for the links.

5.2 WRITABLE OVERLAYS

Ordinarily each overlay link built by LINK is copied by the overlay handler from the OVL file to the address space at runtime. The contents of any locations that have been modified will be lost each time the overlay link is copied from the OVL file. This can be prevented by the use of writable overlays.

If a link is specified as writable, the overlay handler copies that link to a temporary file on disk before overwriting it. Later, when the copied link is needed, the overlay handler retrieves the link from the temporary file rather than the OVL file. In this way, any modified values are preserved. Because writable overlays involve more file I/O, they are slower than the default (nonwritable) overlays and should only be used when the program structure and storage requirements demand dynamic storage in overlay links.

5.2.1 Writable Overlay Syntax

To build a writable overlay, specify the keyword WRITABLE with the /OVERLAY switch in the LINK command line:

```
filespec/OVERLAY:WRITABLE
```

5.2.2 Writable Overlay Error Messages

The overlay handler must write and update a temporary file. In addition to the error messages associated with all overlays, there are two additional error messages for writable overlays:

```
? OVLCWF Cannot write file [filename]:  [reason]

? OVLCUF Cannot update file [filename]:  [reason]
```

If either of these messages appears, you should check for disk quota violations or other conditions that could prevent the overlay handler from writing a temporary file.

5.3 RELOCATABLE OVERLAYS

LINK ordinarily allocates 2000 extra words at the end of the root link and no extra space at the end of each subsequent link. This is adequate for programs with static storage requirements. If a link requires extra storage at run-time, you can use the /SPACE switch to make the necessary allowances for the program's requirements. The /SPACE switch allows you to specify the number of words to be allocated after the current link is loaded.

However, there are programs whose dynamic run-time storage requirements are unpredictable. For example, a program's run-time storage requirements may vary according to the program's input. For this class of programs, relocatable overlays can be useful.

For relocatable overlays LINK places extra relocation information in the OVL file, permitting overlay links to be relocated at runtime. The overlay handler, using the FUNCT. subroutine, can determine where the link will fit in the address space and resolve relocatable addresses within the link. This extra processing causes relocatable overlays to run slower than nonrelocatable overlays. Relocatable overlays should only be used when you cannot determine the dynamic storage requirements of a program.

5.3.1 Relocatable Overlay Syntax

To build a relocatable overlay, specify the RELOCATABLE keyword to the /OVERLAY switch in the LINK command line:

```
     filespec/OVERLAY:RELOCATABLE
```

5.3.2 Relocatable Overlay Messages

If /OVERLAY:(LOGFILE,RELOCATABLE) is specified during the loading of a program, informational messages of the following form are sent to the user's terminal:

```
     %OVLRLL Relocating link [linkname] at [address]
```

5.4 RESTRICTIONS ON OVERLAYS

The following restrictions apply to all overlayed programs:

- Overlayed programs cannot be run execute-only.
- PSECTed programs cannot be overlayed.
- Overlayed programs with large buffer requirements must use the /SPACE switch. If an %OVLMAN (Memory not available) error is encountered, the program should be reloaded using the /SPACE switch with each link.
- If the program uses more than 256 links, use the /MAXNODE switch to specify the number of links necessary for the program. LINK will allocate extra space in the the OVL file for tables that require it, based on the number of links you specify.

5.4.1 Restrictions on Absolute Overlays

The following restrictions apply to absolute overlaid programs:

1. Any intermediate results stored in non-root links are lost as soon as the links are overlaid. Do not expect to retain a value stored in a non-root link unless /OVERLAY:WRITABLE has been specified.

2. Certain forms of global, inter-overlay references are not recommended, because you cannot be sure that the necessary modules will be in memory at the right time. Some of these references are:

   - Additive fixups, in the form FOO##+BAR where FOO is in another overlay.

   - Left-hand fixups, in the form XWD FOO##,BAR, where FOO is in another overlay.

   - Fullword fixups, in the form EXP FOO##, where FOO is in another overlay.

   - Similarly, MOVEI 1,FOO##, where FOO is in a different overlay, should not be used, because the necessary module may not be in memory.

   In fact, the only predictable inter-overlay global reference is one that brings the necessary module into memory, such as PUSHJ P,FOO##.

5.4.2 Restrictions on Relocatable Overlays

The following restriction applies to relocatable overlays:

- Complex expressions involving relocatable symbols are not relocated properly in a relocatable overlay. No standard DEC compiler produces such expressions. MACRO programmers should avoid using them in subroutines that are to be loaded as part of an overlayed program. Any expression that causes MACRO to generate a Polish fixup block will not be properly relocated at run-time. The following is an example of such a complex expression:

  ```
  MOVEI 1,A## + B## + C##
  ```

5.4.3 Restrictions on FORTRAN Overlays

The following restriction applies to FORTRAN programs that are written with associate variables and using the overlay facility.

- If the associate variable is declared in a subroutine, that subroutine must be loaded in the root link of the overlay structure. Accessing a file opened with an associate variable changes the value of the specified variable. If this variable is in a nonresident overlay link when the access is made, program execution will produce unpredictable results. Moreover, the value of the variable will be reset to zero each time its overlay link is removed from memory. Only variables declared in routines that are loaded into the root link will always be resident. However, variables declared in COMMON and in the main program will always be resident, and may be safely used as associate variables.

- If you place COMMON in a writable overlay, be sure that all references to the variables in that COMMON are in the same overlay.

- A FORTRAN ASSIGN statement may be used in a relocatable overlay. If the ASSIGN is made in a subroutine, the value of the assigned variable may be preserved from one call of that subroutine to the next. However, the overlay containing that subroutine could then be replaced in memory by a different overlay. If the overlay containing the subroutine is relocated differently when brought back into memory, any subsequent GOTO may fail.

5.5 SIZE OF OVERLAY PROGRAMS

Although most programs have a consistent size, the size of an overlay program depends on which overlays are in memory. This can be ascertained by using the /COUNTER switch when linking the program. To do this, place /COUNTER after the /LINK switch for the overlay you want to know the size of, but before the next /NODE switch. This will give you the size of the program when the overlay is actually loaded into memory. The display will include all routines loaded from the runtime libraries. This allows you to determine which overlay is the largest, and whether the program can be loaded without restructuring.

5.6 DEBUGGING OVERLAYED PROGRAMS

COBDDT and ALGDDT can be used to debug overlay programs, but FORDDT cannot. To use DDT with an overlayed program, the program should be loaded using /SYMSEG:LOW, with local symbols for the desired modules.

To set breakpoints in an overlay, put a subroutine in the root node, and call the subroutine from the overlay. Such a subroutine need consist only of a SUBROUTINE statement, a RETURN, and an END. The breakpoint can be set at this subroutine before the program starts running.

When a FORTRAN program starts running, it calls RESET. in FOROTS, which removes the symbol table. The symbol table will return after the first overlay is called. If you need the symbols for debugging the root link, insert a CALL INIOVL at the beginning of the main program (refer to Section 5.7.1 for more information). This call will reinstall the symbol table. LINK builds a separate symbol table for each overlay, so that all the symbols known to DDT are for modules that are currently in memory. Note that it is not possible to single-step through RESET. ($X and $$X will not work). Set a breakpoint after RESET. if you are debugging a root link, and use $G.

5.7 THE OVERLAY HANDLER

LINK's overlay handler is the program that supervises execution of overlay structures defined by LINK switches.

The overlay handler is in the file SYS:OVRLAY.REL. Some installations will install LINK Version 5 without the overlay handler that was shipped with it. To find the version of the overlay handler, type the following:

```
@LINK RET
*SYS:OVRLAY RET
*/VALUE:%OVRLA RET
[LNKVAL Symbol %OVRLA 402000056 defined]
*Z
```

The left halfword of $OVRLA contains the version number of the overlay handler, and should be 402, corresponding to Version 5 of LINK. The right halfword is the edit number, and should be 000056 if field image, or greater if edits have been installed.

When you load an overlay structure, the overlay handler is loaded into the root link of the structure. From there it can supervise overlaying operations, because the root link is always in your virtual address space during execution. During execution, when a link not in memory is called, the overlay handler brings in the link, possibly overlaying one or more links already in memory. The overlay handler consists of self-modifying code and data, and two 128-word buffers. One of these buffers, IDXBFR, contains a 128-word section of the link number index table. This allows 256 links to be directly referenced at any one time. The second buffer, INBFR, contains the preambles and relocation tables, if required, of the individual links.

There are two ways of overlaying links during execution:

1. A call to a link not in memory implicitly calls the overlay handler to overlay one or more links with the required links. This action of the overlay handler is transparent to the user.

2. An explicit call to one of several entry points in the overlay handler can cause one or more links to be overlaid. These entry points and calls to them are discussed in the sections below.

5.7.1 Calls to the Overlay Handler

Overlays can be used transparently, or they can be explicitly called from the program. Such calls are made to one of the entry points in the overlay handler.

The overlay handler has five entry points that are available for calls from user programs. To call the overlay handler from a MACRO program, you must use the standard calling sequence, which is:

```
     MOVEI     16,arglst
     PUSHJ     17,entry-name
```

Where arglst is the address of the first argument in the argument list, and entry-name is the entry-point name.

The argument list must be of the form:

```
               -n,,0           ;n is number of arguments
     arglst:   Z code,addr1    ;For first argument
                          .
                          .
                          .
               Z code,addrn    ;For nth argument
```

Where addr... is the address of the argument.

The legal values of "code" are 2 (for a link number), 17 (for an ASCIZ string), and 15 (for a character string descriptor).

For each word of the argument list, the code indicates the type of argument. The code occupies the AC field, bits 9 through 12. The address gives the location of the argument; it can be indirect and indexed.

To call the overlay handler from a FORTRAN program, the call must be of the form:

```
CALL subroutine (arglst)
```

Where subroutine is the name of the desired subroutine, and arglst is a list of arguments separated by commas.

5.7.2 Overlay Handler Subroutines

Each of the seven callable subroutines in the overlay handler has an entry name symbol for use with MACRO, and a subroutine name for use with FORTRAN, as follows:

MACRO Entry Name Symbol	FORTRAN Subroutine	Subroutine Function
CLROV.	CLROVL	Specifies a non-writable overlay.
GETOV.	GETOVL	Brings specified links into memory.
INIOV.	INIOVL	Specifies the file from which the overlay program will be read, if the load time specification is to be overridden.
LOGOV.	LOGOVL	Specifies or closes the file in which runtime messages from the overlay handler will be written.
REMOV.	REMOVL	Removes specified links from memory.
RUNOV.	RUNOVL	Moves into memory a specified link and begins execution at its start address.
SAVOV.	SAVOVL	Specifies a writable overlay.

Declaring a Non-Writable Link (CLROV.)

You can declare an overlay link to be non-writable, using the CLROV. entry point. This does not immediately affect the program, but waits until the link is about to be overlaid or read in. If the link is already non-writable, this entry point has no effect.

Example

```
                MOVEI          16,arglst
                PUSHJ          17,CLROV.

                -n,,0                               ;n is number of arguments
        arglst: Z 17,addrl                          ;for first ASCIZ linkname
                     .
                     .
                     .
                Z 17,addrn                          ;for nth ASCIZ linkname

                                 OR

                -n,,0                               ;n is number of arguments
        arglst: Z 2,addrl                           ;for first link number
                     .
                     .
                     .
                Z 2,addrn                           ;for nth link number
```

Where addr... is the address of the argument.

Getting a Specific Path (GETOV.)

The subroutine to bring a specific path into core can be used to make sure that a particular path is used when otherwise the overlay handler might have a choice of paths. It is illegal to specify a path that overlays the calling link.

To call the subroutine from a FORTRAN program, use:

```
CALL GETOVL (linkname,...,linkname)
```

where each linkname is the ASCIZ name of a link in the desired path.

To call the subroutine from a MACRO program, use the standard FORTRAN calling sequence:

```
MOVEI     16,arglst
PUSHJ     17,GETOV.
```

The argument list has one word for each link required to be in the path.

Example

```
                -n,,0                               ;n is number of arguments
        arglst: Z 17,addrl
                     .
                     .
                     .
                Z 17,addrn

                                 OR

                -n,,0                               ;n is number of arguments
        arglst: Z 2,addrl
                     .
                     .
                     .
                Z 2,addrn
```

Where addr... is the address of the argument.

Initializing an Overlay (INIOV.)

The overlay initializing subroutine specifies a file from which the overlay program will be read. This subroutine is used to override the file specified at load time. The file specified to INIOV. can have any valid specification, but it must be in the correct format for an overlay (OVL) file.

To call the subroutine from a FORTRAN program, use:

```
     CALL INIOVL ('filespec')
```

where `filespec' is a literal constant that can give a device, a filename, a file type, and a project-programmer number (PPN).

To call the subroutine from a MACRO program, use the standard FORTRAN calling sequence:

```
     MOVEI     16,arglst
     PUSHJ     17,INIOV.
```

The argument list is of the form:

```
             -1,,0
     arglst: Z 17,address of ASCIZ filespec
```

where filespec is an ASCIZ string (ASCII ending with nulls) that can give a device, a filename, a file type, and a PPN

NOTE

> If you call INIOV. with no arguments, it initiates the overlay handler and reads in the symbols for the root link, using the overlay file specified at load time. This can be useful for debugging the root link before any successor links have been read in, because symbols are not normally available until the first link comes into memory.

Specifying an Overlay Log File (LOGOV.)

You can specify an output file for runtime messages from the overlay handler. These messages are listed in Section 5.5. The log file entry includes the elapsed run time since the first call to the overlay handler.

To call this subroutine from a FORTRAN program, use:

```
     CALL LOGOVL ('filespec')
```

where `filespec' is a literal constant that can give a device, a filename, a file type, and a PPN.

To close the file, use

```
     CALL LOGOVL (0)
```

To call the subroutine from a MACRO program, use the standard FORTRAN calling sequence:

```
MOVEI       16,arglst
PUSHJ       17,LOGOV.
```

The argument list is of the form:

```
          -1,,0
arglst: Z 17,address of ASCIZ filespec
```

Where filespec is an ASCIZ string that can give a device, a filename, a file type, and a PPN.

To close the log file, the argument list is:

```
          -1,,0
arglst: Z 17,address of word containing zero
```

Removing Specific Links from Memory (REMOV.)

The subroutine to remove specific links from memory, once they are no longer required, can be used to reduce core image size for faster execution. Specifying removal of the calling link causes an error.

To call the subroutine from a FORTRAN program, use:

```
CALL REMOVL (linkname,...,linkname)
```

Where each linkname is the ASCIZ name of a link to be removed from memory.

To call the subroutine from a MACRO program, use the standard FORTRAN calling sequence:

```
MOVEI       16,arglst
PUSHJ       17,REMOV.
```

The argument list has one word for each link to be removed.

Example

```
            -n,,0                           ;n is number of arguments
arglst:     Z 17,addr1
                 .
                 .
                 .
            Z 17,addrn

                        OR

            -n,,0                           ;n is number of arguments
arglst:     Z 2,add1
                 .
                 .
                 .
            Z 2,addrn
```

Where addr... is the address of the argument.

Running a Specific Link (RUNOV.)

The subroutine for running a specific link allows you to transfer program execution to the start address of a particular link. (An error occurs if the link has no start address.) If the link is not already in memory, it and its path are brought in.

You can use this subroutine to overlay the calling link, because the next instruction executed is the start address of the named link; therefore, there is no automatic return to the calling link.

> NOTE
>
> The FORTRAN compiler does not generate start addresses for subroutines. FORTRAN main programs cannot be loaded into non-root links. Therefore, to use RUNOVL to transfer control to a FORTRAN subroutine in a non-root link, you must use the /START switch at load time to define a start address for the link.

To call the subroutine RUNOVL from a FORTRAN program, use:

```
     CALL RUNOVL (linkname)
```

Where linkname is the ASCIZ name of the link to be run.

To call the subroutine from a MACRO program, use the standard FORTRAN calling sequence:

```
     MOVEI     16,arglst
     PUSHJ     17,RUNOV.
```

The argument list is of the form:

```
               -1,,0
     arglst: Z 17,address of ASCIZ linkname
```

OR

```
               -1,,0
     arglst: Z 2,address of link number
```

Declaring A Writable Link (SAVOV.)

You can dynamically declare an overlay link to be writable by calling SAVOV. This does not affect the current state of the code immediately, but waits until the link is about to be overlaid. If the link already writable, this symbol has no effect.

Example

```
MOVEI      16,arglst
PUSHJ      17,SAVOV.

           -n,,0                         ;n is number of arguments

arglst:    Z 17,addr1                    ;for first ASCIZ linkname
                .
                .
                .
           Z 17,addrn                    ;for nth ASCIZ linkname
                             OR

           -n,,0                         ;n is number of arguments
arglist:   Z 2,addr1                     ;for first link number
                .
                .
                .
           Z 2, addrn                    ;for nth link number
Where addr... is the address of the argument.
```

If called with no arguments, SAVOV. only initializes the temporary file.

5.7.3 Overlay Handler Messages

This section lists all of the overlay handler's messages. (The messages from LINK, which have the LNK prefix, are given in Appendix B.)

For each overlay handler message, the last three letters of the six-letter code, the severity, and the text of the message are given in boldface. Then, in lightface type, comes an explanation of the message.

When a message is issued, the three letters are suffixed to the letters OVL, forming a 6-letter code of the form OVLxxx. The explanation of the message will be printed only if you use the /OVERLAY:LOG switch.

The severity of a message determines whether the job will be terminated when the message is issued. Level 31 messages terminate program execution. Level 8 messages are warnings: they do not terminate execution, but the error may affect the execution of the program. Level 1 messages are informational and are printed on the terminal only if you specified /OVERLAY:LOGFILE.

Code	Sev	Message and Explanation
ARC	31	**Attempt to remove caller from link [name or number]** The named link attempted to remove the link that called it. This error occurs when the call to the REMOV. subroutine requests removal of the calling link.
ARL	8	**Ambiguous request in link number [number] for [symbol], using link number [number]** More than one successor link satisfies a call from a predecessor link, and none of these successors is in memory. Since all their paths are of equal length, the overlay handler has selected an arbitrary link.
CDL	31	**Cannot delete link [name or number], FUNCT. return status [number]** This is an internal LINK error, and is not expected to occur. If it does, please notify your Software Specialist, or send a Software Performance Report (SPR) to DIGITAL. Return status is one of the following: 1 Core already deallocated 3 Illegal argument passed to FUNCT. module
CGM	31	**Cannot get memory from OTS, FUNCT. return status [octal]** The system does not have enough free memory to load the link. The status 3 (illegal argument) is returned from the object-time system.
CRF	31	**Cannot read file [file] [reason]** An error occurred when reading the overlay file. The file was closed after the last successful read operation.
CSM	31	**Cannot shrink memory, FUNCT. return status [octal]** A request to the object-time system to reduce memory, if possible, failed. This error is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
CUF	31	**Cannot update file [file] [reason]** An error occurred when updating the TMP file into which non-resident writable overlay links are written.
CWF	31	**Cannot write file [file] [reason]** An error occurred when creating the TMP file used to store non-resident writable overlay links.
DLN	1	**Deleting link [name or number] after [hh:mm:ss]** The named link has been removed from memory as a result of a call to the REMOV. subroutine. The time given is elapsed time since the first call to the overlay handler. This message is output only to the overlay log file, if any.

