TOPS-20

LINK Reference Manual

AA-4183D-TM

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

This document revises the document of the same name, Order No. AA-4183C-TM, published April 1982 and its update, Order No. AD-413C-T1, published March 1983.

OPERATING SYSTEM AND VERSION: TOPS-20, Version 4.1, 5.1

SOFTWARE VERSION: LINK-20, Version 6.0

First Printing, January 1976
Revised, January 1978
April 1982
Updated, March 1983
Revised, May 1985

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 be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

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

The following are trademarks of Digital Equipment Corporation.

DIGITAL	TOPS-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOMM	TOPS-20	TYPESET-11

CONTENTS

FIGURES

TABLES

PREFACE

This manual is the reference document for LINK, the TOPS-20 linking loader.

Document Structure

Chapter 1 provides a general introduction to LINK, a discussion of libraries, library searches, and extended addressing. Chapter 1 also contains a summary of LINK switches.

Chapter 2 describes automatic use of LINK through one of the TOPS-20 DEBUG, EXECUTE, or LOAD commands. This chapter is sufficient for most loading tasks.

Chapter 3 describes running and exiting LINK, LINK's command format, and LINK switches. This chapter contains the alphabetical listing of LINK switches and an example of using LINK directly.

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

Chapter 6 discusses PSECTs (Program SECTions), loading programs with PSECTS, and PSECT attributes.

Chapter 7 discusses PDVs (Program Data Vectors), their format, memory map, and symbol table vector.

Appendix A gives a technical description of REL (RELocatable) Blocks, the main input to LINK.

Appendix B lists all LINK messages except the OVERLAY handler runtime messages that are contained in Chapter 5.

Appendix C describes JOBDAT, the Job Data Area.

A Glossary follows the appendixes.

Reference Material

TOPS-20 User's Guide

TOPS-20 Commands Reference Manual

MACRO Assembler Reference Manual

FORTRAN Language Manual

COBOL-68 Language Manual

COBOL-74 Language Manual

DECsystem-10/DECSYSTEM-20 Processor Reference Manual

TOPS-20 Monitor Calls Reference Manual

Conventions

<ESC>	indicates that you are to enter an ESCAPE key.
<RET>	indicates that you are to enter a carriage-return.
bold italic	is used in reference section of Chapter 3 to show defaults and Digital supported software.

Loading Map

To load most programs,

1. Become familiar with the concepts in the TOPS-20 User's Guide and TOPS-20 Commands Reference Manual.

2. Refer to Chapter 2 for a review of the DEBUG, EXECUTE, and LOAD commands that automatically use LINK to load programs.

To load programs that use overlays,

1. Refer to Chapter 5.

2. Refer to the /ARSIZE, /LINK, /MAXNODE, /NODE, /OVERLAY, /PLOT, and /SPACE switches in Chapter 3.

To load programs that use PSECTs,

1. Refer to Chapter 6.

2. Refer to the /SET and /LIMIT switches in Chapter 3.

To load extended addressing programs,

1. Become familiar with the concepts in

   the DECsystem-10/DECSYSTEM-20 Processor Reference Manual

   the Chapter 8 of the TOPS-20 Monitor Calls User's Guide

   TOPS-20 Monitor Calls Reference Manual
2. Refer to Section 3.7.
3. Refer to Section 6.2 on loading PSECTs.

CHAPTER 1

INTRODUCTION TO LINK

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

This merging process requires LINK to:

1. Convert relocatable addresses to absolute addresses, and bind program segments and PSECTs (Program SECTions) to addresses.

   Relocatable addresses are addresses within a module that are specified as an offset from the first location in that module.

   Absolute addresses are fixed locations in your address space.

   Program segments define the program as a single-segment or two segment program.

   PSECTs are programmer- or system-defined regions.

2. Resolve global symbol references using chained fixups, Polish fixups and library searches.

   Global symbols are defined in one module and used in other modules. LINK resolves the references to global symbols by storing the reference until LINK loads the module that contains the global symbol's definition. After LINK loads that module, it "fixes up" the location in memory where LINK stored the reference.

   Chained fixups are a list of locations that require the global symbols' definition. One location points to the next location that requires the global symbol's definition.

   Polish fixups use an algorithm to find the locations that require the global symbol's definition.

3. Produce an executable program, a JOBDAT area or a PDV (Program Data Vector) and a debugger symbol table.

   An executable program is the memory image of the program.

   A JOBDAT or PDV contain information about the program such as its version number.

   A debugger symbol table contains entries and values for the symbols defined or used in a program.

1.1 INPUT TO LINK

LINK accepts three kinds of input:

- Object modules that are the main input to LINK, and are output by a language translator.
- LINK commands
- Libraries that contain object modules for specific languages and applications.

1.1.1 Object Modules

Object modules contain machine language that corresponds to the source program, and are contained in relocatable binary files. A relocatable binary file can contain many modules or one module, and is formatted into REL (RELocatable) Blocks. LINK recognizes REL Blocks and handles them appropriately. The format of each REL Block Type is described in Appendix A. The default file type for a relocatable binary file is REL, and throughout this manual relocatable binary files are referred to as REL files.

As their name implies, REL files contain relocatable code or addresses. LINK converts relocatable addresses to absolute addresses by loading relocatable code at an arbitrary memory address, and adding a constant to each address referenced in the program.

In the process of converting addresses, LINK binds program segments, and PSECTS to addresses.

A program that uses program segments can be a single-segment or two-segment program. A single segment program uses one relocation counter for LINK to use in the loading process. A two-segment program uses two relocation counters, and is divided into a high-segment or low-segment. The LINK /ONLY, /REDIRECT, /SET, and /SEGMENT switches

can be used to manipulate segments during loading. A relocation counter is an address counter that LINK uses while loading relocatable code.

A program that uses PSECTs can have one or multiple PSECTs. Each PSECT in a program has its own relocation counter. For example, if a program has 5 PSECTs, it also has 5 relocation counters, one for each PSECT. Refer to Chapter 6 for information on PSECTs.

Besides relocating and loading your object modules, LINK resolves values for local, global and entry name symbols. Local symbols are those that are defined and referenced only within a module. Global symbols are those that are defined in one module and used in others. Entry name symbols are special global symbols in a module that contain an entry point name for other modules.

1.1.2 Commands to LINK

LINK commands control and modify the loading process. Commands consist of file specifications and switches. LINK commands are summarized at the end of this chapter and are detailed in Chapter 3.

1.1.3 Libraries And Searches

Libraries are files that contain object modules that may be needed to resolve global symbol references. LINK only loads the object modules that contain entries for the global symbols to be resolved. There are two kinds of libraries: system or user.

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

LINK searches system libraries before it finishes loading a program. LINK determines which system library to load from the object module that contains translator information for the program. LINK performs system library searches when it finds a LINK /GO or /LINK switch. For example, if you load a FORTRAN-compiled module, LINK searches the system FORTRAN library SYS:FORLIB.REL when a /GO or /LINK switch is processed. This search resolves requests for FORTRAN-defined subroutines and functions in FORLIB.REL.

The /SYSLIB and /NOSYSLIB switches allow you to manipulate system library searches.

The /SYSLIB switch requires LINK to search specified system libraries

no matter what kind of modules were loaded. The /NOSYSLIB switch prevents searching of specified system libraries. Using these two switches, you can select the order for searching system libraries.

User libraries are libraries that you create for LINK to search.

The /USERLIB and /NOUSERLIB switches allow you to specify user library searches. /USERLIB specifies that 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 suspends the effect of a /USERLIB switch.

The /SEARCH and /NOSEARCH switches allow you to specify how an input file is to be loaded. /SEARCH specifies that LINK search input files as libraries, and load only those modules that resolve a global symbol. /NOSEARCH specifies that LINK is to load all modules from each input file. /NOSEARCH is the default.

Using combinations of these switches gives you precise control of library searches. See Chapter 3 for information on these switches.

1.2 OUTPUT FROM LINK

LINK outputs an executable program in memory, messages, and special files.

1.2.1 Executable Program

The main output from LINK is an executable program. In this executable program, all addresses are resolved to absolute memory locations, and all symbols (including subroutine calls) are resolved to absolute values or addresses.

The executable program can be executed immediately or saved as a sharable save file. A sharable save file has a default file type of EXE, and is referred to to as an EXE file. LINK automatically creates an EXE file when you use /SAVE with LINK.

You can also execute the executable program under the control of a debugger using the /DEBUG or /TEST switch. LINK outputs a debugger symbol table for this purpose. A debugger symbol table contains the program's symbols and their definitions.

LINK also generates a JOBDAT or a PDV for a program. Both JOBDAT and PDV contain information about the program such as debugger symbol table pointers, version numbers, and memory use. Refer to Appendix C for JOBDAT information and Chapter 7 for PDV information.

1.2.2 Messages

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

1.2.3 Special Files

At your option, LINK can generate several special files:

- a map file that contains information about where the program was loaded, and what symbols are used in the program modules.
- a log file that contains a record of the messages that LINK returned during the linking and loading process.
- a symbol file that contains a symbol table for the program that was loaded.
- a plotter file that contains a tree diagram of an overlay program.
- an overlay file that contains information about the overlay links.

LINK's output files are described in Chapter 4.

1.3 LINK'S OVERLAY FACILITY

If a program is too large to execute in one piece, you can use LINK's overlay facility to define an overlay structure for the program. 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 LINK AND EXTENDED ADDRESSING

The KL Model B processor is capable of using an address space consisting of 32 sections, each containing 512 pages. As of TOPS-20 Version 5, programs have been able to reference this expanded address space. For information on using extended addressing with a programming language, consult the documentation for that language.

Use PSECTs to load a program into a nonzero section. See Section 6.1

for information on loading PSECTs.

When loading a program that uses extended addressing, pay particular attention to the use of 18-bit and 30-bit addresses. If a program uses 30-bit addresses and you reference the 30-bit addresses as a 18-bit address, LINK truncates the 30-bit address and notifies you with the following message:

```
%LNKFTH Fullword value [symbol] truncated to halfword
```

Refer to Appendix B for more information about this message.

LINK issues this warning if the truncation results in the loss of a section number.

While writing an extended addressing program, keep the following restrictions in mind:

- Programs that use overlays cannot use nonzero sections.
- LINK does not set up JOBDAT for a program loaded in any nonzero section. LINK stores information about an extended addressing program in a PDV. PDVs are described in Chapter 7. JOBDAT is described in Appendix C.
- Programs should not store executable code into locations 0 through 17 of nonzero sections. However, you can store data that is not executed in these locations. If data is stored in locations 0 through 17, use global addresses to reference the locations. If local addresses are used, the locations are referenced as ACs (accumulators). In section one, locations 0 through 17 are ACs.

Table 1-1: Summary of LINK Switches

Switch	Description
/ARSIZE	Sets the size of the overlay handler's table of multiply-defined global symbols.
/COMMON	Allocates words of labeled COMMON storage for FORTRAN and FORTRAN-compatible programs.

Table 1-1 (cont.)

Switch	Description
/CONTENTS	Specifies the symbol types to be included in the map file if the file is generated.
/COUNTERS	Displays relocation counters on the terminal.
/DDEBUG	Specifies a default debugger to be loaded if the /DEBUG or /TEST switch appears without an argument.
/DEBUG	Requests loading of a debugger and sets the start address for execution at the start address of the debugger.
/DEFAULT	Changes default specifications for input or output files.
/DEFINE	Assigns a decimal value to a symbol.
/ENTRY	Displays all entry name symbols that have been loaded on the terminal.
/ERRORLEVEL	Suppresses terminal display of LINK messages.
/EXCLUDE	Prevents loading of the specified modules from the current file.
/EXECUTE	Directs LINK to execute the loaded program beginning at the program's start address.
/EXIT	Exits LINK.
/FRECOR	Requires LINK to maintain a minimum amount of free memory after any expansions.

Table 1-1 (cont.)

Switch	Description
/GO	Ends loading after the current file and exits LINK.
/HASHSIZE	Gives a minimum for the initial size of the global symbol table.
/HELP	Displays information about LINK on the terminal.
/INCLUDE	Specifies modules to be loaded regardless of any global requests for them.
/LIMIT	Specifies an upper bound for a PSECT.
/LINK	Closes an overlay link.
/LOCALS	Includes local symbols from a module in the symbol table.
/LOG	Specifies a file specification for the log file.
/LOGLEVEL	Suppresses logging of LINK messages.
/MAP	Specifies a file specification for the map output file.
/MAXNODE	Specifies the number of links to be defined when the overlayed program requires more than 256 links.
/MESSAGE	Displays messages on the terminal in the format specified by keyword.

Table 1-1 (cont.)

Switch	Description
/MISSING	Displays on the terminal the number of modules requested with the /INCLUDE switch that have not yet been loaded.
/NEWPAGE	Sets the relocation counter to the first word of the next page.
/NODE	Opens an overlay link.
/NOENTRY	Deletes entry name symbols from LINK's overhead tables when loading overlays.
/NOINCLUDE	Clears requests for modules that were specified in a previous /INCLUDE.
/NOINITIAL	Prevents loading of LINK's initial global symbol table (JOBDAT).
/NOJOBDAT	Prevents LINK from filling in JOBDAT's or vestigial JOBDAT's address space, and creates a PDV.
/NOLOCAL	Suspends the effect of a preceding /LOCALS switch so that local symbol tables are not loaded with their modules.
/NOREQUEST	Deletes references to overlay links from LINK's overhead tables when loading overlay programs.
/NOSEARCH	Suspends the effect of a previous /SEARCH switch.
/NOSTART	Directs LINK to disregard any start addresses found after the /NOSTART switch.

Table 1-1 (cont.)

Switch	Description
/NOSYMBOL	Prevents construction of user symbol tables.
/NOSYSLIB	Prevents automatic searching of system libraries.
/NOUSERLIB	Discontinues automatic searching of files at each /LINK or /GO switch.
/ONLY	Directs LINK to load the specified segment of two-segment modules.
/OPTION	Reads the SWITCH.INI file to determine switch defaults for LINK.
/OTSEGMENT	Specifies the time and manner of loading the object-time system.
/OVERLAY	Initiates construction of an overlay structure.
/PATCHSIZE	Allocates words of storage to precede the symbol table.
/PLOT	Directs LINK to output a tree diagram of an overlay structure.
/PLTTYP	Specifies the type of plot file to be generated by the /PLOT switch.
/PSCOMMON	Specifies where LINK is to load COMMON blocks.
/PVBLOCK	Requests a PDV (Program Data Vector) from LINK and gives you control over where the vector goes.

Table 1-1 (cont.)

Switch	Description
/PVDATA	Changes the contents of a PDV block.
/REDIRECT	Loads two-segment formatted REL files as part of a program using PSECTs.
/REQUEST	Displays external overlay link references on the terminal.
/REQUIRE	Generates global requests for the specified symbols.
/RUN	Directs LINK to run a program after it is loaded.
/RUNAME	Assigns a job name for execution of your program.
/RUNOFFSET	Runs the program specified in a /RUN switch with an offset.
/SAVE	Directs LINK to create an EXE file.
/SEARCH	Directs LINK to search the input files and load only the modules whose entry point names resolves a global symbol reference.
/SEGMENT	Specifies whether the high segment or the low segment of a two-segment program is to be used for loading the following modules.
/SET	Sets the origin of a PSECT, or sets the .HIGH. or .LOW. relocation counter.
/SEVERITY	Specifies that messages of a certain severity level terminate the load.

Table 1-1 (cont.)

Switch	Description
/SPACE	Specifies that n words of memory follow the current link at execution time.
/START	Specifies the start address for the loaded program.
/SUPPRESS	Suppresses a previously defined global symbol.
/SYFILE	Requests LINK to output a symbol file to the given filespec, and sets the /SYMSEG:DEFAULT switch.
/SYMSEG	Allows you to specify where the symbol table is to be placed.
/SYSLIB	Forces searching of one or more system libraries, immediately after you end the command line.
/TEST	Loads a debugger.
/UNDEFINED	Displays undefined global symbols on the terminal.
/UPTO	Sets an upper limit to which the symbol table can expand.
/USERLIB	Directs LINK to search a user library.
/VALUE	Displays global symbol values on the terminal.
/VERBOSITY	Specifies the length of LINK messages.
/VERSION	Allows you to specify a version number.

CHAPTER 2

USING LINK AUTOMATICALLY

The TOPS-20 LOAD, EXECUTE, and DEBUG commands invoke LINK automatically. Each of these commands uses a simple command string that the system converts 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 are discussed here. For a full discussion, see the TOPS-20 Commands Reference Manual.

Each of these commands invoke LINK:

- The **LOAD** command uses LINK to load your object modules into memory. LOAD compiles source files first if necessary, but does not execute the program.

- 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 loads your program into memory along with a debugger. The program is then executed under the control of the debugger. The debugger that is loaded depends on the type of program being loaded. See the /TEST switch for a list of languages. TOPS-20 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 your programs' files. Standard file types are listed in the TOPS-20 Commands Reference Manual.

2.1 COMMAND FORMATS

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

@command/switches input-filespec/switches, input-filespec/switches,...

Where:

command	is LOAD, EXECUTE, or DEBUG.
input-spec	is the file specification of the program you want to load. This input file specification accepts a 6-character device name, 39-character directory name, a 6-character filename and a 3-character extension.
switches	are any of the valid switches for the command. Table 2-1 contains switches that apply to LINK.

If you separate the input file specifications with commas, each source file is compiled into a separate object (relocatable) file. For example,

```
@LOAD PROGA,PROGB,PROGC
```

compile into separate object modules, PROGA.REL, PROGB.REL and PROGC.REL.

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

```
@LOAD PROGD+PROGE+LIBRAY
```

compile into a single object module, LIBRAY.REL

Section 2.3 shows additional examples of using LINK automatically.

2.2 COMMAND SWITCHES

You can use switches with the LOAD, EXECUTE, and DEBUG commands to control the program'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 TOPS-20 Commands

Switch	Meaning

Table 2-1 (cont.)

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 that is based on the file type of the first file in the command string.
/MAP	Produces a map file at the end of loading.
/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 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 /SYMSEG:HIGH switch to LINK by using the command:

```
@EXECUTE MYPROG %"SYMSEG:HIGH"<RET>
```

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

```
@EXECUTE MYPROG %"SYMSEG:HIGH/COUNTERS"<RET>
```

LINK program switches are described in Chapter 3.

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:

```
@EXE<ESC>CUTE (FROM) MYPROG.FOR<RET>
FORTRAN: MYPROG
MAIN.
LINK: Loading
[LNKXCT MYPROG execution]
This is written by MYPROG
CPU time 0.21   Elapsed time 1.31
```

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

```
@DEBUG MYPROG.FOR<RET>
FORTRAN: MYPROG
MAIN.
LINK: Loading
[LNKDEB FORDDT execution]

STARTING FORTRAN DDT

>>START<RET>
This is written by MYPROG
CPU time 0.17   Elapsed time 0.46
```

CHAPTER 3

USING LINK DIRECTLY

You can choose to use the TOPS-20 LOAD, EXECUTE, and DEBUG commands to load a program, or you can use LINK directly. Using LINK directly is useful when you have a large or complicated program, are loading overlays, or are using PSECTs because you can better control the loading process. For example, in the case of a program that uses PSECTs, you can use LINK switches to specify the origins of the PSECTs. In the case of an overlay program, you can use LINK switches to define the program's tree structure.

Before you load a program with LINK, you must have compiled or assembled all required object modules.

3.1 RUNNING AND EXITING LINK

To run LINK, type LINK after the TOPS-20 system prompt. LINK returns an asterisk (*) and waits for you to enter commands. For example,

```
@LINK<RET>
*
```

To exit LINK, use the /GO switch.

The /GO switch finishes loading the program and exits. /GO exits to TOPS-20 command level, the loaded program for execution, or to a debugger. /GO passes control to the loaded program if you specified /EXECUTE, and to a debugger if you specified /DEBUG. For example, in the following /GO passes control to the loaded program.

```
@LINK<RET>
*MYPROG/EXECUTE/GO
[LNKXCT MYPROG execution]
This is written by MYPROG
CPU time 0.21   Elapsed time 0.82
```

If LINK encounters a fatal error, LINK prints an error message and

exits to TOPS-20.

3.2 COMMAND FORMATS

A LINK command can contain file specifications and switches. The general format of a LINK command is:

```
*/switch input-filespec /switch output-filespec /switch
```

where:

switch — is a LINK switch. LINK switches are described in Section 3.3. Section 3.3 also discusses switch arguments, abbreviations, and placement.

filespec — consists of 1- to 6-character device name, 1- to 6-character filename, a 1- to 3-character file type, and a PPN (project-programmer number).

You only need to supply a filename as LINK supplies the missing parts of the file specifications from its defaults. These defaults are listed in Section 3.2. You can use a logical name to specify a device name, and the TOPS-20 TRANSLATE command to determine a PPN. See Sections 3.2.2 and 3.2.3 for additional information on logical names and TRANSLATE respectively.

You can continue a command, include a comment, or use indirect command files with LINK.

By typing a hyphen before entering a carriage return, you can continue a command onto the next line. LINK continues the line by prompting with a hash sign (#). For example,

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

You can include a comment in a command by beginning the comment with a semicolon. LINK does not process the text after the semicolon. Commenting commands can be useful in indirect command files. For example,

```
/CONTENTS:(LOCALS,UNDEFINED)  ;Include local and undefined
                              ;symbols in my map file.
PROMAP/MAP:ERROR              ;Give me a map file named PROMAP
                              ;when LINK encounters an error.
```

```
PROGAM
/GO                          ;Load my program
```

These commands and comments when placed in an indirect command file provide you with a description of how you want a program loaded.

To have LINK read commands from an indirect command file, prefix the command file specification with an at-sign (@) and enter it after the LINK prompt. For example,

```
@LINK<RET>
*@LNKPRG
```

LINK reads the commands in LNKPRG and processes them. LNKPRG can have a .CCL or .CMD file type. Both are default file types for indirect command files. LINK first looks for a .CCL file type and then a .CMD file type.

3.2.1 LINK File Specification Defaults

For input files, the defaults are:

device	logical name (DSK:)
type	REL
directory	the connected directory. If you are using a PPN, refer to Section 3.3.3 for information on translating directories.

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 in decimal, with any leading zeros.
type	one or all of the following depending on the switches you use.

/EXECUTE	executable file	EXE
/LOG	log file	LOG
/MAP	map file	MAP
/PLOT	plotter file	PLT
/SYFILE	symbol file	SYM

directory	the connected directory. If you are using a PPN, refer to Section 3.3.3 for information on translating directories.

You can change these defaults by using the /DEFAULT switch. See Section 3.3.4.

3.2.2 Logical Names

To use a logical name:

1. Give the TOPS-20 DEFINE command to define a logical name. A logical name cannot be longer than six characters.

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

See the TOPS-20 User's Guide for more information.

3.2.3 Translating Directories

To find a corresponding PPN:

1. Use the TOPS-20 TRANSLATE command to find the corresponding PPN for the given directory name.

2. Include the PPN enclosed in square brackets ([]) at the end of the file specification in the format:

```
device:filename.type[PPN]
```

3.3 LINK SWITCHES

LINK switches control and modify the loading process. This section describes switch abbreviations, arguments, and placements followed by an alphabetical listing of the LINK switches.

3.3.1 Switch Abbreviation

You can abbreviate switches to save typing. However, when abbreviating a switch, you must must include enough characters to make the abbreviation unique for the switch. For example, both /NOSYMBOL and /NOSYSLIB begin with the characters /NOSY, therefore you cannot abbreviate the /NOSYMBOL switch to /NOSY. You can abbreviate /NOSYMBOL to /NOSYM because /NOSYM is a unique set of characters for /NOSYMBOL.

The following switches can be abbreviated to a single letter by

default:

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

Use the full switch name when placing switches in batch or indirect command files to avoid incorrect or ambiguous abbreviations.

3.3.2 Switch Arguments

Many LINK switches require arguments. An argument can be a value, a symbol name, a module name, an output file specification, a PSECT name, a keyword, or an address. The specific requirements of each switch are discussed in the FORMAT section of the switch's description.

A value can be specified in decimal or octal. The default is provided in the FORMAT information. 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 hash sign (#) before the octal number. For example, /ARSIZE:39 can be specified in octal as /ARSIZE:#47.

A symbol name, a module name, or a PSECT name consists of from 1 to 72 SIXBIT-compatible ASCII characters. You cannot embed spaces in any of the names. If the name is not Radix-50 compatible, you must enclose it in double quotes. For example, NOT^RADIX would be specified as "NOT^RADIX" in a LINK command. If you enclose a name in quotes, be aware that how you enter the name (all in UPPERCASE, all in lowercase, or in Mixed cAse) affect the interpretation of the symbol. For example,

```
"DIFFERENT" "different" "Different"
```

all define individual symbols. However, if you do not enclose a name in quotes and enter the name in mixed case, the name defines the same symbol. For example,

```
SAME same Same
```

all define the symbol same and are equivalent to the quoted name

"SAME".

An overlay link name consists of from 1- to 6-Radix-50 characters that are compatible with ASCII characters.

A keyword is a unique argument that is associated with the switch. For example, /DEBUG's keywords identify the debugger to be loaded with the program.

An address is an octal number.

An output file specification specifies a file that LINK is to create. See Section 3.2 for format and default information on output file specifications. The following switches produce an output file:

```
/DEFAULT       /LOG
/MAP           /OVERLAY
/PLOT          /NOUSERLIB
/SAVE          /SYFILE
/USERLIB
```

When specifying output file specifications with these switches, append the switch to the output file specification in the format:

```
outputfilespec/switch
```

For example,

```
*LOGFIL/LOG
```

directs LINK to write LINK messages into the output file LOGFIL.LOG.

The other file specifications in a LINK command specify input files. For example, the following command tells LINK to use an input file called MYREL.REL to generate a saved output file called MYEXE.EXE. /SAVE specifies MYEXE as the output filename, and .EXE is the default file type:

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

3.3.3 Switch Placement

The /LOCALS, /NOLOCAL, /NOSTART, /SEGMENT, /INCLUDE, /ONLY, /SEARCH, /START and /NOSEARCH switches can be local or global switches. A local switch applies to a single file specification and is appended to that file specification in the command. The switch is appended before the comma in a list of file specifications. For example, in the following command, /SEARCH is used as a local switch to load FILE2 in search mode. /SEARCH is appended to FILE2 before the comma.

```
*FILE1,FILE2/SEARCH,FILE3<RET>
```

A global switch applies to all the files in a command and is disabled at the end of the command when you enter a carriage-return. A global switch is not appended to a file specification, and can be placed after the LINK prompt (*), or after a filename. A global switch is appended after the comma in a list of file specifications. For example, in the following commands /SEARCH is a global switch used to load all the files in search mode. In the first command, /SEARCH is placed after the LINK prompt. In the second, note that /SEARCH is placed after the comma following the first file specification.

```
*/SEARCH FILE4,FILE5,FILE6<RET>
*FILE4,/SEARCH FILE5,FILE6<RET>
```

A global switch applies to all the files entered after the asterisk and before the carriage-return, unless it is overridden by another switch. If this second switch is a global switch, it persists for all the files that follow in the command. For example, in the following command, the /NOSEARCH disables the /SEARCH switch for files FILE10 and FILE11.

```
*FILE7,/SEARCH FILE8, FILE9,/NOSEARCH FILE10, FILE11<RET>
```

If the second switch is a local switch, it overrides the first switch only for the file to which it is appended. For example, in the following command string, the global /SEARCH switch is overridden by the local /NOSEARCH switch, but only for FILE2. FILE1 and FILE3 are loaded in search mode.

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

NOTE

When the /GO switch is present in the same command as a global switch, /GO causes the effects of the global switch to go beyond the command and apply to any modules loaded during library searches. For example,

```
*/SEGMENT:LOW FILEA,FILEB,FILEC/60<RET>
```

causes the input files on this line and files loaded during a library search to be loaded in the program's low-segment. However, defaults for certain languages can override the global switches that you specify.

3.3.4 Alphabetical Listing of LINK Switches

In this section the following information is shown, if appropriate, for each switch:

FORMAT
FUNCTION
EXAMPLES
OPTIONAL NOTATIONS
RELATED SWITCHES

/ARSIZE

FORMAT

/ARSIZE:n

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 give instructions for the argument to the /ARSIZE switch.

EXAMPLES

```
@LINK<RET>
*/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.

/COMMON

FORMAT

/COMMON:name:n

name is a common name. The symbol name rules described in Section 3.3.2 apply to common name.

n is 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 a global 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

```
@LINK<RET>
*/COMMON:ALPHA:1000<RET>
*
```

Creates a labeled COMMON area of 1000 words.

```
@LINK<RET>
*/COMMON:.COMM.:1000<RET>
*
```

Creates an unlabeled COMMON area of 1000 words.

```
@LINK<RET>
*/COMMON::1000<RET>
*
```

Creates an unlabeled COMMON area of 1000 words.

OPTIONAL NOTATIONS

You can specify the number of words in octal.

RELATED SWITCHES

/PSCOMMON

/CONTENTS

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

FUNCTION specifies the symbol types 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 shown in **bold**:

Keyword	Description
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, for example, JOBDAT.)
RELOCATABLE	**Include relocatable symbols.**
UNDEFINED	Include undefined symbols.
ZERO	Include symbols in zero-length programs.

Only those symbols that satisfy all conditions in the keyword list will appear in the MAP file. For example, if both the NOGLOBAL and RELOCATABLE settings are in force, all global symbols are excluded regardless of their relocatability.

EXAMPLES

```
@LINK<RET>
*/CONTENTS:(NOCOMMON,NOENTRY)<RET>
*
```

Excludes COMMON and entry-name symbols.

```
@LINK<RET>
*/CONTENTS:ALL<RET>
*
```

Includes all symbols.

OPTIONAL NOTATIONS

You can omit the parentheses if you give only one keyword.

RELATED SWITCHES

/MAP

/COUNTERS

FORMAT /COUNTERS

FUNCTION Displays information about relocation counters on the terminal. A relocation counter is an address counter that LINK uses while loading relocatable code.

/COUNTERS returns the name, initial value, current value, and limit value of each counter. /COUNTERS first prints a header:

Reloc. ctr. initial value current value limit value

Reloc. ctr. gives the name of relocation counter.

initial value is the origin of the relocation counter.

current value is the address of the next free location after the relocation counter has been loaded.

limit value is an upper bound that you set using /LIMIT or that LINK sets by default for the relocation counter. This upper bound defines a point the relocation counter should not load beyond. If /LIMIT is used and the counter loads beyond this bound, LINK returns messages. See /LIMIT for more information.

/COUNTERS 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.

You can also use /COUNTERS to determine the size of PSECTs when loading extended addressing programs or programs that use PSECTs to conserve memory space. Refer to Chapter 6.

EXAMPLES The following examples illustrate the various /COUNTERS displays.

The following display results from loading a module that does not contain code.

```
@LINK<RET>
*/COUNTERS<RET>
[LNKRLC No relocation counters set]
```

The following display results from loading only absolute code.

```
@LINK<RET>
*ABCODE/COUNTERS<RET>
[LNKRLC No relocation counters set
 Absolute code loaded]
*
```

The following display results from loading only PSECT code.

```
@LINK<RET>
*PSCODE/COUNTERS<RET>

[LNKRLC Reloc. ctr.     initial value   current value   limit value
        PSCODE          20              25              1000000]
*
```

The following display results from loading code that contains both absolute and PSECT code.

```
@LINK<RET>
*MIXED/COUNTERS<RET>
[LNKRLC Reloc. ctr.     initial value   current value   limit value
        PSECTA          1400000         1400001         4000000
        PSECTB          2500000         2500001         4000000
        PSECTC          3600000         3600001         4000000
        Absolute code loaded]
*
```

The following display results from loading two-segmented formatted code.

```
@LINK<RET>
*TWOPRT/COUNTERS<RET>
[LNKRLC Reloc. ctr.     initial value   current value   limit value
        .LOW.           0               1642            1000000
        .HIGH.          400000          400753          1000000]
*
```

RELATED SWITCHES /NEWPAGE, /SET, /LIMIT

/CPU

FORMAT /CPU:(keyword,keyword)

Keyword: KA10
KI10
KL10
KS10

FUNCTION Overrides LINK's handling of the processor information found in the REL files being loaded. (See the description of Block Type 6 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).

EXAMPLES
```
@LINK<RET>
*/CPU:KI10<RET>
*
```

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

OPTIONAL NOTATIONS You can omit the parentheses if you specify only one CPU.

/DDEBUG

FORMAT /DDEBUG:keyword

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

The keywords and the debuggers they specify are listed below. Only those shown in **bold** are supported by DIGITAL.

Keyword	Debugger
ALGDDT	**ALGDDT**
ALGOL	**ALGDDT**
COBDDT	**COBDDT**
COBOL	**COBDDT**
DDT	**DDT**
FAIL	SDDT (SAIL debugger)
FORDDT	**FORDDT**
FORTRAN	**FORDDT**
MACRO	**DDT**
PASCAL	PASDDT
PASDDT	PASDDT
SAIL	SDDT (SAIL debugger)
SDDT	SDDT (SAIL debugger)
SIMDDT	SIMDDT
SIMULA	SIMDDT

EXAMPLES
```
@LINK<RET>
*/DDEBUG:FORTRAN<RET>
*
```

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

RELATED SWITCHES /DEBUG, /TEST

/DEBUG

FORMAT /DEBUG:keyword

FUNCTION Requests loading of a debugger and sets the start address for execution at the start address of the debugger. 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.

Local symbols for the debugger 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 keywords and the programs they load are listed below. Only those shown in **bold** supported by DIGITAL.

Keyword	Debugger
ALGDDT	**ALGDDT**
ALGOL	**ALGDDT**
COBDDT	**COBDDT**
COBOL	**COBDDT**
DDT	**DDT**
FAIL	SDDT (SAIL debugger)
FORDDT	**FORDDT**
FORTRAN	**FORDDT**
MACRO	**DDT**
PASCAL	PASDDT
PASDDT	PASDDT
SAIL	SDDT (SAIL debugger)
SDDT	SDDT (SAIL debugger)
SIMDDT	SIMDDT
SIMULA	SIMDDT

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

EXAMPLES

```
@LINK<RET>
*/DEBUG:FORDDT<RET>
*
```

Loads FORDDT, sets the /EXECUTE switch, and specifies that FORDDT control execution.

OPTIONAL NOTATIONS

You can abbreviate /DEBUG to /D.

RELATED SWITCHES

/DDEBUG, /TEST

/DEFAULT

FORMAT 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	Your connected directory

For output files, the initial defaults are:

device	DSK:
filename	Name of main program
directory	Your connected directory

EXAMPLES

```
@LINK<RET>
*/DEFAULT:INPUT .BIN<RET>
*
```

Resets input file default extension to BIN.

```
@LINK<RET>
*/DEFAULT:OUTPUT MYSTR:<RET>
*
```

Resets output file default device to MYSTR:.

OPTIONAL NOTATIONS If you omit the keyword, INPUT is assumed.

/DEFINE

FORMAT

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

symbol is a global symbol.

value is a decimal number.

FUNCTION

Assigns a decimal value to a symbol. This assignment causes the symbols to be global symbols. To get a list of any undefined symbols, use the /UNDEFINED switch.

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

EXAMPLES

```
*/UNDEFINED<RET>
[LNKUGS  2 UNDEFINED GLOBAL SYMBOLS]
     ALPHA   400123
     BETA    402017

*/DEFINE:(ALPHA:591,BETA:1)<RET>
*
```

Gives the decimal values 591 and 1 to ALPHA and BETA, respectively.

OPTIONAL NOTATIONS

You can give the value in octal by typing a hash sign (#) before the value. You can omit the parentheses if you define only one symbol.

RELATED SWITCHES

/UNDEFINED, /VALUE

/ENTRY

FORMAT /ENTRY

FUNCTION Displays all entry name symbols that have been loaded on the terminal. Each entry name symbol is defined by an ENTRY statement (MACRO, FORTRAN, or BLISS), a FUNCTION statement (FORTRAN), a SUBROUTINE statement (FORTRAN, or COBOL), or a PROCEDURE declaration (ALGOL, or PASCAL).

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

EXAMPLES
```
@LINK<RET>
/ENTRY<RET>
[LNKLSS  LIBRARY SEARCH SYMBOLS (ENTRY POINTS)]
     SQRT.  3456
*
```

RELATED SWITCHES /NOENTRY

/ERRORLEVEL

FORMAT /ERRORLEVEL:n

n is a positive decimal integer.

FUNCTION Suppresses the terminal display of LINK messages with message level less than n, where n is a decimal number between 0 and 31 inclusive. You cannot suppress level 31 messages. LINK's default is /ERRORLEVEL:8.

See Appendix B for the level of each LINK message.

EXAMPLES
```
@LINK<RET>
*/ERRORLEVEL:15<RET>
*
```

Suppresses all messages less than level 15.

```
@LINK<RET>
*/ERRORLEVEL:0<RET>
*
```

Permits typeout of all messages.

RELATED SWITCHES /VERBOSITY, /MESSAGE

/EXCLUDE

FORMAT /EXCLUDE:(module,...,module)

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.
- If several modules contain the same symbol, you can select the module with the symbol definition you want by excluding the others. This also prevents modules from giving multiple definitions of a symbol.
- In defining an overlay structure, you can delay loading of a module until a later overlay link by excluding the module.

EXAMPLES
```
@LINK<RET>
*/SEARCH LIBFIL.REL/EXCLUDE:(MOD1,MOD2)<RET>
*
```

Searches LIBFIL as a library but prevents loading of MOD1 and MOD2 even if they resolve global symbol references.

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

RELATED SWITCHES /INCLUDE, /NOINCLUDE

/EXECUTE

FORMAT /EXECUTE

FUNCTION Directs LINK to execute the loaded program beginning at the program's start address. LINK continues loading until a /GO switch is found.

Use either the /EXECUTE or /DEBUG switches to load and execute a program.

EXAMPLES
```
@LINK<RET>
*/EXECUTE<RET>
*PROGRM/GO
```

OPTIONAL NOTATIONS You can abbreviate /EXECUTE to /E.

RELATED SWITCHES /DEBUG, /GO, /RUN, /TEST

/FRECOR

FORMAT /FRECOR:nK

n is a positive decimal integer.

K is the representation for 1024.

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 ten 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).
10. Argument typechecking (TP). +

Areas that can overflow are marked with a +. 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 ten spaces is at least the space given by the /FRECOR switch, if possible.

LINK recovers free memory by concatenating these ten free areas. When all this recovered space is used and one or more of the ten areas overflows then free memory is no longer maintained because the free memory has been used.

EXAMPLES
```
@LINK<RET>
*/FRECOR:7K<RET>
*
```

Maintains 7K of free memory, if possible.

OPTIONAL NOTATIONS You can specify the free memory in octal using a hash sign (#). For example, /FRECOR:#11K.

/GO

FORMAT /GO

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

- Begins execution at the start address of the loaded program if you used /EXECUTE.
- Begins execution at the start address of the debugger if you used /DEBUG.
- Exits to the monitor if you used no execution switch.

EXAMPLES

```
@LINK<RET>
*MYPROG/EXECUTE/GO<RET>

[LNKXCT  MYPROG execution]
```

Begins execution of the loaded program at its start address.

```
@LINK<RET>
*MYPROG/DEBUG/GO<RET>

[LNKDEB DDT execution]
```

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

OPTIONAL NOTATIONS You can abbreviate /GO to /G.

RELATED SWITCHES /DEBUG, /EXECUTE, /RUN

/EXIT

FORMAT /EXIT

FUNCTION Exits LINK. LINK does not finish loading the program. To finish loading the program and to exit LINK, use /GO.

EXAMPLES
```
@LINK
*MYPROG<RET>
*/EXIT
@
```

/HASHSIZE

FORMAT /HASHSIZE:n

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.

LINK automatically expands a symbol table. However, if you know that you 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 at least 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
```
@LINK<RET>
/HASHSIZE:1000<RET>
*
```

Sets the hash size to the prime number 1021.

/HELP

FORMAT /HELP:keyword

DEFAULTS /HELP:TEXT

FUNCTION Displays information about LINK on the terminal. The keywords are:

Keyword	Description
SWITCHES	prints a list of valid switches.
TEXT	prints a more detailed description of command formats and switches.

OPTIONAL NOTATIONS You can abbreviate /HELP to /H.

/INCLUDE

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

FUNCTION Requires the loading of the specified modules whether or not there are any global requests for them. For example,

```
*/INCLUDE:(MODU6,MODU9)<RET>
```

requires that modules MODU6 and MODU9 be loaded.

/INCLUDE can be used as a local or global switch. If /INCLUDE is used as a local switch the request for the module is cleared after the file is loaded. For example,

```
*FILE8/INCLUDE:(MODU7,MODU8)<RET>
```

loads module MODU7 and MODU8 from FILE8.

If /INCLUDE is used as a global switch, the request for the modules are not cleared until the modules are loaded. For example,

```
*/INCLUDE:(MODU2,MODU3,MODU4) FILEA,FILEB,FILEC<RET>
```

requires loading of MODU2, MODU3, and MODU4. If these modules do not exist in FILEA, FILEB, and FILEC, /INCLUDE request continues until they are loaded.

If you used /SEARCH specifying library search mode and specify /INCLUDE, the /INCLUDE switch causes the modules specified in the switch to be loaded in addition to modules that are loaded to satisfy global symbol references. For example,

```
*LIBARY/SEARCH/INCLUDE:(DATMOD,NAMMOD,NUMMOD)<RET>
```

Causes LIBARY to be searched and DATMOD, NAMMOD, and NUMMOD to be loaded along with other modules that are needed to resolve global symbol references.

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

```
@LINK<RET>
*/SEARCH LIB1/INCLUDE:(MOD1,MOD2)<RET>
*
```

Searches LIB1 and loads MOD1 and MOD2 even if they are not needed to resolve global symbol references.

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

RELATED SWITCHES — /EXCLUDE, /NOINCLUDE, /MISSING

/LIMIT

FORMAT /LIMIT:psect:origin

psect is a PSECT name.

origin is a thirty-bit octal address or a defined symbol.

FUNCTION specifies an upper bound for a PSECT. If the PSECT loads beyond this bound, LINK returns error messages.

Before using /LIMIT, the PSECT name must be defined with either the /SET switch or in one of the modules already loaded.

The upper bound can be expressed in either numeric or symbolic form. This address is one greater than the highest location which may be loaded in the PSECT 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 sends a warning to your terminal, but continues to process input files and to load code. The warning message takes the following form:

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

Although LINK does continue to process input files and load code, the program is incomplete and should NOT be used. LINK produces the following fatal error:

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

Chained fixups are a method that LINK uses to resolve symbol references.

Using /LIMIT to define an upper bound prevents unintended PSECT overlaps. PSECT overlaps can cause LINK to loop and produces other unpredictable behavior, because LINK uses the unresolved address relocation chains in the user program being built.

EXAMPLES In the following example, the LRGPRO and BIGPRO programs each contain two PSECTs, BIG and GRAND. LRGPRO is loaded first using /COUNTERS to check the origin and the current value of the PSECTs. The PSECT origin is found by looking under the initial value column and the current value is found by looking under the current value of the /COUNTERS output.

```
@LINK<RET>
*/SET:BIG:1000<RET>
```

```
*/SET:GRAND:5400<RET>
*LRGPRO<RET>
*/COUNTERS<RET>
[LNKRLC Reloc. ctr.     initial value   current value   limit value
        .LOW.           0               140             1000000
        BIG             1000            5100            1000000
        GRAND           5400            10500           1000000
        Absolute code loaded]
```

/COUNTERS shows that the current value for PSECT BIG and the initial value for PSECT GRAND are close together in memory. The current value for BIG is 5100 and the PSECT origin for GRAND is 5400. The /LIMIT switch can now be used to restrict PSECT BIG upper bound to PSECT GRAND's initial value using:

```
*/LIMIT:BIG:GRAND<RET>
```

This prevents an unintended overlap when PSECT BIG from BIGPRO is loaded because the upper bound for BIG is set to GRAND's origin. Now when BIGPRO is loaded, if PSECT BIG exceeds the limit, LINK produces the LNKPEL warning indicating that the upper bound needs to be increased. The /COUNTERS switch shows a new current value greater than 5400. Notice that the limit set with the /LIMIT switch is shown in the LIMIT VALUE column.

```
*BIGPRO<RET>
%LNKPEL PSECT BIG exceeded limit of 5100
        detected in module .MAIN from file BIGPRO.REL
*/COUNTERS<RET>
[LNKRLC Reloc. ctr.     initial value   current value   limit value
        .LOW.           0               140             1000000
        BIG             1000            6300            5400
        GRAND           5400            10500           1000000]
        Absolute code loaded]
```

/GO continues loading the program, and LINK produces the POV warning message and the CFS fatal error message.

```
*/GO<RET>
%LNKPOV PSECTs BIG and GRAND overlap from address 5400 to 6300
?LNKCFS chained fixups have been suppressed
```

To avoid the overlap, move the PSECTs farther apart in memory. In this example, GRAND's origin is reset from 5400 to 7000.

```
@LINK<RET>
*/SET:BIG:1000<RET>
*/SET:GRAND:7000<RET>
*/LRGPRO<RET>
```

```
*/LIMIT:BIG:GRAND<RET>
*BIGPRO<RET>
*/GO<RET>
@
```

/LINK

FORMAT

/LINK:name

name is up to 6 Radix-50 compatible characters.

FUNCTION

Closes an overlay link. /LINK directs LINK to give the specified overlay link name to the current memory image and outputs the memory image to the overlay file. LINK first performs any required library searches and assigns a number to the overlay link.

For a discussion of overlay structures, see Chapter 5.

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

EXAMPLES

```
@LINK<RET>
*SPEXP/LINK:ALPHA<RET>
*
```

Loads module SPEXP and outputs the memory image to the overlay file as an overlay link named ALPHA.

OPTIONAL NOTATIONS

If you omit the overlay link name, LINK uses the overlay link's 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.

/LOCALS and /NOLOCAL may be used as either local or global switches. If the switch is appended to a file specification, it applies only to that file; if it is not appended to a file specification, it applies to all the files that follow in the same command line.

EXAMPLES

```
@LINK<RET>
*/LOCALS ALPHA,BETA/NOLOCAL,CAPPA,/NOLOCAL DELTA<RET>
*
```

Loads ALPHA with local symbols, BETA without local symbols, CAPPA with local symbols, and DELTA 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 encountered are not entered in the log file. Any messages output after the LOG file is closed are also not entered in the file.

EXAMPLES

```
@LINK<RET>
*LOGFIL/LOG<RET>
*
```

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

```
@LINK<RET>
*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
```

n is a positive decimal integer.

FUNCTION

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

See Appendix B for the level of each LINK message.

The default is /LOGLEVEL:8.

EXAMPLES

```
@LINK<RET>
*/LOGLEVEL:0<RET>
*
```

Logs all messages.

RELATED SWITCHES

/ERRORLEVEL, /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.

Keyword	Description
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 do 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 are not performed unless forced.

EXAMPLES

```
@LINK<RET>
*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

n is a positive decimal integer.

FUNCTION Specifies the number of overlay links to be defined when the overlayed program requires more than 256 overlay links. LINK allocates extra space in the OVL file for certain fixed-length tables based on the number of overlay 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
```
@LINK<RET>
*TEST/OVERLAY/MAXNODE:500<RET>
*
```

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

RELATED SWITCHES /OVERLAY

/MESSAGE

FORMAT /MESSAGE:keyword

FUNCTION Displays messages on the terminal in the format specified by keyword. Both /MESSAGE and /VERBOSITY display messages on the terminal, but use different keywords and return slightly different output.

Keyword	Description
PREFIX	displays the message code.
NOPREFIX	prevents the display of the message code.
FIRST	displays the short message.
NOFIRST	prevents the display of the short message.
CONTINUATION	displays the longer message.
NOCONTINUATION	prevents the display of the longer message.

EXAMPLES In all the following examples, an incorrect file specification (KSD:MISTKE) is entered. In the following example, /NOPREFIX prevents the display of the message code.

```
@LINK<RET>
*/MESSAGE:NOPREFIX<RET>
*KSD:MISTKE<RET>
% Non-existent device KSD:
[ Please retype the incorrect parts of the file
specification ]
*
```

In the next example, a short message is returned.

```
@LINK<RET>
*/MESSAGE:FIRST<RET>
*KSD:MISTKE<RET>
%LNKNED Non-existent device KSD:
[ Please retype the incorrect parts of the file
specification]
```

In this example, a long message is returned.

```
@LINK<RET>
*/MESSAGE:CONTINUATION<RET>
*KSD:MISTKE<RET>
%LNKNED Non-existent device KSD:
 You gave a device that does  not  exist  on  this  system.
 Correct your input files and reload.
[ Please retype the incorrect parts of the file specification]
```

RELATED SWITCHES /ERRORLEVEL, /VERBOSITY

/MISSING

FORMAT /MISSING

FUNCTION Displays on the terminal the number of modules requested with the /INCLUDE switch that have not yet been loaded.

EXAMPLES In the following example, MYPROG is loaded, library searches are requested, MOD1 and MOD2 are to be loaded during the search, and LIB1 is to be loaded.

```
@LINK<RET>
*MYPROG<RET>
*/SEARCH/INCLUDE:(MOD1,MOD2) LIB1<RET>
```

Next, /MISSING is used to check if both modules were loaded.

```
*/MISSING<RET>
[LNKIMM  1 included module missing]
```

Now, LIB2 is loaded, MOD2 is to be included, and /MISSING is used to check if MOD2 was loaded.

```
*LIB2/INCLUDE:(MOD2)<RET>
*/MISSING<RET>
[LNKIMM  no included modules missing]
*
```

RELATED SWITCHES /EXCLUDE, /INCLUDE, /UNDEFINED

/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.

Keyword	Description
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
```
@LINK<RET>
*/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, /COUNTERS

/NODE

FORMAT /NODE:argument

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

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

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

NOTE

It is recommended that you use an overlay 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 overlay link numbers.

EXAMPLES In the following example, /NODE opens overlay link FATHER, SON1 and SON2 are loaded, and /LINK closes the overlay link.

```
@LINK<RET>
*/NODE:FATHER<RET>
*SON1<RET>
*SON2<RET>
*/LINK:FATHER<RET>
*
```

See Chapter 5 for more examples.

RELATED SWITCHES /LINK, /OVERLAY, /PLOT

/NOENTRY

FORMAT

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

symbol is an entry name 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 does 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

```
@LINK<RET>
*/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.

EXAMPLES
```
@LINK<RET>
*LIB1/INCLUDE:(MOD1,MOD3)<RET>
*/NOINCLUDE<RET>
*
```

Loads MOD1 and MOD3 from LIB1. However, if the modules are not found immediately, LINK stops 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 is 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 eventually runs in executive mode (for example, a monitor or bootstrap loader).
- Building a TOPS-20 native program which does not use a JOBDAT area.

EXAMPLES
```
@LINK<RET>
*/NOINITIAL<RET>
*
```

RELATED SWITCHES /NOJOBDAT

/NOJOBDAT

FORMAT /NOJOBDAT

FUNCTION Prevents LINK from filling in JOBDAT's or vestigial JOBDAT's address space and causes LINK to create a PDV. JOBDAT contains program information such as debugger symbol table location and version number. Vestigial JOBDAT is used in two segment programs.

LINK creates either a JOBDAT or a PDV (Program Data Vector) when loading a program. LINK creates a JOBDAT for most section zero programs. LINK creates a PDV:

- for extended addressing programs.
- if a PSECT has an origin below 140.
- if you specify a PDV using /PVBLOCK or /PVDATA.
- if you specify /NOJOBDAT.

/NOJOBDAT does not suppress symbols.

EXAMPLES
```
@LINK<RET>
*PROGAM/NOJOBDAT<RET>
*
```

Loads PROGAM and keeps LINK from filling in JOBDAT information during loading.

RELATED SWITCHES /NOINITIAL

/NOLOCAL

FORMAT

```
/NOLOCAL
```

FUNCTION

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

The /LOCALS and /NOLOCAL switches may be used as local or global switches. If the switch is appended to a file specification, it applies only to that file; if it is not appended to a file specification, it applies to all the following files in the same command.

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

EXAMPLES

```
@LINK<RET>
*/LOCALS MODA,MODB/NOLOCAL,MODC,/NOLOCAL MODD<RET>
*
```

Loads MODA with local symbols, MODB without local symbols, MODC with local symbols, and MODD without local symbols.

OPTIONAL NOTATIONS

Abbreviate /NOLOCAL to /N.

RELATED SWITCHES

/LOCALS

/NOREQUEST

FORMAT

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

FUNCTION

Deletes references to overlay links from LINK's overhead tables when loading overlay programs. If you know that the execution of the current load does not require certain overlay 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

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

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

EXAMPLES
```
@LINK<RET>
*/SEARCH ALPHA,BETA/NOSEARCH,CAPPA,/NOSEARCH DELTA<RET>
*
```

Searches ALPHA, loads BETA, searches CAPPA, and loads DELTA.

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
```
@LINK<RET>
*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 free some memory space using the /NOSYMBOL switch.

EXAMPLES
```
@LINK<RET>
*/NOSYMBOL<RET>
*
```

/NOSYSLIB

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

FUNCTION Prevents automatic search of the system libraries named as keywords. LINK 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 continues 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 keywords and the libraries they specify are listed below. Only those shown in **bold** specify libraries supported by DIGITAL.

Keyword	Library
ANY	**all**
ALGOL	**ALGLIB**
BCPL	BCPLIB
COBOL	**LIBOL or C74LIB**
FORTRAN	**FORLIB**
NELIAC	LIBNEL
PASCAL	PASLIB
SAIL	SAILIB
SIMULA	SIMLIB

EXAMPLES
```
@LINK<RET>
*/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.

RELATED SWITCHES /SYSLIB

/NOUSERLIB

FORMAT

/NOUSERLIB

FUNCTION

Discontinues automatic searching of the user libraries at each /LINK or /GO switch. You can specify which library to stop searching by appending /USERLIB to a file specification. For example,

*MYLIB/NOUSERLIB

stops searching MYLIB.

For programs that use overlays, if you need a file searched for some overlay links but not others, you can use the /USERLIB and /NOUSERLIB switches to enable and disable automatic searching of the file.

EXAMPLES

```
@LINK<RET>
*FILE/OVERLAY<RET>
*FILE4/USERLIB:FORTRAN<RET>
*LINKA/LINK:ROOT<RET>
*/NODE:ROOT<RET>
*LINKB/LINK:ONE<RET>
*/NODE:ROOT<RET>
*LINKA,LINKB/NOUSERLIB<RET>
*
```

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 the specified segments of two-segment modules. The keywords are:

Keyword	Segments
HIGH	high
LOW	low
BOTH	both

If you are loading a module using /ONLY:HIGH or /ONLY:LOW, it cannot contain chained fixups that cross segments.

/ONLY and /SEGMENT are incompatible switches.

EXAMPLES
```
@LINK<RET>
*/ONLY:HIGH MOD1,MOD2<RET>
*MOD3/ONLY:BOTH<RET>
*
```

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

RELATED SWITCHES /SEGMENT

/OPTION

FORMAT /OPTION:name

FUNCTION Reads the SWITCH.INI file to determine the specified switch defaults for LINK. /NOOPTION ignores the SWITCH.INI file.

A SWITCH.INI contains switches for LINK to use, and allows you to override system defaults. For example, the following line when placed in SWITCH.INI changes the default for input files from DSK: to MYSTR:.

```
LINK/DEFAULT:INPUT MYSTR:
```

The name argument specifies which line to read in SWITCH.INI file. For example,

```
/OPTION:LINK
```

reads the LINK line in the SWITCH.INI file.

EXAMPLES:

```
@LINK<RET>
*/OPTION:LINK<RET>
*INPUT<RET>
*
```

/OTSEGMENT

FORMAT /OTSEGMENT:keyword

FUNCTION Specifies the time and manner of loading the object-time system. An object-time system is a collection of modules that is called by compiled code for a particular language in order to perform various utility functions such as I/O and trap-handling. For example, FOROTS is the object-time system for FORTRAN. For more information on object-time systems for the language you are using, refer to the appropriate language manual.

Keyword	Description
DEFAULT	Suspends the effect of a previous /OTSEGMENT:SHARABLE or /OTSEGMENT:NONSHARABLE switch.
NONSHARABLE	Loads the object-time system into the executable program at load time. Both the user program and the object-time system may have code in both the high and low 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, except in the following cases. If **any** of the following cases occur, a non-sharable object-time system is loaded as part of the program.

- You specify /OTSEGMENT:NONSHARABLE.
- You have loaded any code into the high segment.
- You have specified /SYMSEG:HIGH.
- Your low segment is too big for the sharable object-time systems to fit, and your program is loaded in section 0.

EXAMPLES
```
@LINK<RET>
*MYPROG/SYSLIB/OTSEGMENT:NONSHAR<RET>
*
```

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

RELATED /SEGMENT
SWITCHES

/OVERLAY

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

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

You can append the /OVERLAY switch to a file specification in the form:

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

The keywords and their meanings are listed below. The default settings are shown in **bold**

Keyword	Description
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.**
NOWARNING	Suppresses overlay warning messages.
RELOCATABLE	Specifies that links are relocatable.
TREE	**Specifies that the overlay has 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 @LINK<RET>
*/OVERLAY:(TREE,LOGFILE)

Specifies that an overlay is to be constructed, and that

runtime messages are to be printed on the terminal.

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

RELATED SWITCHES — /LINK, /NODE, /PLOT

/PATCHSIZE

FORMAT

```
/PATCHSIZE:n
```

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

```
@LINK<RET>
*/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
```
@LINK<RET>
*OVLPRO/OVERLAY:(TREE,LOGFILE)<RET>
*LPT:OVLPRO/PLOT<RET>
*
```

requests a tree diagram for the overlay. The diagram is formatted for the line printer because LPT: is specified.

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 automatically designs subtrees for diagrams on more pages.

RELATED SWITCHES /LINK, /NODE, /OVERLAY

/PLTTYP

FORMAT /PLTTYP:keyword

FUNCTION Specifies 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 @LINK

```
*OVLPRO/OVERLAY
*OVLPRO/PLOT/PLTTYP:PRINTER
```

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

RELATED SWITCHES /PLOT

/PSCOMMON

FORMAT /PSCOMMON:psect:common

FUNCTION Specifies where LINK is to load COMMON blocks. This switch causes the FORTRAN common specified by the argument common to be loaded into the PSECT specified in the argument psect. Use the /PSCOMMON switch before loading the specified common and before declaring the common's size with the /COMMON switch.

/PSCOMMON only affects common blocks defined with the /COMMON switch. If the common block is created by a REL block, /PSCOMMON is ignored, and the PSECT specified by the REL file is used.

EXAMPLES In the following example, /SET defines the SECTA PSECT's origin, /PSCOMMON specifies that SECTA is loaded into COMABC, and /COMMON defines the common size.