Code	Sev	Message and Explanation
IAT	31	**Illegal argument type on call to [subroutine]** A user call to the named overlay handler subroutine gave an illegal type of argument.
IEF	31	**Input error for file [file] [reason]** An error occurred while reading the OVL or TMP file.
ILN	31	**Illegal link number [number]** A user call to one of the overlay handler subroutines gave an illegal link number as an argument.
IMP	31	**Impossible error condition at PC=[address]** This is an internal error caused by monitor call error returns that should not occur. This message is issued instead of the HALT message. This error is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
IPE	31	**Input positioning error for file [file] [reason]** An error occurred while reading the OVL or TMP file.
IVN	8	**Inconsistent version numbers** The OVL and EXE files found were not created at the same time, and may not be compatible.
LNM	31	**Link number [decimal] not in memory** A call to the REMOV. subroutine has removed the named link from memory. It must be restored by a call to GETOV. or RUNOV.
MAN	31	**Memory not available for absolute [link], FUNCT. return status [octal]** There is not enough room for the overlay handler to load the specified link into the part of memory the link was built for. Two options are available: a) Use the /SPACE switch at load time to reserve more space for the link, or b) Build a relocatable overlay using the RELOCATABLE option to the /OVERLAY switch at load time.
MEF	31	**Memory expansion failed, FUNCT. return status [octal]** The overlay handler was unable to get free space from the memory manager. Restructure your overlay so that the minimum number of links are in memory at any time.
NMS	8	**Not enough memory to load symbols, FUNCT. return status [octal]** There was not enough free space available to load symbols into memory.

Code	Sev	Message and Explanation
NRS	31	No relocation table for symbols A relocation table was not included for the symbol table. It is possible that LINK failed to load the relocation table because there wasn't enough room in memory.
NSA	31	No start address for link [name or number] A user call to the RUNOV. subroutine requests execution to continue at the start address of the named link, but that link has no start address.
NSD	31	No such device for [file] An invalid device was specified.
OEF	31	Output error for file [file] [reason] An error occurred when writing the overlay file. The file was closed after the last successful write operation.
OPE	31	Output positioning error for file [file] [reason] An error occurred while writing the TMP file used to hold non-resident writable overlay links.
OPP	31	Overlay handler in private page The overlay handler has been loaded into a non-sharable page of the program. Your program should not write into the pages occupied by the overlay handler. Ask LINK for a map of your program if there is doubt. If the program is not writing into these pages, this error may reflect an internal LINK error. This error is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
RLL	1	Relocating link [name or number] at [address] The named relocatable link has been loaded at the given address. This message is output only to the overlay log file.
RLN	1	Reading in link [name or number] after [time] The named link has been loaded. The time given is elapsed time since the first call to the overlay handler. This message is output only to the overlay log file.
RMP	31	RMAP JSYS failed This is an internal error and is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
RPA	31	RPACS JSYS failed This is an internal error and is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.

Code	Sev	Message and Explanation
STS	8	**OTS reserved space too small** The object-time system does not have space for its minimum number of buffers. Reload, using the /SPACE switch for the root link with an argument greater than 2000 (octal).
ULN	31	**Unknown link name [name]** A call to one of the overlay handler subroutines gave an invalid link name as an argument. Correct the call.
USC	8	**Undefined subroutine [name] called from [address]** A required subroutine was not loaded. The instruction at the given program counter address calls for an undefined subroutine. Correct the call or load the required subroutine.
WLN	1	**Writing [link] after [time]** The overlay handler is writing out a writable overlay link.

5.7.4 The FUNCT. Subroutine

Each DIGITAL-supplied object-time system has a subroutine that the overlay handler uses for memory management, I/O, and message handling. This subroutine has a single entry point, FUNCT., and is called by the sequence:

```
MOVEI      16,arglst
PUSHJ      17,FUNCT.
```

The format of the argument list is:

```
        -<n+3>,,0
arglst: Z 2,address of integer function code
        Z 2,address for error code on return
        Z 2,address for status code on return
        Z code,address of first argument
           .
           .
           .
        Z code,address of nth argument
```

Where function code is one of the function codes described below; error code is a 3-letter ASCII mnemonic output by the object-time system (after ?, %, or [); and status (on return) contains one of the following values:

-1	Function not implemented
0	Successful return
n	Number of the error message

Most object-time systems allocate separate space for their own use and for the use of the overlay handler. This minimizes the possibility that the overlay handler will request space that the object-time system is already using.

The permitted function code arguments, their names, and their meanings are:

Code	Name	Function
0	ILL	Illegal function; returns -1 status.
1	GAD	Get a specific segment of memory.
2	COR	Get a given amount of memory from anywhere in the space allocated to the overlay handler.
3	RAD	Return a specific segment of memory.
4	GCH	Get an I/O channel.
5	RCH	Return an I/O channel.
6	GOT	Get memory from the space allocated to the object-time system.
7	ROT	Return memory to the object-time system.
10	RNT	Get the initial runtime, in milliseconds, from the object-time system.
11	IFS	Get the initial runtime file specification of the program being run.
12	CBC	Cut back core (if possible) to reduce job size.
13	F.RRS	Read retain status (DBMS)
14	F.WRS	Write retain status (DBMS)
15	F.GPG	Get pages
16	F.RPG	Return pages
17	F.GPSI	Get TOPS-20 PSI channel
20	F.RPSI	Return TOPS-20 PSI channel

All FUNCT. codes are reserved to DEC.

The following subsections describe each function of the FUNCT. subroutine (except the reserved functions).

ILL (0) Function

This function is illegal. The argument list is ignored, and the status returned is -1.

OVERLAYS

GAD (1) Function

The GAD function gets memory from a specific address in the space allocated to the overlay handler. The argument list points to:

arg 1 Address of requested memory
arg 2 Size of requested allocation (in words)

A call to GAD with arg 2 equal to -1 requests all available memory.

On return, the status is one of the following:

0 Successful allocation
1 Not enough memory available
2 Memory not available at specified address
3 Illegal arguments (address + size > 256K)

COR (2) Function

The COR function gets memory from any available space allocated to the overlay handler. The arguments are:

arg 1 Undefined (address of allocated memory on return)
arg 2 Size of requested allocation

On return, the status is:

0 Core allocated
1 Not enough memory available
3 Illegal argument (size > 256K)

RAD (3) Function

The RAD function returns the memory starting at the specified address to the overlay handler. The arguments are:

arg 1 Address of memory to be returned
arg 2 Size of memory to be returned (in words)

On return, the status is one of the following:

0 Successful return of memory
1 Memory cannot be returned
3 Illegal argument (address or size > 256K)

GCH (4) Function

Returns a status of 1. The channel is not available.

RCH (5) Function

Returns a status of 1. The channel is not available.

GCH (4) Function

The GCH function gets an input/output channel. The arguments are:

arg 1 Undefined (channel number allocated on return)
arg 2 Ignored

On return, the status is one of the following:

0 Successful channel allocation
1 No channels available

RCH (5) Function

The RCH function returns an input/output channel. Its arguments are:

arg 1 Number of channel to be returned
arg 2 Ignored

On return, the status is one of the following:

0 Channel released
1 Channel number invalid for user

GOT (6) Function

The GOT function gets memory from the space allocated to the object-time system. Its arguments are:

arg 1 Undefined (address of allocated memory on return)
arg 2 Size of memory requested

On return, the status is one of the following:

0 Successful allocation
1 Not enough memory available
3 Illegal argument (size > 256K)

ROT (7) Function

The ROT function returns memory to the object-time system. Its arguments are:

arg 1 Address of memory to be returned
arg 2 Size of memory to be returned (in words)

On return, the status is one of the following:

0 Successful return of memory
1 Memory cannot be returned
3 Illegal argument (address or size > 256K)

RNT (10) Function

The RNT function returns the initial runtime, in milliseconds, from the object-time system. (At the beginning of the program, the object-time system will have executed a RUNTIM UUO; the result is the time returned by RNT.) Its arguments are:

```
arg 1  Undefined (contains initial runtime on return)
arg 2  Ignored
```

On return, the runtime is in arg 1, and the status is 0. The status is 0.

IFS (11) Function

Always returns a value of -1. This function is not implemented.

CBC (12) Function

The CBC function cuts back memory if possible, which reduces the size of the job. It uses no arguments, and the returned status is 0.

RRS (13) Function (Reserved for DBMS)

Returns ARG1 = 0. On return, the status is always 0.

WRS (14) Function (Reserved for DBMS)

Returns ARG1 = 0. On return, the status is always 0.

GPG (15) Function

The GPG function is used to fetch a page. The arguments are:

```
arg2:  size to be allocated, in words
```

On return,

```
arg1 = address of allocated memory, on page boundary
```

and the status is one of the following:

```
0 if allocated OK
1 if not enough memory
3 if argument error
```

RPG (16) Function

The RPG function is used to return pages. The arguments are:

```
arg1:  address (a word)
arg2:  size (in words)
```

On return, the status is:

```
0 if deallocated OK
1 if wasn't allocated
3 if argument error
```

GPSI (17)

The GPSI function can be used to get a PSI channel for programs running in a TOPS-20 environment. This entry point provides only controlled access to the PSI tables. It will arrange that the tables exist and that SIR and EIR have been done but does not do AIC or any other JSYS necessary to set up the channel (ATI or MTOPR, for example).

The arguments are:

```
arg1: channel number,
        or -1 to allocate any user-assignable channel
arg2: level number
arg3: address of interrupt routine
```

On return, arg1 contains the channel number allocated (if -1 was originally specified). On return, the status is:

```
0 if OK
1 if channel was already assigned
2 if no free channels
3 if argument error
```

NOTE

This function is used by TOPS-20 programs. It is a reserved function in the TOPS-10 environment.

RPSI (20) Function

This entry point provides only controlled access to the PSI tables. It does not do DIC or any other JSYS necessary to release a channel. It just clears the level and interrupt address fields in CHNTAB.

This function accepts the following argument:
arg1: channel number

On return the status is one of the following:

```
0 if OK
1 if channel wasn't in use
3 if argument error
```

NOTE

This function is used by TOPS-20 programs. It is a reserved function in the TOPS-10 environment.

5.8 THE OVERLAY (OVL) FILE

This section contains diagrams of the contents of the overlay file output by LINK as a result of the /OVERLAY switch. The following diagram shows the overall scheme of the file:

Scheme of the Overlay (OVL) File

```
|========================================================|
|                                                        |
|                    Directory Block                     |
|                                                        |
|========================================================|
|                                                        |
|                   Link Number Table                    |
|                                                        |
|========================================================|
|                                                        |
|                    Link Name Table                     |
|                                                        |
|========================================================|
|                                                        |
|               Writable Link Flags Table                |
|                                                        |
|========================================================|
|                                                        |
|                         Link                           |
|                                                        |
|--------------------------------------------------------|
                            .
                            .
                            .
|--------------------------------------------------------|
|                                                        |
|                         Link                           |
|                                                        |
|========================================================|
```

5.8.1 The Directory Block

The following diagram shows the contents of the Directory Block:

Directory Block

```
         |=========================================================|
.DIHDR:  |       0 (Reserved)        | Length of Directory Block |
         |---------------------------------------------------------|
.DIRGN:  |                      0 (Reserved)                        |
         |---------------------------------------------------------|
.DIVER:  |       Version Number of Corresponding EXE file        |
         |---------------------------------------------------------|
.DILPT:  | -(Size of Link No. Table) |Link Number Table Block No.|
         |---------------------------------------------------------|
.DINPT:  |-(Size of Link Name Table) | Link Name Table Block No. |
         |---------------------------------------------------------|
.DIWPT:  |-(Size of Writable Flg Tbl)| Writable Flg Tbl Block No |
         |---------------------------------------------------------|
.DIFLG:  |                         Flags                            |
         |---------------------------------------------------------|
         |                      0 (Reserved)                        |
         |=========================================================|
```

In the fourth word above, the size of the Link Number Table (in words) is half the number of links (rounded upward); the Link Number Table Block No. is the number of the 128-word disk block containing the Link Number Table. (There are four disk blocks per disk page.)

In the fifth word above, the size of the Link Name Table (in words) is twice the number of links; the Link Name Table Block No. is the number of the 128-word disk block containing the Link Name Table.

The table defined by the .DIWPT word above consists of a string of two-bit bytes. The first bit, OW.WRT, indicates whether the corresponding overlay link is writable. This bit is set under the control of a REL block of type 1045 (writable links). The second bit, OW.PAG, indicates whether the corresponding overlay link is currently paged into the runtime overlay temporary file. This is strictly a run-time flag and should be zero in the overlay file. This flag is defined in the overlay file to allow the overlay handler to set up its flag table with a single read operation.

The .DIFLG word in the directory block contains a single bit flag (bit 0). If this bit is set the overlay file contains at least one writable overlay. This information is also contained in the Writable Link Table. However, by having the information available in the directory block the overlay handler can determine if any links are writable without scanning the Writable Link Table. All other bits in the .DIFLG word are reserved and must be zero.

NOTE

> If a user requests both writable and relocatable overlays, only halfwords known to be relocatable at load time will be correctly relocated when the link is refetched.

5.8.2 The Link Number Table

The following diagram shows the contents of the Link Number Table:

Link Number Table

```
|=========================================================|
|     Pointer to Link 0      |      Pointer to Link 1     |
|---------------------------------------------------------|
|     Pointer to Link 2      |      Pointer to Link 3     |
|---------------------------------------------------------|
                             .
                             .
                             .
|---------------------------------------------------------|
|    Pointer to Link n-1     |      Pointer to Link n     |
|=========================================================|
```

Each pointer is a disk block number. Any unused words in the last disk block of the Link Number Table are zeros.

5.8.3 The Link Name Table

The following diagram shows the contents of the Link Name Table:

Link Name Table

```
|=========================================================|
|                       Link Number                       |
|---------------------------------------------------------|
|                    SIXBIT Link Name                     |
|=========================================================|
                             .
                             .
                             .
|=========================================================|
|                       Link Number                       |
|---------------------------------------------------------|
|                    SIXBIT Link Name                     |
|=========================================================|
```

Any unused words in the last disk block of the Link Name Table are zeros.

5.8.4 The Overlay Link

The following diagram shows the overall scheme of each overlay link in the overlay file:

Scheme of an Overlay Link

```
|=======================================================|
|                                                       |
|                        Preamble                       |
|                                                       |
|=======================================================|
|                                                       |
|                      Code for Link                    |
|                                                       |
|=======================================================|
|                                                       |
|                  Link Control Section                 |
|                                                       |
|=======================================================|
|                                                       |
|                         EXTTAB                        |
|                                                       |
|=======================================================|
|                                                       |
|                         INTTAB                        |
|                                                       |
|=======================================================|
|                                                       |
|                    Relocation Table                   |
|                                                       |
|=======================================================|
|                                                       |
|                 Other Relocation Tables               |
|                                                       |
|=======================================================|
```

The Preamble

The following diagram shows the contents of the preamble for an overlay link:

Preamble

```
|========================================================|
|        0 (Reserved)         |    Length of Preamble    |
|--------------------------------------------------------|
|        0 (Reserved)         |       0 (Reserved)       |
|--------------------------------------------------------|
|        0 (Reserved)         |        Link Number       |
|--------------------------------------------------------|
|                    SIXBIT Link Name                    |
|--------------------------------------------------------|
|Pointer to List of Bound Links Starting with Root Link |
|--------------------------------------------------------|
| Pointer to List of Bound Links Ending with Root Link  |
|--------------------------------------------------------|
|                  Equivalence Pointer                   |
|--------------------------------------------------------|
|              Address of Control Section                |
|--------------------------------------------------------|
|                         Flags                          |
|--------------------------------------------------------|
|          Absolute Address at Which Link Loaded         |
|--------------------------------------------------------|
|          Length of Link (Code through INTTAB)          |
|--------------------------------------------------------|
|           Disk Block Number of Start of Link Code      |
|--------------------------------------------------------|
|                      0 (Reserved)                      |
|--------------------------------------------------------|
|            Disk Block Number of Relocation Table       |
|--------------------------------------------------------|
|         Disk Block Number of Other Relocation Tables   |
|--------------------------------------------------------|
|                      0 (Reserved)                      |
|--------------------------------------------------------|
|            Disk Block Number of Radix-50 Symbols       |
|--------------------------------------------------------|
|Block Number of Relocation Tables for Radix-50 Symbols |
|--------------------------------------------------------|
|         Next Free Memory Location for Next Link        |
|========================================================|
```

Code for the Link

The code for each link consists of a core image that was constructed from the REL files placed in the link. This core image contains the code and data for the link.

The Control Section

The following diagram shows the contents of the Control Section:

Control Section

```
|========================================================|
|       0 (Reserved)          |      Length of Header     |
|--------------------------------------------------------|
|       0 (Reserved)          |        0 (Reserved)       |
|--------------------------------------------------------|
|       0 (Reserved)          |         Link Number       |
|--------------------------------------------------------|
|                     SIXBIT Link Name                     |
|--------------------------------------------------------|
|  Ptr to Ancestor in Core    | Ptr to Successor in Core  |
|--------------------------------------------------------|
| -(Length of Symbol Table)   |  Address of Symbol Table  |
|--------------------------------------------------------|
|       0 (Reserved)          |  Start Address for Link   |
|--------------------------------------------------------|
|Memory Needed to Load Link   |   First Address in Link   |
|--------------------------------------------------------|
|    -(Length of EXTTAB)      |     Pointer to EXTTAB     |
|--------------------------------------------------------|
|    -(Length of INTTAB)      |     Pointer to INTTAB     |
|--------------------------------------------------------|
|                Address of Symbols on Disk                |
|--------------------------------------------------------|
|                    Relocation Address                    |
|--------------------------------------------------------|
|               Copy of Block Number for Code              |
|--------------------------------------------------------|
|-(Length of Radix-50 SymTab)|Blk No. of Radix-50 SymTab|
|========================================================|
```

The EXTTAB Table

The following diagram shows the contents of the EXTTAB table:

EXTTAB

```
|========================================================|
|                      JSP 1,.OVRLA                      |
|--------------------------------------------------------|
|           Flags            |Address of Callee's INTTAB |
|--------------------------------------------------------|
|    Callee's Link Number    |Ptr to Callee's Control Sec|
|--------------------------------------------------------|
|      Backward Pointer      |       Forward Pointer     |
|========================================================|
                             .
                             .
                             .
|========================================================|
|                      JSP 1,.OVRLA                      |
|--------------------------------------------------------|
|           Flags            |Address of Callee's INTTAB |
|--------------------------------------------------------|
|    Callee's Link Number    |Ptr to Callee's Control Sec|
|--------------------------------------------------------|
|      Backward Pointer      |       Forward Pointer     |
|========================================================|
```

The flags in the left half of the second word have the following meanings:

Bit	Meaning (if bit is on)
0	Module is in core.
1	Module is in more than one link.
2	Relocatable link is already relocated.

The INTTAB Table

The following diagram shows the contents of the INTTAB table:

```
                              INTTAB

     |=========================================================|
     |        0 (Reserved)        |  Address of Entry Point    |
     |---------------------------------------------------------|
     |        0 (Reserved)        |      Forward Pointer       |
     |=========================================================|
                                  .
                                  .
                                  .
     |=========================================================|
     |        0 (Reserved)        |  Address of Entry Point    |
     |---------------------------------------------------------|
     |        0 (Reserved)        |      Forward Pointer       |
     |=========================================================|
```

The Relocation Table

The following diagram shows the contents of the Relocation Table:

Relocation Table

```
|=====================================================|
|                   Relocation Word                   |
|-----------------------------------------------------|
                          .
                          .
                          .
|-----------------------------------------------------|
|                   Relocation Word                   |
|=====================================================|
```

The Relocation Table contains one bit for each halfword of the link. If the bit is on, the halfword is relocatable; if it is off, the halfword is not relocatable.

The first word contains the relocation bits for the first 22 (octal) words of the link; the second word contains the relocation bits for the next 22 (octal) words; and so forth for all words in the link.

This table exists only when relocatable overlays are requested with the /OVERLAY:RELOCATABLE switch.

The Other Relocation Tables

The following diagram shows the contents of the Other Relocation Tables:

Other Relocation Tables

```
|=======================================================|
|         Number of Words Following for This Link       |
|-------------------------------------------------------|
|         Link Number         |   Planned Load Address  |
|-------------------------------------------------------|
|     Relocation Halfword     |   Ptr to Words of Code  |
|-------------------------------------------------------|
                              .
                              .
                              .
|-------------------------------------------------------|
|     Relocation Halfword     |   Ptr to Words of Code  |
|=======================================================|
                              .
                              .
                              .
|=======================================================|
|         Number of Words Following for This Link       |
|-------------------------------------------------------|
|         Link Number         |   Planned Load Address  |
|-------------------------------------------------------|
|     Relocation Halfword     |   Ptr to Words of Code  |
|-------------------------------------------------------|
                              .
                              .
                              .
|-------------------------------------------------------|
|     Relocation Halfword     |   Ptr to Words of Code  |
|=======================================================|
```

This table exists only when relocatable overlays have been requested with the OVERLAY/RELOCATABLE switch. The Other Relocation Tables are used to hold internal LINK references.

APPENDIX A

REL BLOCKS

The object modules that LINK loads are output from the language translators. These object modules are formatted into REL Blocks, each of which contains information for LINK.

This appendix describes each type of REL Block and gives its format. Terms used throughout this discussion are defined as follows:

Header Word: a fullword giving the REL Block Type in its left half and a short count or long count in its right half.

Short Count: a halfword giving the length of the REL Block, excluding relocation words (which appear before each group of 18 (decimal), or 22 (octal) words) and excluding the header word.

Long Count: a halfword giving the length of the REL Block, including all words in the block except the header word itself.

Relocation Word: a fullword containing the relocation bits for up to 18 (decimal) following words. Each relocation bit is either 1, indicating a relocatable halfword, or 0, indicating a nonrelocatable halfword.

The first two relocation bits give the relocatability of the left and right halves, respectively, of the next following word; the next two bits give the relocatability of the two halves of the second following word; and so forth for all bits in the word, except any unused bits, which will be zero.

If a REL Block has relocation words, the first one follows the header word. If more than 18 (decimal) data words follow this relocation word, the next word (after the 18 words) is another relocation word. Thus, a REL Block that has relocation words will have one for each 18 words of data that it contains. If the REL Block does not contain an integral multiple of 18 words, the last relocation word will have unused bits.

NOTE

A block with a zero short count does not include a relocation word.

Data Word: Any word other than a header word or a relocation word.

MBZ: Must Be Zero.

REL BLOCKS

NOTE

All numbers in this appendix are octal unless specifically noted as decimal.

The diagram below shows a REL Block having a short count of 7, and a relocation word.

```
|=======================================================|
|        Block Type         |             7             |
|-------------------------------------------------------|
|                    Relocation Word                    |
|-------------------------------------------------------|
|                      Data Word 1                      |
|-------------------------------------------------------|
|                      Data Word 2                      |
|-------------------------------------------------------|
|                      Data Word 3                      |
|-------------------------------------------------------|
|                      Data Word 4                      |
|-------------------------------------------------------|
|                      Data Word 5                      |
|-------------------------------------------------------|
|                      Data Word 6                      |
|-------------------------------------------------------|
|                      Data Word 7                      |
|=======================================================|
```

REL BLOCKS

The diagram below shows a REL Block having a short count of 31 (octal), and two relocation words.

```
|=========================================================|
|        Block Type           |            31             |
|---------------------------------------------------------|
|                    Relocation Word                      |
|---------------------------------------------------------|
|                      Data Word 1                        |
|---------------------------------------------------------|
                             .
                             .
                             .
|---------------------------------------------------------|
|                     Data Word 22                        |
|---------------------------------------------------------|
|                    Relocation Word                      |
|---------------------------------------------------------|
|                     Data Word 23                        |
|---------------------------------------------------------|
                             .
                             .
                             .
|---------------------------------------------------------|
|                     Data Word 31                        |
|=========================================================|
```

REL Block Types must be numbered in the range 0 to 777777 (octal). The following list shows which numbers are reserved for DIGITAL, and which for customers:

Type Numbers	Use
0 - 37	Reserved for DIGITAL
40 - 77	Reserved for customers
100 - 401	Reserved for DIGITAL
402 - 577	Reserved for customers
600 - 677	Reserved for customer files
700 - 777	Reserved for DIGITAL files
1000 - 1777	Reserved for DIGITAL
2000 - 3777	Reserved for customers
4000 - 777777	Reserved for ASCII text

Block Type 0 (Ignored)

```
|=========================================================|
|             0              |          Short Count        |
|---------------------------------------------------------|
|                     Relocation Word                      |
|---------------------------------------------------------|
|                        Data Word                         |
|---------------------------------------------------------|
                            .
                            .
                            .
|---------------------------------------------------------|
|                        Data Word                         |
|=========================================================|
```

Block Type 0 is ignored by LINK.

If the short count is 0, then no relocation word follows, and the block consists of only one word. This is how LINK bypasses zero words in a REL file.

REL BLOCKS

Block Type 1 (Code)

```
|=========================================================|
|              1              |         Short Count        |
|---------------------------------------------------------|
|                     Relocation Word                      |
|---------------------------------------------------------|
|                        Data Word                         |
|---------------------------------------------------------|
                             .
                             .
                             .
|---------------------------------------------------------|
|                        Data Word                         |
|=========================================================|
```

Block Type 1 contains data and code. The first data word gives the address at which the data is to be loaded. This address can be relocatable or absolute, depending on the value of bit 1 of the relocation word. The remaining data words are loaded beginning at that address.

If the start address is given in symbolic, the following format of Block Type 1 is used:

```
|=========================================================|
|              1              |         Short Count        |
|---------------------------------------------------------|
|                     Relocation Word                      |
|---------------------------------------------------------|
|                         Symbol                           |
|---------------------------------------------------------|
|                         Offset                           |
|---------------------------------------------------------|
                             .
                             .
                             .
|---------------------------------------------------------|
|                        Data Word                         |
|---------------------------------------------------------|
```

In this alternate format, the first four bits of the first data word (Symbol) are 1100 (binary), and the word is assumed to be a Radix-50 symbol of type 60. The load address is calculated by adding the value of the global symbol to the offset given in the following word. The third and following data words are loaded beginning at the resulting address. The global symbol must be defined when the Type 1 Block is found.

Block Type 2 (Symbols)

```
|===================================================|
|              2            |        Short Count     |
|---------------------------------------------------|
|                   Relocation Word                 |
|---------------------------------------------------|
|Code |              Radix-50 Symbol                |
|---------------------------------------------------|
|                 Second Word of Pair               |
|---------------------------------------------------|
                            .
                            .
                            .
|---------------------------------------------------|
|Code |              Radix-50 Symbol                |
|---------------------------------------------------|
|                 Second Word of Pair               |
|===================================================|
```

The first word of each pair has a code in bits 0 to 3 and a Radix-50 symbol in bits 4 to 35 (decimal). The contents of the second word of a pair depends on the given code. The octal codes and their meanings are:

Code **Meaning**

00 This code is illegal in a symbol block.

04 The given symbol is a global definition. Its value, contained in the second word of the pair, is available to other programs.

10 The given symbol is a local definition, and its value is contained in the second word of the pair. If the symbol is followed by one of the special pairs or by a Polish REL Block (as explained below, under code 24), the symbol is considered a partially defined local symbol. Otherwise, it is considered fully defined.

14 The given symbol is a block name (from a translator that uses block structure). The second word of the pair contains the block level. The symbol is considered local; if local symbols are loaded, the value of the block name is entered in the symbol table as its block level.

24 The given symbol is a global definition. However, it is only partially defined at this time, and LINK cannot yet use its value. If the symbol is defined in terms of another symbol, then the next entry in the REL file must be a word pair in a Block Type 2 as follows:

```
|===================================================|
| 60  |                Other Symbol                 |
|---------------------------------------------------|
| 50  |                This  Symbol                 |
|===================================================|
```

In this format, code 50 indicates that the right half of the word depends on the other symbol.

Code **Meaning**

If the partially defined symbol is defined in terms of a Polish expression, then the next entry in the REL file must be Block Type 11 (Polish), whose store operator gives this symbol as the symbol to be fixed up. The store operator must be -4 or -6.

30 The given symbol is a global definition. However, it is only partially defined at this time, and LINK cannot yet use its value. If the symbol is defined in terms of another symbol, then the next entry in the REL file must be a word pair in a Block Type 2 as follows:

```
|=========================================================|
| 60  |                 Other Symbol                      |
|---------------------------------------------------------|
| 70  |                 This  Symbol                      |
|=========================================================|
```

In this format, code 70 indicates that the left half of the word depends on the other symbol.

If the partially defined symbol is defined in terms of a Polish expression, then the next entry in the REL file must be Block Type 11 (POLISH), whose store operator gives this symbol as the symbol to be fixed up. The store operator must be -5.

34 The given symbol is a global definition. However, it is only partially defined at this time, and LINK cannot yet use its value. If the symbol is defined in terms of another symbol, then the next entry in the REL file must be a word pair in a Block Type 2 as follows:

```
|=========================================================|
| 60  |                 Other Symbol                      |
|---------------------------------------------------------|
| 50  |                 This  Symbol                      |
|---------------------------------------------------------|
| 60  |                 Other Symbol                      |
|---------------------------------------------------------|
| 70  |                 This  Symbol                      |
|=========================================================|
```

This format indicates that both halves of the word depend on the other symbol.

44 The given symbol is a global definition exactly as in code 04, except that it is not output by DDT.

50 The given symbol is a local symbol exactly as in code 10, except that it is not output by DDT.

60 The given symbol is a global request. LINK's handling of the symbol depends on the value of the code in the first four bits of the second word of the pair. These codes and their meanings are:

00 The right half of the word gives the address of the first word in a chain of requests for the global memory address. In each request, the right half of the word gives the address of the next request. The addresses in the chain must be strictly descending; the chain ends when the address is 0.

Code **Meaning**

40 The right half of the word contains an address. The right half of the value of the requested symbol is added to the right half of this word.

50 The rest of the word contains a Radix-50 symbol whose value depends on the requested global symbol. (If the given Radix-50 symbol is not the one defined in the previous word pair, then this word is ignored.) When the value of the requested symbol is resolved, it is added to the right half of the value of the Radix-50 symbol.

60 The right half of the word contains an address. The right half of the value of the requested symbol is added to the left half of this word.

70 The rest of the word contains a Radix-50 symbol whose value depends on the requested global symbol. (If the given Radix-50 symbol is not the one defined in the previous word pair, then this word is ignored.) When the value of the requested global symbol is resolved, it is added to the left half of the value of the Radix-50 symbol.

64 The given symbol is a global definition exactly as in code 24, except that it is not output by DDT.

70 The given symbol is partially defined, where the left half is deferred, as in code 30, except that it is not output by DDT.

74 The given symbol is partially defined, where the right half is deferred, as in code 34, except that it is not output by DDT.

Symbols are placed in the symbol table in the order that LINK finds them. However, DDT expects to find the symbols in a specific order.

For a non-block-structured program, that order is:

Program Name

Symbols for Program

For a block-structured program whose structure is:

```
Begin Block 1 (same as program name)
     Begin Block 2
     End Block 2
     Begin Block 3
          Begin Block 4
          End Block 4
     End Block 3
End Block 1
```

the order is:

```
Program Name (Block 1)
Block Name 2
Symbols for Block 2
Block Name 4
Symbols for Block 4
Block Name 3
Symbols for Block 3
Block Name 1
Symbols for Block 1
```

This ordering follows the rule that the name and symbols for each block must occur in the symbol table **in the order of the block endings** in the program.

NOTES

1. Only one fixup by a Type 2, 10, 11, 1070, or 1072 Block is allowed for a given field. (There can be separate fixups for the left and right halves of the same word.)

2. Fixups are not necessarily performed in the order LINK finds them.

Block Type 3 (HISEG)

```
|=========================================================|
|                3                |         1 or 2        |
|---------------------------------------------------------|
|                     Relocation Word                     |
|---------------------------------------------------------|
|High-Segment Program Break |    High-Segment Origin      |
|---------------------------------------------------------|
|(Low-Segment Program Break)|   (Low-Segment Origin)      |
|=========================================================|
```

Block Type 3 tells LINK that code is to be loaded into the high segment.

If the left half of the first data word is 0, subsequent Type 1 blocks found are assumed to have been produced by the MACRO pseudo-op HISEG. This usage is not recommended. It means that the addresses in the blocks are relative to 0, but are to be placed in the program high segment. The right half of the first data word is the beginning of the high segment (usually 400000).

If the left half of the first data word is nonzero (the preferred usage), subsequent Type 1 blocks found are assumed to have been produced by the MACRO pseudo-op TWOSEG.

The right half is interpreted as the beginning of the high segment, and the left half is the high-segment break; the high-segment length is the difference of the left and right halves.

(One-pass translators that cannot calculate the high-segment break should set the left half equal to the right half.)

If the second word appears in the HISEG block, its left half shows the low-segment program break, and its right half shows the low-segment origin (usually 0).

Block Type 4 (Entry)

```
|=============================================================|
|              4              |         Short Count           |
|-------------------------------------------------------------|
|                  Relocation Word (Zero)                     |
|-------------------------------------------------------------|
|                     Radix-50 Symbol                         |
|-------------------------------------------------------------|
                              .
                              .
                              .
|-------------------------------------------------------------|
|                     Radix-50 Symbol                         |
|=============================================================|
```

Block Type 4 lists the entry name symbols for a program module. If a Type 4 block appears in a module, it must be the first block in the module. A library file contains a Type 4 block for each of its modules.

When LINK is in library search mode, the symbols in the block are compared to the current list of global requests for the load. If one or more matches occur, the module is loaded and the name of the module is marked as an entry point in map files, etc. If no match occurs, the module is not loaded.

If LINK is not in library search mode, no comparison of requests and entry names is made, and the module is always loaded. Refer to Block Type 17 for more information about libraries.

Block Type 5 (End)

```
|=======================================================|
|               5              |             2           |
|-------------------------------------------------------|
|                   Relocation Word                     |
|-------------------------------------------------------|
|                   First Data Word                     |
|-------------------------------------------------------|
|                  Second Data Word                     |
|=======================================================|
```

Block Type 5 ends a program module. A Block Type 6 must be encountered earlier in the module than the Type 5 block.

If the module contains a two-segment program, the first data word is the high-segment break and the second data word is the low-segment break. If the module contains a one-segment program, the first data word is the program break and the second data word is the absolute break.

Each PRGEND pseudo-op in a MACRO program generates a Type 5 REL block. Therefore a REL file may contain more than one Type 5 block.

A library REL file has a Type 5 block at the end of each of its modules.

Block Type 6 (Name)

```
|==========================================================|
|              6              |           1 or 2           |
|----------------------------------------------------------|
|                     Relocation Word                      |
|----------------------------------------------------------|
|                     Radix-50 Symbol                      |
|----------------------------------------------------------|
| (CPU)  |    (Compiler)       | (Length of Blank Common)   |
|----------------------------------------------------------|
```

Block Type 6 contains the program name, and must precede any Type 2 blocks. (A module should begin with a Type 6 block and end with a Type 5 block.)

The first data word is the program name in Radix-50 format; this name cannot be blanks. The second data word is optional; if it appears, it contains CPU codes in bits 0 to 5, a compiler code in bits 6 to 17 (decimal), and the length of the program's blank COMMON in the right halfword.

The CPU codes specify processors for program execution as:

Bit 2	KS10
Bit 3	KL10
Bit 4	KI10
Bit 5	KA10

If all these bits are off, then any of the processors can be used for execution.

The compiler code specifies the compiler that produced the REL file. The defined codes are:

0	Unknown	7	SAIL	16	COBOL-74
1	Not used	10	FORTRAN	17	(Reserved)
2	COBOL-68	11	MACRO	20	BLISS-36
3	ALGOL	12	FAIL	21	BASIC
4	NELIAC	13	BCPL	22	SITGO
5	PL/I	14	MIDAS	23	G-floating FORTRAN
6	BLISS	15	SIMULA	24	PASCAL
				25	JOVIAL

Block Type 7 (Start)

```
|=========================================================|
|              7              |            1 or 2          |
|---------------------------------------------------------|
|                     Relocation Word                      |
|---------------------------------------------------------|
|                      Start Address                       |
|---------------------------------------------------------|
|(60) |                (Radix-50 Symbol)                  |
|=========================================================|
```

Block Type 7 contains the start address for program execution. LINK uses the start address in the last such block processed by the load, unless /START or /NOSTART switches specify otherwise. However, if the left halfword of the start address is nonzero, the halfword is interpreted as the length of the entry vector, which is located at the address specified in the right halfword. Refer to Section 4.2.1 for further information.

If the second (optional) word is present, it must be a Radix-50 symbol with the code 60; LINK forms the start address by adding the value of the symbol to the value in the right half of the preceding word (Start Address).

Block Type 10 (Internal Request)

```
|=======================================================|
|           10              |        Short Count        |
|-------------------------------------------------------|
|                    Relocation Word                    |
|-------------------------------------------------------|
|  Pointer to Last Request  |           Value           |
|-------------------------------------------------------|
                            .
                            .
                            .
|-------------------------------------------------------|
|  Pointer to Last Request  |           Value           |
|-------------------------------------------------------|
```

Block Type 10 is generated by one-pass compilers to resolve requests caused by forward references to internal symbols. The MACRO assembler also generates Type 10 blocks to resolve requests for labels defined in literals; a separate chain is required for each PSECT in a PSECTed program.

Each data word contains one request for an internal symbol. The left half is the address of the last request for a given symbol. The right half is the value of the symbol. The right half of the last request contains the address of the next-to-last request, and so on, until a zero right half is found. (This is exactly analogous to Radix-50 code 60 with second-word code 00 in a Block Type 2.)

If a data word contains -1, then the following word contains a request for the left (rather than right) half of the specified word. In this case, the left half of the word being fixed up contains the address of the next-to-last left half request, and so on, until a zero left half is found. (This is a left half chain analogous to the right half chain described above.)

NOTES

1. Only one fixup by a Type 2, 10, 11, 1070, or 1072 Block is allowed for a given field. (There can be separate fixups for the left and right halves of the same word.)

2. Fixups are not necessarily performed in the order LINK finds them.

Block Type 11 (Polish)

```
|=======================================================|
|            11             |        Short Count        |
|-------------------------------------------------------|
|                    Relocation Word                    |
|-------------------------------------------------------|
|       Data Halfword       |       Data Halfword       |
|-------------------------------------------------------|
                            .
                            .
                            .
|-------------------------------------------------------|
|       Data Halfword       |       Data Halfword       |
|=======================================================|
```

Block Type 11 defines Polish fixups for operations on relocatable values or external symbols. Only one store operator code can appear in a Block Type 11; this store operator code can be either a symbol fixup code or a chained fixup code. The store operator code appears at the end of the block.

NOTES

1. Only one fixup by a Type 2, 10, 11, 1070, or 1072 Block is allowed for a given field. (There can be separate fixups for the left and right halves of the same word.)