```
@LINK<RET>
*/SET:SECTA:3000000
*/PSCOMMON:SECTA:COMABC
*/COMMON:COMABC:10000
*PROG
*
```

RELATED SWITCHES /COMMON

/PVBLOCK

FORMAT /PVBLOCK:keyword

FUNCTION Requests a PDV (Program Data Vector) from LINK and gives you control over where the vector goes.

NOTE

PDVs are useful only with TOPS-20 Version 5 and later monitors. Earlier monitors ignore PDVs. Refer to Chapter 7 for more information about PDVs.

Keyword	Description
DEFAULT	disables the previous /PVBLOCK:HIGH or /PVBLOCK:LOW, and restores the default. LINK supplies a PDV by default: o for an extended addressing program, o if /PVDATA is specified o if /NOJOBDAT is specified. o if a PSECT's start address is below 140. The default PDV is placed at the end of the low segment even for programs that use PSECTs.
HIGH	places the PDV at the end of the high segment.
LOW	places the PDV at the end of the low segment.
NONE	prevents loading of the PDV.
PSECT:name	places the PDV at the end of the named PSECT.

EXAMPLES In the following example, PVTEST is loaded and a PDV is requested. The PDV is placed at the end of the low segment as indicated by the LOW keyword.

```
@LINK<RET>
```

```
*PVTEST<RET>
*/PVBLOCK:LOW/GO<RET>

Next, the TOPS-20 INFORMATION VERSION command is  used  to
display the program and PDV name.

@INFO VERSION<RET>
       .
       .
       .
Program is PVBLCK
PDVs: Program name PVTEST, version
@
```

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

RELATED SWITCHES — /PVDATA, /NOJOBDAT

/PVDATA

FORMAT /PVDATA:keyword:value

FUNCTION Changes the contents of a PDV block. The /PVDATA switch also allocates storage for the PDV. 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 PDV information into the executable program or sharable save (EXE) file. Refer to Chapter 7 for more information on PDVs.

NOTE

PDVs are useful only with TOPS-20 Version 5 and later monitors. Earlier monitors ignore PDVs. Refer to Chapter 7 for more information about PDVs.

Keyword	Description
NAME	is an optionally quoted ASCII string that identifies the PDV. This string is not converted to uppercase. If you do not quote the program name, you can use only alphanumerics and the period (.), dollar sign ($), and percent sign (%) characters to specify program name.
VERSION	is a hash sign (#) followed by an octal value, a version number that starts with a number, or a global symbol.
MEMORY	is the address of the user supplied memory map in octal, or a previously defined global symbol that contains the address of a user supplied memory map. The user supplied memory map suppresses the map generated by LINK.
PROGRAM	is the address of a program-specific data block in octal value, or a previously defined global symbol that contains the address of a program-specific data block.

CBLOCK	is the address of a customer-defined block in octal, or a previously defined global symbol that contains the address of a customer-defined block.
EXPORT	is the address of a block of information defined by a program in octal, or a previously defined global symbol that contains the address of a block of information defined by a program.

EXAMPLES In the following example, a PDV name and version number are defined. /PVDATA:NAME:T.S$T% defines the T.S$T% as the PDV name, and /PVDATA:VERSION:1A(3) defines 1.1(3) as the version number.

```
@LINK<RET>
*/PVDATA:NAME:T.S$T%<RET>
*/PVDATA:VERSION:1A(3)<RET>
*PVTEST/GO<RET>
```

Next, the TOPS-20 INFORMATION VERSION command is used to display the PDV name and version.

```
@INFO VERSION<RET>
       .
       .
       .
Program is PVBLCK
PDVs: Program name T.S$T%, version 1.1(3)
@
```

The next example also specifies a PDV name and version, but in a different format.

```
@LINK<RET>
*/PVDATA:NAME:"TST@ "<RET>
*/PVDATA:VERSION: 101000003<RET>
*PVTEST/GO<RET>

@INFO VERSION<RET>
       .
       .
       .
Program is PVBLCK
PDVs: Program name TST@ , version 1.1(3)
@
```

RELATED /PVBLOCK

SWITCHES

/REDIRECT

FORMAT /REDIRECT:Lowpsect:Highpsect

FUNCTION Loads two-segmented formatted REL files as part of a program using PSECTs. The argument Lowpsect is the name of the PSECT to receive the low segment code and Highpsect is the name of the PSECT to receive the high segment code.

You must redirect both segments, you cannot redirect just the high or the low segment.

EXAMPLES The following example loads a two-segment program (TWOPRT), and displays the low- and high-segment values using /COUNTERS.

```
@LINK<RET>
*TWOPRT<RET>
*/COUNTERS<RET>
[LNKRLC   Reloc. ctr.    initial value current value limit
value
     .LOW.          0     1642  1000000
     .HIGH.         400000  400753  1000000]
*
```

Next, PSECT origins are set for PSHI and PSLO, .LOW. is redirected into PSLO, .HIGH. is redirected into PSHI, and /COUNTERS is used to display PSHI and PSLO values.

```
@LINK<RET>
*/SET:PSHI:400010<RET>
*/SET:PSLO:3500<RET>
*/REDIRECT:PSLO:PSHI<RET>
*TWOSEG<RET>
*/COUNTERS<RET>
[LNKRLC   Reloc. ctr.    initial value current value limit
value
     PSHI      400010     400753  1000000
     PSLO      3500      5202  1000000]
```

/REQUEST

FORMAT /REQUEST

FUNCTION Displays external global symbol references on the terminal. Use /REQUEST to determine if a module uses a global symbol that is loaded in another module.

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

```
@LINK<RET>
*/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 an overlay link common to all overlay links that reference the module.

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

EXAMPLES
```
@LINK<RET>
*/UNDEFINED<RET>
[LNKUGS  NO UNDEFINED GLOBAL SYMBOLS]
*/REQUIRE:(ROUTN.,SQRT.)<RET>
*/UNDEFINED<RET>
[LNKUGS  2 UNDEFINED GLOBAL SYMBOLS]
     SQRT.      0
     ROUTN.     0]
*
```

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

RELATED SWITCHES /SEARCH, /NOSEARCH

/RUN

FORMAT /RUN:file

FUNCTION Directs LINK to run the program after it is loaded. LINK ignores this switch if you used /EXECUTE or /DEBUG.

EXAMPLES:

```
@LINK<RET>
*/RUN:MYPROG<RET>
*MYPROG<RET>
*
```

/RUNAME

FORMAT /RUNAME:name

FUNCTION Assigns a job name for execution of your program. This job name is used in the SYSTAT display.

EXAMPLES
```
@LINK<RET>
*/RUNAME:LNKDEV<RET>
*PROGRM
*
```

Assigns the name LNKDEV for program execution regardless of the file names that are loaded.

/RUNOFFSET

FORMAT /RUNOFFSET:n

FUNCTION Runs the program specified in a /RUN switch with an offset of n. If you omit the switch the default is 0. If you omit the n argument, the default is 1.

EXAMPLES In the following example, MACRO is run with an offset of 1 after MCPROG is loaded.

```
@LINK<RET>
*MCPROG<RET>
*/RUN:MACRO/RUNOFFSET:1/GO<RET>
```

/SAVE

FORMAT filespec/SAVE

FUNCTION Directs LINK to create a sharable save file using the specified file specification. If you omit the file type, LINK uses EXE.

To run the sharable save file with the TOPS-20 RUN command, the file must have an EXE file type.

EXAMPLES
```
@LINK<RET>
*MYPROG<RET>
*DSKZ:RUNPRO.EXE/SAVE/GO<RET>
@
```

Directs LINK to save the loaded version of MYPROG as RUNPRO.EXE on DSKZ:.

/SEARCH

FORMAT /SEARCH

FUNCTION Directs LINK to search the input files and load only the modules whose entry point names resolves a global symbol request.

The /NOSEARCH discontinues library searching of input files, and loads all the modules in the input files. Although /NOSEARCH is used, LINK continues to search system libraries unless you used the /NOSYSLIB, and searches user libraries if you used the /USERLIB switch.

For example, the FILE9, FILE8, and FILE7 input files are to be searched as libraries and only the modules that resolve global symbols are to be loaded, but all the modules in FILE1, FILE2, and FILE3 are to be loaded.

```
@LINK<RET>
*/SEARCH FILE9,FILE8,FILE7<RET>
*/NOSEARCH FILE1,FILE2,FILE3<RET>
```

The /SEARCH and /NOSEARCH switches can be used as local or global switches. See Section 3.3.3 for a discussion of switch placement.

Note that search requests in .TEXT blocks may be processed in the reverse order of entered /SEARCH switches. Keep this in mind when specifying the order in which modules are to be searched. See Block types greater than 3777 for more information.

EXAMPLES

```
@LINK<RET>
*/SEARCH MODA,MODB/NOSEARCH,MODC,/NOSEARCH MODD<RET>
*
```

Searches MODA, loads MODB, searches MODC, and loads MODD.

This examples illustrates the use of /SEARCH and /NOSEARCH as local and global switches. The /SEARCH switch is a global switch because it is entered after the asterisk, and applies to MODA and MOBC. The first /NOSEARCH switch is a local switch because it is appended to MODB. Notice its placement before the comma. The second /NOSEARCH switch is a global switch and applies to the remainder of the line.

OPTIONAL NOTATIONS You can abbreviate /SEARCH to /S.

RELATED /NOSEARCH

SWITCHES

/SEGMENT

FORMAT /SEGMENT:keyword

FUNCTION Specifies whether the high segment or the low segment of a two-segment program is to be used for loading the modules that follow. If /SEGMENT is omitted, LINK follows the specifications in the program.

LINK does not follow the programs specifications for FORTRAN object code, but loads both segments into the low segment. LINK does not load FORTRAN object code into the low segment if:

- 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.
- A high segment already exists.

The keywords for the /SEGMENT switch are:

Keyword	Description
DEFAULT	Suspends effect of /SEGMENT:LOW or /SEGMENT:HIGH.
HIGH	Loads into high segment.
LOW	Loads into low segment.
NONE	Same as DEFAULT.

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

/SEGMENT and /ONLY are incompatible.

EXAMPLES
```
@LINK<RET>
*/SEGMENT:LOW MOD1,MOD2,/SEGMENT:HIGH MOD3<RET>
*
```

Loads MOD1 and MOD2 into the low segment; loads MOD3 into the high segment.

RELATED /OTSEGMENT

SWITCHES

/SET

FORMAT /SET:name:address

name is .HIGH., .LOW., or a PSECT name.

address is a thirty-bit octal address or a defined symbol.

FUNCTION Sets the origin of a PSECT, or sets the .HIGH. or .LOW. relocation counter.

For setting the origin 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 make up the specified PSECT. The /SET switch is not needed if the REL files already contain the PSECT origin information.

EXAMPLES
```
@LINK<RET>
*/SET:MEMLOCA:200000<RET>
*
```

Specifies that the PSECT named MEMLOCA is to be loaded with an origin of 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 /COUNTERS, /LIMIT

/SEVERITY

FORMAT /SEVERITY:n

n is a positive decimal integer.

FUNCTION Specifies that errors of severity level greater than or equal to n terminate the load, where n is a decimal number between 0 and 31 inclusive. Level 31 errors always terminate the load.

The defaults are /SEVERITY:28 for timesharing jobs, and /SEVERITY:16 for batch jobs. See Appendix B for a list of severity codes.

EXAMPLES
```
@LINK<RET>
*/SEVERITY:29<RET>
*
```

Specifies that level 29 errors and above are fatal.

/SPACE

FORMAT /SPACE:n

n is a positive decimal integer.

FUNCTION Specifies that n words of memory follow the current overlay link at execution time. This memory allocation does not increase the size of the overlay file, but it does 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: such as 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 use 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 overlay 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
```
@LINK<RET>
*/OVERLAY<RET>
*TEST/SPACE:90/LINK:MAIN<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
```

symbol is thirty-bit octal address or a symbol.

FUNCTION

Specifies the start address for the loaded program, and prevents replacement by any start addresses found in later modules.

The /START switch without an argument disable a previously given /NOSTART switch.

LINK starts a program using a TOPS-10 style entry vector if the entry length vector is zero.

LINK starts a program using a TOPS-20 style entry vector:

- if the program contains nonzero sections and the length of the entry vector is not specified
- if the length of the entry vector is 1
- if /NOJOBDAT equals 1

See the TOPS-20 Monitor Calls Reference Manual for more information on entry vectors.

EXAMPLES

```
@LINK<RET>
*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 Suppresses a previously defined global symbol. If the symbol is unknown, this switch has no effect. Use this switch if the same global symbol is defined in two modules and you wish to suppress one of the definitions.

LINK suppresses a defined global symbol by setting its definition to undefined in the global symbol table. LINK does not remove the symbol definition from the symbol table. As a result, the symbol table built for debugging contains both the old and new values of the symbol.

Since LINK sets the symbol to undefined in the symbol table, it expects that a subsequent module will be loaded that contains a global definition for the symbol. If the symbol is not defined later, LINK issues the Undefined Global Symbol (LNKUGS) error.

EXAMPLES In the following example, the ENTPTR symbol is used in both the TEST and TEST2 programs. First, TEST is loaded, and the value of ENTPTR is show using the /VALUE switch.

```
@LINK<RET>
*TEST<RET>
*/VALUE:ENTPTR<RET>
[LNKVAL   Symbol ENTPTR 140 defined]
```

Next, ENTPTR's value is suppressed using /SUPPRESS and its current value is shown. Note that the value is now undefined.

```
*/SUPPRESS:ENTPTR<RET>
*/VALUE:ENTPTR<RET>
[LNKVAL   Symbol ENTPTR 0 undefined]
```

Finally TEST2 is loaded and the value is shown again.

```
*TEST2<RET>
*/VALUE:ENTPTR<RET>
[LNKVAL   Symbol ENTPTR 200 defined]
```

In the next example, TEST and TEST2 are loaded, but ENTPTR is not suppressed after TEST is loaded. In this example, LINK issues the Multiply-defined global symbol warning.

```
@LINK<RET>
*TEST<RET>
```

```
TEST2<RET>
%LNKMDS   Multiply-defined global symbol ENTPTR
     Detected in module .MAIN from file TEST2.REL.1
     Defined value = 140, this value = 200
*
```

/SYFILE

FORMAT filespec/SYFILE:keyword

FUNCTION Requests LINK to output a symbol file to the given file specification, 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, and has a SYM file type. If you used the /LOCALS switch, this file also contains local symbols, module names, and module lengths.

Keyword	Description
ALGOL	Requests symbols in ALGOL's format.
RADIX-50	Requests symbols in Radix-50 format.
TRIPLET	Requests symbols in triplet format.

See Chapter 4 for symbol table formats.

EXAMPLES
```
@LINK<RET>
*SYMBOL/SYFILE<RET>
*
```

Creates a symbol file called SYMBOL with the symbols in Radix-50 format and a file type of .SYM.

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

/SYMSEG

FORMAT /SYMSEG:keyword

FUNCTION Allows you to specify where the symbol table is to be placed.

Keyword	Description
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 at the end of the PSECT after allocating the patch space.

EXAMPLES
```
@LINK<RET>
*/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 enter a carriage-return ending the command.

LINK automatically searches a system library, by default, if code from the corresponding translator has been loaded. This search is performed after all the modules of the program are loaded. /SYSLIB forces this search to take place immediately rather than after all the program modules are loaded.

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

The keywords and the libraries they specify are listed below. Keywords shown in **bold** indicate libraries supported by DIGITAL.

Keyword	Library
ANY	**all**
ALGOL	**ALGLIB**
BCPL	BCPLIB
COBOL	**LIBOL or C74LIB**
FORTRAN	**FORLIB**
NELIAC	LIBNEL
PASCAL	PASLIB
SAIL	SAILIB
SIMULA	SIMLIB

EXAMPLES
```
@LINK<RET>
*MAIN/SYSLIB:COBOL<RET>
*SUB1<RET>
*SUB2/NOSYSLIB<RET>
*
```

OPTIONAL NOTATIONS You can omit the keyword. LINK searches all libraries for which corresponding code has been loaded.

RELATED SWITCHES /NOSYSLIB

/TEST

FORMAT /TEST:keyword

FUNCTION Loads the debugger 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.

Local symbols for the debugging module itself are never loaded.

The keywords and the programs they load are listed below. Only those shown in **bold** are supported by DIGITAL.

Keyword	Debugger
ALGDDT	**ALGDDT**
ALGOL	**ALGDDT**
COBDDT	**COBDDT**
COBOL	**COBDDT**
DDT	**DDT**
FAIL	SDDT (SAIL debugger)
FORDDT	**FORDDT**
FORTRAN	**FORDDT**
MACRO	**DDT**
PASCAL	PASDDT
PASDDT	PASDDT
SAIL	SDDT (SAIL debugger)
SDDT	SDDT (SAIL debugger)
SIMDDT	SIMDDT
SIMULA	SIMDDT

EXAMPLES
```
@LINK<RET>
*MYPROG/TEST:FORTRAN<RET>
*
```

Loads MYPROG and FORDDT.

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

You can abbreviate /TEST to /T.

RELATED /DDEBUG, /DEBUG

SWITCHES

/UNDEFINED

FORMAT /UNDEFINED

FUNCTION Displays undefined global symbols on the terminal. An undefined global symbol is one that LINK has not yet resolved. You can use /UNDEFINED to get a list of undefined symbols, and then define them with the /DEFINE switch.

EXAMPLES

```
@LINK<RET>
*/UNDEFINED<RET>
[LNKUGS  2 undefined global symbols]
     ALPHA   400123
     IOTA    402017
*/DEFINE:(ALPHA:591,IOTA:1)<RET>
*
```

Displays two undefined global symbols (ALPHA and IOTA), and defines a decimal value of 591 to ALPHA and a decimal value of 1 to IOTA.

OPTIONAL NOTATIONS You can abbreviate /UNDEFINE to /U.

RELATED SWITCHES /DEFINE, /VALUE

/UPTO

FORMAT /UPTO:address

address is a thirty-bit octal address or a defined symbol.

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

EXAMPLES
```
@LINK<RET>
*/UPTO:550000<RET>
*FORPRO<RET>
```

Defines a 550,000 upper limit for the FORTRAN symbol table. This switch overrides the default upper bound for the FORTOS 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 in the file specification before searching system libraries. The keyword indicates that the given library is to be searched only if code from the corresponding translator was loaded.

Keywords and their meanings are given below. Only those shown in **bold** are supported by DIGITAL.

Keyword	Library
ALGOL	**ALGOL**
ANY	**This library**
BCPL	BCPL
COBOL	**COBOL**
FORTRAN	**FORTRAN**
NELIAC	NELIAC
PASCAL	PASCAL
SAIL	SAIL
SIMULA	SIMULA

EXAMPLES
```
@LINK<RET>
*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, /SYSLIB

/VALUE

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

FUNCTION Displays the values of each specified global symbol on the terminal. LINK issues the LNKVAL message, giving the symbol, the symbol's current value, and symbol's status. 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

```
@LINK<RET>
*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]
*
```

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

/VERBOSITY

FORMAT /VERBOSITY:keyword

FUNCTION Specifies the length of LINK messages. This switch is similar to /MESSAGE. Both switches determine message length, but use different keywords and output.

Keyword	Description
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 In all the examples that follow an incorrect file specification (ABC:WRONG) is used. In the following example, only the 6-letter code and instructions for correcting the error is returned.

```
@LINK<RET>
*/VERBOSITY:SHORT
*ABC:WRONG<RET>
%LNKNED
[    Please retype the incorrect parts of the file
specification]
*
```

In the next example, a medium length message is returned.

```
@LINK<RET>
*/VERBOSITY:MEDIUM<RET>
*ABC:WRONG<RET>

%LNKNED   Non-existent device ABC:
[    Please retype the incorrect parts of the file specification]
*
```

In this example, a longer explanation is returned.

```
@LINK<RET>
/VERBOSITY:LONG<RET>
```

```
            *ABC:WRONG<RET>
            %LNKNED   Non-existent device ABC:
                 You gave a device that does  not  exist  on  this  system.
                 Correct your input files and reload.
            [    Please retype the incorrect parts of the file specification]
            *
```

RELATED SWITCHES /ERRORLEVEL, /MESSAGE

/VERSION

FORMAT /VERSION:ic(j)-k

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 Allows you to specify a version number. /VERSION changes the value of .JBVER (location 137 in JOBDAT) and .JBHVR in the vestigial job data area.

If the switch is appended to 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
```
@LINK<RET>
*/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.

OPTIONAL NOTATIONS You can abbreviate /VERSION to /V.

3.5 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. After running LINK, the command 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. Finally the /EXECUTE switch directs LINK to execute the loaded program, and the /GO switch tells LINK that there are no more commands.

```
@LINK<RET>
*MYPROG,MYLIB/USERLIB/NOSYSLIB/EXECUTE/GO<RET>
[LNKXCT     MYPROG execution]
This is written by MYPROG
CPU time 0.21   Elapsed time 1.31
@
```

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

```
@LINK<RET>
*/DEBUG:FORDDT MYPROG,MYLIB/USERLIB/NOSYSLIB/GO<RET>
[LNKDEB     FORDDT execution]

STARTING FORTRAN DDT

>>START<RET>
This is written by MYPROG
CPU time 0.17   Elapsed time 0.46
@
```

CHAPTER 4

OUTPUT FROM LINK

The primary output from LINK is the executable program formed from 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 consists mostly of data and machine instructions from your object modules. In the executable program, 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 passes control to your program for execution.

- Execute the program under the control of a debugger. 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 DEBUG to the system to enter the debugger.

- Save the executable as a sharable save file or 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:

- Sharable save file.
- Log file.
- Map file.
- Symbol file.
- Plotter file (see Section 5.1).
- Overlay file (see Section 5.1).

4.2.1 Sharable Save Files

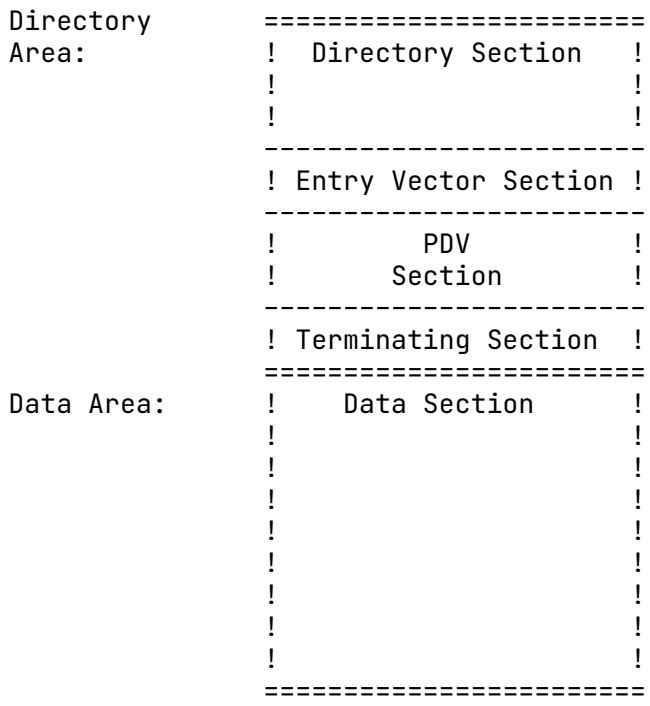
The sharable file, save file or EXE file, is a copy of the completed executable program generated by LINK. You can create a sharable save file by supplying the /SAVE switch before the /GO switch when you are loading the program with direct commands to LINK. The sharable save file retains the same file name as the source program, with a file type of EXE.

Alternatively, you can type the file specification, followed by /SAVE, and the sharable save file is written to the file you specified. If you load the program with the executable LOAD TOPS-20 command, you may then save the sharable save file by typing the TOPS-20 SAVE command.

You can run the sharable save file later, without running LINK, by using the TOPS-20 RUN command, or the two TOPS-20 GET and START commands. The following section describes the internal format of the sharable save file.

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

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



NOTE

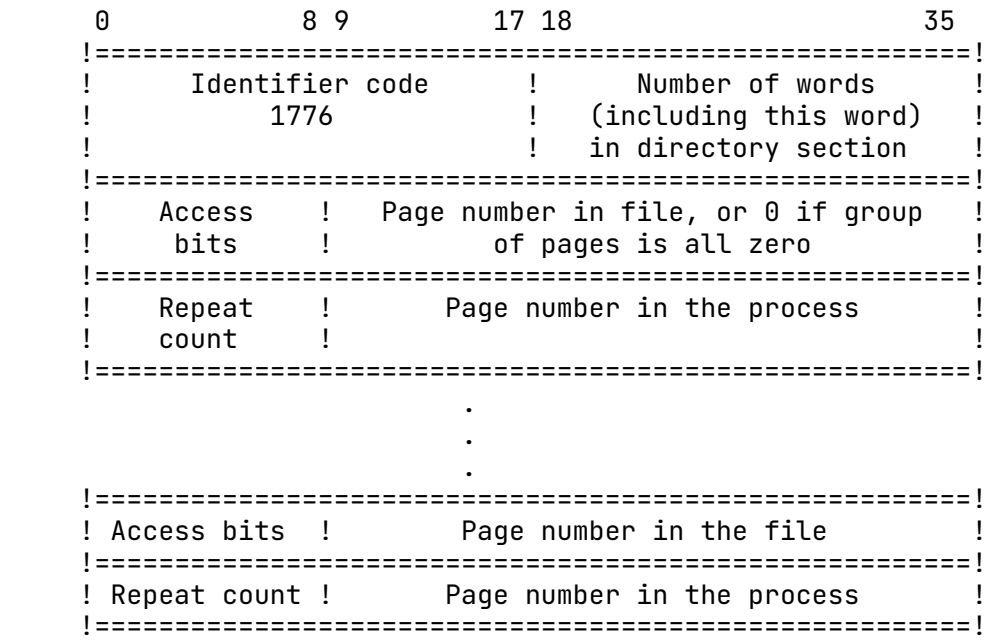
The PDV area is useful only with TOPS-20 version 5 and later monitors. Earlier monitors ignore this area. See Chapter 7 for more information on PDVs.

The directory area of the sharable save file has four distinct sections: the directory section, the entry vector section, the PDV 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.

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 four 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:



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.

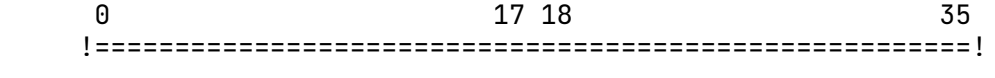
2. The pages have the same access.

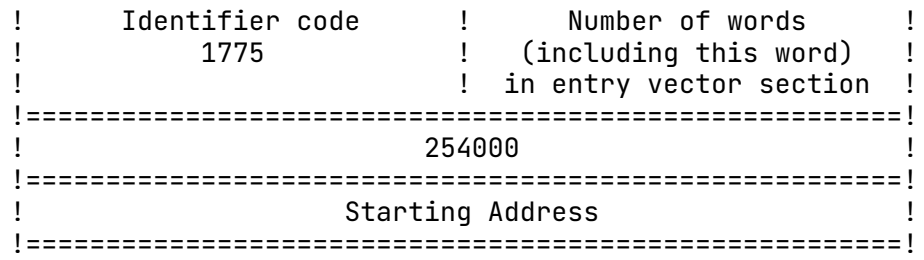
3. The pages are allocated but not loaded.

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

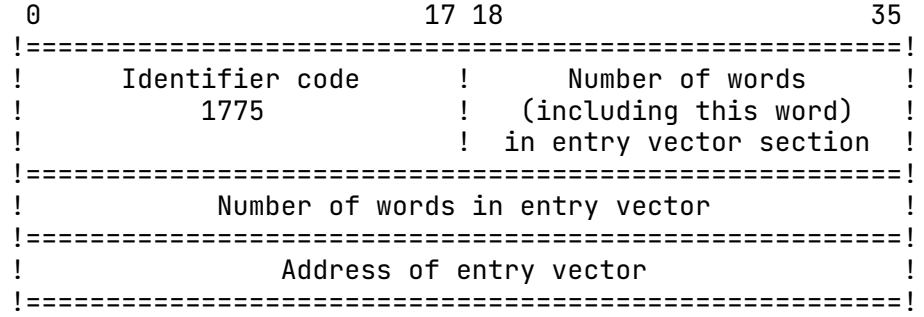
The word pairs are repeated for each group of pages in the address space.

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.



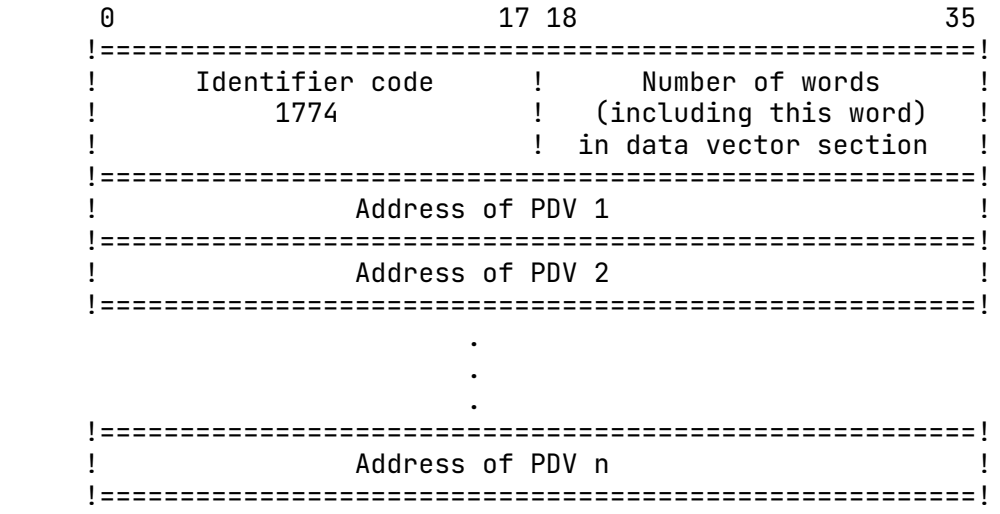


This format is the default. However, if you make special provisions in your program, the format becomes the following. (Refer to the description of Block Type 7 in Appendix A and the description of the SFRKV% JSYS in the TOPS-20 Monitor Calls Reference Manual for further information.)