2. Fixups are not necessarily performed in the order LINK finds them.

The data words of a Type 11 block form one Polish string of halfwords. Each halfword contains one of the following:

1. A symbol fixup store operator code.

   A symbol fixup defines the value to be stored in the value field of the symbol table for the given symbol. A symbol fixup store operator code is followed by four data halfwords.

2. A chained fixup store operator code.

   A chained fixup takes a relocatable address whose corrected virtual address is the location for storing or chaining. A chained fixup store operator code is followed by one data halfword.

3. A data type code. Data type code 0 is followed by a data halfword; a data type code 1 or 2 is followed by two data halfwords.

4. An arithmetic or logical operator code.

5. A PSECT index code. This code defines a PSECT index to be used for calculating the relocated addresses that appear in this block. PSECT indexes are needed only for PSECTed programs.

   A global PSECT index is associated with a Block Type 11. This index appears as the first halfword after the relocation word, and it defines the PSECT for the store address or store symbol. Any addresses for a different PSECT must be preceded by a different PSECT index.

   Thus, a relocatable data halfword in a different PSECT must appear in one of the following formats:

```
|---------------------------------------------------------|
|          400nnn          |     (operator code)          |
|---------------------------------------------------------|
|                       (operands)                        |
|---------------------------------------------------------|
```

OR

```
|---------------------------------------------------------|
|          . . .           |          400nnn              |
|---------------------------------------------------------|
|      (operator code)     |        (operands)            |
|---------------------------------------------------------|
```

   where the different PSECT index is nnn+1.

   Any relocatable address that does not have an explicit preceding PSECT index code preceding its data type code is assumed to be in the same PSECT as the store address for the block. The current PSECT may be set by a previous REL Block type.

6. A halfword of data (preceded by a data type 0 halfword) or two halfwords of data (preceded by a data type 1 or 2 halfword).

   A sequence of halfwords containing a data type code 0 and a data halfword can begin in either half of a word.

The codes and their meanings are:

Symbol Fixup Store Operator Codes:

-7 Fullword replacement. No chaining is done.

-6 Fullword symbol fixup. The following one or two words contain the Radix-50 symbol(s) (with their 4-bit codes). The first is the symbol to be fixed up, and the second is the block name for a block-structured program (0 or nonexistent for other programs).

-5 Left half symbol fixup. The following one or two words contain the Radix-50 symbols. The first is the symbol to be fixed up, and the second is the block name for a block-structured program (0 or nonexistent for other programs).

-4 Right half symbol fixup. The following one or two words contain the Radix-50 symbols. The first is the symbol to be fixed up, and the second is the block name for a block-structured program (0 or nonexistent for other programs).

Chained Fixup Store Operator Codes:

-3 Fullword chained fixup. The halfword following points to the first element in the chain. The entire word pointed to is replaced, and the old right half points to the next fullword.

-2 Left half chained fixup. The halfword following points to the first element in the chain.

-1 Right half chained fixup. The halfword following points to the first element in the chain.

Data Type Codes:

0 The next halfword is an operand.

1 The next two halfwords form a fullword operand.

2 The next two halfwords form a Radix-50 symbol that is a global request. The operand is the value of the symbol.

Arithmetic and Logical Operator Codes:

NOTE

Operands are read in the order that they are encountered.

3 Add.

4 Subtract.

5 Multiply.

6 Divide.

7 Logical AND.

10 Logical OR.

11 Logical shift. (A positive second operand causes a shift to the left. A negative operand causes a shift to the right.)

12 Logical XOR.

13 Logical NOT (one's complement).

14 Arithmetic negation (two's complement).

15 Count leading zeros (like JFFO instruction). Refer to the MACRO Assembler Reference Manual for information about the ^L operand, which this code implements.

16 Remainder.

17 Magnitude.

20 Maximum.

21 Minimum.

22 Comparison. Returns 0 if the two operands are different; -1 if they are equal.

23 Used to resolve the links in a chain. Refer to Block Type 12.

24 Symbol definition test. Returns 0 if the operand (a Radix-50 symbol) is unknown; 1 if it is known but undefined; -1 if it is known and defined.

25 Skip N words of Polish.

26 Skip N words of Polish on some condition.

27 Return contents of location N.

PSECT Index Codes:

400nnn PSECT index nnn, where nnn is a 3-digit octal integer.

For an example of a Type 11 block, the MACRO statements

```
        EXTERN B
A:      EXP <A*B+A>
```

Generate (assuming that A has a relocatable value of zero):

```
|==========================================================|
|             11              |              6             |
|----------------------------------------------------------|
|00|01|00|00|10|10|                  0                    |
|----------------------------------------------------------|
|          3 (Add)            |        5 (Multiply)        |
|----------------------------------------------------------|
| 0 (Halfword Operand Next)   |       0 (Relocatable)      |
|----------------------------------------------------------|
|2 (Fullword Radix-50 Next)   |   1st Half of Radix-50 B   |
|----------------------------------------------------------|
|  2nd Half of Radix-50 B     | 0 (Halfword Operand Next)  |
|----------------------------------------------------------|
|      0 (Relocatable)        |-3 (Rh Chained Fixup Next)  |
|----------------------------------------------------------|
|  0 (Chain Starts at 0')     |            . . .           |
|==========================================================|
```

The first word contains the block type (11) and the short count (6). The second word is the relocation word; it shows that the following halfwords are to be relocated: right half of second following word, left half of fifth following word, left half of sixth following word.

The next word shows that the two operations to be performed are addition and multiplication; because this is in Polish prefix format, the multiplication is to be performed on the first two operands first, then addition is performed on the product and the third operand.

The next two halfwords define the first operand. The first halfword is a data type code 0, showing that the operand is a single halfword; the next halfword is the operand (relocatable 0).

The next three halfwords define the second operand. The first of these halfwords contains a data type code 2, showing that the operand is two halfwords containing a Radix-50 symbol with code 60. The next two halfwords give the symbol (B).

The next two halfwords define the third operand. The first of these halfwords contains a data type code 0, showing that the operand is a single halfword; the next halfword gives the value of the operand (relocatable 0).

The next two halfwords give the store operator for the block. The first of these halfwords contains the chained fixup store operator code -3, showing that a fullword chained fixup is required; the next halfword contains the operand (relocatable 0), showing that the chain starts at relocatable zero.

The last halfword is irrelevant, and should be zero. If it is not, LINK issues the LNKJPB error message.

Block Type 12 (Chain)

```
|=========================================================|
|              12             |          Short Count      |
|---------------------------------------------------------|
|                      Relocation Word                    |
|---------------------------------------------------------|
|                       Chain Number                      |
|---------------------------------------------------------|
|        Chain Address        |        Store Address      |
|---------------------------------------------------------|
                              .
                              .
                              .
|---------------------------------------------------------|
|                       Chain Number                      |
|---------------------------------------------------------|
|        Chain Address        |        Store Address      |
|=========================================================|
```

Block Type 12 chains together data structures from separately compiled modules. (The MACRO pseudo-ops .LINK and .LNKEND generate Type 12 blocks.) Block Type 12 allows linked lists that have entries in separately compiled modules to be constructed so that new entries can be added to one module without editing or recompiling any other module.

The data words in a Type 12 block are paired. The first word of each pair contains a chain number between 1 and 100 (octal). (The chain number is negative if the pair was generated by a .LNKEND pseudo-op.) The second word contains a store address in the right half, and a chain address in the left half. The store address points to the location where LINK will place the chain address of the last entry encountered for the current chain. The first entry in a chain has a zero in the word pointed to by the store address.

A MACRO statement of the form:

```
     .LINK chain-number,store-address,chain-address
```

generates a word pair in a Type 12 block as shown above. A MACRO statement of the form:

```
     .LINK chain-number,store-address
```

generates a word pair in a Type 12 block with a 0 for the chain address field in the REL block. A MACRO statement of the form:

```
     .LNKEND chain-number,store-address
```

generates a word pair in a Type 12 Block with a 0 for the chain address and a negative chain number.

As LINK processes a load, it performs a separate chaining for each different chain number found; thus a word pair in a Type 12 block is related to all other word pairs having the same chain number (even in other loaded modules). Type 12 pairs having different chain numbers (even in the same module), are not related.

To show how the chains are formed, we will take some pairs from different programs having the same chain number (1 in the example). The following four programs contain .LINK or .LNKEND pseudo-ops for the chain numbered 1. After each program, the word pair generated in the Type 12 block appears.

NOTE

> When LINK stores an address resulting from a Type 12 REL Block, only the right half of the receiving location is written. You can safely store another value in the left half; it will not be overwritten.

Example

```
TITLE MOD0
          .
          .
          .
TAG0:   BLOCK 1
          .
          .
          .
        .LNKEND 1,TAG0
          .
          .
          .
        END

|=======================================================|
|                          -1                           |
|-------------------------------------------------------|
|             0             |       Value of TAG0       |
|=======================================================|

TITLE MOD1
          .
          .
          .
TAG1:   BLOCK 1
          .
          .
          .
        .LINK 1,TAG1
          .
          .
          .
        END

|=======================================================|
|                           1                           |
|-------------------------------------------------------|
|             0             |       Value of TAG1       |
|=======================================================|

TITLE MOD2
          .
          .
          .
TAG2:   BLOCK 1
          .
          .
          .
        .LINK 1,TAG2
          .
          .
          .
        END

|=======================================================|
|                           1                           |
|-------------------------------------------------------|
|             0             |       Value of TAG2       |
|=======================================================|

TITLE MOD3
          .
          .
          .
TAG3:   BLOCK 1
          .
          .
          .
TAG33:  BLOCK 1
          .
          .
          .
        .LINK 1,TAG33,TAG3
          .
          .
          .
        END

|=======================================================|
|                           1                           |
|-------------------------------------------------------|
|       Value of TAG3       |       Value of TAG33      |
|=======================================================|
```

Suppose we load MOD0 first. The .LNKEND statement for MOD0 generates a negative chain number. LINK sees the negative chain number (-1) and recognizes this as the result of a .LNKEND statement for chain number 1. LINK remembers the store address (value of TAG0) as the base of the chain.

Next we load MOD1. The .LINK statement for MOD1 does not use the third argument, so the chain address is 0. LINK sees that this is the first entry for chain number 1. Because it is the first entry, LINK places a 0 in the store address (value of TAG1). LINK then remembers the value of TAG1 for use in the next chain entry. (If the chain address is 0, as it is in MOD1, LINK remembers the store address; if the chain address is nonzero, LINK remembers the chain address.)

Next we load MOD3. The .LINK statement in MOD3 uses a third argument (TAG3), therefore, the value of TAG3 is used as the chain address. LINK places its remembered address (value of TAG1) in the store address (value of TAG33). Because the chain address (value of TAG3) is nonzero, LINK remembers it for the next entry.

Finally we load MOD2. Like MOD1, the .LINK statement for MOD2 does not take a third argument, and thus the chain address is 0. LINK places the remembered address (value of TAG3) in the store address (value of TAG2). Because the chain address is 0, LINK remembers the store address (value of TAG2).

At the end of loading, LINK places the last remembered address (value of TAG2) at the address (value of TAG0) given by the .LNKEND statement in MOD0.

The results of the chaining can be seen in the following diagram of the loaded core image:

```
                MOD0                      MOD2
            -----------------        ----------------
            |                |       |               |
            |                |       |               |
TAG0:       |Value of TAG2   |  TAG2:|Value of TAG3  |
            |                |       |               |
            |                |       |               |
            |                |       |               |
            -----------------        ----------------

                 MOD3                     MOD1
            -----------------        ----------------
            |                |       |               |
            |                |       |               |
TAG3:       |                |  TAG1:|      0        |
            |                |       |               |
TAG33:      |Value of TAG1   |       |               |
            |                |       |               |
            -----------------        ----------------
```

Note that the order of loading for modules with .LINK statements is critical. (A module containing a .LNKEND statement can be loaded any time; its treatment is not affected by the order of loading.)

For example, if we load the four programs in the order MOD2, MOD3, MOD0, MOD1, we get a different resulting core image:

```
           MOD0                    MOD1
        -----------------       ----------------
        |               |       |              |
        |               |       |              |
TAG0:   |Value of TAG1  |  TAG1:|Value of TAG3 |
        |               |       |              |
        |               |       |              |
        |               |       |              |
        -----------------       ----------------

           MOD3                    MOD2
        -----------------       ----------------
        |               |       |              |
        |               |       |              |
TAG3:   |               |  TAG2:|       0      |
        |               |       |              |
TAG33:  |Value of TAG2  |       |              |
        |               |       |              |
        -----------------       ----------------
```

Block Type 14 (Index)

```
|=======================================================|
|             14            |            177             |
|-------------------------------------------------------|
|                       Sub-Block                        |
|                                                       |
|-------------------------------------------------------|
                            .
                            .
                            .
|-------------------------------------------------------|
|                                                       |
|                        Sub-Block                       |
|                                                       |
|-------------------------------------------------------|
|             -1            | Ptr To Nxt Rel Blk Typ 14 |
|=======================================================|
```

Each sub-block is of the form:

```
|=======================================================|
|     Index-Version Number   |      Count of Symbols      |
|-------------------------------------------------------|
|                     Radix-50 Symbol                    |
|-------------------------------------------------------|
                            .
                            .
                            .
|-------------------------------------------------------|
|                     Radix-50 Symbol                    |
|-------------------------------------------------------|
|       Pointer to Module Containing Entry Symbols       |
|=======================================================|
```

Block Type 14 contains a list of all entry points in a library produced by MAKLIB. The block contains 177 (octal) data words (with no relocation words); if the index requires more entries, additional Type 14 blocks are used.

The Type 14 block consists of a header word, a number of sub-blocks, and a trailer word containing the disk block address of the next Type 14 block, if any. Each disk block is 128 words.

Each sub-block is like a Type 4 block, with three differences:

1. The sub-block has no relocation words.

2. The last word of the sub-block points to the module that contains the entry points listed in the sub-block. The right half of the pointer has the disk block number of the module within the file; the left half has the number of words (in that block) that precede the module. If there is no next block, then the word after the last sub-block is 0.

3. The index-version number is used so that old blocks can still be loaded, even if the format changes in the future.

Block Type 15 (ALGOL)

```
|==========================================================|
|            15              |        Short Count          |
|----------------------------------------------------------|
|                     Relocation Word                      |
|----------------------------------------------------------|
|        Load Address        |          Length             |
|----------------------------------------------------------|
|        Chain Address       |          Offset             |
|----------------------------------------------------------|
                             .
                             .
                             .
|----------------------------------------------------------|
|        Chain Address       |          Offset             |
|==========================================================|
```

Block Type 15 is used to build the special ALGOL OWN block.

The first data word contains the length of the module's OWN block in the right half, and the desired load address for the current OWN block in the left half. Each following word contains an offset for the start of the OWN block in the right half, and the address of a standard righthalf chain of requests for that word of the OWN block in the left half.

When LINK sees a REL Block Type 15, it allocates a block of the requested size at the requested address. The length of the block is then placed in the left half of the first word, and the address of the last OWN block seen is placed in the right half. If this is the first OWN block seen, 0 is stored in the right half of the first word.

The remaining data words are then processed by adding the address of the first word of the OWN block to each offset, and then storing the resulting value in all the locations chained together, starting with the chain address.

At the end of loading, LINK checks to see if the symbol %OWN is undefined. If it is undefined, then it is defined to be the address of the last OWN block seen. In addition, if LINK is creating an ALGOL symbol file, the file specification of the symbol file is stored in the first OWN block loaded. This file specification must be standard TOPS-10 format and can include (in order): device, file name, file type, and project-programmer number.

Block Type 16 (Request Load)

```
|=======================================================|
|              16             |        Short Count       |
|-------------------------------------------------------|
|               Relocation Word (Zero)                   |
|-------------------------------------------------------|
|                  SIXBIT Filename                       |
|-------------------------------------------------------|
|              Project-Programmer Number                 |
|-------------------------------------------------------|
|                   SIXBIT Device                        |
|-------------------------------------------------------|
                            .
                            .
                            .
|-------------------------------------------------------|
|                  SIXBIT Filename                       |
|-------------------------------------------------------|
|              Project-Programmer Number                 |
|-------------------------------------------------------|
|                   SIXBIT Device                        |
|=======================================================|
```

Block Type 16 contains a list of files to be loaded. The data words are arranged in triplets; each triplet contains information for one file: file name, project-programmer number, and device. The file type is assumed to be REL, unless you have specified otherwise with /DEFAULT.

LINK saves the specifications for the files to be loaded, discarding duplicates. At the end of loading, LINK loads all specified files immediately before beginning library searches.

The MACRO pseudo-op .REQUIRE generates a Type 16 REL Block.

Block Type 17 (Request Library)

```
|=========================================================|
|             17              |          Short Count       |
|---------------------------------------------------------|
|                 Relocation Word (Zero)                   |
|---------------------------------------------------------|
|                     SIXBIT Filename                      |
|---------------------------------------------------------|
|                Project-Programmer Number                 |
|---------------------------------------------------------|
|                      SIXBIT Device                       |
|---------------------------------------------------------|
                             .
                             .
                             .
|---------------------------------------------------------|
|                     SIXBIT Filename                      |
|---------------------------------------------------------|
|                Project-Programmer Number                 |
|---------------------------------------------------------|
|                      SIXBIT Device                       |
|=========================================================|
```

Block Type 17 is identical to Block Type 16 except that the specified files are loaded in library search mode. The specified files are searched after loading files given in Type 16 blocks, but before searching system or user libraries.

The MACRO pseudo-op .REQUEST generates a Type 17 REL Block.

Block Type 20 (Common)

```
|=========================================================|
|              20             |          Short Count       |
|---------------------------------------------------------|
|                 Relocation Word (Zero)                  |
|---------------------------------------------------------|
|                    Radix-50 Symbol                      |
|---------------------------------------------------------|
|             Length of Labeled Common Block              |
|---------------------------------------------------------|
                             .
                             .
                             .
|---------------------------------------------------------|
|                    Radix-50 Symbol                      |
|---------------------------------------------------------|
|             Length of Labeled Common Block              |
|=========================================================|
```

Block Type 20 allocates labeled COMMON areas. The label for unlabeled COMMON is ".COMM.". If a Block Type 20 appears in a REL file, it must appear before any other block that causes code to be loaded or storage to be allocated in the core image.

The data words are arranged in pairs. The first word of each pair contains a COMMON name in Radix-50 format (the four-bit code field must contain 60). The second contains the length of the area to be allocated.

For each COMMON entry found, LINK first determines whether the COMMON area is already allocated. If not, LINK allocates it. If the area has been allocated, the allocated area must be at least as large as the current requested allocation.

COMMON blocks can be referenced from other block types as standard globally defined symbols. However, a COMMON block must be initially allocated by a REL Block Type 20, by a REL Block Type 6 (for blank COMMON), or by the /COMMON switch to LINK. Any attempt to initially define a COMMON block with a standard global symbol definition causes the LNKSNC error when the redefining Block Type 20 is later seen.

Block Type 21 (Sparse Data)

```
|=========================================================|
|              21             |          Short Count       |
|---------------------------------------------------------|
|                 Relocation Word (Zero)                   |
|---------------------------------------------------------|
|                                                          |
|                        Sub-Block                         |
|                                                          |
|---------------------------------------------------------|
                              .
                              .
                              .
|---------------------------------------------------------|
|                                                          |
|                        Sub-Block                         |
|                                                          |
|=========================================================|
```

Each sub-block is of the form:

```
|=========================================================|
|          Long Count         |            Address         |
|---------------------------------------------------------|
|                        Data Word                         |
|---------------------------------------------------------|
                              .
                              .
                              .
|---------------------------------------------------------|
|                        Data Word                         |
|=========================================================|
```

Block Type 21 contains data to be loaded sparsely in a large area. The first word of each sub-block contains the long count for the sub-block in the left half, and the address for loading the data words in the right half.

If the first four bits of the first data word of each sub-block are 1100 (binary) then the word is assumed to be a Radix-50 symbol of type 60; in this case the left half of the second word is the sub-block count, and the right half plus the value of the symbol is the load address.

Block Type 22 (PSECT Origin)

```
|=========================================================|
|            22             |           Short Count       |
|---------------------------------------------------------|
|                      Relocation Word                    |
|---------------------------------------------------------|
|          (SIXBIT PSECT Name) or (PSECT Index)           |
|---------------------------------------------------------|
|                       PSECT Origin                      |
|=========================================================|
```

Block Type 22 contains the PSECT origin (base address).

Block Type 22 tells LINK to set the value of the relocation counter to the value of the counter associated with the given PSECT name. All following REL blocks are relocated with respect to this PSECT until the next Block Type 22 or 23 is found.

When data or code is being loaded into this PSECT, all relocatable addresses are relocated for the PSECT counter.

MACRO generates a Block Type 22 for each .PSECT and .ENDPS pseudo-op it processes. These Type 22 blocks are interleaved with the other blocks to indicate PSECT changes. A Type 22 block is also generated at the beginning of each symbol table to show which PSECT the table belongs in.

Block Type 23 (PSECT End Block)

```
|============================================================|
|              23              |         Short Count         |
|------------------------------------------------------------|
|                         PSECT Index                        |
|------------------------------------------------------------|
|                         PSECT Break                        |
|============================================================|
```

Block Type 23 contains information about a PSECT.

The PSECT index uniquely identifies the PSECT within the module being loaded. The Type 24 block assigns the index.

The PSECT break gives the length of the PSECT. This break is relative to the zero address of the current module, not to the PSECT origin.

Block Type 24 (PSECT Header Block)

```
|=======================================================|
|              24           |           Count           |
|-------------------------------------------------------|
|                   Relocation Word                     |
|-------------------------------------------------------|
|                 PSECT Name (SIXBIT)                   |
|-------------------------------------------------------|
|         Attributes        |         PSECT Index       |
|-------------------------------------------------------|
|                     PSECT Origin                      |
|=======================================================|
```

Block Type 24 contains information concerning a specified PSECT. The first word contains the block type number and the number of words associated with the block. The second word contains the relocation information. The third word contains the PSECT name in SIXBIT. The fourth word is the PSECT origin specified for this module.

Bit	Interpretation	MACRO .PSECT Keyword
11	PSECT is all within section one. This is the default.	
12	PSECT is in a nonzero section.	
B13	PSECT is page-aligned.	PALIGNED
B14	Concatenate parts of PSECTs seen in distinct modules.	CONCATENATE
B15	Overlay parts of PSECTs seen in distinct modules.	OVERLAY
B16	Read-only	RONLY
B17	Read and write	RWRITE

LINK must find a Type 24 block for a PSECT before it finds the index for that PSECT. (MACRO generates a complete set of Type 24 blocks for all PSECTS in a module before generating Type 2 (Symbol Table) Blocks and Type 11 (POLISH) Blocks.)

Block Type 37 (COBOL Symbols)

```
|=========================================================|
|              37              |       Short Count        |
|---------------------------------------------------------|
|                     Relocation Word                     |
|---------------------------------------------------------|
|                        Data Word                        |
|---------------------------------------------------------|
                               .
                               .
                               .
|---------------------------------------------------------|
|                        Data Word                        |
|=========================================================|
```

Block Type 37 contains a debugging symbol table for COBDDT, the COBOL debugging program. If local symbols are being loaded, the table is loaded.

If a REL file contains a Block Type 37, it must appear after all other blocks that cause code to be loaded or storage to be allocated in the core image.

This block is in the same format as the Type 1 REL Block.

Block Type 100 (.ASSIGN)

```
|=========================================================|
|              100              |        Short Count      |
|---------------------------------------------------------|
|                    Relocation Word                      |
|---------------------------------------------------------|
|Code |                Radix-50 Symbol 1                  |
|---------------------------------------------------------|
|Code |                Radix-50 Symbol 2                  |
|---------------------------------------------------------|
|                         Offset                          |
|=========================================================|
```

Block Type 100 defines Symbol 1 (in the diagram above) as a new global symbol with the current value of Symbol 2, and then increases the value of Symbol 2 by the value of the given offset.

NOTE

Symbol 2 must be completely defined when the Block Type 100 is found.

Block Type 776 (Symbol File)

```
|==========================================================|
|             776             |         Long Count         |
|----------------------------------------------------------|
|          .JBSYM-Style Symbol Table Pointer                |
|----------------------------------------------------------|
|          .JBUSY-Style Symbol Table Pointer                |
|----------------------------------------------------------|
|                         Data Word                         |
|----------------------------------------------------------|
                              .
                              .
                              .
|----------------------------------------------------------|
|                         Data Word                         |
|==========================================================|
```

Block Type 776 must begin in the first word of the file, if it occurs at all. This block type shows that the file is a Radix-50 symbol file.

The data words form a Radix-50 symbol table for DDT in the same format as the table loaded for the switches /LOCALS/SYMSEG or the switch /DEBUG.

Block Type 777 (Universal File)

```
|=========================================================|
|           777            |          Long Count          |
|---------------------------------------------------------|
|                       Data Word                         |
|---------------------------------------------------------|
                            .
                            .
                            .
|---------------------------------------------------------|
|                       Data Word                         |
|=========================================================|
```

Block Type 777 is included in a universal (UNV) file that is produced by MACRO so that LINK will recognize when a UNV file is being loaded incorrectly. When a Block Type 777 is found, LINK produces a ?LNKUNS error.

Block Type 1000 (Ignored)

```
|=========================================================|
|            1000            |          Long Count        |
|---------------------------------------------------------|
|                        Data Word                        |
|---------------------------------------------------------|
                             .
                             .
                             .
|---------------------------------------------------------|
|                        Data Word                        |
|=========================================================|
```

Block Type 1000 is ignored by LINK.

Block Type 1001 (Entry)

```
|=========================================================|
!        1001                 |             Count         !
|---------------------------------------------------------|
!                          Symbol                         !
|---------------------------------------------------------|
!                          Symbol                         !
|=========================================================|
```

Block type 1001 is used to declare symbolic entry points. Each word contains one SIXBIT symbol.

Block Type 1002 (Long Entry)

```
|=========================================================|
|              1002            |             Count         |
|---------------------------------------------------------|
|                       Symbol Name                        |
|---------------------------------------------------------|
                              .
                              .
                              .
|---------------------------------------------------------|
|                        Symbol Name                       |
|=========================================================|
```

Block type 1002 is used to declare a symbolic entry point with a long name in SIXBIT. The count reflects the symbol length in words.

Block Type 1003 (Long Title)

```
|=========================================================|
|             1003            |            Count          |
|---------------------------------------------------------|
|            640000           |              0            |
|---------------------------------------------------------|
|                      Program Title                      |
|---------------------------------------------------------|
|                            0                            |
|---------------------------------------------------------|
|            040000           |              0            |
|---------------------------------------------------------|
|                       More Title                        |
|---------------------------------------------------------|
|                       More Title                        |
|---------------------------------------------------------|
                             .
                             .
                             .
|---------------------------------------------------------|
|            044000           |              0            |
|---------------------------------------------------------|
|                      ASCII Comment                      |
|---------------------------------------------------------|
|                      More Comment                       |
|---------------------------------------------------------|
                             .
                             .
                             .
|---------------------------------------------------------|
|            042000           |              0            |
|---------------------------------------------------------|
|                      Compiler Name                      |
|---------------------------------------------------------|
|             Code            |             CPUs          |
|---------------------------------------------------------|
|            042000           |              0            |
|---------------------------------------------------------|
|                   More Compiler Name                    |
|---------------------------------------------------------|
|                   More Compiler Name                    |
|---------------------------------------------------------|
                             .
                             .
                             .
```

```
|-------------------------------------------------------|
|          041000           |              0            |
|-------------------------------------------------------|
|                 Compile Date and Time                 |
|-------------------------------------------------------|
|               Compiler Version Number                 |
|-------------------------------------------------------|
|          040400           |              0            |
|-------------------------------------------------------|
|                      Device Name                      |
|-------------------------------------------------------|
|                        UFD                            |
|-------------------------------------------------------|
|          040200            |             0            |
|-------------------------------------------------------|
|                      File Name                        |
|-------------------------------------------------------|
|          File Ext          |                          |
|-------------------------------------------------------|
|          040100            |             0            |
|-------------------------------------------------------|
|                        SFD1                           |
|-------------------------------------------------------|
|                        SFD2                           |
|-------------------------------------------------------|
                              .
                              .
                              .
|-------------------------------------------------------|
|           040040           |             0            |
|-------------------------------------------------------|
|               Source Version Number                   |
|-------------------------------------------------------|
|       Date                 |              Time        |
|=======================================================|
```

Block type 1003 is used to declare long Title symbols in SIXBIT and to furnish other information about the source module. This Block Type is printed in the map file that is produced by LINK.

For TOPS-20, the UFD and SFD words are 0, and the file extension is replaced by the file type.

NOTE

> For the compiler code and the CPU code, refer to the explanation of Block Type 6, where these codes are listed.

Block Type 1004 (Byte Initialization)

```
|=======================================================|
|            1004            |        Short Count       |
|-------------------------------------------------------|
|                   Relocation Word                     |
|-------------------------------------------------------|
|                     Byte Count                        |
|-------------------------------------------------------|
|                    Byte Pointer                       |
|-------------------------------------------------------|
|                    Byte String                        |
|-------------------------------------------------------|
                             .
                             .
                             .
```

Block Type 1004 is used to move a character string into static storage.

The byte count is the number of bytes in the string. The byte pointer is relocated and used to initialize a string in the user's program.

A second form of Type 1004 block is formatted as follows:

```
|=======================================================|
|            1004            |        Short Count       |
|-------------------------------------------------------|
|                   Relocation Word                     |
|-------------------------------------------------------|
|                    Global Symbol                      |
|-------------------------------------------------------|
|                     Byte Count                        |
|-------------------------------------------------------|
|                    Byte Pointer                       |
|-------------------------------------------------------|
|                    Byte String                        |
|-------------------------------------------------------|
                             .
                             .
                             .
```

In this form, the global symbol (in Radix-50 format) is used to relocate the byte pointer, which, consequently, becomes an offset to the static storage. The symbol must be defined when this REL block is encountered.

Block Types 1010 - 1037 (Code Blocks)

Block types 1010 through 1037 are similar in function to blocks of Type 1. They contain code and data to be loaded. These blocks also contain relocation bytes that permit inclusion of PSECT indexes local to the module. For PSECTed programs with many inter-PSECT references this permits a substantial decrease in the size of the REL files. The number of PSECTs that can be encoded in this manner is limited by the size of the relocation byte. A set of parallel code blocks differing only in the size of the relocation byte permits the compiler or assembler to select the most space efficient representation according to the number of PSECTs referenced in a given load module.

This set of blocks is divided by the type of relocation:

Right Relocation	Block types 1010 - 1017
Left/Right Relocation	Block types 1020 - 1027
Thirty-bit Relocation	Block types 1030 - 1037

Blocks 1010 - 1017 (Right Relocation)

```
|=========================================================|
|            1010             |             N             |
|---------------------------------------------------------|
|     b1    |    b2    |    b3    |   . . .   |     bi     |
|---------------------------------------------------------|
|                   Beginning Address                     |
|---------------------------------------------------------|
|                        Data 1                           |
|---------------------------------------------------------|
|                        Data 2                           |
|---------------------------------------------------------|
                            .
                            .
                            .
|---------------------------------------------------------|
|                       Data (i-1)                        |
|=========================================================|
```

Block Types 1010 - 1017 are identical in function. They differ only in the size and number of relocation bytes. Each relocation byte applies to the right half of the corresponding data word.

N — is the length of the REL block, including all words in the block except the Header word.

b1,b2...bi — are the relocation bytes.

Each relocation byte contains a PSECT index number. A zero byte means no relocation (absolute data). All PSECT index numbers must reference predefined PSECTS.

Size	I-value	Block Type
2	18	1010
3	12	1011
6	6	1012
9	4	1013
18	2	1014

A size of 2 allows 3 PSECTs; a size of 3 allows 7 (2**3-1) PSECTs, etc.

Beginning Addr — is the address where the block of code is to be loaded. This address is relocated with respect to the PSECT in "b1". "b1" also defines the current PSECT. It is not necessary to declare the current PSECT with a block of Type 22.

Data1...Data(i-1) — are the words to be loaded. The right halves of these words are relocated with respect to the various PSECTs that are specified by the corresponding relocation bytes, b2,b3,...bi.

Block Types 1020-1027 (Left/Right Relocation Blocks)

```
|=========================================================|
|           1020              |             N             |
|---------------------------------------------------------|
|   L1  | R1    |    L2  |  R2  |  . . .  |   Li  |   Ri  |
|---------------------------------------------------------|
|                  Beginning Address                      |
|---------------------------------------------------------|
|                        Data 1                           |
|---------------------------------------------------------|
|                        Data 2                           |
|---------------------------------------------------------|
                            .
                            .
                            .
|---------------------------------------------------------|
|                      Data (i-1)                         |
|=========================================================|
```

Block Types 1020 - 1027 are identical in function. They differ only in the size and number of relocation bytes. Each pair of bytes applies to the left and right halves, respectively, of the corresponding data word.

N — is the length of the REL block, including all words except the Header word.

L1,R1 — are the relocation byte pairs for the left and right halves respectively.

Size	I-Value	Block Type
2	9	1020
3	6	1021
6	3	1022
9	2	1023

(Block Types 1024-1027 are reserved)

Polish blocks must be used to do left relocation if there are more than (2**9)-1 (decimal 511) PSECTs local to the module.

Beginning Addr — is the address of the block of code to be loaded. This address is relocated with respect to the PSECT in "R1". "R1" also defines the current PSECT. "L1" must be zero.

Data1,..Data(i-1) — is the block of code to be loaded, whose left and right halves are relocated with respect to the various PSECTs as specified by the corresponding byte pairs. The L2 index relocates the left half of data word 1 and R2 relocates the right half of data word 1. Note that these blocks contain 2 bytes for each data word as compared to one byte for Block Types 1010 - 1017.

Block Types 1030 - 1037 (Thirty-bit Relocation Blocks)

```
|===========================================================|
|            103x              |              N             |
|-----------------------------------------------------------|
|      b1    |    b2    |     . . .     |        bi         |
|-----------------------------------------------------------|
|                    Beginning Address                      |
|-----------------------------------------------------------|
|                          Data 1                           |
|-----------------------------------------------------------|
|                          Data 2                           |
|-----------------------------------------------------------|
                              .
                              .
                              .
|-----------------------------------------------------------|
|                        Data (i-1)                         |
|===========================================================|
```

Block Types 1030 - 1037 are identical in function. They differ only in the size and number of relocation bytes. Each relocation byte applies to the entire 30-bit address field of the corresponding data word.

103x — is the Block Type number

N — is the length of the REL block, including all words in the block except the Header word.

b1,b2..bi — are the relocation bytes.

Each relocation byte contains a PSECT index number. A zero byte means no relocation (absolute data). All PSECT index numbers must reference predefined PSECTS.

Size	I-Value	Block Type	Maximum No. of PSECTs
2	18	1030	3
3	12	1031	7
6	6	1032	63
9	4	1033	511
18	2	1034	More than 511

(Block Types 1035 - 1037 are reserved)

Beginning Addr — is the address where the block of code is to be loaded. This address is relocated with respect to the PSECT in b1. B1 also defines the current PSECT. It is not necessary to declare the current PSECT with a block of Type 22.

Data1...Data(i-1) — are the words to be loaded. The 30-bit address field of these words is relocated with respect to the various PSECTs that are specified by the corresponding relocation bytes, b2,b3,...bi.

Block Type 1042 (Request Load for SFDs)

```
|=========================================================|
|            1042            |        Short Count         |
|---------------------------------------------------------|
|                          Device                         |
|---------------------------------------------------------|
|                     SIXBIT Filename                     |
|---------------------------------------------------------|
|       File Extension       |     Directory Count        |
|---------------------------------------------------------|
|               Project-Programmer Number                 |
|---------------------------------------------------------|
|                          SFD1                           |
|---------------------------------------------------------|
|                          SFD2                           |
|---------------------------------------------------------|
                            .
                            .
                            .
```

Block Type 1042 contains a list of files to be loaded. It is similar to blocks of Type 16, but it supplies TOPS-10 sub-file directories for the files being requested. The first three data words (device, file name, and extension) are required. The right half of the third word (directory count) specifies the number of directory levels that are included. For example, the directory [27,5434,SFD1,SFD2] would have a directory count of 3.

LINK saves the specifications for the files to be loaded, discarding duplicates. LINK loads all specified files at the end of loading, and immediately before beginning library searches.

Block Type 1043 (Request Library for SFDs)

```
|=========================================================|
|            1043            |         Short Count        |
|---------------------------------------------------------|
|                          Device                         |
|---------------------------------------------------------|
|                      SIXBIT Filename                    |
|---------------------------------------------------------|
|       File Extension       |     Directory Count        |
|---------------------------------------------------------|
|                Project-Programmer Number                |
|---------------------------------------------------------|
|                           SFD1                          |
|---------------------------------------------------------|
|                           SFD2                          |
|---------------------------------------------------------|
                             .
                             .
                             .
```

Block Type 1043 specifies the files to be searched as libraries. It is similar to Type 17 Blocks, except that it provides TOPS-10 sub-file directories. The first three data words (device, file name, and extension) are required. The right half of the third word (directory count) specifies the number of directory levels that are included. For example, the directory [27,5434,SFD1,SFD2] would have a directory count of 3.

The specified files are searched after requested files are loaded, but before user and system libraries are searched.

Block Type 1044 (ALGOL Symbols)

```
|==========================================================|
|            1044             |          Long Count        |
|----------------------------------------------------------|
|                         Data Word                        |
|----------------------------------------------------------|
                               .
                               .
                               .
|----------------------------------------------------------|
|                         Data Word                        |
|==========================================================|
```

Block Type 1044 contains a debugging symbol table for ALGDDT, the ALGOL debugging program.

If an ALGOL main program has been loaded, or if you have used the /SYFILE:ALGOL switch, LINK writes the data words into a SYM file. In addition, if any Type 15 (ALGOL OWN) REL blocks have been seen, LINK stores the file specification of the file into the first OWN block loaded.

NOTE

> If you have specified the /NOSYMBOLS switch, or if you have specified the /SYFILE switch with an argument other than ALGOL, then LINK ignores any Type 1044 blocks found.

Block Type 1045 (Writable Links)

```
|=========================================================|
|              1045          |            Count             |
|---------------------------------------------------------|
|                          Flags                           |
|---------------------------------------------------------|
|                          Symbol                          |
|---------------------------------------------------------|
|                          Symbol                          |
|---------------------------------------------------------|
                             .
                             .
                             .
|---------------------------------------------------------|
|                          Symbol                          |
|=========================================================|
```

Block type 1045 declares as writable either the link containing the current module or the links containing the definitions of the specified symbols or both. This block type must follow any common block declarations (Types 20 or 6) in a module.