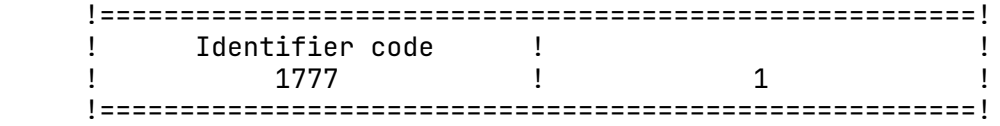
The data for this section is the address of the entry vector.

The PDV section may follow the entry vector section and contains the addresses at which the PDVs begin (PDVAs). This section is optional and only appears for an extended addressing program, if the program declared a PDV, or if /NOJOBDAT is used causing LINK to create a PDV. The format of the PDV section is as follows:



For information on the format of a PDV, see Chapter 7.

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



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

4.2.2 LOG Files

A LOG file, .LOG, 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, .MAP, 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, /NOINITIAL, and /NOSYMBOLS switches. For an example of a map file, see Section 5.1. For a list of /MAP options, refer to Chapter 3.

4.2.4 Symbol Files

The symbol file, 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.

The format of the symbol file is defined using /SYFILE:ALGOL, /SYFILE:RADIX-50, or /SYFILE:TRIPLET.

/SYSFILE:ALGOL creates a symbol file where the first word of the table is XWD 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.

/SYFILE:RADIX-50 creates a symbol file where 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. This is the default if just /SYFILE is used.

/SYFILE:TRIPLET creates internal format used by LINK.

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 /COUNTERS, /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 the message is 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:8 and /LOGLEVEL:8.

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. The /MESSAGE switch can also be used to specify message length. See Chapter 3 for more information on /MESSAGE and the other switches discussed in this section.

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

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.

Art work number: MRS-2595-83

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 summarized in Table 5-1 below and described in detail in Section 3.2.2. In addition to the overlay-related switches, you can use other LINK switches such as /ERRORLEVEL or /LOG when loading overlays.

Table 5-1: Summary of LINK's Overlay-Related Switches

Switch	Description

Table 5-1 (cont.)

Switch	Description
/ARSIZE	Sets the size of the overlay handler's table of multiply-defined global symbols.
/LINK	Closes an overlay link.
/MAXNODE	Specifies the number of links to be defined when the overlayed program requires more than 256 links.
/NODE	Opens an overlay link.
/NOREQUEST	Deletes references to links from LINK's overhead tables when loading overlay programs.
/OVERLAY	Initiates construction of an overlay structure.
/PLOT	Directs LINK to output a tree diagram of your overlay structure.
/PLTTYP	Specifies the type of plot file to be generated by the /PLOT switch.

The following example shows commands for defining the overlay diagrammed in Figure 5-1. Each command is followed by an explanation.

```
*TEST/LOG/LOGLEVEL:2
```

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.

```
*/ERRORLEVEL:5
```

Directs LINK to return messages of level 5 and above to the terminal.

```
*TEST/OVERLAY
```

Tells LINK that an overlay structure is to be defined, and that the file for the overlay is to be TEST.OVL.

```
*TEST/MAP
```

Defines the to contain symbol maps for each link.

```
*LPT:TEST/PLOT
```

Directs that a tree diagram of the overlay links be printed on the line printer.

```
*OVL0,OVL1/LINK:TEST
```

Loads the OVL0.REL and OVL1.REL files 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.

```
*/NODE:TEST OVL2/LINK:LEFT
```

Defines a link named LEFT. /NODE:TEST tells LINK that the link being defined (LEFT) is to be an immediate successor to TEST, the root link. OVL2/LINK:LEFT loads the OVL2.REL file, ends the link, and names it LEFT.

```
*/NODE:LEFT OVL5/LINK:LEFT1
```

Defines a link named LEFT1. /NODE:LEFT tells LINK that the link being defined (LINK1) is an immediate successor to LEFT. OVL5/LINK:LEFT1 loads the OVL5.REL file, ends the link, and names it LEFT1.

```
*/NODE:LEFT OVL6/LINK:LEFT2
```

Defines another immediate successor to LEFT named LEFT2.

```
*/NODE:TEST OVL3,OVL4/LINK:RIGHT
```

Defines the last link, RIGHT. /NODE:TEST defines RIGHT as an immediate successor to TEST, loads the OVL3.REL and OVL4.REL files, and names the link RIGHT.

```
*TEST/SAVE
```

Directs LINK to create the saved file TEST.EXE.

```
*/EXECUTE/GO
```

Specifies that the loaded program (TEST) is to be executed, and that all commands to LINK are completed.

This 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 samples of the files used in the previous example.

5.1.2.1 Source Files - Copies of the FORTRAN source files used in the overlay are displayed on the terminal using the TOPS-20 TYPE command.

```
@TYPE OVL0.FOR<RET>

CTHIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY ONLY BE USED
C  OR COPIED IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
C
CCOPYRIGHT (C) DIGITAL EQUIPMENT CORPORATION 1982, 1983

C       Simple overlay tests

C                        OVL0
C                       /    \
C                 OVL2A         OVL4
C               /     \               \
C          OVL5      OVL6            OVL1
C                       \           /
C                        OVL2B   OVL3

        IMPLICIT DOUBLE PRECISION (D)
        IMPLICIT COMPLEX (C)
        IMPLICIT INTEGER (A)

        TYPE 1
1       FORMAT ('1','Execution begins in main program OVL0')

        TYPE 11
11      FORMAT (1X,'OVL0 CALLS OVL2A')
        IVAR=0
        CALL OVL2A(IVAR)
        TYPE 2
2       FORMAT (/1X,'RETURN TO OVL0')
        IF (IVAR.NE.1) TYPE 100
100     FORMAT(' ?Error. Value not returned from OVL2A')

        TYPE 21
21      FORMAT (1X,'OVL0 CALLS OVL4')
        NUMB=0
        CALL OVL4(NUMB)
        TYPE 2
        IF (IVAR.NE.1) TYPE 200
200     FORMAT(' ?Error. Value not returned from NUMB')

        TYPE 3
3       FORMAT (/1X,'Execution ends in main program OVL0'//)

        STOP
        END
```

```
@TYPE OVL1.FOR


        SUBROUTINE OVL1(IVAR)

C THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY ONLY BE USED
C   OR COPIED IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
C
C COPYRIGHT (C) DIGITAL EQUIPMENT CORPORATION 1982, 1983

        IMPLICIT DOUBLE PRECISION (D)
        IMPLICIT COMPLEX (C)
        IMPLICIT INTEGER (A)

        TYPE 1
1       FORMAT (30x,'   OVL1 CALLS OVL3'/)

        NUM=0
        CALL OVL3(NUM)
        TYPE 2
2       FORMAT (30x,'   RETURN TO OVL1'/)
        IF (NUM.NE.1) TYPE 100
100     FORMAT(' ?Error. Variable not returned from OVL3.')

        IVAR=1
        RETURN
        END


@TYPE OVL2.FOR


        SUBROUTINE OVL2A(ARG)

C THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY ONLY BE USED
C   OR COPIED IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
C
C COPYRIGHT (C) DIGITAL EQUIPMENT CORPORATION 1982, 1983

        IMPLICIT DOUBLE PRECISION (D)
        IMPLICIT COMPLEX (C)
        IMPLICIT INTEGER (A)
        INTEGER OVL6

        TYPE 1
1       FORMAT (1X,'    OVL2A CALLS OVL5'/)
        ARG=0
        CALL OVL5(ARG)
        TYPE 2
2       FORMAT (1X,'    RETURN TO OVL2A'/)
        IF (ARG.NE.1) TYPE 100
100     FORMAT(' ?Error. Value not returned from OVL5')
```

```
        TYPE 3
3       FORMAT (1X,'    OVL2A CALLS OVL6'/)
        NUM=0
        NUM=OVL6(NUM)
        TYPE 2
        IF (NUM.NE.1) TYPE 300
300     FORMAT(' ?Error. Value not returned from OVL6')

        ARG=1           !Return value
        RETURN
        END

        SUBROUTINE OVL2B

        IMPLICIT DOUBLE PRECISION (D)
        IMPLICIT COMPLEX (C)
        IMPLICIT INTEGER (A)

        TYPE 1
1       FORMAT (/1X,'           OVL2B DOESN''T CALL ANYTHING')

        RETURN
        END


@TYPE OVL3.FOR

        SUBROUTINE OVL3(IVAR)

C THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY ONLY BE USED
C   OR COPIED IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
C
C COPYRIGHT (C) DIGITAL EQUIPMENT CORPORATION 1982, 1983

        IMPLICIT DOUBLE PRECISION (D)
        IMPLICIT COMPLEX (C)
        IMPLICIT INTEGER (A)

        IVAR=1  !Got here!

        TYPE 1
1       FORMAT (30x,'           OVL3 DOESN''T CALL ANYTHING'/)

        RETURN
        END


@TYPE OVL4.FOR

        SUBROUTINE OVL4(ARGU)

C THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY ONLY BE USED
```

```
C   OR COPIED IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
C
C COPYRIGHT (C) DIGITAL EQUIPMENT CORPORATION 1982, 1982

        IMPLICIT DOUBLE PRECISION (D)
        IMPLICIT COMPLEX (C)
        IMPLICIT INTEGER (A)

        TYPE 1
1       FORMAT (30x,'   OVL4 CALLS OVL1')
        NUM=0
        CALL OVL1(NUM)
        IF (NUM.NE.1) TYPE 100
100     FORMAT(' ?Error. Value not returned from OVL1')

        TYPE 2
2       FORMAT (30x,'   RETURN TO OVL4')

        ARGU=1
        RETURN
        END


@TYPE OVL5.FOR


        SUBROUTINE OVL5(NUM)

C THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY ONLY BE USED
C   OR COPIED IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
C
C COPYRIGHT (C) DIGITAL EQUIPMENT CORPORATION 1982, 1983

        IMPLICIT DOUBLE PRECISION (D)
        IMPLICIT COMPLEX (C)
        IMPLICIT INTEGER (A)
        COMMON C

        C=(1,1)
        C=C**C
        TYPE 1
1       FORMAT (30x,'           OVL5 CALLS CEXP3.'/)

        NUM=1
        RETURN
        END



@TYPE OVL6.FOR

        INTEGER FUNCTION OVL6(ARG)
```

```
C THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY ONLY BE USED
C   OR COPIED IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE.
C
C COPYRIGHT (C) DIGITAL EQUIPMENT CORPORATION 1982, 1983

        IMPLICIT DOUBLE PRECISION (D)
        IMPLICIT COMPLEX (C)
        IMPLICIT INTEGER (A)

        TYPE 1
1       FORMAT (30x,'   OVL6 CALLS OVL2B'/)
        NUM=0
        CALL OVL2B(NUM)
        TYPE 2
2       FORMAT (30x,'   RETURN TO OVL6'/)
        IF (NUM .NE. 0) TYPE 100
100     FORMAT(' ?Error. Value not returned from OVL2A')

        OVL6=1
        RETURN
        END
```

5.1.2.2 Source File Compilation - Source files are compiled using the TOPS-20 COMPILE command.

```
@compile ovl0,ovl1,ovl2,ovl3,ovl4,ovl5,ovl6
FORTRAN: OVL0
OVL0
FORTRAN: OVL1
OVL1
FORTRAN: OVL2
OVL2A
OVL2B
FORTRAN: OVL3
OVL3
FORTRAN: OVL4
OVL4
FORTRAN: OVL5
OVL5
FORTRAN: OVL6
OVL6
```

5.1.2.3 Interactive use of LINK - LINK is run and used to define and execute the overlay.

```
@LINK
TEST/LOG/LOGLEVEL:5
/ERRORLEVEL:5
```

```
TEST/OVERLAY
TEST/MAP
LPT:TEST/PLOT
OVL0,OVL1/LINK:TEST
[LNKLMN Loading module OVL0 from file OVL0.REL.1]
[LNKLMN Loading module OVL1 from file OVL1.REL.1]
[LNKLMN Loading module OVRLAY from file SYS:OVRLAY.REL.10]
[LNKLMN Loading module FORHST from file SYS:FORLIB.REL.1]
[LNKLMN Loading module FORSHR from file SYS:FORLIB.REL.1]
[LNKLMN Loading module FORGET from file SYS:FORLIB.REL.1]
[LNKLMN Loading module FDBDUM from file SYS:FORLIB.REL.1]
[LNKLMN Loading module KSORT from file SYS:FORLIB.REL.1]
[LNKLMN Loading module FORMSL from file SYS:FORLIB.REL.1]
[LNKLMN Loading module FORPSE from file SYS:FORLIB.REL.1]
[LNKELN End of link number 0 name TEST]
NODE:TEST OVL2/LINK:LEFT
[LNKLMN Loading module OVL2A from file OVL2.REL.1]
[LNKLMN Loading module OVL2B from file OVL2.REL.1]
[LNKELN End of link number 1 name LEFT]
/NODE:LEFT OVL5/LINK:LEFT1
[LNKLMN Loading module OVL5 from file OVL5.REL.1]
[LNKLMN Loading module CEXP3. from file SYS:FORLIB.REL.1]
[LNKLMN Loading module CDLOG from file SYS:FORLIB.REL.1]
[LNKLMN Loading module CGLOG from file SYS:FORLIB.REL.1]
[LNKLMN Loading module CGLOG from file SYS:FORLIB.REL.1]
[LNKLMN Loading module GATAN. from file SYS:FORLIB.REL.1]
[LNKLMN Loading module GLOG. from file SYS:FORLIB.REL.1]
[LNKLMN Loading module DATAN. from file SYS:FORLIB.REL.1]
[LNKLMN Loading module DEXP. from file SYS:FORLIB.REL.1]
[LNKLMN Loading module DLOG. from file SYS:FORLIB.REL.1]
[LNKLMN Loading module DSIN. from file SYS:FORLIB.REL.1]
[LNKLMN Loading module DFL.4 from file SYS:FORLIB.REL.1]
[LNKLMN Loading module EXP2. from file SYS:FORLIB.REL.1]
[LNKLMN Loading module MTHMSG from file SYS:FORLIB.REL.1]
[LNKELN End of link number 2 name LEFT1]
/NODE:LEFT OVL6/LINK:LEFT2
[LNKLMN Loading module OVL6 from file OVL6.REL.1]
[LNKELN End of link number 3 name LEFT2]
/NODE:TEST OVL3,OVL4/LINK:RIGHT
[LNKLMN Loading module OVL3 from file OVL3.REL.1]
[LNKLMN Loading module OVL4 from file OVL4.REL.1]
[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 CALLS CEXP3.
```

```
        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


CPU time 0.85   Elapsed time 11.85
```

5.1.2.4 TEST.LOG - LINK generated the following TEST.LOG file. It shows the log messages issued during the load.

```
15:30:48   6   1   LMN  Loading module OVL0 from file OVL0.REL.1
15:30:49   6   1   LMN  Loading module OVL1 from file OVL1.REL.1
15:30:50   6   1   LMN  Loading module OVRLAY from file SYS:OVRLAY.REL.10
15:30:52   6   1   LMN  Loading module FORHST from file SYS:FORLIB.REL.1
15:30:52   6   1   LMN  Loading module FORSHR from file SYS:FORLIB.REL.1
15:30:52   6   1   LMN  Loading module FORGET from file SYS:FORLIB.REL.1
15:30:53   6   1   LMN  Loading module FDBDUM from file SYS:FORLIB.REL.1
15:30:53   6   1   LMN  Loading module KSORT from file SYS:FORLIB.REL.1
15:30:54   6   1   LMN  Loading module FORMSL from file SYS:FORLIB.REL.1
15:30:54   6   1   LMN  Loading module FORPSE from file SYS:FORLIB.REL.1
15:31:23   7   1   ELN  End of link number 0 name TEST

15:31:42   6   1   LMN  Loading module OVL2A from file OVL2.REL.1
15:31:42   6   1   LMN  Loading module OVL2B from file OVL2.REL.1
15:31:48   7   1   ELN  End of link number 1 name LEFT
```

```
15:32:04   6   1   LMN  Loading module OVL5 from file OVL5.REL.1
15:32:06   6   1   LMN  Loading module CEXP3. from file SYS:FORLIB.REL.1
15:32:06   6   1   LMN  Loading module CDLOG from file SYS:FORLIB.REL.1
15:32:07   6   1   LMN  Loading module CGLOG from file SYS:FORLIB.REL.1
15:32:07   6   1   LMN  Loading module CGLOG from file SYS:FORLIB.REL.1
15:32:07   6   1   LMN  Loading module GATAN. from file SYS:FORLIB.REL.1
15:32:08   6   1   LMN  Loading module GLOG. from file SYS:FORLIB.REL.1
15:32:09   6   1   LMN  Loading module DATAN. from file SYS:FORLIB.REL.1
15:32:09   6   1   LMN  Loading module DEXP. from file SYS:FORLIB.REL.1
15:32:09   6   1   LMN  Loading module DLOG. from file SYS:FORLIB.REL.1
15:32:09   6   1   LMN  Loading module DSIN. from file SYS:FORLIB.REL.1
15:32:10   6   1   LMN  Loading module DFL.4 from file SYS:FORLIB.REL.1
15:32:10   6   1   LMN  Loading module EXP2. from file SYS:FORLIB.REL.1
15:32:10   6   1   LMN  Loading module MTHMSG from file SYS:FORLIB.REL.1
15:32:12   7   1   ELN  End of link number 2 name LEFT1

15:32:36   6   1   LMN  Loading module OVL6 from file OVL6.REL.1
15:32:37   7   1   ELN  End of link number 3 name LEFT2

15:32:55   6   1   LMN  Loading module OVL3 from file OVL3.REL.1
15:32:56   6   1   LMN  Loading module OVL4 from file OVL4.REL.1
15:32:56   7   1   ELN  End of link number 4 name RIGHT
```

5.1.2.5 TEST.MAP - LINK generated the following TEST.MAP file. It shows symbol maps for the overlay.

```
              LINK symbol map of      OVL0            page 1
        Produced by LINK version 6(2353) on 11-Feb-85 at 11:53:18

        Overlay no.     0       name    TEST
        Overlay is absolute
        Low  segment starts at       0 ends at   10243 length    10244 =   9P
        Control Block address is  10202, length     32 (octal), 26. (decimal)
        348 words free in Low segment
        188 Global symbols loaded, therefore min. hash size is 209
        Start address is    235, located in program OVL0

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

JOBDAT-INITIAL-SYMBOLS
        %JOBDT      43200000454 Global  Absolute    Suppressed  .JB41              41 Gl
        .JBAPR              125 Global  Absolute                .JBBLT              45 Gl
        .JBCHN              131 Global  Absolute                .JBCNI             126 Gl
        .JBCOR              133 Global  Absolute                .JBCST             136 Gl
        .JBDA               140 Global  Absolute                .JBDDT              74 Gl
        .JBEDV              112 Global  Absolute                .JBERR              42 Gl
        .JBFF               121 Global  Absolute                .JBH41               1 Gl
        .JBHCR                2 Global  Absolute    Suppressed  .JBHDA              10 Gl
        .JBHGA                7 Global  Absolute    Suppressed  .JBHGH          400000 Gl
        .JBHNM                5 Global  Absolute    Suppressed  .JBHRL             115 Gl
```

```
        .JBHRN              3 Global  Absolute    Suppressed  .JBHSA              0 Gl
        .JBHSM              6 Global  Absolute    Suppressed  .JBHVR              4 Gl
        .JBINT            134 Global  Absolute                .JBOPC            130 Gl
        .JBOPS            135 Global  Absolute                .JBOVL            131 Gl
        .JBPFH            123 Global  Absolute                .JBPFI             74 Gl
        .JBREL             44 Global  Absolute                .JBREN            124 Gl
        .JBSA             120 Global  Absolute                .JBSYM            116 Gl
        .JBTPC            127 Global  Absolute                .JBUSY            117 Gl
        .JBUUO             40 Global  Absolute                .JBVER            137 Gl

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

OVL0    from DSK:OVL0.REL       created by FORTRAN on 11-Feb-85 at 11:52:27
        Low  segment starts at     140 ends at     234 length       75 (octal),    61. (de
        High segment starts at     235 ends at     345 length      111 (octal),    73. (de

        MAIN.             235 Global  Relocatable            OVL0              235 En

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

OVL1    from DSK:OVL1.REL       created by FORTRAN on 11-Feb-85 at 11:52:28
        Low  segment starts at     346 ends at     377 length       32 (octal),    26. (de
        High segment starts at     400 ends at     442 length       43 (octal),    35. (de

        OVL1              401 Entry   Relocatable            FOROT%         400010 Gl

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

OVRLAY  from SYS:OVRLAY.REL     created by MACRO on 25-Jan-85 at  5:25:39
        Low  segment starts at    4252 ends at    5245 length      774 (octal),   508. (de
        High segment starts at     443 ends at    4150 length     3506 (octal),  1862. (de


               LINK symbol map of      OVL0            page 2
OVRLAY
        BOUT%      104000000051 Global  Absolute              CLOSF%     104000000022 Gl
        CLROV.             2501 Entry   Relocatable           ERJMP      320700000000 Gl
        ERSTR%     104000000011 Global  Absolute              GCVEC%     104000000300 Gl
        GETOV.             2376 Entry   Relocatable           GJ%FOU     400000000000 Gl
        GJ%OLD     100000000000 Global  Absolute    Suppressed  GJ%SHT          1000000 Gl
        GJ%TMP      10000000000 Global  Absolute    Suppressed  GTJFN%     104000000020 Gl
        HALTF%     104000000170 Global  Absolute              INIOV.             2354 En
        JFNS%      104000000030 Global  Absolute              JS%DEV     300000000000 Gl
        JS%DIR      70000000000 Global  Absolute    Suppressed  JS%NAM       7000000000 Gl
        JS%PAF              1 Global  Absolute    Suppressed  LOGOV.             2520 En
        OF%BSZ     770000000000 Global  Absolute    Suppressed  OF%RD            200000 Gl
        OF%WR           100000 Global  Absolute    Suppressed  OPENF%     104000000021 Gl
        PA%EX       20000000000 Global  Absolute    Suppressed  PA%PRV        200000000 Gl
        PBOUT%     104000000074 Global  Absolute              REMOV.             2412 En
        RMAP%      104000000061 Global  Absolute              RPACS%     104000000057 Gl
        RUNOV.             2430 Entry   Relocatable           RUNTM%     104000000015 Gl
        SAVOV.             2454 Entry   Relocatable           SFPTR%     104000000027 Gl
```

```
        SIN%       104000000052 Global  Absolute                SOUT%      104000000053 Gl
        %OVRLA        501000212 Global  Absolute    Suppressed  .FHJOB           777773 Gl
        .FHSLF           400000 Global  Absolute    Suppressed  .GJEXT                5 Gl
        .GJGEN                0 Global  Absolute    Suppressed  .JSAOF                1 Gl
        .NULIO           377777 Global  Absolute    Suppressed  .OVRLA             4273 En
        .OVRLO             4314 Global  Relocatable             .OVRLU             3254 En
        .OVRWA             4313 Global  Relocatable
               *************

FORHST  from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29

        Zero length module

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

FORSHR  from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29

        Zero length module

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

FORGET  from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        Low  segment starts at    5725 ends at    7134 length     1210 (octal),   648. (de
        High segment starts at    5246 ends at    5724 length      457 (octal),   303. (de

        ABORT.             5603 Entry   Relocatable
        ALCHN.             5557 Entry   Relocatable             ALCOR.             5553 En
        CFIBF%     104000000100 Global  Absolute                CHRPT.             5725 Gl
        CLOSE.             5515 Entry   Relocatable             DEC.               5533 En
        DECHN.             5561 Entry   Relocatable             DECOR.             5555 En
        ENC.               5531 Entry   Relocatable             ERROR.             5511 En
        ESOUT%     104000000313 Global  Absolute                EXIT.              5551 En
        EXIT1.             5517 Entry   Relocatable             FDBMS.             5567 En
        FIN.               5543 Entry   Relocatable             FIND.              5547 En
        FORER.             5511 Entry   Relocatable             FOROP.             5573 En
        FOROT$             5610 Entry   Relocatable             FOROT.             5246 En
        FUNCT.             5565 Entry   Relocatable             GET%       104000000200 Gl

               LINK symbol map of      OVL0            page 3
FORGET
        GEVEC%     104000000205 Global  Absolute                GJ%PHY          10000000 Gl
        GT%NOV            40000 Global  Absolute    Suppressed  IFI.               5575 En
        IFO.               5577 Entry   Relocatable             IN.                5521 En
        INIT.              5507 Entry   Relocatable             INQ.               5571 En
        IOLST.             5541 Entry   Relocatable             MTHER.             5601 En
        MTOP.              5545 Entry   Relocatable             NLI.               5535 En
        NLO.               5537 Entry   Relocatable             OPEN.              5513 En
        OUT.               5523 Entry   Relocatable             PA%PEX       10000000000 Gl
        PM%CNT     400000000000 Global  Absolute    Suppressed  PM%RD      100000000000 Gl
        PM%RWX     160000000000 Global  Absolute    Suppressed  PM%WR       40000000000 Gl
        PMAP%      104000000056 Global  Absolute                PSOUT%     104000000076 Gl
        RD%JFN       4000000000 Global  Absolute    Suppressed  RD%TOP     200000000000 Gl
```

```
        RF%LNG     400000000000 Global  Absolute    Suppressed  RFSTS%     104000000156 Gl
        RTB.               5525 Entry   Relocatable             SEVEC%     104000000204 Gl
        TEXTI%     104000000524 Global  Absolute                TRACE.             5563 En
        WTB.               5527 Entry   Relocatable             .PRIIN              100 Gl
        .PRIOU              101 Global  Absolute    Suppressed  .RFSFL                4 Gl

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

FDBDUM  from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        High segment starts at    7135 ends at    7135 length        1 (octal),     1. (de

        FDBMS%             7135 Global  Relocatable            KDBMS.             7135 Gl
        %DBSTP             7135 Entry   Relocatable
                ************

KSORT   from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        Low  segment starts at    7152 ends at    7154 length        3 (octal),     3. (de
        High segment starts at    7136 ends at    7151 length       14 (octal),    12. (de

        KSORT.             7136 Entry   Relocatable
        %PASRT             7154 Global  Relocatable            %SRTAD             7152 Gl

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

FORMSL  from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        High segment starts at    7155 ends at    7714 length      540 (octal),   352. (de

        F.AQS              7267 Entry   Relocatable            F.CFS              7360 En
        F.CGP              7333 Entry   Relocatable            F.CGS              7367 En
        F.CLE              7233 Entry   Relocatable            F.CPP              7376 En
        F.CWU              7223 Entry   Relocatable            F.ICE              7242 En
        F.IDU              7173 Entry   Relocatable            F.IJE              7424 En
        F.IOE              7155 Entry   Relocatable            F.IPN              7405 En
        F.MXD              7342 Entry   Relocatable            F.NCA              7260 En
        F.NCS              7251 Entry   Relocatable            F.NOF              7213 En
        F.NOR              7164 Entry   Relocatable            F.NSS              7351 En
        F.SNH              7415 Entry   Relocatable            F.SRE              7311 En
        F.SSE              7276 Entry   Relocatable            F.TMA              7324 En
        F.UNO              7203 Entry   Relocatable
                ************


                LINK symbol map of      OVL0            page 4


FORPSE  from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        Low  segment starts at   10121 ends at   10201 length       61 (octal),    49. (de
        High segment starts at    7715 ends at   10120 length      204 (octal),   132. (de

        PAUS.              7716 Entry   Relocatable
        RFMOD%     104000000107 Global  Absolute                SFMOD%     104000000110 Gl
        STOP.              7721 Entry   Relocatable            TT%OSP     400000000000 Gl
```

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


        Index to LINK symbol map of     OVL0            page 5

        Name    Page    Name    Page    Name    Page    Name    Page

        FDBDUM   3      FORMSL   3      KSORT    3      OVL1     1
        FORGET   2      FORPSE   4      OVL0     1      OVRLAY   1
        FORHST   2      FORSHR   2


                LINK symbol map of      OVL0    #1      page 6

        Overlay no.     1       name    LEFT
        Overlay is absolute
        Low  segment starts at   14244 ends at   14501 length      236 =   1P
        Control Block address is  14440, length     30 (octal), 24. (decimal)
        Path is 0
        190 words free in Low segment
        6 Global symbols loaded, therefore min. hash size is 7

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

OVL2A   from DSK:OVL2.REL       created by FORTRAN on 11-Feb-85 at 11:52:30
        Low  segment starts at   14244 ends at   14315 length       52 (octal),    42. (de
        High segment starts at   14316 ends at   14412 length       75 (octal),    61. (de

        OVL2A             14317 Entry   Relocatable

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

OVL2B   from DSK:OVL2.REL       created by FORTRAN on 11-Feb-85 at 11:52:30
        Low  segment starts at   14413 ends at   14425 length       13 (octal),    11. (de
        High segment starts at   14426 ends at   14437 length       12 (octal),    10. (de

        OVL2B             14427 Entry   Relocatable

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

                LINK symbol map of      OVL0    #2      page 7

        Overlay no.     2       name    LEFT1
        Overlay is absolute
        Low  segment starts at   14502 ends at   22675 length     6174 =   7P
        Control Block address is  22510, length     20 (octal), 16. (decimal)
        Path is 0, 1
        66 words free in Low segment
        53 Global symbols loaded, therefore min. hash size is 59