The flag word indicates which links are writable. If bit one is set then the link containing the current module and the links containing the definitions of the specified symbols are writable. If bit one of the flag word is not set then the link containing the current module is not writable, but the links containing the specified symbols are writable. All unused flag bits are reserved and should be zero.

Any symbols specified in a block of Type 1045 must be defined in the path of links leading from the root link to the current link. A module cannot declare a parallel or inferior link to be writable.

If the symbol name contains six or fewer characters it is represented in a single word, left justified, with the following format:

```
|=========================================================|
|                    SIXBIT Symbol Name                    |
|=========================================================|
```

If the symbol name contains more than six characters it is represented in the following format:

```
Bits
0              5 6                  29  30              35
|=========================================================|
|         0     |     Reserved (0)     |         N       |
|---------------------------------------------------------|
|             Word 1 of SIXBIT Symbol Name                |
|---------------------------------------------------------|
|             Word 2 of SIXBIT Symbol Name                |
|---------------------------------------------------------|
                             .
                             .
                             .
|---------------------------------------------------------|
|           Word (N-1) of SIXBIT Symbol Name              |
|=========================================================|
```

The first six bits of a long symbol are always 0. This distinguishes a long symbol name from a single word symbol name. N is the length of the symbol name including the header word. The remaining words contain the symbol name in SIXBIT, six characters to a word, left justified.

Block Type 1050 (PSECT Name Block)

```
|=======================================================|
|             1050            |          Count           |
|-------------------------------------------------------|
|              0              |          Index           |
|-------------------------------------------------------|
|                  SIXBIT  PSECT  Name                   |
|-------------------------------------------------------|
|                 Additional Name Word                   |
|-------------------------------------------------------|
|                 Additional Name Word                   |
|-------------------------------------------------------|
```

Block Type 1050 creates a PSECT with the given name, if none currently exists. It also assigns a unique index number to the PSECT. This index is binding only in the current module.

At least one block type 1050 is required for each PSECT being loaded, and this block must be loaded prior to any other blocks that reference its PSECT (that is, use the unique index number).

Field	Function
Index	contains the 18-bit PSECT index number. It is unique throughout the module.
SIXBIT PSECT Name	the name of the PSECT being defined (in SIXBIT).
Additional Name Word	contains additional characters of the PSECT name.

Block Type 1051 (PSECT Attribute)

```
|========================================================|
|            1051             |           Count          |
|--------------------------------------------------------|
|              0              |           Index          |
|--------------------------------------------------------|
|                        Attribute                       |
|--------------------------------------------------------|
|                          Origin                        |
|========================================================|
```

Blocks Type 1051 assigns attributes to a PSECT and specifies the PSECT's origin address. The attributes that can be assigned are:

Bit	Interpretation	MACRO .PSECT Keyword
11	PSECT is all within section one. This is the default.	
12	PSECT is all in a nonzero section.	
13	PSECT is page-aligned.	PALIGNED
14	Concatenate parts of PSECTs seen in distinct modules.	CONCATENATE
15	Overlay parts of PSECTs seen in distinct modules.	OVERLAY
16	Read-only	RONLY
17	Read and write	RWRITE

Block Type 1052 (PSECT End)

```
|=======================================================|
|           1052           |             N              |
|-------------------------------------------------------|
|             0            |   PSECT Index Number       |
|-------------------------------------------------------|
|                      PSECT Break                      |
|-------------------------------------------------------|
                           .
                           .
                           .
|-------------------------------------------------------|
|             0            |   PSECT Index Number       |
|-------------------------------------------------------|
|                      PSECT Break                      |
|=======================================================|
```

Block Type 1052 allocates additional space for a given PSECT. This space is located between the last address in the PSECT containing data and the address given by the PSECT break. A block of Type 1052 can contain more than one pair of PSECT indexes and breaks.

A module must contain a block of Type 1050 (PSECT Name Block) with the given PSECT index before a block of Type 1052 is generated. If a given PSECT has more than one block 1052 in a single module, the block with the largest break address is used.

Block Type 1066 (Trace Block Data)

```
|========================================================|
|           1060           |         Long Count          |
|--------------------------------------------------------|
|                   SIXBIT Edit Name                     |
|--------------------------------------------------------|
|        Active Code       |        Last Changer         |
|--------------------------------------------------------|
|       Creator Code       |     15-Bit Date Created     |
|--------------------------------------------------------|
|      Installer Code      |    15-Bit Date Installed    |
|--------------------------------------------------------|
|                       Reserved                         |
|--------------------------------------------------------|
|        Edit Count        |       PCO Group Count       |
|--------------------------------------------------------|
|                                                        |
\                                                        /
            Associated Edit Names And Codes
/                                                        \
|                                                        |
|--------------------------------------------------------|
|                                                        |
\                                                        /
               Program Change Order Groups
/                                                        \
|                                                        |
|========================================================|
```

Block Type 1060 contains data used by the MAKLIB program. LINK ignores this block type.

Block Type 1070 (Long Symbol Names)

```
|=====================================================|
|              1070           |      Block Length       |
|-----------------------------------------------------|
|   Code  |  0  |  N  |  R  |           0             |
|-----------------------------------------------------|
|         (Left PSECT #)      |     (Right PSECT #)     |
|-----------------------------------------------------|
|                          Value                        |
|-----------------------------------------------------|
|                       SIXBIT Name                     |
|-----------------------------------------------------|
|                 Additional Name Words                 |
|-----------------------------------------------------|
                            .
                            .
                            .
|-----------------------------------------------------|
|                 Additional Value Words                |
|-----------------------------------------------------|
```

This block defines either a long global symbol or a long local symbol. A symbol defined with this block:

- is output to the DDT symbol table.
- is output to LINK MAP if requested.
- has its value relocated as specified.
- has global requests resolved.

The Long Symbol Name Block is divided into two sections, the basic and the extension sections.

The basic section consists of three words. The first word contains the flags that provide information about the type of symbol, the length of the symbol name, and the relocation type. The second word contains the value. The third word contains the symbol name. If the name or the value cannot fit in a single word, the block contains an extension section that consists of as many words as are necessary to accommodate the symbol name and the value. The length of the symbol name is stored in the flag word and determines how many words are allocated for the long symbol name in the extension section. In the case of a short symbol name only the basic section is used.

The next entry or entries repeat the block, starting at the Flag Word (word 1).

Field	Bits	Description
Header Word		
Rel-block Type	0-17	1070
Block Length	18-35	Number of words used in this block
Flag Word (Word 1)		
Code	0-8	A nine-bit code field:
		bit 0 Must Be Zero
		000 Program name
		100 Local symbol definitions
		101 Extended value
		110 Suppressed to DDT
		120 MAP only
		200 Global symbols completely defined by one word
		201 Completely defined by extended value
		202 Not defined
		203 Right fixup
		204 Left fixup
		205 Right and left fixups
		206 30-bit fixup
		207 Fullword fixup
		210-217 Suppress to DDT
		220-227 MAP only
		24x Global symbol request for chain fixup
		240 No fixup
		241 Right half fixup
		242 Left half fixup
		243 (Not defined)
		244 30-bit fixup
		245 Fullword fixup
		25x Global request for additive fixups (the value of x has the same meaning as in 24x)
		26x Global request for symbol fixups (the value of x has the same meaning as in 24x)
		300 Block names

NOTE

All symbols that require a fixup for their definition must have the fixup block immediately following the entry.

Field	Bits	Description
(Unused field)	9-10	Must Be Zero
N--Name length	11-17	If not zero, extended name field of length n words is used, so that the name occupies N+1 words.

R--Relocation Type	18-21	4-bit relocation type field.
		bit 18=0 relocate with respect to the current PSECT. No PSECT numbers are needed.
		bit 18=1 relocate with respect to the PSECT specified in the next word.
		R=0 Absolute
		R=1 Right half
		R=2 Left half
		R=3 Both halves
		R=4 30-bit
		R=5 Fullword
		R=6 Indirect

(Unused field)	22-35	Not used
PSECT numbers (optional)		Exists only if bit 18=1 in the Flag word. Contains Left and Right PESECT numbers. Bit 18 and bit 0 of this word are zeros.
Value Word (word 2)		Contains the symbol value, it may be relocated as specified by the relocation type and the PSECT numbers provided.
Name (word 3)		Contains the symbol name in SIXBIT.
Additional name field		Optional. It exists only if N > 0. It contains the additional characters when a long symbol name is used.
Additional value field		Optional. It exist only if the code equals xxl. The first word contains the length of the extended field.

NOTES

1. Only one fixup by a Type 2, 10, 11, 1070, or 1072 Block is allowed for a given field. (There can be separate fixups for the left and right halves of the same word.)

2. Fixups are not necessarily performed in the order LINK finds them.

Block Type 1072 (Long Polish Block)

```
|=========================================================|
|          1072          |          Block Length          |
|---------------------------------------------------------|
|                Half-Word Polish String                  |
|---------------------------------------------------------|
                            .
                            .
                            .
|=========================================================|
```

Long Polish Blocks of type 1072 define Polish fixups for operations on relocatable long external symbols. This type block is interpreted as a string of 18-bit operators and operands. The block is in Polish prefix format, with the stored operator at the end of the block. Each halfword can contain one of he following:

- A halfword code in which the first 9 bits contain the data length (when applicable) and the second 9 bits contain the basic code telling LINK how to interpret the data that follows. The largest that can be encoded is 377.
- A halfword data or a part of a larger data packet to be interpreted by LINK as indicated by the code that immediately precedes it.
- A PSECT index of the format 400000+N. The PSECT index field of a long Polish block is handled by LINK the same way that a Polish block type 11 is handled. The first PSECT index that is found sets up the current relocation counter
- A Polish operator.

NOTE

Operands are performed in the order in which they are encountered.

CODE DEFINITIONS

Category	Code	Description
Operand	xxxyyy	next "xxx" halfwords contain data of type "yyy"
	000000	halfword - absolute
	001000	fullword - absolute
	000001	halfword - relocatable
	001001	fullword - relocatable
	000010	fullword symbol name in Radix-50
	xxx010	xxx halfwords of symbol name in SIXBIT (xxx <= 377)

Category	Code	Description
Operator		
	000100	Add
	000101	Subtract
	000102	Multiply
	000103	Divide
	000104	Logical AND
	000105	Logical OR
	000106	Logical shift
	000107	Logical XOR
	000110	One's complement (not)
	000111	Two's complement (negative)
	000112	Count leading zeros
	000113	Remainder
	000114	Magnitude
	000115	Maximum
	000116	Minimum
	000117	Equal relation
	000120	link
	000121	Defined
	000122-00177	Reserved
Store Operator		
	xxx777-xxx770	Chained fixup Next xxx+1 halfwords contain start address of a: (xxx+1 halfwords)
	xxx777	Right half chain fixup
	xxx776	Left half chain fixup
	xxx775	30-bit chain fixup
	xxx774	Fullword chain fixup
	xxx773	Indirect fixup
	000767-000760	Chained fixup with absolute address. Next halfword contains an absolute address.
	001767-001760	Chained fixup with absolute fullword address. Next two halfwords contain absolute address.
	xxx757-xxx750	Symbol fixup. For xxx=0 next two halfwords contain a Radix-50 symbol to be fixed-up. For 1<=xxx<=377 the next xxx+1 halfwords contain a SIXBIT symbol name to be fixed-up
	xxx757	Right half symbol fixup
	xxx756	Left half symbol fixup
	xxx755	30-bit symbol fixup
	xxx754	Fullword symbol fixup
	xxx747-xxx700	Not defined
PSECT index	4000000+N	PSECT index for PSECT N.

NOTES

1. Only one fixup by a Type 2, 10, 11, 1070, or 1072 Block is allowed for a given field. (There can be separate fixups for the left and right halves of the same word.)

2. Fixups are not necessarily performed in the order LINK finds them.

Block Type 1100 - 1107 (Program Data Vector)

```
|=========================================================|
|        1100 - 1107          |            Count          |
|---------------------------------------------------------|
|                  Relocation Byte Word                   |
|---------------------------------------------------------|
|                        Address                          |
|---------------------------------------------------------|
|                  Relocation Byte Word                   |
|---------------------------------------------------------|
|                        Address                          |
|---------------------------------------------------------|
                            .
                            .
                            .
```

Block Types 1100 - 1107 are used to declare the location of program data vectors to LINK. The relocation byte word uses the n-bit byte relocation format. This format permits compact representation of PSECT relocation and thirty-bit address resolution. For these program data vector blocks all nonzero PSECT indexes result in thirty-bit fixups.

Block types 1120-1127 (Argument Descriptor Blocks)

```
|===========================================================|
|       1120 - 1127         |            Count               |
|-----------------------------------------------------------|
|             N-Bit Byte Relocation Information             |
|-----------------------------------------------------------|
|                Argument Block Address  or 0               |
|-----------------------------------------------------------|
|                Associated Call Address or 0               |
|-----------------------------------------------------------|
|                   Loading Address or 0                    |
|-----------------------------------------------------------|
|             Length of Function Name (in bytes)            |
|-----------------------------------------------------------|
|                   Function Name (ASCIZ)                   |
|-----------------------------------------------------------|
                              .
                              .
                              .
|-----------------------------------------------------------|
|      Flag Bits             |       Argument Count         |
|-----------------------------------------------------------|
|       First Argument's Primary Descriptor                 |
|-----------------------------------------------------------|
|       First Argument's Secondary Descriptor               |
|-----------------------------------------------------------|
|       Second Argument's Primary Descriptor                |
|-----------------------------------------------------------|
|       Second Argument's Secondary Descriptor              |
|-----------------------------------------------------------|
                              .
                              .
                              .
|-----------------------------------------------------------|
|       nth Argument's Primary Descriptor                   |
|-----------------------------------------------------------|
|       nth Argument's Secondary Descriptor                 |
|===========================================================|
```

A block of this type is generated for the argument list to each subroutine call. The subroutine entry point also specifies one block with this format, though for the callee the argument block address is zero. If a descriptor block is associated with an argument list it must always follow the loading of the argument list.

The associated call address is used by LINK in diagnostic error messages and its value is determined by the compiler. The argument block address is nonzero if the descriptor block is associated with a call. In this case the argument block address points to the base of the argument block.

The argument block address, associated call address and the loading address are all relocatable.

The argument descriptors in these type blocks describe the properties of each formal (in the case of an entry point) or actual (in the case of a call). In either case the name of the associated routine is specified as a byte count followed by an ASCIZ string. Each primary description is optionally followed by a secondary descriptor.

There are five flag bits in the Descriptor Block:

Bit	Usage
0	If bit 0 is 1 then a difference between the actual number of arguments and the expected number of arguments is flagged as a warning at load time. If bit 0 is 0 no action is taken.
1	If bit 1 is 1 then the block is associated with a function call. If bit 1 is 0 then the block is associated with the function definition.
2	If bit 2 is 1 then the descriptor block is loaded into the user image at the loading address.
3	If bit 3 is 1 then the callee returns a value and the value's descriptor is the last descriptor specified.
4	If bit 4 is 1, and the caller expects a return value, which is not provided by the called function, or if the called function unexpectedly returns a value, then LINK will issue an error. The severity of the error is controlled by the coercion block.

The format for the argument descriptors is as follows:

Bit	Usage
0	(Reserved)
1	No update. In a caller block the argument is a literal, constant, or expression. In a callee block the argument won't be modified.
2-4	Passing mechanism 000 - pass by address 001 - pass by descriptor 010 - pass immediate value Others - reserved
5	Compile-time constant
6-11	Argument type code (see below)
12-17	(Reserved)
18	Implicit argument descriptor
19-26	(Reserved)
27-35	Number of secondary descriptors

The argument type codes are as follows:

Type-Code	Usage
0	Don't care
1	FORTRAN logical
2	Integer
3	(Reserved)
4	Real
5	(Reserved)
6	36-bit string
7	Alternate return (label)
10	Double real
11	Double integer
12	Double octal
13	G-floating real
14	Complex
15	COBOL format byte string descriptor (for constant strings), or FORTRAN character
16	BASIC shared string descriptor
17	ASCIZ string
20	Seven-bit ASCII string

Secondary descriptors are used to convey information about the length of a data object passed as an argument and (in the case of the callee's argument descriptor block) whether or not a mismatched length is permissible. Secondary descriptors have the following format:

Bit Pos	Usage
0-2	(For callee only) Defines the permissable relationships between formal and actual lengths. The values are: 000 - Any relationships are allowed 001 - Lengths must be equal 010 - Actual < formal 011 - Actual <= formal 100 - Actual > formal 101 - Actual >= formal 110 - Reserved 111 - Reserved
3-5	Length of argument (in words)

Block Type 1130 (Coercion Block)

```
|==========================================================|
|            1130            |            Count            |
|----------------------------------------------------------|
|         Field Code         |            Action           |
|----------------------------------------------------------|
|     Formal Attribute       |      Actual Attribute       |
|----------------------------------------------------------|
|         Field Code         |            Action           |
|----------------------------------------------------------|
|     Formal Attribute       |      Actual Attribute       |
|----------------------------------------------------------|
                             .
                             .
                             .
|----------------------------------------------------------|
|         Field Code         |            Action           |
|----------------------------------------------------------|
|     Formal Attribute       |      Actual Attribute       |
|==========================================================|
```

Block Type 1130 specifies which data type associations are permissible and what action LINK should take if an illegal type association is attempted. It may also specify actions to be taken by LINK to modify an actual parameter.

The Coercion Block must be placed before any instance of the caller/callee descriptor block in the REL file. If more than one coercion block is seen during a load, the union of the blocks is used for type checking. However, when different actions are specified for the same type association, the more severe action is used.

When a caller's argument descriptor block is compared to the descriptor block provided by the callee, LINK first checks bit 0 and the argument counts of the descriptor block. If bit 0 is set and the argument counts differ, a warning is given.

Next LINK compares the argument descriptors. The particular formal/actual pair is looked up in the internal table LINK builds using the information in the coercion block. The item field code designates which field of the argument descriptor is being checked. The field codes are defined as follows:

Field Code	**Condition**
0	Check update
1	Check passing mechanism
2	Check argument type code
3	Check if compile-time constant
4	Check number of arguments
5	Check for return value
6	Check length of argument

If the fields of the formal/actual pair do not match, LINK searches the internal table set up by the coercion block. If the table does not specify an action to take in the event of such a mismatch, LINK issues an informational message. If the formal/actual pair differs in more than one field then LINK takes the most severe action specified for the mismatches.

If an actual/formal pair differ and no coercion block has been seen, LINK ignores the difference. If the caller has specified a descriptor block but the subroutine has not, or if the subroutine has specified a descriptor and the caller has not, LINK does not flag the condition as an error and does not take any special action.

If LINK finds an entry in its internal table for a particular actual/formal mismatch, it uses the action code found in the entry to select one of the following five possible responses:

Code (18 Bits)	Action
0	Informational message
1	Warning
2	Error
3	Reserved for the specific conversion of static descriptor pointers (in the argument list) into addresses. The descriptor pointers are supplied by FORTRAN blocks of types 112x. NOTE The actual conversion process involves the following actions: • If byte descriptor's P field is not word-aligned, issued a warning and continue. • Pick up word address of start of string. • If the string is not in the same section as the argument block, nonfatal error and continue. • Put the address of the string into the associated argument block in place of the address of the string descriptor.
4	Suppress the message.
5-777776	Reserved
777777	Fatal error

These messages can be displayed or suppressed. Refer to the descriptions of the /ERRORLEVEL and /LOGLEVEL switches.

Block Type 1140 (PL/I debugger information)

```
|=====================================================|
|            1140           |          Long count     |
|-----------------------------------------------------|
|                      Data Word                      |
|-----------------------------------------------------|
                           .
                           .
                           .
|-----------------------------------------------------|
|                      Data Word                      |
|=====================================================|
```

Block type 1140 is ignored by LINK.

Block Type Greater Than 3777 (ASCIZ)

```
|=======================================================|
|  ASCII  |  ASCII  |  ASCII  |  ASCII  |  ASCII  |  0  |
|-------------------------------------------------------|
|  ASCII  |  ASCII  |  ASCII  |  ASCII  |  ASCII  |  0  |
|-------------------------------------------------------|
                            .
                            .
                            .
|-------------------------------------------------------|
|  ASCII  |  ASCII  |  ASCII  |  ASCII  |    0    |  0  |
|=======================================================|
```

When LINK reads a number larger than 3777 in the left half of a REL Block header word, the block is assumed to contain ASCIZ text. If the module containing the text is being loaded, LINK reads the ASCII characters as if they were a command string, input from the user's terminal.

LINK reads the string as five 7-bit ASCII characters per word; bit 35 of each word is ignored. The string and the block end when the first null ASCII character (000) is found in the fifth 7-bit byte of a word (bits 28-34).

APPENDIX B

LINK MESSAGES

This appendix lists all of LINK's messages. (The messages from the overlay handler, which have the OVL prefix, are given in Chapter 5.)

B.1 DESCRIPTION OF MESSAGES

Section B.2 lists LINK's messages. For each message, the last three letters of the 6-letter code, the level, the severity, and its medium-length message are given in **boldface**. Then, in lightface type, comes the long message.

When a message is issued, the three letters are suffixed to the letters LNK, forming a 6-letter code of the form LNKxxx.

The level of a message determines whether it will be issued to the terminal, the log file, or both. You can use the /ERRORLEVEL and /LOGLEVEL switches to control message output. For some messages an asterisk (*) is given for the level or severity. This means that the value is variable, and depends on the conditions that generated the message.

The severity of a message determines whether the load will be terminated when the message is issued. Table B-1 lists the severity codes used in LINK, along with their meanings. The /SEVERITY switch provides a means for lowering the severity that is considered fatal.

The severity also determines the first character on the message line printed to the terminal. This character can then be detected by the batch system. For all informational messages, the character is [. Warnings use %, and fatal errors use ?.

Table B-1
Severity Codes

Code	Meaning
1-7	Informational; messages of this severity generally indicate LINK's progress through the load.
8-15	Warning; LINK is able to recover by itself and continue the load.
16	Warning if timesharing, but fatal and stops the load if running under batch.
20	Fatal; LINK can only partially recover and continue the load. The loaded program may be incorrect. Undefined symbols cause this action.
24	This is for file access errors. Under batch, this is fatal and stops the load. Under timesharing, this is a warning, and LINK prompts for the correct file specification if possible.
31	Always fatal; LINK stops the load.

The /VERBOSITY switch determines whether the medium-length and long messages are issued. If you use /VERBOSITY:SHORT, only the 6-letter code, the level, and the severity are issued. If you use /VERBOSITY:MEDIUM, the medium-length message is also issued. If you use /VERBOSITY:LONG, the code, level, severity, medium-length message, and long message are issued.

Those portions of the medium-length messages enclosed in braces ({ and }) are optional, and are only printed in appropriate circumstances.

Those portions of the medium-length messages enclosed in square brackets are filled in at run-time with values pertinent to the particular error. Table B-2 describes each of these bracketed quantities.

Table B-2
Special Message Segments

[area]	The name of one of LINK's internal memory management areas.
[date]	The date when LINK is running.
[decimal]	A decimal number, such as a node number.
[device]	A device name.
[file]	A file specification.
[label]	An internal label in LINK.
[memory]	A memory size, such as 17P.
[name]	The name of the loaded program or a node in an overlaid program.
[octal]	An octal number, such as a symbol value.
[reason]	The reason for a file access failure, one of the messages shown in Section B.3.
[switch]	The name of a switch associated with the error.
[symbol]	The name of a symbol, such as a subroutine or common block name.
[type]	The type or attributes associated with a symbol.

Whenever possible, LINK attempts to indicate the module and file associated with an error. This information represents the module currently being processed by LINK, and may not always be the actual module containing the error. For instance, if LINK detects a multiply-defined symbol, either value may be the incorrect one. In this case, LINK reports only the second (and subsequent) redefinition and the module containing it.

B.2 LIST OF MESSAGES

Code	Lev	Sev	Message
ABT	31	31	**Load aborted due to %LNKTMA errors, max. /ARSIZE: needed was [decimal]** You loaded programs containing more ambiguous subroutine requests than can fit in the tables of one or more overlay links. You received a LNKARL message for each ambiguous request, and a LNKTMA message for each link with too many requests. You can solve this problem by using the /ARSIZE switch just before each /LINK switch to expand the tables separately.
AIC	31	31	**Attempt to increase size of {blank common} {common [symbol]} from [decimal] to [decimal] {Detected in module [symbol] from file [file]}** FORTRAN common areas cannot be expanded once defined. Either load the module with the largest definition first, or use the /COMMON: switch to reserve the needed space.
AMM	†	†	**Argument mismatch in argument [decimal] in call to routine [symbol] called from module [symbol] at location [octal]** The caller supplied argument does not match the argument expected by the callee.
AMP	8	8	**ALGOL main program not loaded** You loaded ALGOL procedures, but no main program. The missing start address and undefined symbols will cause termination of execution.
ANM	31	31	**Address not in memory** LINK expected a particular user address to be in memory, but it is not there. This is an internal LINK error. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
ARL	8	8	**Ambiguous request in link [decimal] {name [name]} for [symbol] defined in links [decimal], [decimal], ...** More than one successor link can satisfy a call from a predecessor link. The predecessor link requested an entry point that is contained in two or more of its successors. You should revise your overlay structure to remove the ambiguity.

† The level and severity of this message is determined by compiler-generated coercion block. See Block Type 1130 in Appendix A.

Code	Lev	Sev	Message
			If you execute the current load, one of the following will occur when the ambiguous call is executed: • If only one module satisfying the request is in memory, that module will be called. • If two or more modules satisfying the request are in memory, the one with the most links in memory will be called. • If no modules satisfying the request are in memory, the one with the most links in memory will be called. If a module cannot be selected by the methods 2 or 3 above, an arbitrarily selected module will be called.
AZW	31	31	**Allocating zero words** LINK's memory manager was called with a request for 0 words. This is an internal LINK error. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
CCD	31	31	**CPU conflict** **{Detected in module [symbol] from file [file]}** You have loaded modules compiled with conflicting CPU specifications, such as loading a MACRO program compiled with the statement .DIRECTIVE KL10 and another compiled with .DIRECTIVE KI10. Recompile the affected modules with compatible CPU specifications.
CFS	31	31	**Chained fixups have been suppressed** The specified PSECT grew beyond the address specified in the /LIMIT switch. The program is probably incorrect. Use the /MAP or /COUNTER switch to check for accidental PSECT overlaps. Refer to Section 3.2.2 for more information about the /LIMIT switch.
CLF	1	1	**Closing log file, continuing on file [file]** You have changed the log file specification. The old log file is closed; further log entries are written in the new log file.
CMC	31	31	**Cannot mix COBOL-68 and COBOL-74 compiled code** **{Detected in module [symbol] from file [file]}** You cannot use COBOL-68 and COBOL-74 files in the same load. Compile all COBOL programs with the same compiler and reload.

Code	Lev	Sev	Message
CMF	31	31	**COBOL module must be loaded first [Detected in module [symbol] from file [file]]** You are loading a mixture of COBOL-compiled files and other files. Load one of the COBOL-compiled files first.
CNW	31	31	**Code not yet written at [label]** You attempted to use an unimplemented feature. This is an internal LINK error. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
CRS	1	1	**Creating section [octal]** LINK prints this informational message when a module is loaded into a new section. The message is printed only if you have specified /ERROR:0.
CSF	1	1	**Creating saved file** LINK is generating your executable (.EXE) file.
DEB	31	1	**[name] execution** LINK is beginning program execution at the named debugger.
DLT	31	1	**Execution deleted** Though you have asked for program execution, LINK cannot proceed due to earlier fatal compiler or LINK errors. Your program is left in memory or in an executable file.
DNA	31	31	**DDT not available** A monitor call to obtain DDT failed. This can happen if you have redefined the logical name SYS: and neglected to include any directory that contains DDT.
DNS	8	8	**Device not specified for switch [switch]** You used a device switch (for example, /REWIND, /BACKSPACE), but LINK cannot associate a device with the switch. Neither LINK's default device nor any device you gave with the /DEFAULT switch can apply. Give the device with or before the switch (in the same command line).
DRC	8	8	**Decreasing relocation counter [symbol] from [octal] to [octal] [Detected in module [symbol] from file [file]]** You are using the /SET switch to reduce the value of an already defined relocation counter. Unless you know exactly where each module is loaded, code may be overwritten.

Code	Lev	Sev	Message
DSC	31	31	**Data store to common [symbol] not in link number [decimal]** **{Detected in module [symbol] from file [file]}** You loaded a FORTRAN-compiled module with DATA statement assignments to a common area. The common area is already defined in an ancestor link. Restructure the load so that the DATA statements are loaded in the same link as the common area to which they refer.
DSL	31	*	**Data store to location [octal] not in link number [decimal]** **{Detected in module [symbol] from file [file]}** You have a data store for an absolute location outside the specified link. Load the module into the root link. NOTE If the location is less than 140, this message has level 8 and severity 8.
DUZ	31	31	**Decreasing undefined symbol count below zero** LINK's undefined symbol count has become negative. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
EAS	31	31	**Error creating area AS overflow file [reason] [file]** LINK could not make the ALGOL symbol table on disk. You could be over your disk quota, or the disk could be full or have errors.
ECE	31	31	**Error creating EXE file [reason] [file]** LINK could not write the saved file on disk. You could be over your disk quota, or the disk could be full or have errors.
EHC	31	31	**Error creating area HC overflow file [reason] [file]** LINK could not write your high-segment code on disk. You could be over your disk quota, or the disk could be full or have errors.
EIF	31	31	**Error for input file [file]** A read error has occurred on the input file. Use of the file is terminated and the file is released.

Code	Lev	Sev	Message
ELC	31	31	**Error creating area LC overflow file [reason] [file]** LINK could not write your low-segment code on the disk. You could be over your disk quota, or the disk could be full or have errors.
ELF	1	1	**End of log file** LINK has finished writing your log file. The file is closed.
ELN	1	1	**End of link number [decimal] {name [name]}** The link is loaded.
ELS	31	31	**Error creating area LS overflow file [reason] [file]** LINK could not write your local symbol table on the disk. You could be over your disk quota, or the disk could be full or have errors.
EMS	1	1	**End of MAP segment** The map file is completed and closed.
EOE	31	31	**EXE file output error [file]** LINK could not write the saved file on the disk.
EOI	31	31	**Error on input [file]** An error has been detected while reading the named file.
EOO	31	31	**Error on output [file]** An error has been detected while writing the named file.
EOV	31	31	**Error creating overlay file [reason] [file]** LINK could not write the overlay file on the disk.
ESN	31	31	**Extended symbol not expected** Long symbol names (more than six characters) are not implemented. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
EXS	1	1	**EXIT segment** LINK is in the last stages of loading your program (for example, creating EXE and symbol files, preparing for execution if requested).

Code	Lev	Sev	Message
FCF	1	1	**Final code fixups** LINK is reading one or both segment overflow files backwards to perform any needed code fixups. This may cause considerable disk overhead, but occurs only if your program is too big for memory.
FIN	1	1	**LINK finished** LINK is finished. Control is passed to the monitor, or to the loaded program for execution.
FSI	8	8	**FORTRAN requires FOROTS, /FORSE switch ignored** You gave the /FORSE switch while loading FORTRAN-compiled code. LINK ignored the switch and will use the FORTRAN run-time system.
FSN	31	31	**FUNCT. subroutine not loaded** During final processing of your root link, LINK found that the FUNCT. subroutine was not loaded. This would cause an infinite recursion if your program were executed. The FUNCT. subroutine is requested by the overlay handler, and is usually loaded from a default system library. Either you prevented searching of system libraries, or you did not load a main program from an overlay-supporting compiler into the root link.
FTH	15	15	**Fullword value <name> truncated to halfword** This message is printed when a symbol that has a value greater than 777777 is used to resolve a halfword reference. This warning message helps you to be sure that global addresses are used properly throughout the modules in a load.
GFE	31	31	**GTJFN% JSYS failed for file [file]** While attempting to run your program from the named file, LINK received an error from the monitor. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
HCL	31	31	**High segment code not allowed in an overlay link {Detected in module [symbol] from file [file]}** You have attempted to load high segment code into an overlay link other than the root. Any high segment code in an overlaid program must be in the root.

Code	Lev	Sev	Message
HSL	31	31	**Attempt to set high segment origin too low {Detected in module [symbol] from file [file]}** You have set the high-segment counter to a page containing low-segment code. Reload, using the /SET:.HIGH.:n switch, or (for MACRO programs) reassemble after changing your TWOSEG pseudo-op.
HTL	31	31	**Symbol hash table too large** Your symbol hash table is larger than the maximum LINK can generate (about 50P). This table size is an assembly parameter. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
IAS	31	31	**Error inputting area as [file]** An error occurred while reading in the ALGOL symbol table.
ICB	8	8	**Invalid chain REL block (type 12) link number [octal] {Detected in module [symbol] from file [file]}** REL block type 12 (Chain), generated by the MACRO pseudo-op .LINK and .LNKEND, must contain a number from 1 to 100 (octal) in its first word. The link word is ignored.
IDM	31	31	**Illegal data mode for device [device]** You specified an illegal combination of device and data mode (for example, terminal and dump mode). Specify a legal device.
IHC	31	31	**Error inputting area HC [file]** An error occurred while reading in your high-segment code.
ILC	31	31	**Error inputting area LC [file]** An error occurred while reading in your low-segment code.
ILS	31	31	**Error inputting area LS [file]** An error occurred while reading in your local symbol table.
IMA	8	8	**Incremental maps not yet available** The INCREMENTAL keyword for the /MAP switch is not implemented. The switch is ignored.
IMI	31	31	**Insufficient memory to initialize LINK** LINK needs mcre memory than is available.

Code	Lev	Sev	Message
IMM	*	1	**[Decimal] included modules missing {from file [file]}** You have requested with the /INCLUDE switch that the named modules (if any) be loaded. Specify files containing these modules.
IOV	31	31	**Input error for overlay file [file]** An error occurred when reading the overlay file.
IPO	31	31	**Invalid Polish operator [octal] {Detected in module [symbol] from file [file]}** You are attempting to load a file containing an invalid REL Block Type 11 (Polish). This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
IPX	31	31	**Invalid PSECT index {for PSECT [symbol]} {Detected in module [symbol] from file [file]}** A REL block contains a reference to a nonexistent PSECT. This error is probably caused by a fault in the language translator used for the program. This error is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
IRB	31	31	**Illegal REL block type [octal] {Detected in module [symbol] from file [file]}** The file is not in the proper binary format. It may have been generated by a translator that LINK does not recognize, or it may be an ASCII or EXE file.
IRC	31	31	**Illegal relocation counter {Detected in module [symbol] from file [file]}** One of the new style 1000+ block types has an illegal relocation counter. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
IRR	8	8	**Illegal request/require block {Detected in module [symbol] from file [file]}** One of the REL block types 1042 or 1043 is in the wrong format. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.

Code	Lev	Sev	Message
ISM	31	31	**Incomplete symbol in store operator in Polish block (type 11)** {Detected in module [symbol] from file [file]} The specified module contains an incorrectly formatted Polish Fixup Block (Type 11). The store operator specifies a symbol fixup, but the block ends before the symbol is fully specified. This error is probably caused by a fault in the language translator used for the program. This error is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
ISN	31	31	**Illegal symbol name [symbol]** **{Detected in module [symbol] from file [file]}** The LINK symbol table routine was called with the blank symbol. This error can be caused by a fault in the language translator used for the program. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
ISP	31	31	**Incorrect symbol pointer** There is an error in the global symbol table. This is an internal LINK error. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
ISS	8	8	**Insufficient space for symbol table after PSECT [symbol] -- table truncated** There is insufficient address space for the symbol table between the named PSECT and the next higher one or the end of the address space. Restructure your PSECT layout to allow sufficient room for the symbol table, or use /UPTO to allow more room.
IST	31	31	**Inconsistency in switch table** LINK has found errors in the switch table passed from the SCAN module. This is an internal error. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.

Code	Lev	Sev	Message
ITB	31	31	**Invalid text in ASCII block from file [file]** LINK has failed to complete the processing of an ASCII text REL block from the named file. This is an internal error. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
IVC	31	31	**Index validation check failed at address [octal]** The range checking of LINK's internal tables and arrays failed. The address given is the point in a LINK segment at which failure occurred. This is an internal error. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
JPB	8	8	**Junk at end of Polish block {Detected in module [symbol] from file [file]}** The specified module contains an incorrectly formatted Polish Fixup Block (Type 11). Either the last unused halfword (if it exists) is nonzero, or there are extra halfwords following all valid data.
LDS	1	1	**LOAD segment** The LINK module LNKLOD is beginning its processing.
LFB	1	1	**LINK log file begun on [date]** LINK is creating your log file as a result of defining the logical name LOG:.
LFC	1	1	**Log file continuation** LINK is continuing your log file as a result of the /LOG switch.
LFI	1	1	**Log file initialization** LINK is beginning your log file as a result of the /LOG switch.
LII	8	1	**Library index inconsistent, continuing** A REL Block Type 14 (Index) for a MAKLIB generated library file is inconsistent. The library is searched, but the index is ignored.

Code	Lev	Sev	Message
LIN	1	1	**LINK initialization** LINK is beginning its processing by initializing its internal tables and variables.
LMM	†	†	**Length mismatch for argument [decimal] in call to routine [symbol] called from module [symbol] at location [octal]** The length of the argument passed by the caller does not match what the called routine expects it to be.
LMN	6	1	**Loading module [symbol] from file [file]** LINK is loading the naned module.
LNA	8	8	**Link name [name] already assigned to link number [decimal]** You used this name for another link. Specify a different name for this link.
LNL	8	8	**Link number [decimal] not loaded** The link with this number has not yet been loaded. The /NODE switch is ignored. If you have used link numbers instead of link names with the /NODE switch, you may have confused the link numbers. To avoid this, use link names.
LNM	31	31	**Link number [decimal] not in memory** LINK cannot find the named link in memory. This is an internal error. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
LNN	8	8	**Link name [name] not assigned** The name you gave with the /NODE switch is not the name of any loaded link. The switch is ignored.
LNS	31	8	**Low segment data base not same size** The length of LINK's low segment differs from the length stored in the current LINK high segment. This occurs if some but not all of LINK's .EXE files have been updated after rebuilding LINK from sources. Update all of LINK's .EXE files.

† The level and severity of this message is determined by a compiler-generated coercion block. See Block Type 1130 in Appendix A.