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

```
OVL5    from DSK:OVL5.REL       created by FORTRAN on 11-Feb-85 at 11:52:36
        Low  segment starts at   14502 ends at   14517 length       16 (octal),    14. (de
        High segment starts at   14520 ends at   14547 length       30 (octal),    24. (de

        .COMM.           14502 Common  length       2.         OVL5             14521 En

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

CEXP3.  from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        Low  segment starts at   16062 ends at   16144 length       63 (octal),    51. (de
        High segment starts at   14550 ends at   16061 length     1312 (octal),   714. (de

        CEXP2.           14551 Entry   Relocatable             CEXP3.           14655 En

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

CDLOG   from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        Low  segment starts at   16515 ends at   16544 length       30 (octal),    24. (de
        High segment starts at   16145 ends at   16514 length      350 (octal),   232. (de

        CDLOG            16146 Entry   Relocatable

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

CGLOG   from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29

        Zero length module

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

CGLOG   from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        Low  segment starts at   17112 ends at   17145 length       34 (octal),    28. (de
        High segment starts at   16545 ends at   17111 length      345 (octal),   229. (de

        CGLOG.           16546 Entry   Relocatable

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

GATAN.  from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        Low  segment starts at   17715 ends at   17723 length        7 (octal),     7. (de
        High segment starts at   17146 ends at   17714 length      547 (octal),   359. (de

        GATAN.           17147 Entry   Relocatable             GATN2.           17242 En

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

               LINK symbol map of      OVL0    #2      page 8


GLOG.   from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        Low  segment starts at   20075 ends at   20102 length        6 (octal),     6. (de
        High segment starts at   17724 ends at   20074 length      151 (octal),   105. (de
```

```
        GLG10.            17725 Entry   Relocatable             GLOG.             17730 En

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

DATAN.  from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        Low  segment starts at   20652 ends at   20660 length        7 (octal),     7. (de
        High segment starts at   20103 ends at   20651 length      547 (octal),   359. (de

        DATAN.            20104 Entry   Relocatable             DATN2.            20177 En

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

DEXP.   from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        Low  segment starts at   21042 ends at   21044 length        3 (octal),     3. (de
        High segment starts at   20661 ends at   21041 length      161 (octal),   113. (de

        DEXP.             20662 Entry   Relocatable
                ************

DLOG.   from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        Low  segment starts at   21217 ends at   21222 length        4 (octal),     4. (de
        High segment starts at   21045 ends at   21216 length      152 (octal),   106. (de

        DLG10.            21046 Entry   Relocatable
        DLOG.             21051 Entry   Relocatable
                ************

DSIN.   from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        Low  segment starts at   21450 ends at   21454 length        5 (octal),     5. (de
        High segment starts at   21223 ends at   21447 length      225 (octal),   149. (de

        DCOS.             21224 Entry   Relocatable
        DSIN.             21236 Entry   Relocatable
                ************

DFL.4   from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        High segment starts at   21455 ends at   21464 length       10 (octal),     8. (de

        DFL.4             21456 Entry   Relocatable

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

EXP2.   from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        Low  segment starts at   21606 ends at   21610 length        3 (octal),     3. (de
        High segment starts at   21465 ends at   21605 length      121 (octal),    81. (de

        EXP2.             21466 Entry   Relocatable
                ************


                LINK symbol map of      OVL0    #2      page 9
```

```
MTHMSG  from SYS:FORLIB.REL     created by MACRO on 25-Jan-85 at  5:24:29
        High segment starts at   21611 ends at   22507 length      677 (octal),   447. (de

        L.888             22106 Entry   Relocatable
        L.999             22077 Entry   Relocatable             L.AII             21762 En
        L.AIR             21753 Entry   Relocatable             L.AIZ             21735 En
        L.AOI             22043 Entry   Relocatable             L.ARZ             21744 En
        L.ATI             21710 Entry   Relocatable             L.ATZ             21701 En
        L.AZM             21771 Entry   Relocatable             L.BAZ             22052 En
        L.BPI             22016 Entry   Relocatable             L.BPO             21663 En
        L.BPU             21672 Entry   Relocatable             L.IPO             21645 En
        L.IPU             21627 Entry   Relocatable             L.MFX             22124 En
        L.NAA             22025 Entry   Relocatable             L.NNA             22061 En
        L.NOR             22115 Entry   Relocatable             L.ROV             21611 En
        L.RPO             21654 Entry   Relocatable             L.RPU             21636 En
        L.RTI             21717 Entry   Relocatable             L.RTR             21726 En
        L.RUN             21620 Entry   Relocatable             L.ZCI             22007 En
        L.ZIZ             22034 Entry   Relocatable             L.ZNI             22070 En
        L.ZZZ             22000 Entry   Relocatable
               *************


        Index to LINK symbol map of     OVL0    #2      page 10

        Name    Page    Name    Page    Name    Page    Name    Page

        CDLOG    7      DATAN.   8      DSIN.    8      GLOG.    8
        CEXP3.   7      DEXP.    8      EXP2.    8      MTHMSG   9
        CGLOG    7      DFL.4    8      GATAN.   7      OVL5     7
        CGLOG    7      DLOG.    8


               LINK symbol map of      OVL0    #3      page 11

        Overlay no.     3       name    LEFT2
        Overlay is absolute
        Low  segment starts at   14502 ends at   14666 length      165 =   1P
        Control Block address is  14637, length     20 (octal), 16. (decimal)
        Path is 0, 1
        73 words free in Low segment
        4 Global symbols loaded, therefore min. hash size is 5

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

OVL6    from DSK:OVL6.REL       created by FORTRAN on 11-Feb-85 at 11:52:37
        Low  segment starts at   14502 ends at   14550 length       47 (octal),    39. (de
        High segment starts at   14551 ends at   14636 length       66 (octal),    54. (de

        OVL6              14552 Entry   Relocatable

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

```
        LINK symbol map of      OVL0    #4      page 12

Overlay no.     4       name    RIGHT
Overlay is absolute
Low  segment starts at   14244 ends at   14426 length      163 =   1P
Control Block address is  14375, length     22 (octal), 18. (decimal)
Path is 0
233 words free in Low segment
5 Global symbols loaded, therefore min. hash size is 6

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

OVL3    from DSK:OVL3.REL       created by FORTRAN on 11-Feb-85 at 11:52:32
        Low  segment starts at   14244 ends at   14257 length       14 (octal),    12. (de
        High segment starts at   14260 ends at   14277 length       20 (octal),    16. (de

        OVL3              14261 Entry   Relocatable
        *************

OVL4    from DSK:OVL4.REL       created by FORTRAN on 11-Feb-85 at 11:52:34
        Low  segment starts at   14300 ends at   14331 length       32 (octal),    26. (de
        High segment starts at   14332 ends at   14374 length       43 (octal),    35. (de

        OVL4              14333 Entry   Relocatable

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



        Index to overlay numbers of OVL0                page 13

        Overlay Page    Overlay Page    Overlay Page    Overlay Page

        #0       5      #2      10      #3      11      #4      12
        #1       6

        Index to overlay names of OVL0

        Name    Page    Name    Page    Name    Page    Name    Page

        LEFT     6      LEFT2   11      RIGHT   12      TEST     5
        LEFT1   10

        [End of LINK map of     OVL0]
```

5.1.2.6 Tree Diagram - The tree diagram requested by the LPT:TEST/PLOT switch.

5.1.2.7 Executable File – 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 are 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.

To specify that an overlay is writable, use the FORTRAN SAVE statement in the program, and specify /OVERLAY:WRITABLE when loading the program with LINK.

5.2.1 Writable Overlay Syntax

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

```
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:

```
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 your terminal:

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

5.4 RESTRICTIONS ON OVERLAYS

The following restrictions apply to all overlaid programs:

- Overlaid programs cannot be run execute-only.
- Overlaid programs cannot use PDVs.
- PSECTed programs cannot be overlaid.

- Overlaid 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 allocates extra space in the the OVL file for tables that require it, based on the number of links you specify.

- If the program uses argument checking, the callee argument checking block must be seen in the same overlay node as the definition of the global symbol of the same name.

- If the program uses character fixup, the argument block and the character descriptor must be in the same segment (low or high), and they must both be either relocatable or absolute.

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 ABC##+XYZ where ABC is in another overlay.

   - Left-hand fixups, in the form XWD ABC##,XYZ, where ABC is in another overlay.

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

   - Similarly, MOVEI 1,ABC##, where ABC 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,ABC##.

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 DIGITAL compiler produces such expressions. MACRO programmers should avoid using them in modules that are to be loaded as part of an overlaid program. Any expression that causes MACRO to generate a Polish fixup block is not properly relocated at runtime. The following are examples of such complex expressions:

```
MOVEI 1,A##+B##+C##
A,,0
```

5.4.3 Restrictions on FORTRAN Overlays

The following restrictions apply 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 produces unpredictable results. Moreover, the value of the variable is reset to zero each time its overlay link is removed from memory. Only variables declared in routines that are loaded into the root link are always resident. However, variables declared in COMMON and in the root link are always 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 or its successors.

- 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 of which you want to know the size, but before the next /NODE switch. For example,

```
*OVLAY/LINK
*/COUNTER
*/NODE:TOP
```

This gives you the size of the program when the overlay is actually loaded into memory. The display includes 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 OVERLAID PROGRAMS

COBDDT and ALGDDT can be used to debug overlay programs, but FORDDT cannot. To use DDT with an overlaid 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.

RESET. in FOROTS removes the symbol table when a FORTRAN program starts running. 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. The DDT commands $X and $$X do not work. Set a breakpoint after RESET. if you are debugging a root link, and use the DDT command $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 or later 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 501000210 defined]
*
```

The left halfword of $OVRLA contains the version number of the overlay handler, and should be 501, corresponding to the Version 6 of LINK. The right halfword is the edit number, and should be 000210 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 is 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 are 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,addr1                     ;for first ASCIZ linkname
                .
                .
                .
          Z 17,addrn                     ;for nth ASCIZ linkname

                         OR

          -n,,0                          ;n is number of arguments
arglst:   Z 2,addr1                      ;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,addr1
                .
                .
                .
```

```
            Z 17,addrn

                          OR

            -n,,0                          ;n is number of arguments
arglst:     Z 2,addr1
                  .
                  .
                  .
            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 is 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 39-character device, a 39-character filename, a 39-character file type, and a 39-character directory.

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 39-character device, a 39-character filename, a 39-character file type, and a 39-character directory.

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)
```

You can omit the (0) argument to close the file.

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,addr1
                .
                .
                .
          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 is 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. LINK messages, 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 is printed only if you use the /OVERLAY:LOG switch.

The severity of a message determines whether the job is 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 is returned from the object-time system and depends on the particular FUNCT. function the overlay handler used. See Section 5.7.4 for the FUNCT. function codes and status messages.

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.

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.

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.

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 requests 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	RRS	Read retain status (DBMS)
14	WRS	Write retain status (DBMS)
15	GPG	Get pages
16	RPG	Return pages
17	GPSI	Get TOPS-20 PSI channel
20	RPSI	Return TOPS-20 PSI channel
21	MPG	Get contiguous set of pages

All FUNCT. codes are reserved to DIGITAL.

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.

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. GPSI arranges 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.

MPG (21) Function

This function gets a contiguous set of pages. The pages requested are always allocated from the section FOROTS is in. The user cannot depend upon this call to either create or destroy the pages.

arg 1	first page number to allocate. The page number must be in the range 0 to 777.
arg 2	number of pages to allocate

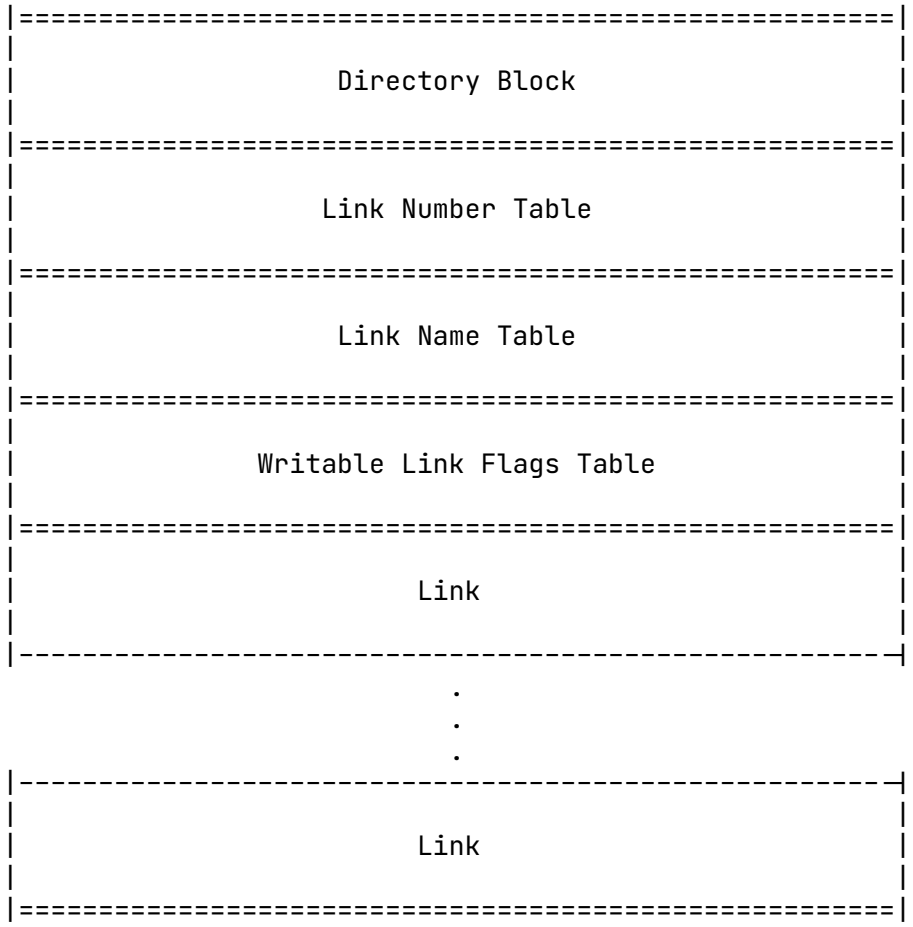
On return, the status is one of the following:

0	successful deallocation of all given pages
1	one or more pages was not allocated through MPG
3	illegal argument (bad page number count)

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:

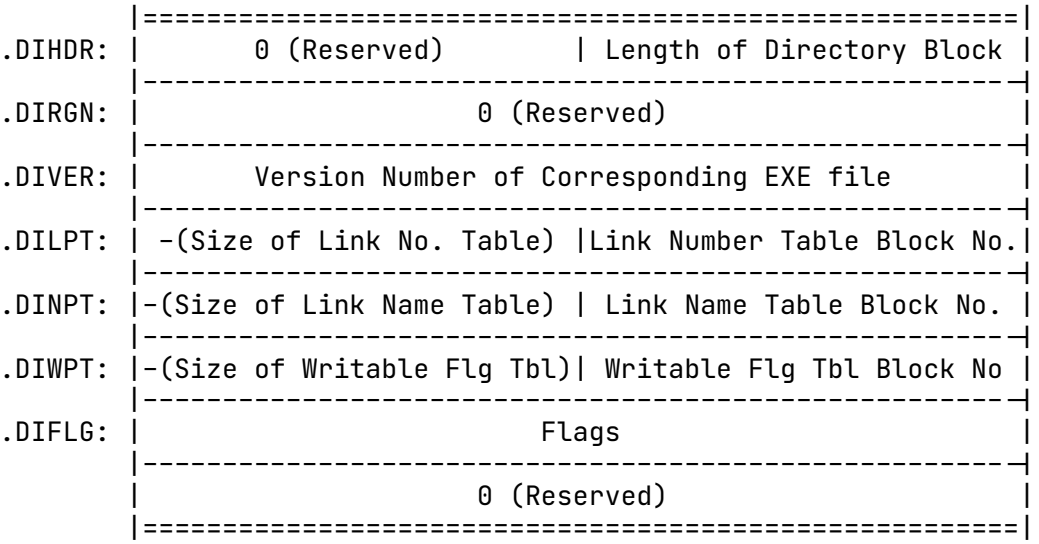
Scheme of the Overlay (OVL) File



5.8.1 The Directory Block

The following diagram shows the contents of the Directory Block:

Directory Block



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.

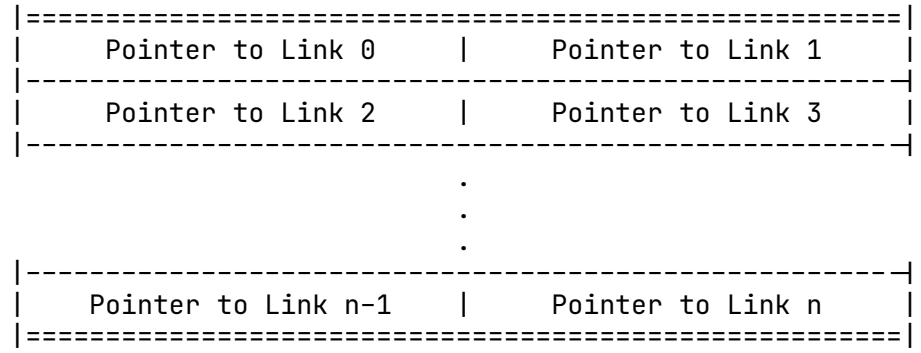
NOTE

If you request 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

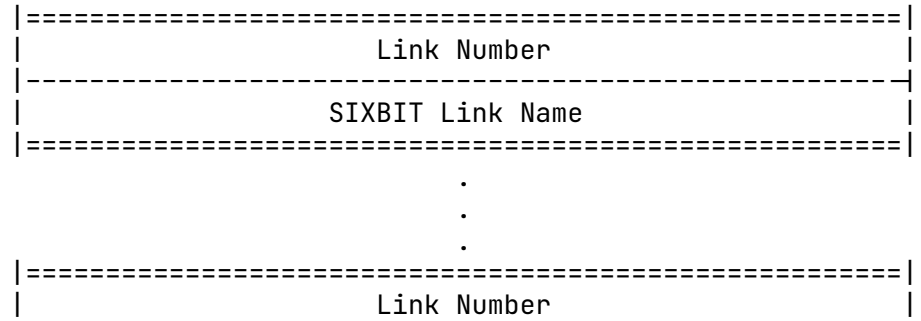


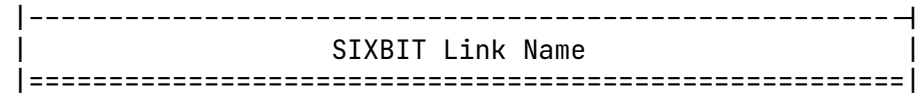
Each pointer is a disk block number. Any unused words in the last disk block of the Link Number Table are zeros.

5.8.3 The Link Name Table

The following diagram shows the contents of the Link Name Table:

Link Name Table



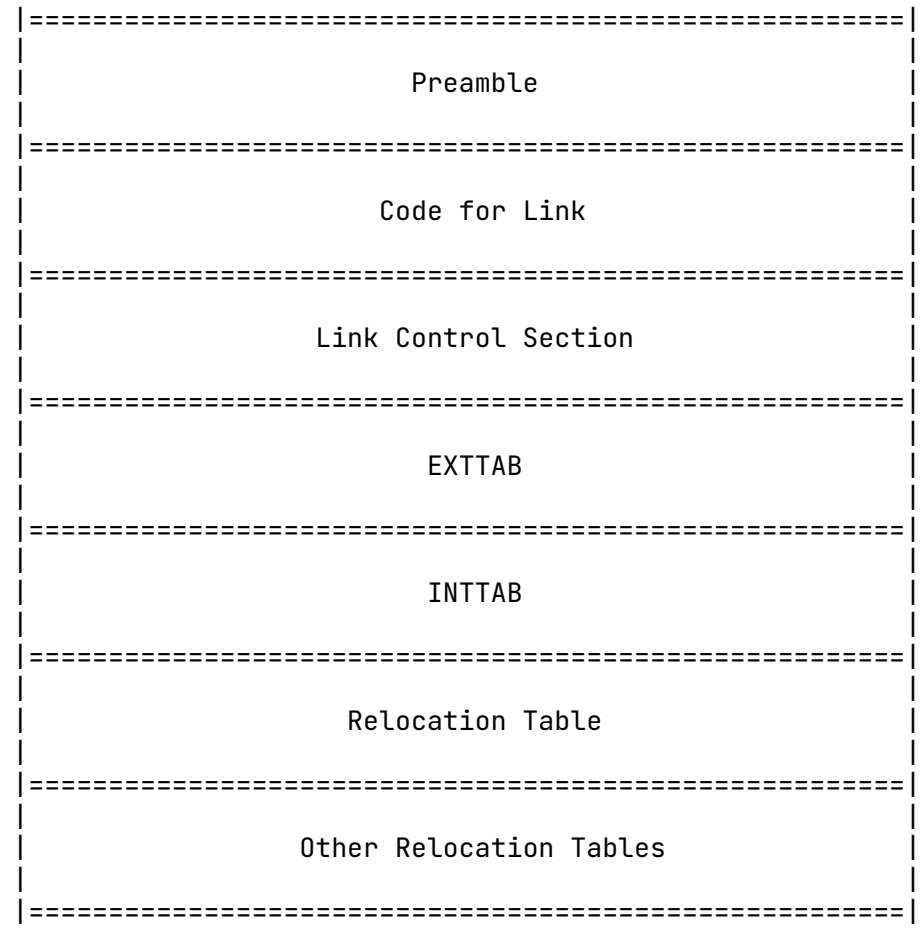


Any unused words in the last disk block of the Link Name Table are zeros.

5.8.4 The Overlay Link

The following diagram shows the overall scheme of each overlay link in the overlay file:

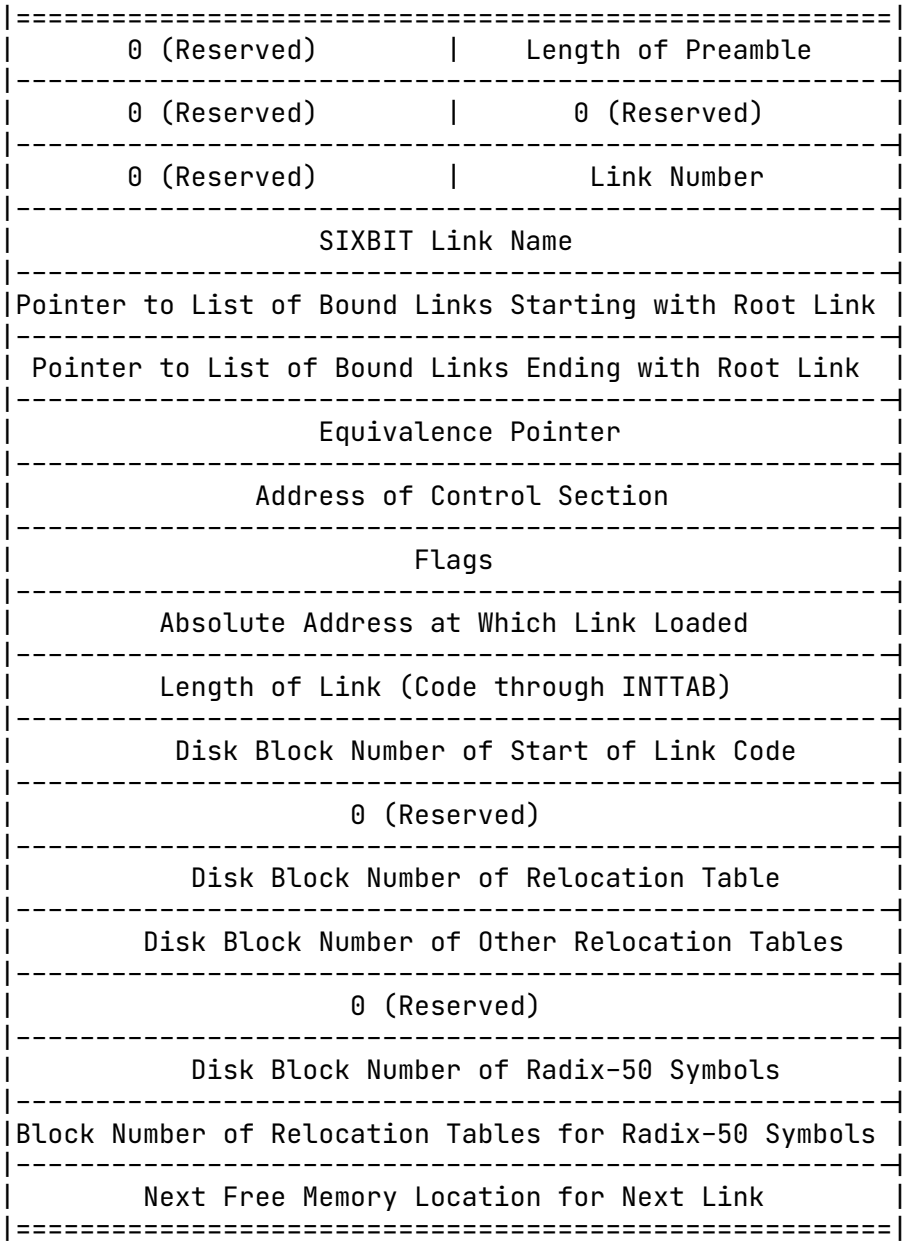
Scheme of an Overlay Link



The Preamble

The following diagram shows the contents of the preamble for an overlay link:

Preamble



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

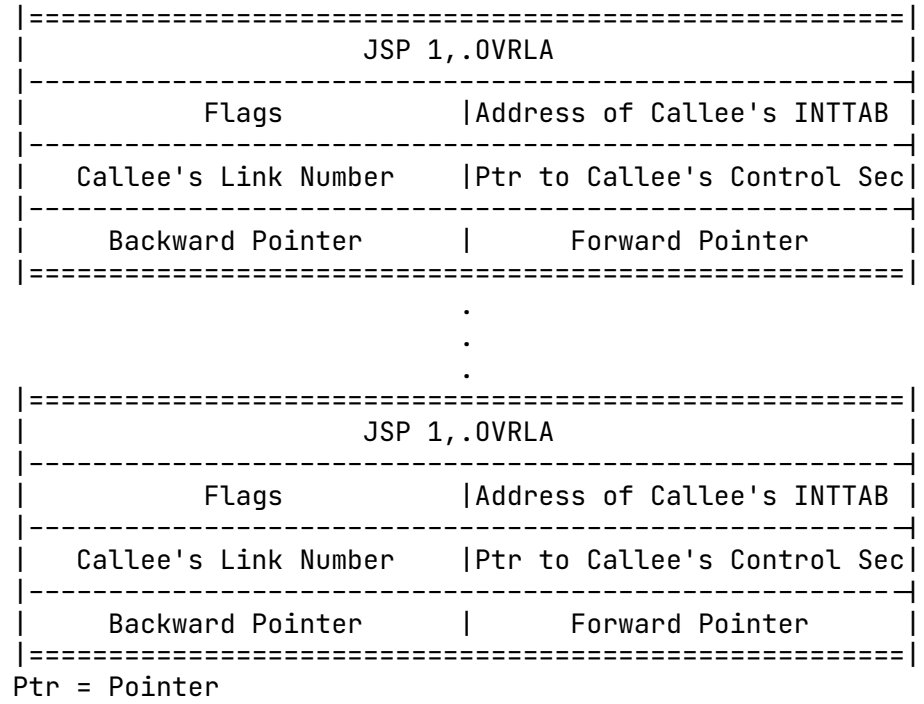
0 (Reserved)	Length of Header
0 (Reserved)	0 (Reserved)
0 (Reserved)	Link Number
SIXBIT Link Name	
Ptr to Ancestor in Core	Ptr to Successor in Core
-(Length of Symbol Table)	Address of Symbol Table
0 (Reserved)	Start Address for Link
Memory Needed to Load Link	First Address in Link
-(Length of EXTTAB)	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

Ptr = Pointer

The EXTTAB Table

The following diagram shows the contents of the EXTTAB table:

EXTTAB



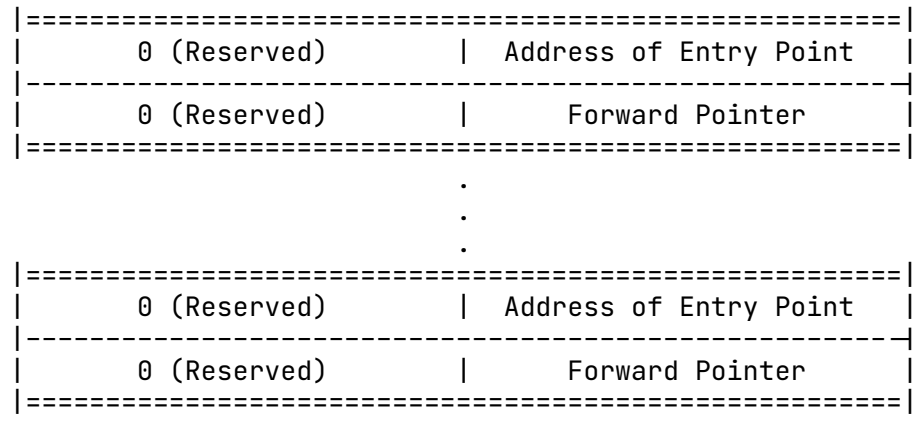
The flags in the left half of the second word have the following meanings:

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:

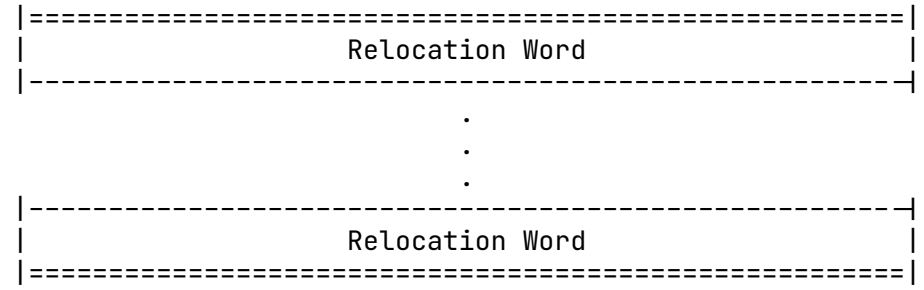
INTTAB



The Relocation Table

The following diagram shows the contents of the Relocation Table:

Relocation Table



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.

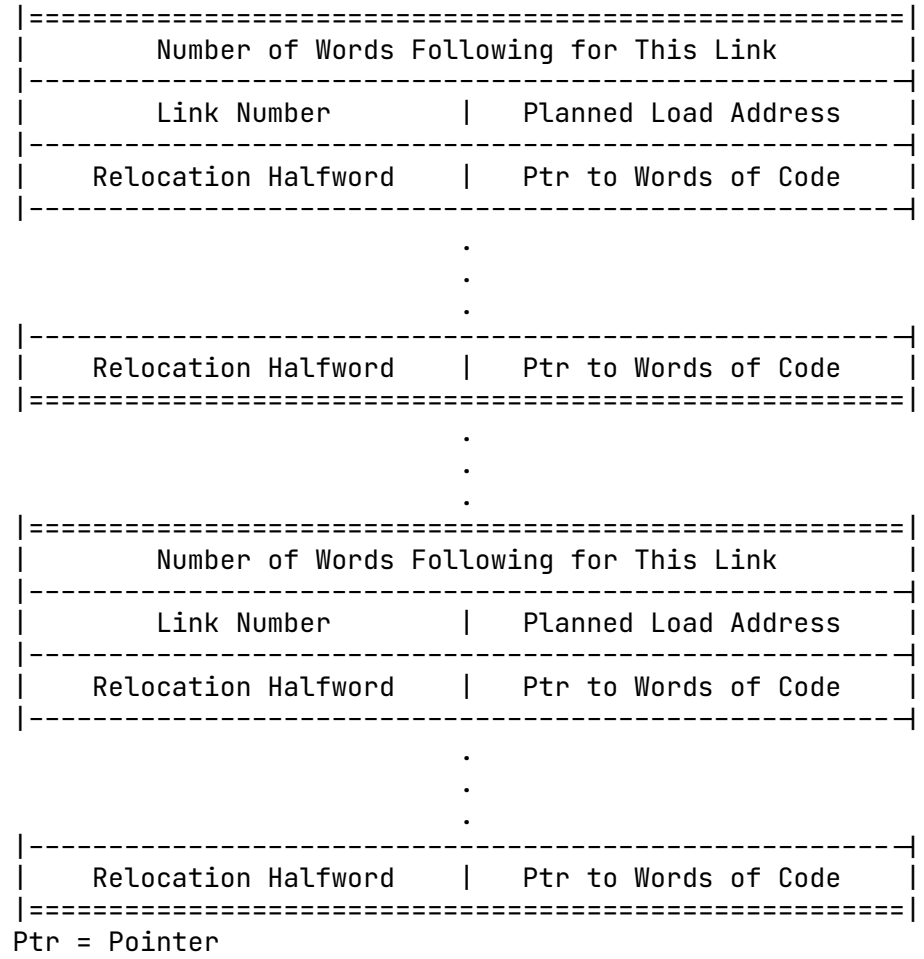
The first word contains the relocation bits for the first 22 (octal) words of the link; the second word contains the relocation bits for the next 22 (octal) words; and so forth for all words in the link.

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



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.

CHAPTER 6

PSECTs

PSECTs (Program SECTions) are programmer- or system-defined regions of code and data that LINK relocates in memory. PSECTs are used for two reasons:

- To load extended addressing programs.
- To structure a program's memory space.

6.1 LOADING PROGRAMS WITH PSECTs

When loading programs with PSECTs (including extended addressing programs), you must specify the origin of the PSECT. LINK then uses this PSECT origin to store the data in the PSECTs.

To specify a PSECT origin, include the origin in the source program or use the LINK /SET switch. See the appropriate language manual for including the origin in the source program and Chapter 3 for the /SET switch.

Defining an upper bound is also important when loading PSECTs. The LINK /LIMIT switch defines an upper bound for a PSECT. If the PSECT loads to this bound, LINK returns a warning and an error message. Despite these messages, LINK continues to process input files and to load code. The warning is:

```
%LNKPEL PSECT [psect] exceeded limit of [address]
```

Although LINK does continue to process input files and load code, the program is incomplete and should not be used. LINK does produce the following fatal error:

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

Chained fixups are a method that LINK uses to resolve symbol references. See the Glossary for a definition of chained fixups.

Using /LIMIT to define an upper bound prevents unintended PSECT overlaps. PSECT overlaps can cause LINK to loop and produce other unpredictable behavior.

For example, the LRGPRO and BIGPRO modules each contain two PSECTs, BIG and GRAND. LRGPRO is loaded and /COUNTERS is used to check PSECT origins and current values. PSECT origins are found by looking under the initial value column and PSECT current values are found by looking under the current value column of the /COUNTERS output. The upper bound is found by looking under the limit value column.

```
@LINK<RET>
*/SET:BIG:1000<RET>
*/SET:GRAND:5400<RET>
*LRGPRO<RET>
*/COUNTERS<RET>
[LNKRLC Reloc. ctr.     initial value   current value   limit value
        .LOW.           0               140             1000000
        BIG             1000            5100            1000000
        GRAND           5400            10500           1000000
        Absolute code loaded]
*
```

/COUNTERS shows that the current value for PSECT BIG and the initial value for PSECT GRAND are close together in memory. The current value for BIG is 5100 and the PSECT origin for GRAND is 5400. The /LIMIT switch can now be used to restrict PSECT BIG's current value to PSECT GRAND's initial value using the following:

```
*/LIMIT:BIG:GRAND<RET>
```

/LIMIT prevents an unintended overlap because it causes LINK to issue a warning if the current value for BIG exceeds GRAND's origin. The warning is:

```
%LNKPEL PSECT [psect] exceeded limit of [address]
```

The warning message indicates that the PSECTs overlapped, and that BIG and GRAND need to be farther apart in memory. The /COUNTERS switch shows a new current value greater than 5400. Notice that the limit set with the /LIMIT switch is shown in the limit value column.

```
*BIGPRO<RET>
%LNKPEL PSECT BIG exceeded limit of 5400
        detected in module .MAIN from file BIGPRO.REL.1
*/COUNTERS<RET>
[LNKRLC Reloc. ctr.     initial value   current value   limit value
        .LOW.           0               140             1000000
        BIG             1000            6300            5400
        GRAND           5400            10500           1000000]
        Absolute code loaded]
```

/GO continues loading the program, and LINK issues a warning and fatal error message. The warning is:

%LNKPOV Psects [psect] and [psect] overlap from address [address] to [address]

The fatal error message is:

?LNKCFS chained fixups have been suppressed

For example,

```
*/GO<RET>
%LNKPOV Psects BIG and GRAND overlap from address 5400 to 6300
?LNKCFS chained fixups have been suppressed
@
```

Now, LINK is re-run and the PSECTs are moved farther apart in memory. In this example, GRAND's origin is reset from 5400 to 7000.

```
@LINK<RET>
*/SET:BIG:1000<RET>
*/SET:GRAND:7000<RET>
*LRGPRO<RET>
*/COUNTERS<RET>
[LNKRLC Reloc. ctr.    initial value  current value  limit value
        .LOW.          0              140            1000000
        BIG            1000           5100           1000000
        GRAND          7000           10500          1000000]

*/LIMIT:BIG:GRAND<RET>
*BIGPRO<RET>
*/COUNTERS<RET>
[LNKRLC Reloc. ctr.    initial value  current value  limit value
        .LOW.          0              140            1000000
        BIG            1000           6300           5400
        GRAND          7000           10500          1000000]
*/GO<RET>
@
```

6.2 PSECT ATTRIBUTES

PSECT attributes specify how LINK stores a PSECT in memory, and the page access of the PSECT.

The CONCATENATED or OVERLAID attribute specifies how LINK stores PSECTs, and the RONLY and RWRITE attributes specify the page access.

6.2.1 CONCATENATED and OVERLAID

LINK uses the CONCATENATED or OVERLAID attributes when loading PSECTs into memory. These attributes are specified when the PSECT is defined in the source program, and are contained in REL Blocks 24 and 1050. See Appendix A for information on these blocks. If the attribute is not specified, LINK uses CONCATENATED.

The following example illustrates how PSECTs are stored in memory. In this example, modules MAINKO and MAINKC contain three PSECTs, ALPHA, BETA, and GAMMA. There is an additional module named SUBMD1. The ALPHA and BETA PSECTs have the CONCATENATE attribute. The GAMMA PSECT, which is a data PSECT declared in each module, has attributes defined differently in each module to illustrate the differences in memory storage. GAMMA has the OVERLAID attribute defined in MAINKO and the CONCATENATED attribute defined in MAINKC.

First, LINK is run and the origin is set for PSECTs ALPHA, BETA, and GAMMA.

```
@LINK<RET>
*/SET:ALPHA:3000/SET:BETA:5000/SET:GAMMA:7000<RET>
```

Next, MAINKO is loaded with GAMMA defined as OVERLAID, and /COUNTERS is used to display the initial, current, and limit values.

```
*MAINKO         ;OVERLAID GAMMA<RET>
*/COUNTERS<RET>
[LNKRLC Reloc. ctr.    initial value  current value  limit value
        ALPHA          3000           3017           1000000
        BETA           5000           5011           1000000
        GAMMA          7000           7025           1000000]
```

Now, SUBMD1 is loaded, /COUNTERS is used, and /GO is used to load the modules and exit LINK. Notice that the current values for ALPHA and BETA have increased, and that the current value for GAMMA remains the same.

```
*SUBMD1<RET>
*/COUNTERS<RET>
[LNKRLC Reloc. ctr.    initial value  current value  limit value
        ALPHA          3000           3033           1000000
        BETA           5000           5041           1000000
        GAMMA          7000           7025           1000000]
*/GO<RET>
@
```

In the following example, LINK is run and the origin is set for PSECTs ALPHA, BETA, and GAMMA.

```
@LINK<RET>
*/SET:ALPHA:3000/SET:BETA:5000/SET:GAMMA:7000<RET>
```

Now, MAINKC is loaded with GAMMA defined as CONCATENATE, and /COUNTERS is used.

```
*MAINKC  ;CONCATENATED GAMMA<RET>
*/COUNTERS<RET>
[LNKRLC Reloc. ctr.    initial value  current value  limit value
        ALPHA          3000           3017           1000000
        BETA           5000           5011           1000000
        GAMMA          7000           7025           1000000]
*
```

Next, SUBMD1 is loaded, /COUNTERS is used, and /GO is used to load the modules and exit LINK. Notice that all current values have increased.

```
*SUBMD1<RET>
*/COUNTERS<RET>
[LNKRLC Reloc. ctr.    initial value  current value  limit value
        ALPHA          3000           3033           1000000
        BETA           5000           5041           1000000
        GAMMA          7000           7035           1000000]
*/GO<RET>
@
```

6.2.2 RONLY and RWRITE

The RONLY and RWRITE attributes set the page access for PSECTs. RONLY defines the page access to be read-only and RWRITE defines the page access to be read/write. These attributes are set in the source program. If the attribute is not set, LINK uses RWRITE.

For example, the following MACRO statement defines read/write access for the RED PSECT.