Code	Lev	Sev	Message
LSM	8	8	**/LINK switch missing while loading link number [decimal] -- assumed** Your use of the /NODE switch shows that you want to begin a new overlay link, but the current link is not yet completely loaded. LINK assumes a /LINK switch immediately preceding the /NODE switch, and loads the link (without a link name).
LSS	31	1	**{No} Library search symbols (entry points) {[symbol] [octal]** The listed symbols and their values (if any) are those that are library search entry points.
MDS	8	8	**Multiply-defined global symbol [symbol] {Detected in module [symbol] from file [file]} Defined value = [octal], this value = [octal]** The named module contains a new definition of an already defined global symbol. The old definition is used. Make the definitions consistent and reload.
MEF	31	31	**Memory expansion failed** LINK cannot expand memory further. All permitted overflows to disk have been tried, but your program is still too large for available memory. A probable cause is a large global symbol table, which cannot be overflowed to disk. It may be necessary to restructure your program, or use overlays, to alleviate this problem.
MOV	1	1	**Moving low segment to expand area [area]** LINK is rearranging its low segment to make more room for the specified area. Area is one of the following: AS ALGOL symbol table BG bound global symbols DY dynamic free memory FX fixup area GS global symbol table HC your high-segment code LC your low-segment code LS local symbol tables RT relocation tables
MPS	1	1	**MAP segment** The LINK module LNKMAP is writing a map file.

Code	Lev	Sev	Message
MPT	31	31	**Mixed PSECT and TWOSEG code in same module {Detected in module [symbol] from file [file]}** This module contains both PSECT code and TWOSEG code. LINK cannot load such a module. Change the source code to use PSECTs .HIGH. and .LOW. as the high and low segments, and remove the TWOSEG or HISEG pseudo-ops.
MRN	1	1	**Multiple regions not yet implemented** The REGION keyword for the /OVERLAY switch is not implemented. The argument is ignored.
MSN	8	8	**Map sorting not yet implemented** Alphabetical or numerical sorting of the map file is not implemented. The symbols in the map file appear in the order they are found in the REL files.
NBR	31	31	**Attempt to position to node before the root** The argument you gave for the /NODE switch would indicate a link before the root link. (For example, from a position after the third link in a path, you cannot give /NODE:-4.)
NEB	8	8	**No end block seen {Detected in module [symbol] from file [file]}** No REL Block Type 5 (End) was found in the named module. This will happen if LINK finds two Type 6 blocks (Name) without an intervening end, or if an end-of-file is found before the end block is seen. LINK simulates the missing end block. However, fatal messages usually follow this, because this condition usually indicates a bad REL file.
NED	31	24	**Non-existent device [device]** You gave a device that does not exist on this system. Correct your input files and reload.
NPS	8	8	**Non-existent PSECT [symbol] specified for symbol table** You have specified the name of a PSECT after which LINK should append the symbol table, but no PSECT with that name was loaded. Load the named PSECT or specify an existing PSECT for the symbols.
NSA	31	1	**No start address** Your program does not have a starting address. This can happen if you neglect to load a main program. Program execution, if requested, will be suppressed unless you specified debugger execution.

Code	Lev	Sev	Message
NSM	31	31	**/NODE switch missing after /LINK switch** You used the /LINK switch, which indicates that you want to begin a new overlay link, but you have not specified a /NODE switch to tell LINK where to put the new overlay link.
NSO	31	31	**No store operator in Polish block (type 11) {Detected in module [symbol] from file [file]}** The specified module contains an incorrectly formatted Polish Fixup Block (Type 11). Either the block does not have a store operator, or LINK was not able to detect it due to the block's invalid format. This error is probably caused by a fault in the language translator used for the program. This error is not expected to occur. If it does, please notify your Software Specialist or send, a Software Performance Report (SPR) to DIGITAL.
NVR	†	†	**No value returned by routine [symbol] called from module [symbol] at location [octal]** The called routine does not return a value, however the caller expected a returned value.
OAS	31	31	**Error outputting area as [file]** An error occurred while writing out the ALGOL symbol table.
OEL	8	8	**Output error on log file, file closed, load continuing {[file]}** An error has occurred on the output file. The output file is closed at the end of the last data successfully output.
OEM	8	8	**Output error on map file, file closed, load continuing [file]** An error has occurred on the output file. The output file is closed at the end of the last data successfully output.
OES	8	8	**Output error on symbol file, file closed, load continuing [file]** An error has occurred on the output file. The output file is closed at the end of the last data successfully output.
OFS	31	31	**Overlay file must be created on a file structure** Specify a disk device for the overlay file.

† The level and severity of this message is determined by a compiler-generated coercion block. See Block Type 1130 in Appendix A.

Code	Lev	Sev	Message
OHC	31	31	**Error outputting area HC [file]** An error occurred while writing out your high-segment code.
OHN	31	31	**Overlay handler not loaded** Internal symbols in the overlay handler could not be referenced. If you are using your own overlay handler, this is a user error; if not, it is an internal error and is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
OLC	31	31	**Error outputting area LC [file]** An error occurred while writing out your low-segment code.
OLS	31	31	**Error outputting area LS [file]** An error occurred while writing out your local symbol table.
OMB	31	31	**/OVERLAY switch must be first** The /OVERLAY switch must appear before you can use any of the following switches: /ARSIZE, /LINK, /NODE, /PLOT, /SPACE. (It is sufficient that the /OVERLAY switch appear on the same line as the first of these switches you use.)
ONS	8	1	**Overlays not supported in this version of LINK** LINK handles overlays with its LNKOV1 and LNKOV2 modules. Your installation has substituted dummy versions of these. You should request that your installation rebuild LINK with the real LNKOV1 and LNKOV2 modules.
OOV	31	31	**Output error for overlay file [file]** An error has occurred while writing the overlay file.
OS2	1	1	**Overlay segment phase 2** LINK's module LNKOV2 is writing your overlay file.
OSL	8	8	**Overlaid program symbols must be in low segment** You have specified /SYMSEG:HIGH or /SYMSEG:PSECT when loading an overlay structure. Specify /SYMSEG:LOW or /SYMSEG:DEFAULT.
PAS	1	1	**Area AS overflowing to disk** The load is too large to fit into the allowed memory and the ALGOL symbol table is being moved to disk.

Code	Lev	Sev	Message
PBI	8	8	**Program break [octal] invalid {Detected in module [symbol] from file [file]}** The highest address allocated in the named module is greater than 512P. This is usually caused by dimensioning large arrays. Modify your programs or load list to reduce the size of the load.
PCL	8	8	**Program too complex to load, saving as file [file]** Your program is too complex to load into memory for one of the following reasons: • There are page gaps between PSECTs (except below the high segment). • There are PSECTs above the origin of the high segment. • Your program will not fit in memory along with LINK's final placement code. • One or more PSECTs has the read-only attribute. LINK has saved your program as an .EXE file on disk and cleared your user memory. You can use a GET or RUN command to load the .EXE file.
PCX	8	1	**Program too complex to load and execute, will run from file [file]** Your program is too complex to load into memory for one of the following reasons: • There are page gaps between PSECTs (except below the high segment). • There are PSECTs above the origin of the high segment. • Your program will not fit in memory along with LINK's final placement code. • One or more PSECTs has the read-only attribute. LINK will save your program as an .EXE file on disk and automatically run it, but the .EXE file will not be deleted.
PEF	31	8	**Premature end of file from file [file]** LINK found an end-of-file inside a REL block (that is, the word count for the block extended beyond the end-of-file). This error may be caused by a fault in the language translator used for the program.

Code	Lev	Sev	Message
PEL	15	15	**PSECT [symbol] exceeded limit of [octal]** The specified PSECT grew beyond the address specified in the /LIMIT switch. The program is probably incorrect. Use the /MAP or /COUNTER switch to check for accidental PSECT overlaps. Refer to Section 3.2.2 for more information about the /LIMIT switch.
PHC	1	1	**Area HC overflowing to disk** The load is too large to fit into the allowed memory and your high-segment code is being moved to disk.
PLC	1	1	**Area LC overflowing to disk** The load is too large to fit into the allowed memory and your low-segment code is being moved to disk.
PLS	1	1	**Area LS overflowing to disk** The load is too large to fit into the allowed memory and your local symbol tables are being moved to disk.
PNO	8	8	**Program Data Vectors not allowed in overlay links** Program data vectors cannot be loaded as part of an overlay program. The load continues, but no program data vector will be provided.
PMA	[†]	[†]	**Possible modification of argument [decimal] in call to routine [symbol] called from module [symbol] at location [octal]** The caller has specified that the argument should not be modified. The called routine contains code which may modify this argument. In some cases this message will occur although the argument is not actually modified by the routine.
POT	1	1	**Plotting overlay tree** LINK is creating your overlay tree file.
POV	8	8	**PSECTs [symbol] and [symbol] overlap from address [octal] to [octal]** The named PSECTs overlap each other in the indicated range of addresses. If you do not expect this message, restructure your PSECT origins with the /SET switch.

[†] The level and severity of this message is determined by a compiler-generated coercion block. See Block Type 1130 in Appendix A.

Code	Lev	Sev	Message
PTL	31	31	**Program too long** **{Detected in module [symbol] from file [file]}** Your program extends beyond location 777777, which is the highest location that LINK is capable of loading. You may be able to make your program fit by moving PSECT origins, lowering the high-segment origin, loading into a single segment, reducing the size of arrays in your program, or using the overlay facility.
RBS	31	31	**REL block type [octal] too short** **{Detected in module [symbol] from file [file]}** The REL block is inconsistent. This may be caused by incorrect output from a translator (for example, missing argument for an end block). Recompile the module and reload.
RED	1	1	**Reducing low segment to [memory]** LINK is reclaiming memory by deleting its internal tables.
RER	*	1	**{No} Request external references (inter-link entry points)** **{[symbol] [octal]}** The listed symbols and their values (if any) represent subroutine entry points in the current link.
RGS	1	1	**Rehashing global symbol table from [decimal] to [decimal]** LINK is expanding the global symbol table either to a prime number larger than your /HASHSIZE switch requested, or by about 50 percent. You can speed up future loads of this program by setting /HASHSIZE this large at the beginning of the load.
RLC	31	1	**Reloc ctr. initial value current value** **{[symbol] [octal] [octal]}** The listed symbols and values represent the current placement of PSECTs in your address space.
RUM	31	31	**Returning unavailable memory** LINK attempted to return memory to the memory manager, but the specified memory was not previously allocated. This is an internal error. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.

Code	Lev	Sev	Message
SFU	8	8	**Symbol table fouled up** There are errors in the local symbol table. Loading continues, but any maps you request will not contain control section lengths. This is an internal error. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
SIF	31	31	**Symbol insert failure, non-zero hole found** LINK's hashing algorithms failed; they are trying to write a new symbol over an old one. You may be able to load your files in a different order. This is an internal error. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
SMP	8	8	**SIMULA main program not loaded** You loaded some SIMULA procedures or classes, but no main program. Missing start address and undefined symbols will terminate execution.
SNC	31	31	**Symbol [symbol] already defined, but not as common** **{Detected in module [symbol] from file [file]}** You defined a FORTRAN common area with the same name as a non-common symbol. You must indicate which definition you want. If you want the common definition, load the common area first.
SNL	1	1	**Scanning new command line** LINK is ready to process the next command line.
SNP	8	8	**Subroutine [symbol] in link number [decimal] not on path for call from link number [decimal] {name [name]}** The named subroutine is in a different path than the calling link. Redefine your overlay structure so that the subroutine is in the correct path.
SNS	31	31	**SITGO not supported** **{Detected in module [symbol] from file [file]}** LINK does not support the REL file format produced by the SITGO compiler. Load your program by using SITGO.
SOE	31	31	**Saved file output error [file]** An error occurred in outputting the EXE file.

Code	Lev	Sev	Message
SRB	8	8	**Attempt to set relocation counter [symbol] below initial value of [octal] {Detected in module [symbol] from file [file]}** You cannot use the /SET switch to set the named relocation counter below its initial value. The attempt is ignored.
SRP	31	31	**/SET: switch required for PSECT [symbol] {Detected in module [symbol] from file [file]}** Relocatable PSECTS are not implemented; you must specify an explicit absolute origin with the /SET switch for the named PSECT.
SSN	8	8	**Symbol table sorting not yet implemented** Alphabetical or numerical sorting of the symbol table is not implemented. The symbols appear in the order they are found.
SST	1	1	**Sorting symbol table** LINK is rearranging the symbol table, and if required, is converting the symbols from the new to old format as indicated on the /SYMSEG, /SYFILE, or /DEBUG switch.
STC	1	1	**Symbol table completed** The symbol table has been sorted and moved according to the /SYMSEG, /SYFILE, or /DEBUG switch.
SUP	1	1	**Loading suppressed** Errors occurred during compilation.
T13	31	31	**LVAR REL block (type 13) not implemented {Detected in module [symbol] from file [file]}** REL Block Type 13 (LVAR) is obsolete. Use the MACRO pseudo-op TWOSEG.
TDS	8	8	**Too late to delete initial symbols** LINK has already loaded the initial symbol table. To prevent this loading, place the /NOINITIAL switch before the first file specification.
TMA	31	8	**Too many ambiguous requests in link [decimal] {name [name]}, use /ARSIZE:[decimal] {Detected in module [symbol] from file [file]}** You have more ambiguous subroutine requests (indicated by LNKARL messages) than will fit in the table for this link. Continue loading. Your load will abort at the end with a LNKABT message; if you have loaded all modules, the message will give the size of the needed /ARSIZE switch for a reload.

Code	Lev	Sev	Message
TMM	†	†	**Type mismatch seen for argument [decimal] in call to routine [symbol] called from module [symbol] at location [octal]** The data type of the argument passed by the caller does not match what the called routine expects.
TTF	8	8	**Too many titles found** In producing the index for a map file, LINK found more program names than there are programs. The symbol table is in error. This is an internal error. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.
UAR	8	8	**Undefined assign for [symbol] {Detected in module [symbol] from file [file]}** The named symbol was referenced in a REL Block Type 100 (ASSIGN), but the symbol is undefined. This is generated with the MACRO pseudo-op .ASSIGN. The assignment is ignored. You should load a module that defines the symbol.
UGS	*	1	**{No} Undefined global symbols {[symbol] [octal]** The listed symbols and their values (if any) represent symbols not yet defined by any module. Each value is the first address in a chain of references for the associated symbol. If this message resulted automatically at the end of loading, this is a user error. In this case, the load will continue, leaving references to these symbols unresolved.
UNS	31	31	**Universal file REL block (type 777) not supported from file [file]** Extraction of symbols from a MACRO universal file is not implemented.
URC	31	1	**Unknown Radix-50 symbol code [octal] [symbol] {Detected in module [symbol] from file [file]}** In a REL Block Type 2 (Symbols), the first 4 bits of each word pair contain the Radix-50 symbol code. LINK found one or more invalid codes in the block. This error can be caused by a fault in the language translator used for the program.

† The level and severity of this message is determined by a compiler-generated coercion block. See Block Type 1130 in Appendix A.

Code	Lev	Sev	Message
URV	†	†	**Unexpected return value in call to routine [symbol] called from module [symbol] at location [octal]** The called routine returns a value which was not expected by the caller.
USA	8	8	**Undefined start address [symbol]** You gave an undefined global symbol as the start address. Load a module that defines the symbol.
USC	31	8	**Undefined subroutine [symbol] called from link number [decimal] {name [name]}** The named link contains a call for a subroutine you have not loaded. If the subroutine is required for execution, you must reload, including the required module in the link.
USI	8	16	**Undefined symbol [symbol] illegal in switch [switch]** You have specified an undefined symbol to a switch that can only take a defined symbol or a number. Specify the correct switch value.
UUA	8	8	**Undefined /UPTO: address [symbol]** You gave the named symbol as an argument to the /UPTO switch, but the symbol was never defined. Load a module that defines the symbol, or change your argument to the /UPTO switch.
VAL	31	1	**Symbol [symbol] [octal] [type]** LINK has printed the specified symbol, its value and its attributes as requested.
WNA	†	†	**Wrong number of arguments in call to routine [symbol] called from module [symbol] at location [octal]** The number of arguments in the routine call is not the number of arguments expected by the called routine.
XCT	31	1	**[Name] execution** LINK is beginning execution of your program.
ZSV	8	8	**Zero switch value illegal** You omitted required arguments for a switch (for example, /REQUIRE with no symbols). Respecify the switch.

† The level and severity of this message is determined by a compiler-generated coercion block. See Block Type 1130 in Appendix A.

APPENDIX C

JOB DATA AREA LOCATIONS SET BY LINK

LINK sets a number of locations between 40 and 140 (octal) in the user's program. These locations are known as the Job Data Area (commonly abbreviated to JOBDAT). They are used by the TOPS-10 monitor and by the Compatibility Package (PA1050) on TOPS-20. In addition, two segment programs will have a Vestigial Job Data Area of eight words following the high segment origin.

Job Data Area

Address	Mnemonic	Use
41	.JB41	HALT if not specified otherwise.
42	.JBERR	Number of errors during loading.
74	.JBDDT	Start address of DDT if loaded.
75	.JBHSO	High segment origin.
115	.JBHRL	Left: High segment length. Right: Highest address in high segment.
116	.JBSYM	Left: Negative length of symbol table. Right: Address of table.
117	.JBUSY	Left: Negative length of undefined symbol table. Right: Address of undefined symbol table.
120	.JBSA	Left: First free location in low segment. Right: Start address of program.
121	.JBFF	First free location in low segment.
124	.JBREN	Reenter address of program.
131	.JBOVL	Address of header block for the root link in an overlaid program.
133	.JBCOR	Left: Highest location of low segment loaded with data.
137	.JBVER	Version number: see description of /VERSION switch in Section 3.2.2.

JOB DATA AREA LOCATIONS SET BY LINK

Vestigial Job Data Area

Offset	Mnemonic	Use
0	.JBHSA	Copy of .JBSA.
1	.JBH41	Copy of .JB41.
2	.JBHCR	Copy of .JBCOR.
3	.JBHRH	LH: left half of .JBHRL. RH: right half of .JBREN.
4	.JBHVR	Copy of .JBVER.
5	.JBHNM	Program Name.
6	.JBHSM	High segment symbol table, if any.
7	.JBHGA	High segment origin page in bits 9-17.

INDEX

TOPS–20
LINK Reference Manual
AA–4183C–TM

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- □ Assembly language programmer
- □ Higher-level language programmer
- □ Occasional programmer (experienced)
- □ User with little programming experience
- □ Student programmer
- □ Other (please specify) ____________________

Name ____________________ Date ____________________

Organization ____________________ Telephone ____________________

Street ____________________

City ____________________ State __________ Zip Code __________
or Country

-Do Not Tear - Fold Here and Tape-

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE PUBLICATIONS
200 FOREST STREET MR1–2/L12
MARLBOROUGH, MASSACHUSETTS 01752

Do Not Tear - Fold Here and Tape

Cut Along Dotted Line

TOPS–20
LINK Reference Manual
AD–4183C–T1

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

____

____

____

____

____

____

____

____

____

Did you find errors in this manual? If so, specify the error and the page number.

____

____

____

____

____

____

____

____

____

____

Please indicate the type of reader that you most nearly represent.

- □ Assembly language programmer
- □ Higher-level language programmer
- □ Occasional programmer (experienced)
- □ User with little programming experience
- □ Student programmer
- □ Other (please specify) ____________________________

Name __________________________ Date ________________

Organization ______________________ Telephone ______________

Street ____

City __________________________ State ______ Zip Code ______
or Country

Do Not Tear – Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE PUBLICATIONS
200 FOREST STREET MRO1-2/L12
MARLBOROUGH, MA 01752

Do Not Tear – Fold Here and Tape

Cut Along Dotted Line