```
     .PSECT RED/RWRITE,1000
```

1000 is the PSECT's origin.

LINK sets the page access to read-only if a memory page contains part or all of a read-only PSECT even if a read/write PSECT exists on the same page.

CHAPTER 7

PDVs

LINK stores information about the program that it is loading into a PDV or JOBDAT. A PDV (Program Data Vector) is a block of information. JOBDAT is the job data area.

LINK stores information in a PDV:

- for extended addressing programs.
- if a PSECT has an origin below 140.
- if you specify a PDV using /PVBLOCK or /PVDATA.
- if /NOJOBDAT is used.

LINK stores information in JOBDAT for most section zero programs.

LINK stores such information as debugger symbol table pointers, version numbers, and memory use.

Although LINK builds no more than one PDV per load, a program can use the TOPS-20 PDVOP% JSYS at runtime to build new PDVs or to find PDVs associated with object time systems and utilities merged into the address space. LINK cannot build a PDV for a program that uses overlays.

At link time, you can use the /PVBLOCK and /PVDATA switches to request a PDV, control its memory location, or change its contents. See Chapter 3 for information on these switches.

Each PDV contains location .PVCST reserved for customer use. A typical use for this location might be to point to the address of a transfer vector that contains entry points to a utility package. Any program that wants to call the utility can then use the PDVOP% JSYS to find the address of the transfer vector. This eliminates the need to build the transfer vector address of the utility package into a program.

For more information about the PDVOP% JSYS, refer to the TOPS-20 Monitor Calls Reference Manual.

7.1 PDV FORMAT

The format of a PDV follows:

Word	Symbol	Meaning
0	.PVCNT	is the length of the PDV including this word.
1	.PVNAM	points to the name in ASCIZ of this PDV. The default PDV name is nnnLNK, where nnn is the job name. The following rules apply to the assignment of PDV names. If these rules are followed, conflicting PDV names can be avoided: 1. PDV names assigned by DIGITAL will contain the percent character (%). PDV names assigned by users should not contain the percent character. 2. All PDV names containing the period character (.) are reserved to DIGITAL for future use. 3. The dollar sign character ($) is reserved for special use: PDV names of the form string1$string2 are reserved for the special class of use named by string1. Rules 1 and 2 still apply in this case. As a general guideline, PDV names should be as specific as possible to avoid name conflicts and confusion. If bit 0 is 1, this word contains a section-local address. If bit 0 is 0, this word contains a global address. A section-local address is an 18-bit address. A global address is a 30-bit address.
2	.PVEXP	points to a counted vector of exported information. A counted vector is a contiguous block of information where the length of the block is contained in the first word. A PDV is an example of a counted vector. Exported information is information that is available for use by other programs. By convention the remainder of the block is defined by the program. If a program class exists, the program class defines the remainder of the program. See the third rule in the PDV naming rules above for program class.

If bit 0 is 1, this word contains a section-local address. If bit 0 is 0, this word contains a global address. A section-local address is an 18-bit address. A global address is a 30-bit address.

3 .PVREE is a word reserved for DIGITAL and must be zero.

4 .PVVER is the program's version number. See .JBVER in Appendix C for more information.

5 .PVMEM points to the static memory map for this program. See Section 7.2 for a description of the memory map.

If bit 0 is 1, this word contains a section-local address. If bit 0 is 0, this word contains a global address. A section-local address is an 18-bit address. A global address is a 30-bit address.

6 .PVSYM points to the symbol table vector for this program. See Section 7.3 for a description of this vector.

If bit 0 is 1, this word contains a section-local address. If bit 0 is 0, this word contains a global address. A section-local address is an 18-bit address. A global address is a 30-bit address.

7 .PVCTM is the time that your program was compiled in TOPS-20 format.

10 .PVCVR is the version number of the main program's compiler. See .JBVER in Appendix C for more information.

11 .PVLTM is the time that the program was loaded in TOPS-20 format.

12 .PVLVR is LINK's version number.

13 .PVMON points to a monitor data block. (Reserved for DIGITAL.)

14 .PVPRG points to a program data block.

If bit 0 is 1, this word contains a section-local address. If bit 0 is 0, this word contains a global address. A section-local address is an 18-bit address. A global address is a 30-bit address.

15 .PVCST points to a customer-defined data block.

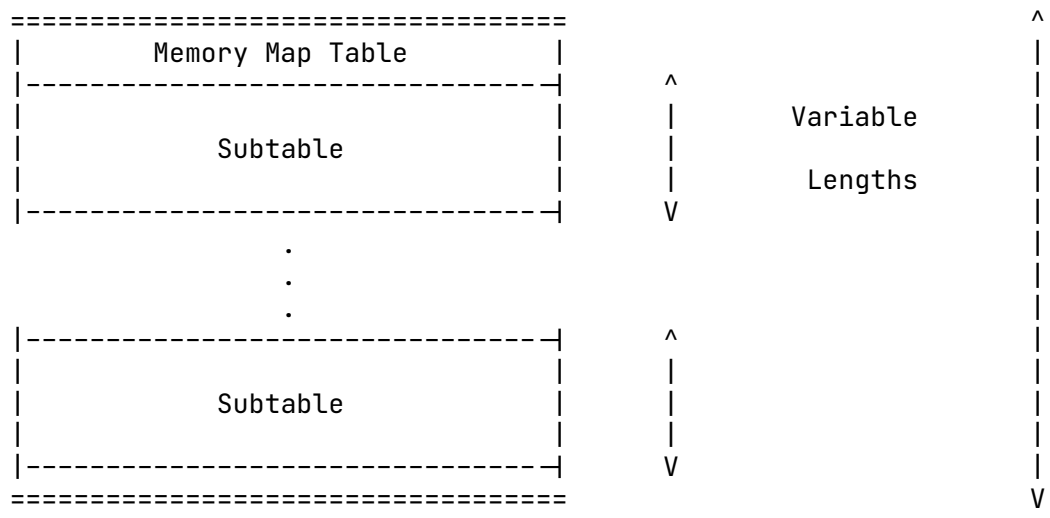
If bit 0 is 1, this word contains a section-local address. If bit 0 is 0, this word contains a global address. A section-local address is an 18-bit address. A global address is a 30-bit address.

7.2 THE PDV STATIC MEMORY MAP

LINK creates the static memory map along with the PDV. This static memory map can be used as a resource in dynamic memory allocation. In addition to containing information about allocated pages in use, this memory map contains information about pages that are allocated but zero.

In the PDV, the .PVMEM (5) word points to the program's memory map.

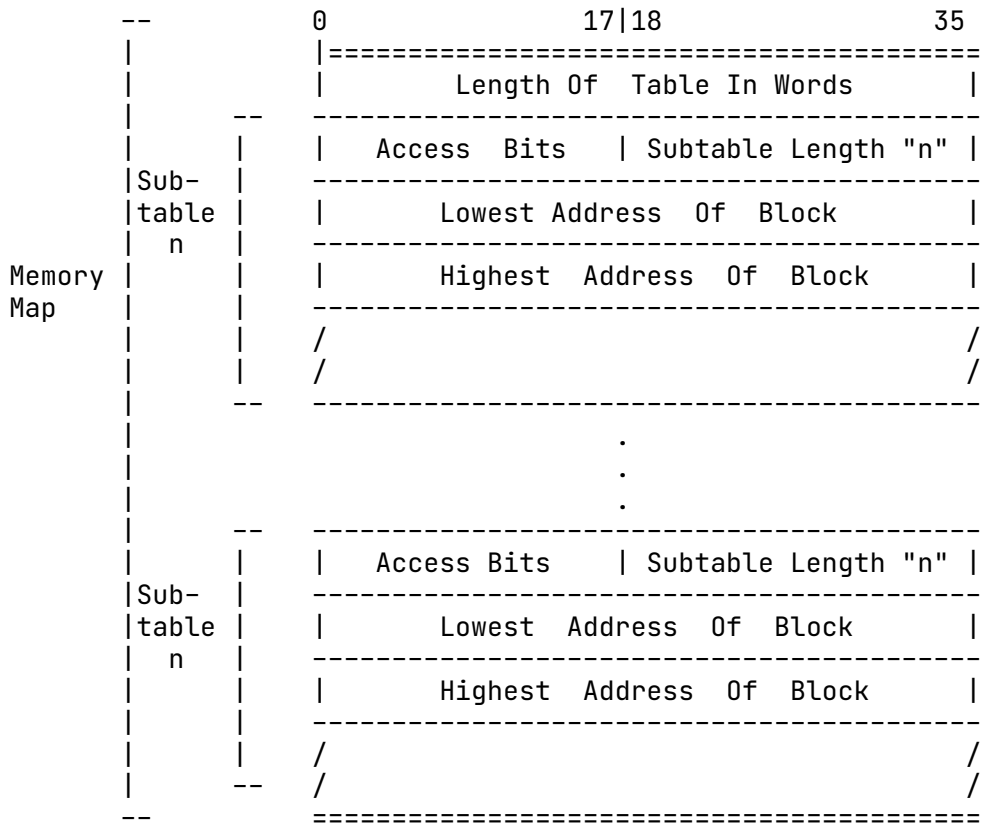
The memory map is a variable length table of variable length subtables.



A subtable defines the beginning and end of a block of contiguous words of memory with the same read/write attributes.

The addresses in the subtable are ordered from lowest to highest with the lowest block of memory entered into the subtable first and the highest entered into the subtable last.

The format of the memory map is illustrated below.



Memory Map

Word	Symbol	Meaning
0	.MMLEN	is the length in words of the entire memory map table, including this word.

Subtable

Word	Symbol	Meaning
0	.MMDAT	is the first subtable word and contains two half-words: MM%ACC and MM%LEN.
	MM%ACC	contains the block access bits. The access bits define the block attributes, namely, whether this block is read-only or read/write.

These bits are set according to how the PSECT attributes are set in the source program. LINK defaults read and write attributes if the source program does not explicitly set attributes. All blocks in the memory map implicitly have read and execute access. If bit 3, MM%WR, is set, the block is writable. Undefined access bits are reserved for DIGITAL and must be zero.

MM%LEN contains the subtable length including the first word. The length of a subtable is normally 3, but can be greater if additional information is stored after .MMHGH. This area after .MMHGH is reserved for DIGITAL.

To obtain the address of the next subtable, add the length of the current subtable to the address of the current subtable. For example, if the length of the current subtable is 5 and its address is 15672, the address of the subtable following is 15677.

1 .MMLOW is the address of the first word of a contiguous block of memory with the same read/write or read-only attributes.

If bit 0 is 1, this word contains a section-local address. If bit 0 is 0, this word contains a global address. A section-local address is an 18-bit address. A global address is a 30-bit address.

2 .MMHGH is the address of the last word of the block.

If bit 0 is 1, this word contains a section-local address. If bit 0 is 0, this word contains a global address. A section-local address is an 18-bit address. A global address is a 30-bit address.

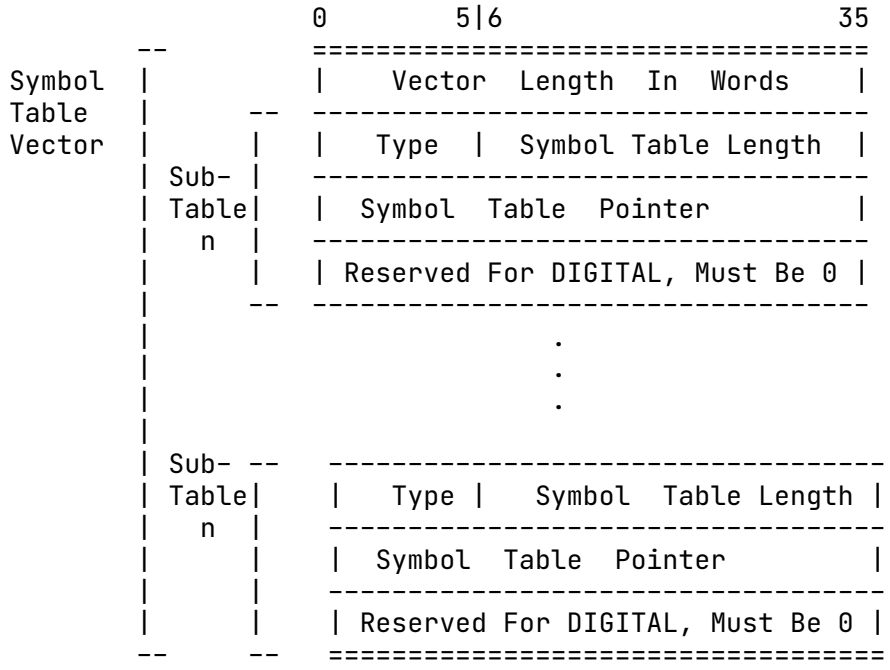
7.3 SYMBOL TABLE VECTOR

A symbol table vector is a pointer to the symbol tables of a program. There is one symbol table vector, and an undefined and defined symbol table per program.

The .PVSYM (6) word in the PDV points to the symbol table vector. If there is no symbol table (for example, when you use /NOSYM at link time), the .PVSYM word is 0. If LINK builds the symbol table vector,

it always contains the defined and undefined symbol table pointers.

The symbol table vector contains subtables that point to each symbol table and gives their length and type. Each subtable is three words long, although only the first two words are currently used. The format of a symbol table vector is illustrated below.



Symbol Table

Word	Symbol	Meaning
0	.STLEN	defines the length in words of the symbol table vector including this word.

Subtable

Word	Symbol	Meaning
0	.STDAT	is the first subtable word and contains two fields: ST%TYP and ST%LEN.
	ST%TYP	is a 6-bit field that contains the symbol table type.
		The types are:

Code	Name	Type
0	.UNDFD	Undefined
1	.R50D	Radix-50 defined symbols
2	.R50U	Radix-50 undefined symbols
3-37		Reserved for DIGITAL
40-77		Reserved for customers

ST%LEN is a 30-bit field that contains the length in words of the particular symbol table.

1 .STPTR is the lowest word in the table.

If bit 0 is 1, this word contains a section-local address. If bit 0 is 0, this word contains a global address. A section-local address is an 18-bit address. A global address is a 30-bit address.

The Reserved word is reserved for DIGITAL and must be zero.

APPENDIX A

REL BLOCKS

REL (RELocatable) Blocks are the main input to LINK and contain information that LINK uses to load a program.

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 that contains the REL Block Type in its left half and a short count or long count in its right half.

Short Count: a halfword that contains the length of the REL Block, excluding relocation words. The short count appears before each group of 18 decimal, or 22 octal words and excludes the header word.

Long Count: a halfword that contains the length of the REL Block, including all words in the block except the header word itself.

Relocation Word: a fullword that contains the relocation bits for up to 18 decimal or 22 octal 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 are 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 has unused bits.

NOTE

A block with a zero short count does not include a relocation word.

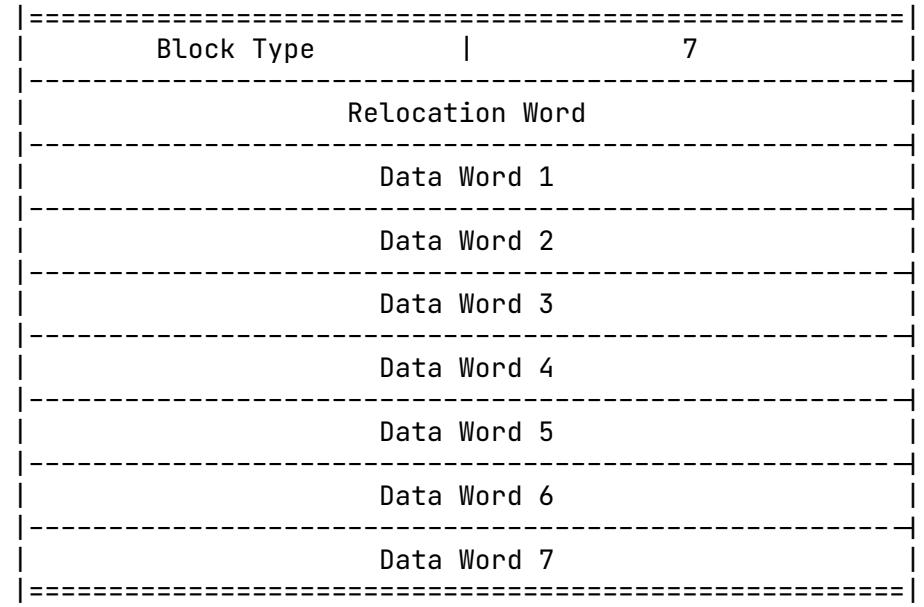
Data Word: Any word other than a header word or a relocation word.

MBZ: Must Be Zero.

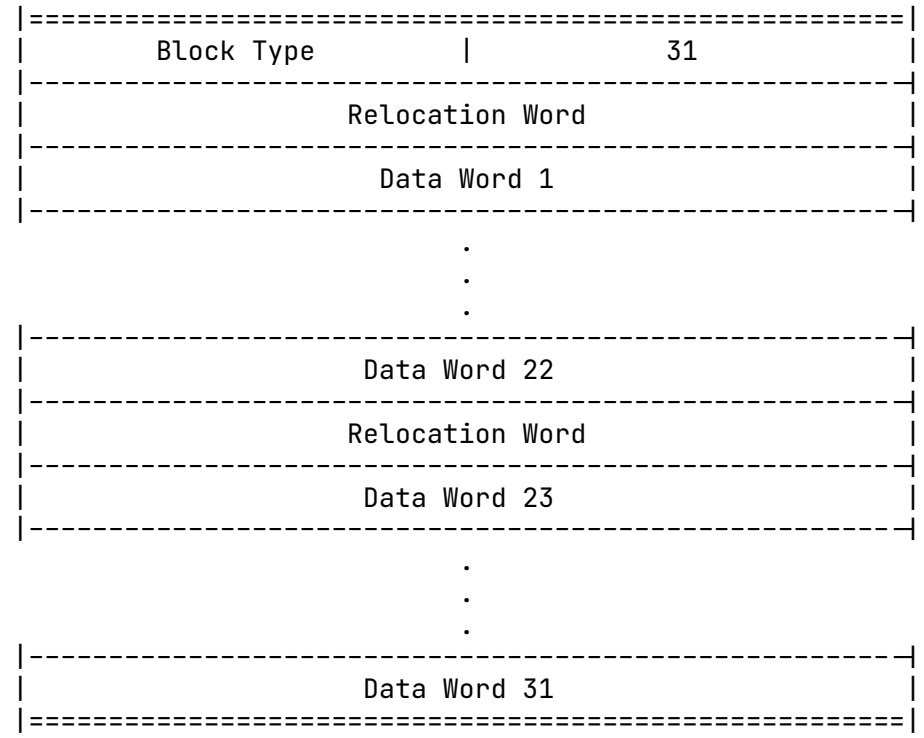
NOTE

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.



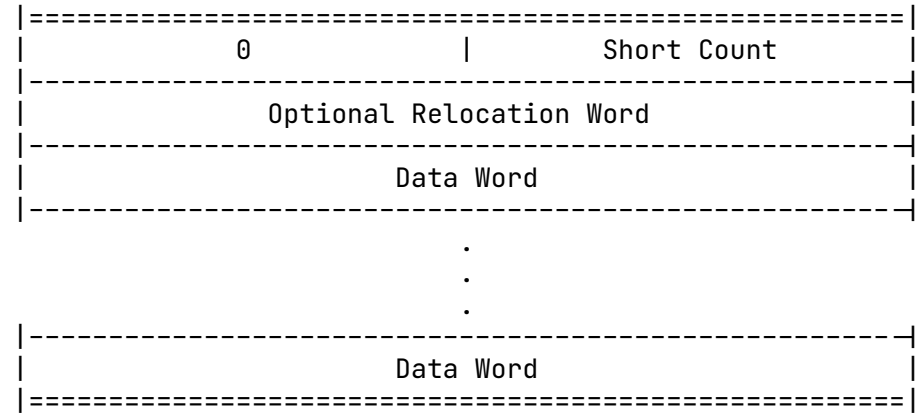
The diagram below shows a REL Block having a short count of 31 (octal) and two relocation words.



REL Block Types must be numbered in the range 0 to 777777. 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

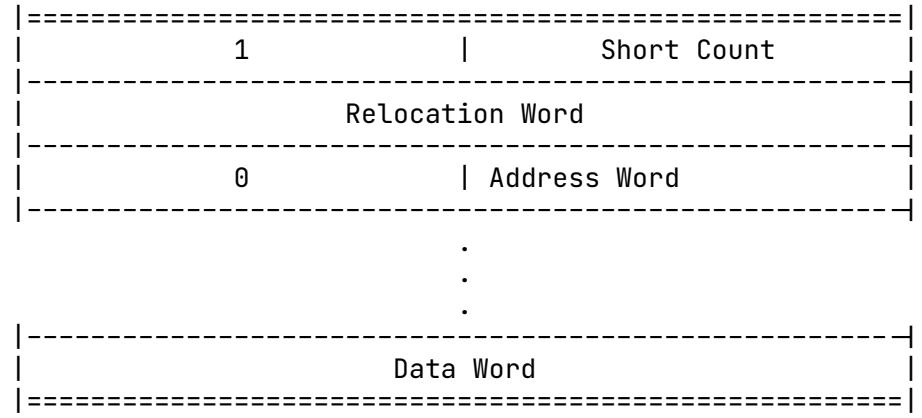
Block Type 0 (Ignored)



Block Type 0 is ignored by LINK.

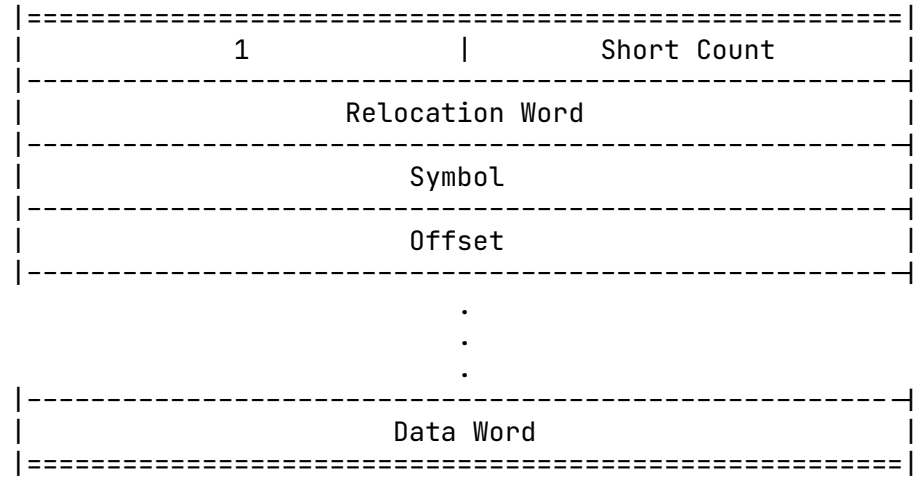
If the short count is 0, then no relocation word follows, and the block consists of only one word. This is how LINK bypasses zero words in a REL file.

Block Type 1 (Code)



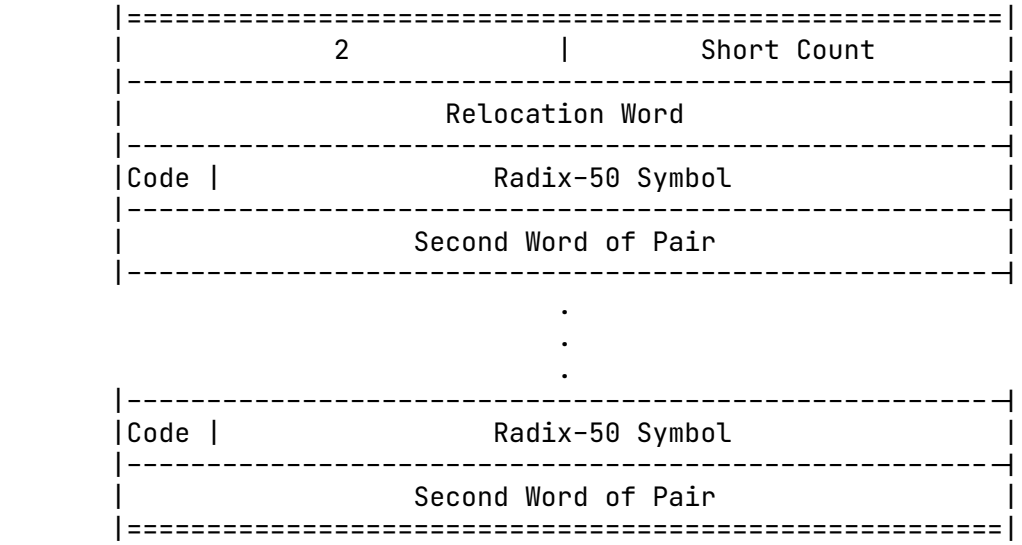
Block Type 1 contains data and code. The address is where LINK is to load the data. This address can be relocatable or absolute, depending on the value of bit 1 of the relocation word. LINK loads the remaining data words beginning at that address.

If a symbol is used to specify the start address, the following format of Block Type 1 is used:



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)



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.

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. A fixup resolves the symbol. 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 DDT does not output the symbol.

50 The given symbol is a local symbol exactly as in code 10, except that DDT does not output the symbol.

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 chain ends when the address is 0.

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 DDT does not output the symbol.

70 The given symbol is partially defined, where the left half is deferred, as in code 30, except that DDT does not output the symbol.

74 The given symbol is partially defined, where the right half is deferred, as in code 34, except that DDT does not output the symbol.

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.

The following fixup rules apply to this block:

1. Only one fixup by a Type 2, 10, 11, 12, 15, 1070, 1072, or 1120 Block is allowed for a given word. (There can be separate fixups for the left and right halves of the same word.)

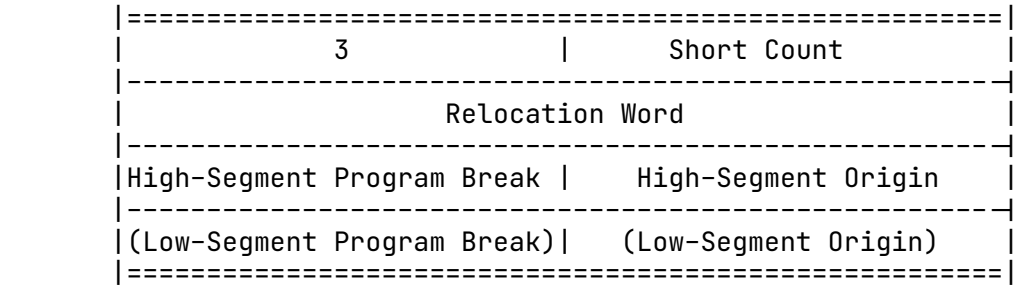
2. Fixups are not necessarily performed in the order LINK finds them.

3. Chained halfword fixups cannot cross section boundaries. Also, they cannot fixup a location which resolves to word zero of a section unless it is the only address in the chain.

4. Chained fixups must be in strict descending address order.

5. A location must contain data before the location can be fixed up.

Block Type 3 (HISEG)



Block Type 3 tells LINK that code is to be loaded into the high segment.

The Short Count is either 1 or 2.

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).

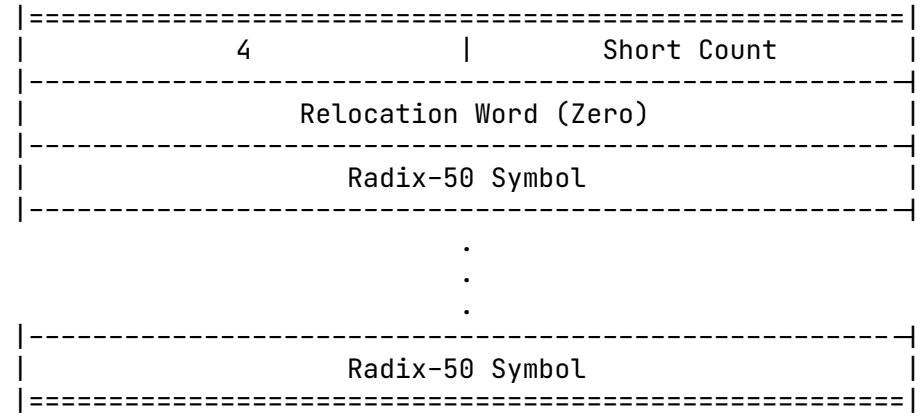
If the left half of the first data word is nonzero (the preferred usage), subsequent Type 1 blocks found are assumed to have been produced by the MACRO pseudo-op TWOSEG.

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)

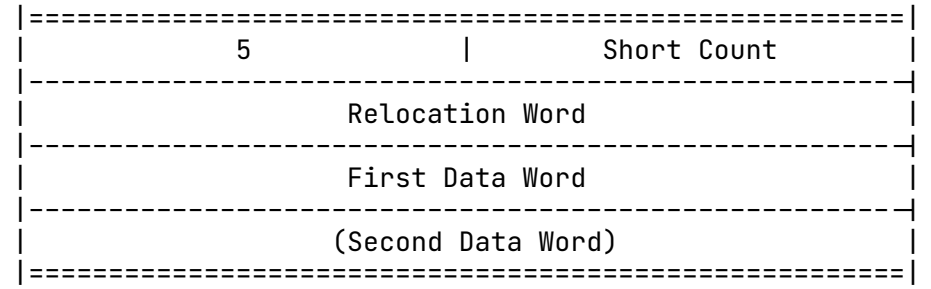


Block Type 4 lists the entry name symbols for a program module. If a Type 4 block appears in a module, it must be the first block in the module. A library file contains a Type 4 block for each of its modules.

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.

The 4 high order bits of the symbol are 0 and are ignored.

Block Type 5 (End)



Block Type 5 ends a program module. A Block Type 6 (Name) must be encountered earlier in the module than the Type 5 block.

The Short Count is 1 or 2.

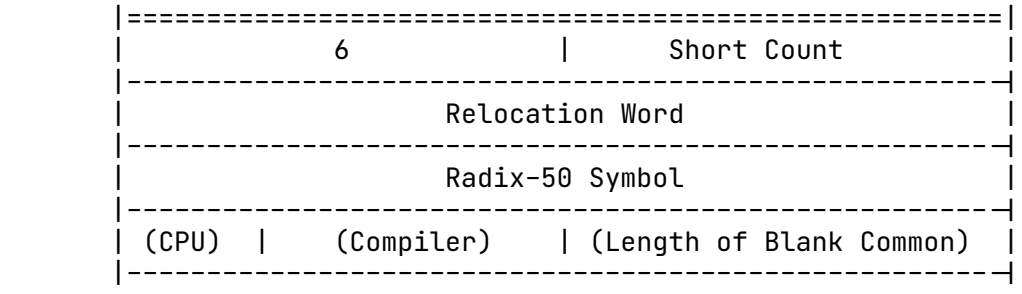
If the module contains a two-segment program, the first data word is the high-segment break and the second data word is the low-segment break. If the module contains a one-segment program, the first data word is the program break and the second data word is the absolute break. If the count is 1, then the second word is assumed to be 0.

If the module contains a program that uses PSECTs, Block Type 5 has no effect except if the /REDIRECT switch was used. If /REDIRECT was used, the break information is used to set the highest location to be loaded for the PSECT specified in /REDIRECT.

Each PRGEND pseudo-op in a MACRO program generates a Type 5 REL block. Therefore, a REL file may contain more than one pair of Type 6 and Type 5 blocks.

A library REL file has a Type 5 block at the end of each of its modules.

Block Type 6 (Name)



Block Type 6 contains the program name, and must precede any Type 2 blocks. (A module should begin with a Type 6 or 1003 block and end with a Type 5 block.)

The Short Count is 1 or 2.

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:

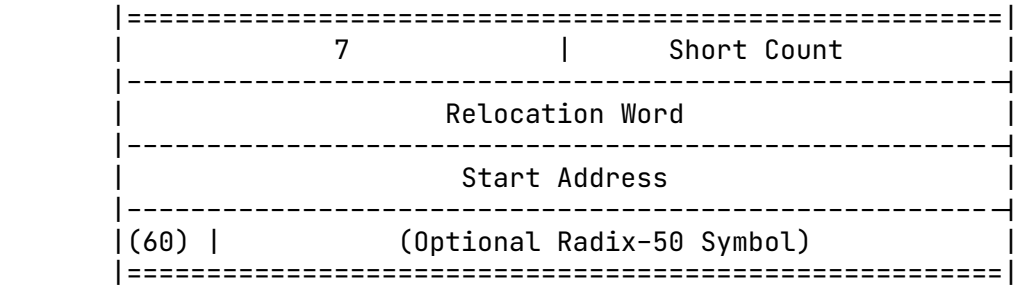
Bit 2	KS10
Bit 3	KL10
Bit 4	KI10
Bit 5	KA10

If none of these bits are on, 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	10	FORTRAN	20	BLISS-36
1	Not used	11	MACRO	21	BASIC
2	COBOL-68	12	FAIL	22	SITGO
3	ALGOL	13	BCPL	23	(Reserved)
4	NELIAC	14	MIDAS	24	PASCAL
5	PL/I	15	SIMULA	25	JOVIAL
6	BLISS	16	COBOL-7	26	ADA
7	SAIL	17	COBOL		

Block Type 7 (Start)



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.

Short Count is 2.

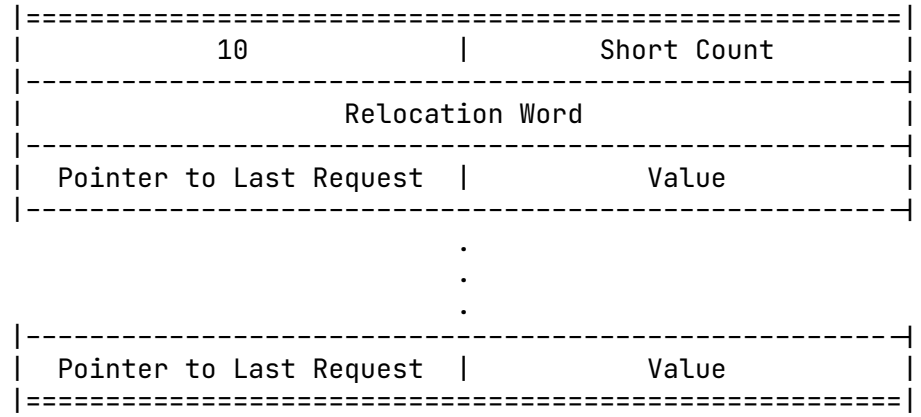
If the Optional Radix-50 Symbol word is present, it must be a Radix-50 symbol with the code 60. LINK forms the start address by adding the value of the symbol to the value in the right half of the preceding word (Start Address).

LINK defaults a TOPS-10 style entry vector if the entry length vector is zero.

LINK defaults a TOPS-20 style entry vector:

- if the program contains nonzero sections and the length of the entry vector is not specified
- if the length of the entry vector is 1
- if /NOJOBDAT equals 1

Block Type 10 (Internal Request)



Block Type 10 is generated by one-pass translators 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 program that contains PSECTs.

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.)

The following fixup rules apply to this block:

1. Only one fixup by a Type 2, 10, 11, 12, 15, 1070, 1072, or 1120 Block is allowed for a given field. (There can be separate fixups for the left and right halves of the same word.)

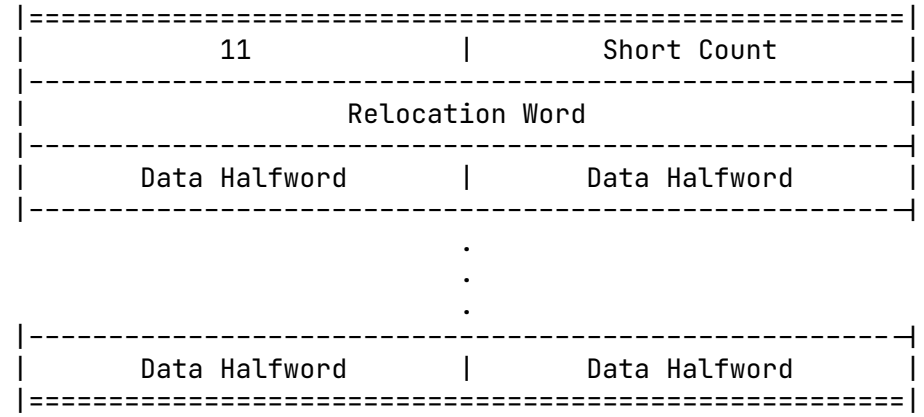
2. Fixups are not necessarily performed in the order LINK finds them.

3. These are halfword fixups and cannot cross section boundaries. They wrap within a section. Also, halfword fixups cannot fixup a location which resolves to word zero of a section unless it is the only address in the chain.

4. Chained fixups must be in strict descending address order.

5. A location must contain data before the location can be fixed up.

Block Type 11 (Polish)



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.

The following fixup rules apply to this block:

1. Only one fixup by a Type 2, 10, 11, 15, 1070, 1072, or 1120 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.

3. Chained halfword fixups cannot cross section boundaries. Also, they cannot fixup a location which resolves to word zero of a section unless it is the only address in the chain.

4. Chained fixups must be in strict descending address order.

5. A location must contain data before the location can be fixed up.

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 two or 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.

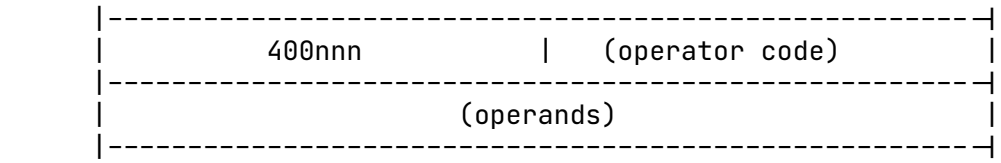
4. An arithmetic or logical operator code.

5. A PSECT index code.

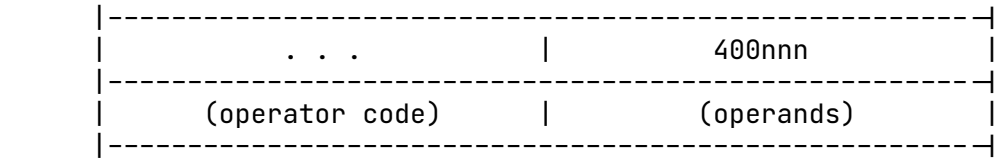
   This code defines a PSECT index to be used for calculating the relocated addresses that appear in this block. PSECT indexes are needed only for PSECTed programs.

   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:



OR



   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:

.LINK fixup codes

-10 does a store into the link that was specified if the address is negative. Stores the address as a link-end. Links the result of the Polish Block.

-7 Fullword replacement. No chaining is done.

Symbol Fixup Store Operator Codes:

-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:

3 Add.

4 Subtract. The first operand is subtracted by the second.

5 Multiply.

6 Divide. The first operand is divided by the second.

7 Logical AND.

10 Logical OR.

11 Logical shift. The first operand is shifted by the second.

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. The first operand is REM by the second.

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. The address of the specified link is used. See -10.

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 the remainder of the REL module if the argument is nonzero, otherwise return the argument. Undefined symbols are not allowed with this store operator in data type 2 operands.

27 Return contents of location N. No fixups may be done on 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 (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

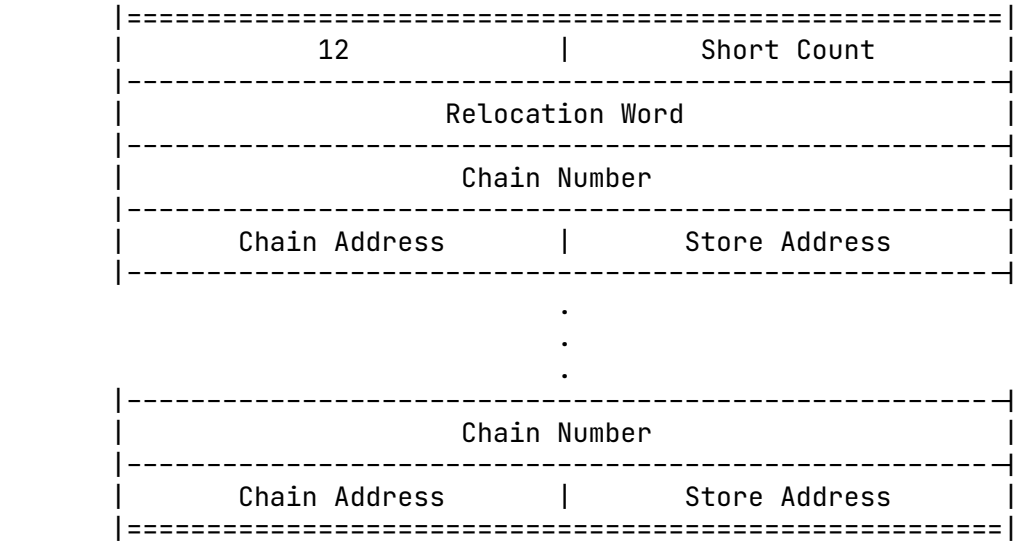
these halfwords contains a data type code 2, showing that the operand is two halfwords containing a Radix-50 symbol with code 60. The next two halfwords give the symbol (B).

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)



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.

The following fixup rules apply to this block:

1. Only one fixup by a Type 2, 10, 11, 12, 15, 1070, 1072, or 1120 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.

3. Chain numbers above 100 (octal) are reserved by DIGITAL.

4. These are halfword fixups and cannot cross section boundaries. They wrap within a section. Also, halfword fixups cannot fixup a location which resolves to word zero of a section unless it is the only address in the chain.

5. A location must contain data before the location can be fixed up.

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
```

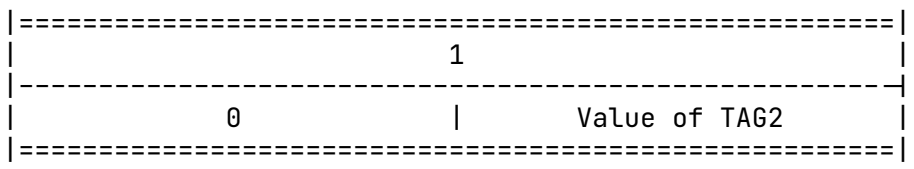
-1	
0	Value of TAG0

```
TITLE MOD1
          .
          .
          .
TAG1:   BLOCK 1
          .
          .
          .
        .LINK 1,TAG1
          .
          .
          .
        END
```

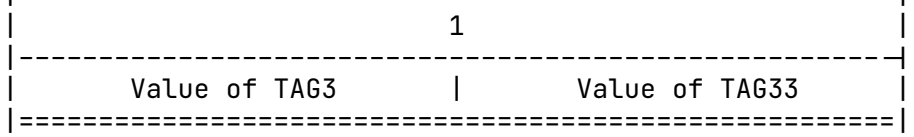
1	
0	Value of TAG1

```
TITLE MOD2
          .
          .
          .
TAG2:   BLOCK 1
          .
          .
          .
        .LINK 1,TAG2
          .
```

```
  .
  .
END
```




```
TITLE MOD3
          .
          .
          .
TAG3:   BLOCK 1
          .
          .
          .
TAG33:  BLOCK 1
          .
          .
          .
        .LINK 1,TAG33,TAG3
          .
          .
          .
        END
```



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.

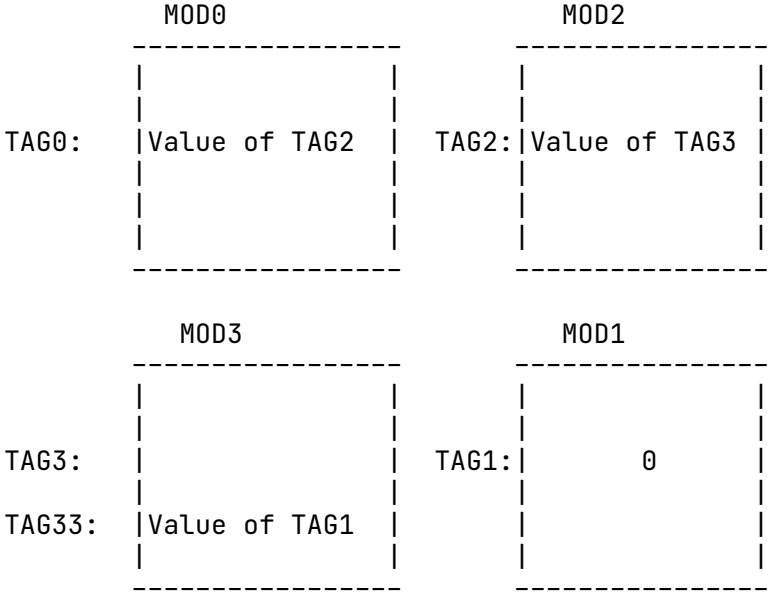
Next we load MOD1. The .LINK statement for MOD1 does not use the third argument, so the chain address is 0. LINK sees that this is the first entry for chain number 1. Because it is the first entry, LINK places a 0 in the store address (value of TAG1). LINK then remembers the value of TAG1 for use in 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).

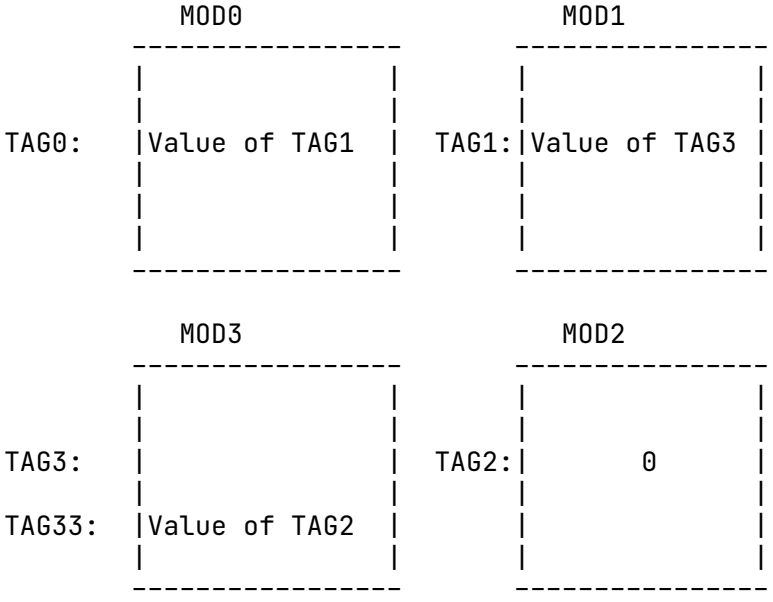
At the end of loading, LINK places the last remembered address (value of TAG2) at the address (value of TAG0) given by the .LNKEND statement in MOD0.

The results of the chaining can be seen in the following diagram of the loaded core image:

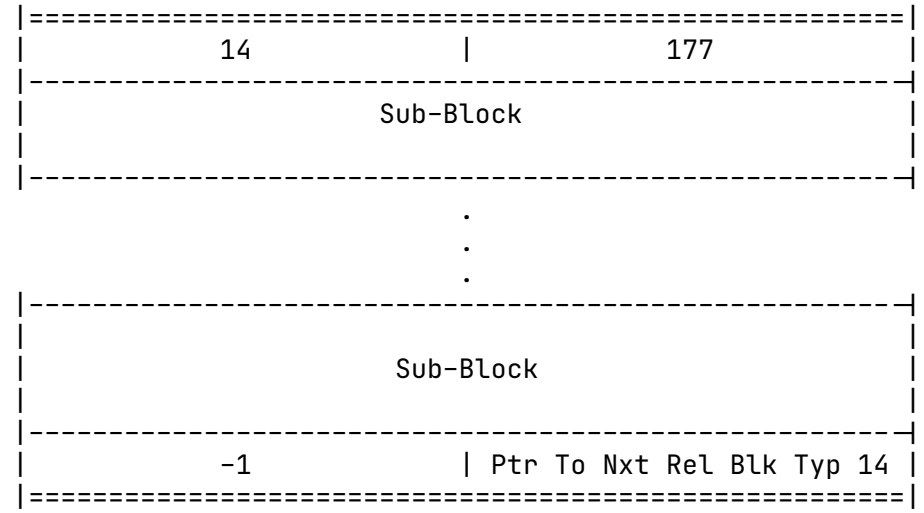


Note that the order of loading for modules with .LINK statements is critical. (A module containing a .LNKEND statement can be loaded any time; its treatment is not affected by the order of loading.)

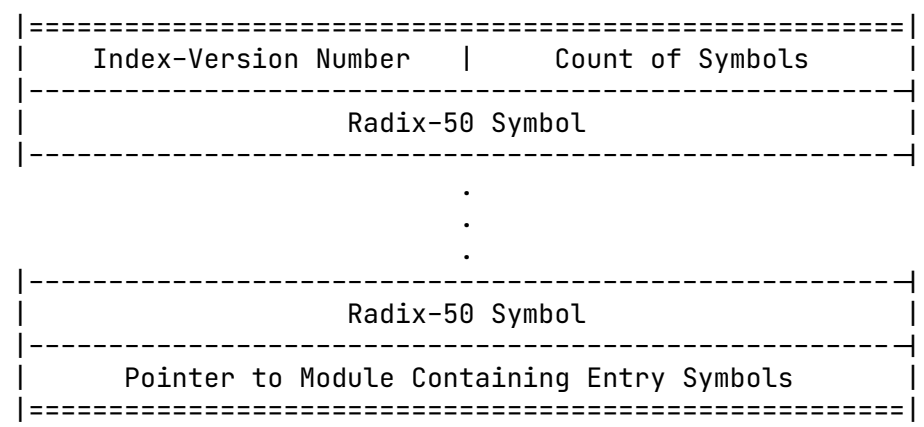
For example, if we load the four programs in the order MOD2, MOD3, MOD0, MOD1, we get a different resulting core image:



Block Type 14 (Index)



Each sub-block is of the form:



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. If 177 data words are not needed, zero words pad the block to a length of 177. -1 indicates the end of the sub-block information.

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.

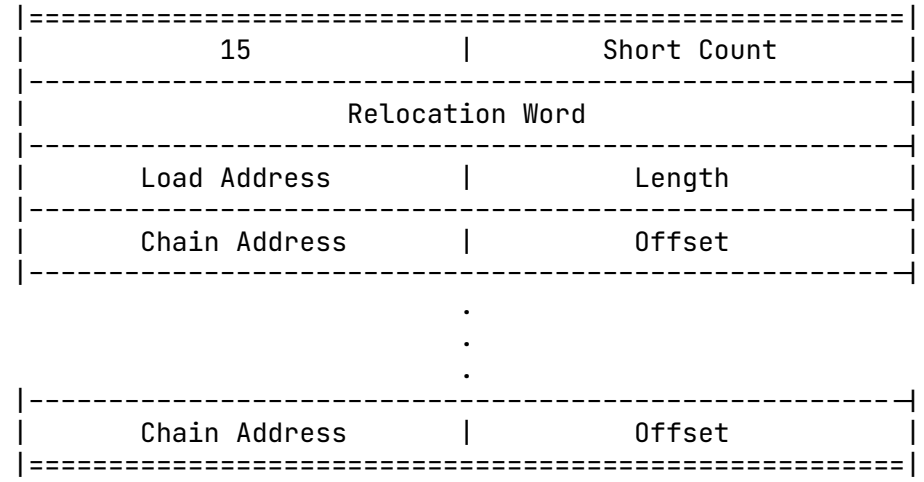
Each sub-block is like a Type 4 block, with three differences:

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 -1.

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)



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.

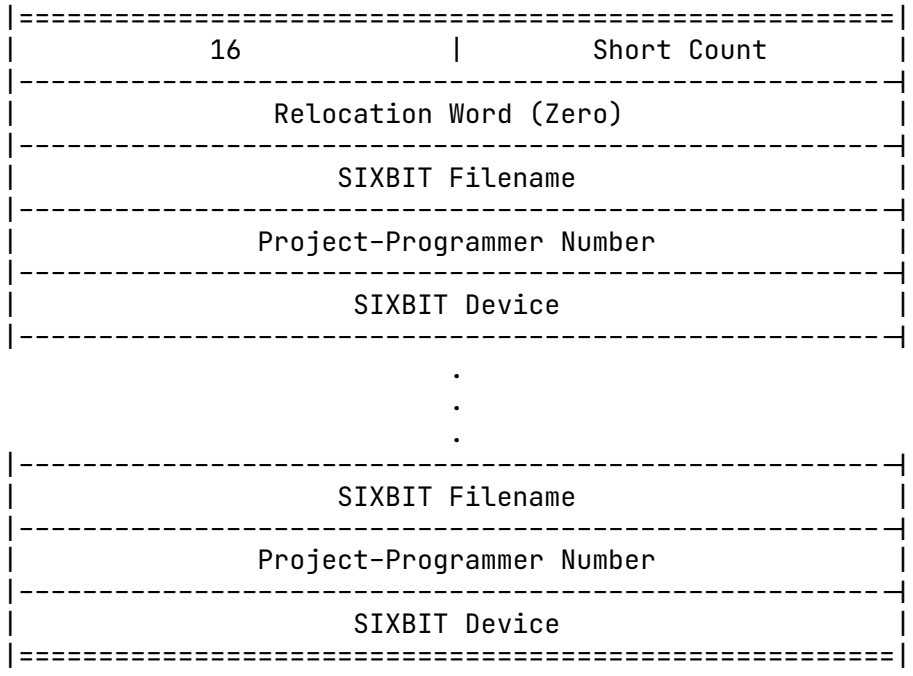
When LINK sees a REL Block Type 15, it allocates a block of the requested size at the requested address. The length of the block is then placed in the left half of the first word, and the address of the last OWN block seen is placed in the right half. If this is the first OWN block seen, 0 is stored 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 use the standard TOPS-10 format below:

```
device:name.type[project-programmer number]
```

Block Type 16 (Request Load)

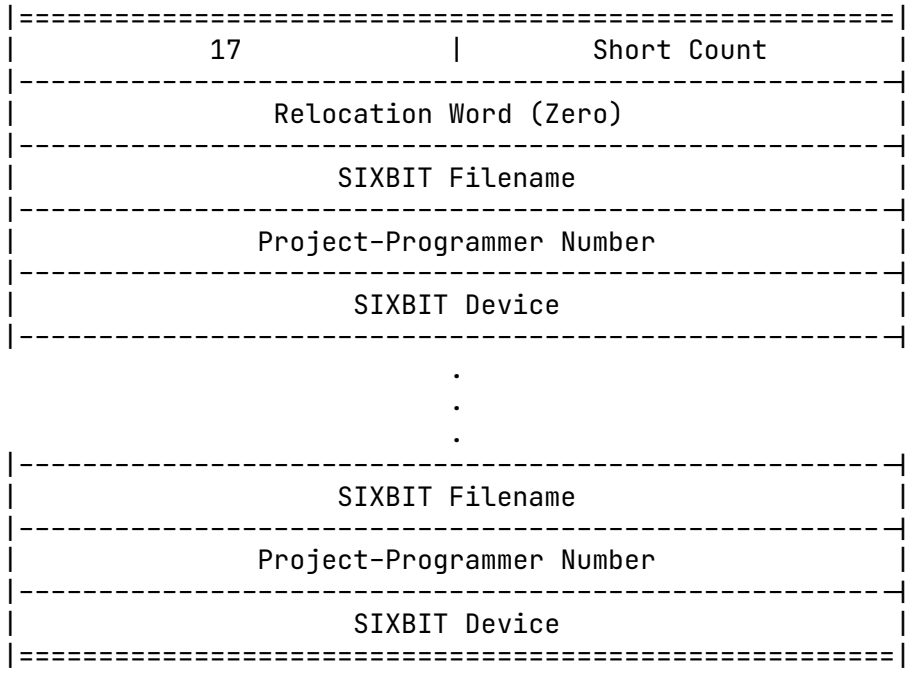


Block Type 16 contains a list of files to be loaded. The data words are arranged in triplets; each triplet contains information for one file: file name, project-programmer number, and device. The file type is assumed to be .REL.

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.

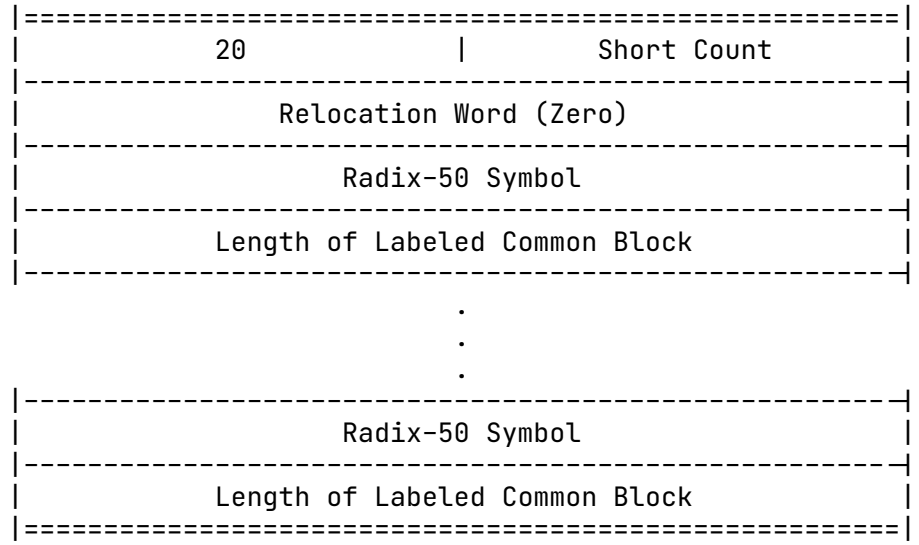
Block Type 17 (Request Library)



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)



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.

LINK allocates the specified COMMON into the current PSECT only when allocating COMMON areas in a program that contains PSECTs. The current PSECT is defined as the PSECT specified in a previous Type 22 PSECT Origin Block or Type 1051 Set PSECT Block.

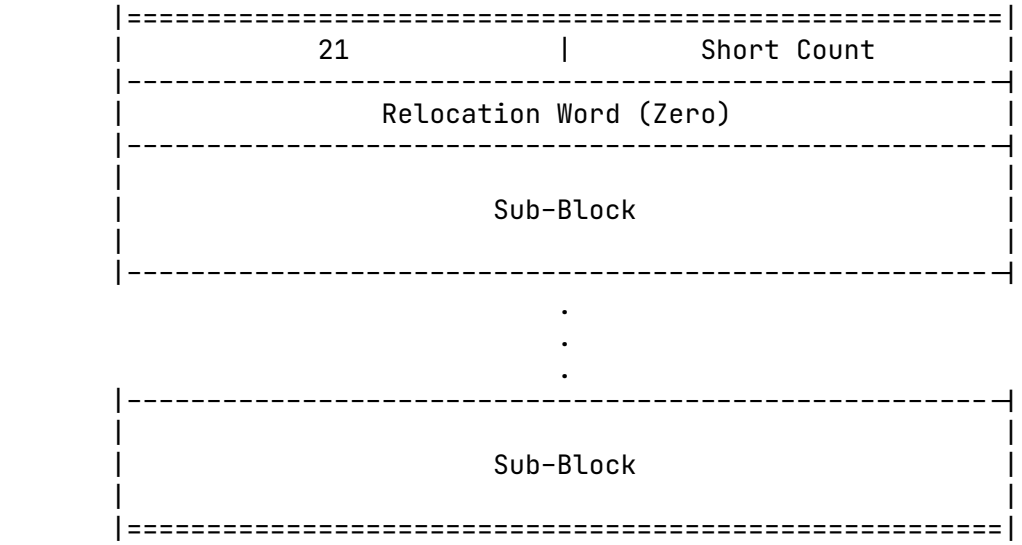
If the program does not use PSECTs, the COMMON areas are allocated into the HISEG or LOSEG as specified with the /SEGMENT switch.

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.

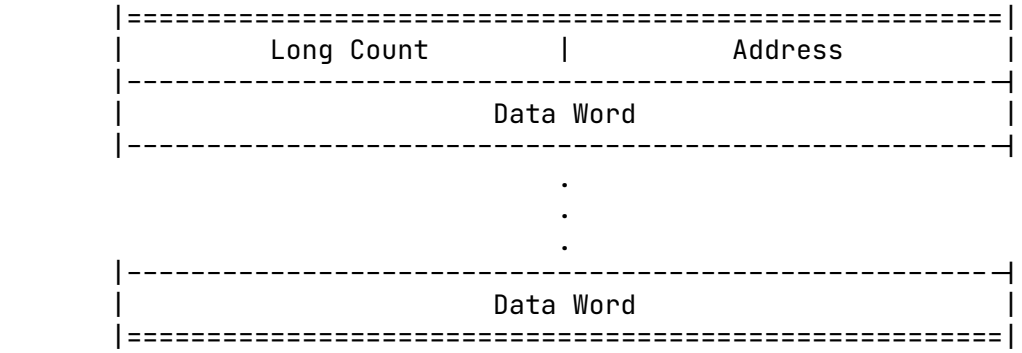
For each COMMON entry found, LINK first determines whether the COMMON area is already allocated. If not, LINK allocates it. If the area has been allocated, the allocated area must be at least as large as the current requested allocation.

COMMON blocks can be referenced from other block types as standard globally defined symbols. However, a COMMON block must be initially allocated by Block Type 20, Block Type 1074, 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)



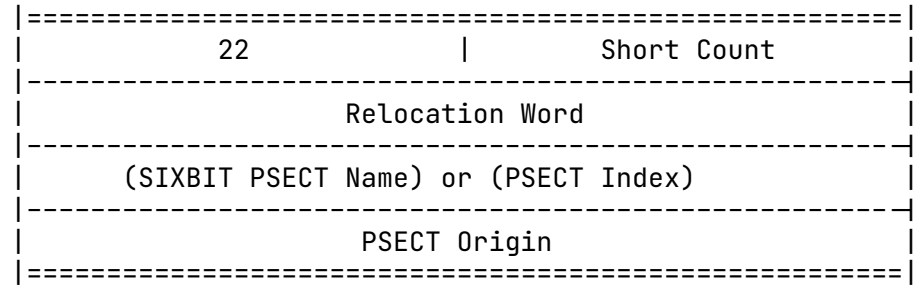
Each sub-block is of the form:



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 previously defined 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)



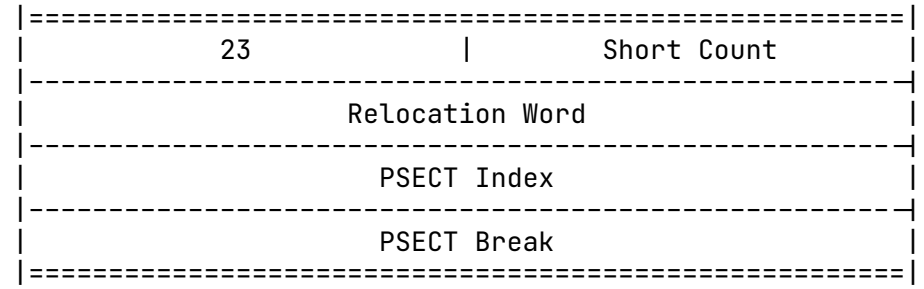
Block Type 22 contains the PSECT origin (base address).

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 TWOSEG REL blocks are relocated with respect to this PSECT until the next Block Type 22, 23, or 1051 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 to which PSECT the table refers.

Block Type 23 (PSECT End Block)

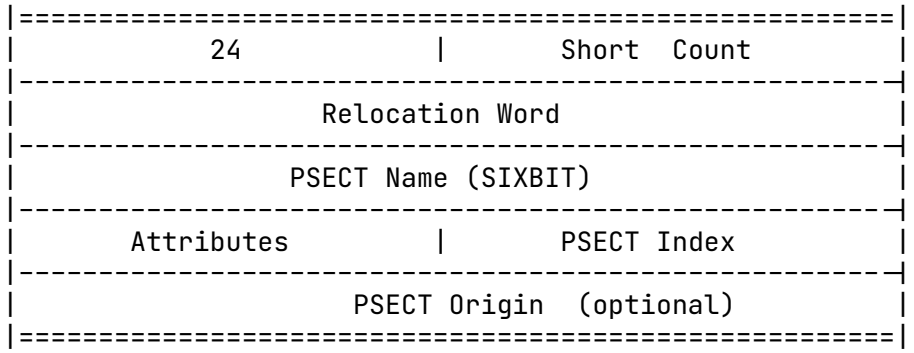


Block Type 23 contains information about a PSECT.

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 interpreted as being relative to the PSECT's origin in the current module.

Block Type 24 (PSECT Header Block)

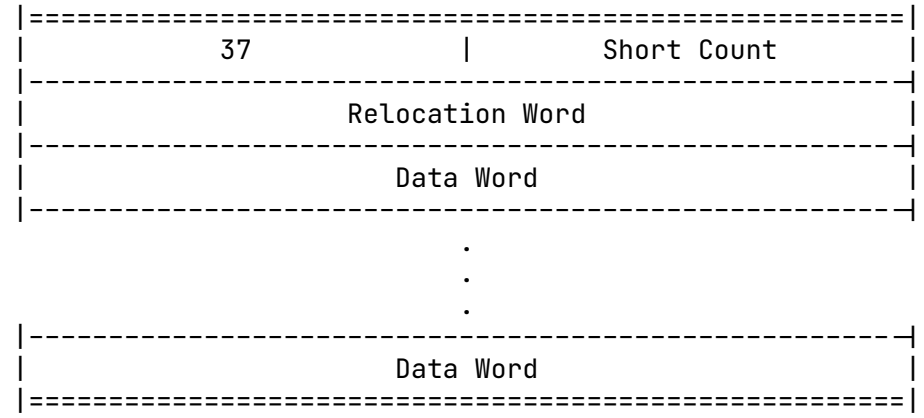


Block Type 24 contains information concerning a specified PSECT. The first word contains the block type number and the number of words associated with the block. The second word contains the relocation information. The third word contains the PSECT name in SIXBIT. The fourth word is the PSECT origin specified for this module.

Bit	Interpretation	MACRO .PSECT Keyword
11	PSECT is all within one section. This is the default.	
12	PSECT is 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

LINK must find a Type 24 or 1050 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)

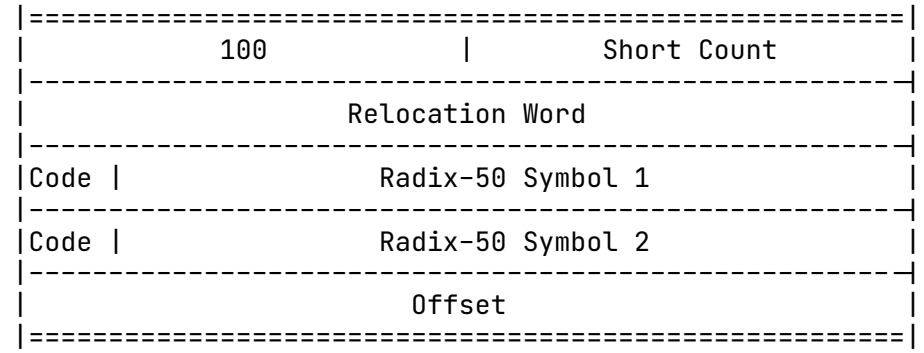


Block Type 37 contains a debugging symbol table for COBDDT, the COBOL debugging program. If local symbols are being loaded, the table is loaded.

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)



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.

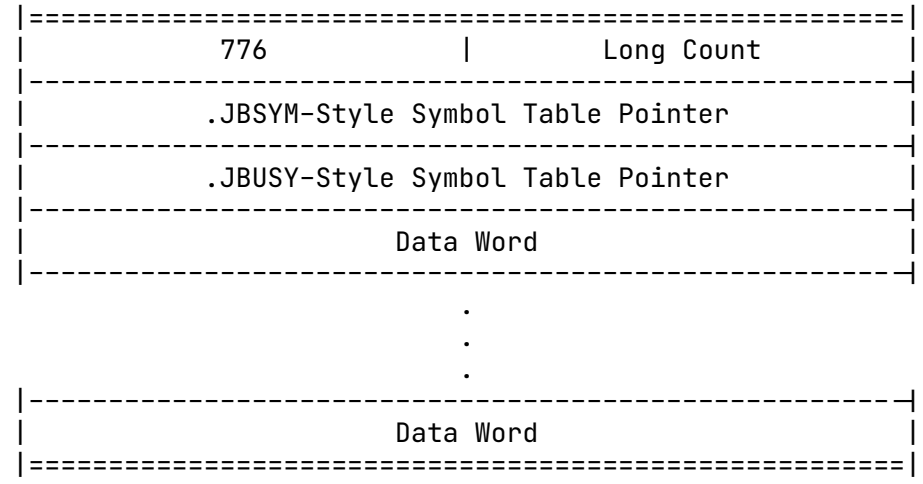
LINK ignores Code in Symbol 2.

NOTE

Symbol 2 must be completely defined when the Block Type 100 is found.

The MACRO pseudo-op .ASSIGN generates a Type 100 REL Block.

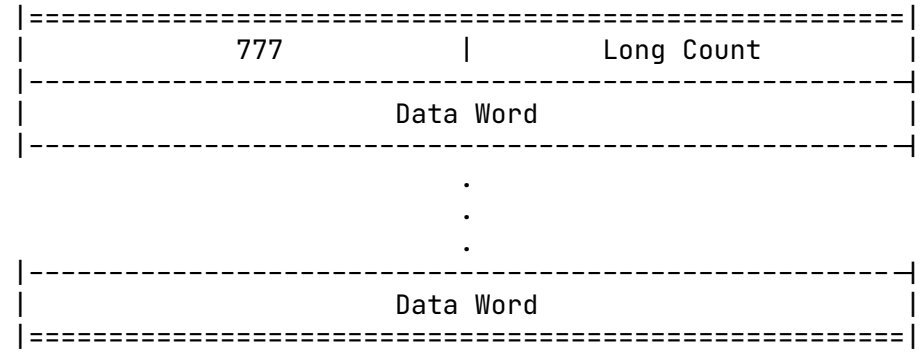
Block Type 776 (Symbol File)



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.

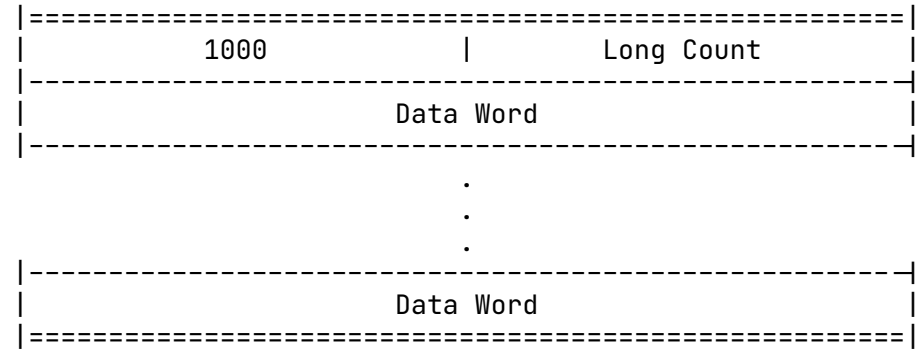
The data words form a Radix-50 symbol table for DDT in the same format as the table loaded for the switches /LOCALS/SYMSEG or the switch /DEBUG.

Block Type 777 (Universal File)



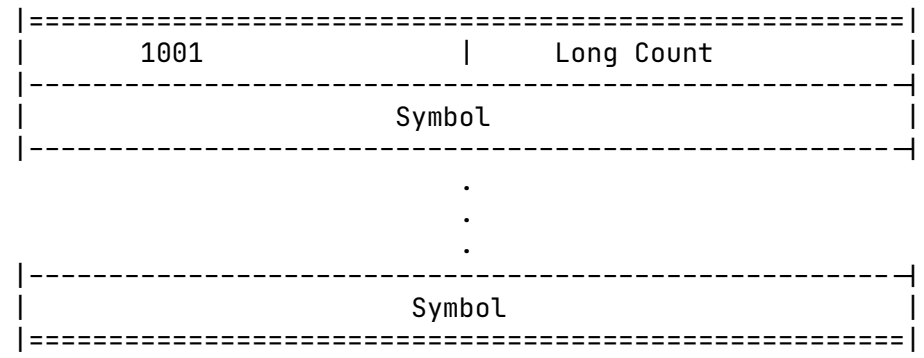
Block Type 777 is included in a universal (UNV) file that is produced by MACRO so that LINK will recognize when a UNV file is being loaded inadvertently. When a Block Type 777 is encountered, LINK produces a ?LNKUNS error.

Block Type 1000 (Ignored)



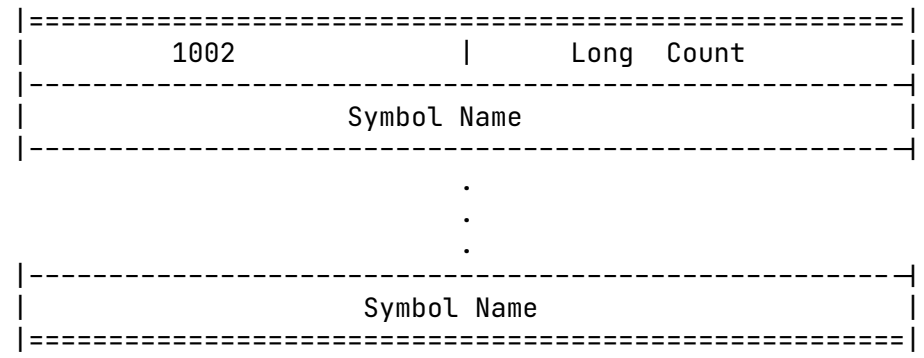
Block Type 1000 is ignored by LINK.

Block Type 1001 (Entry)



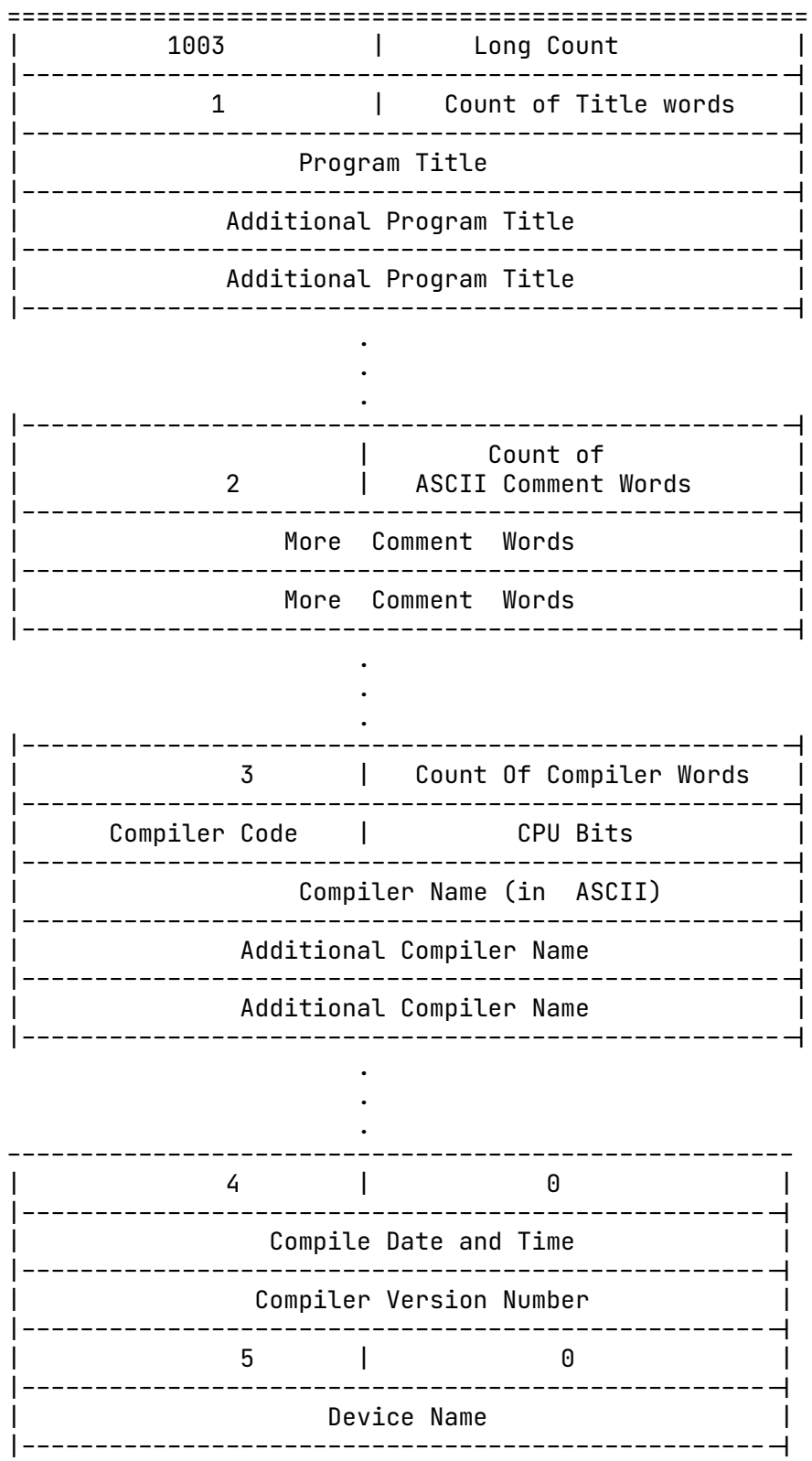
Block type 1001 is used to declare symbolic entry points. Each word contains one SIXBIT symbol. This block is similar in function to block type 4.

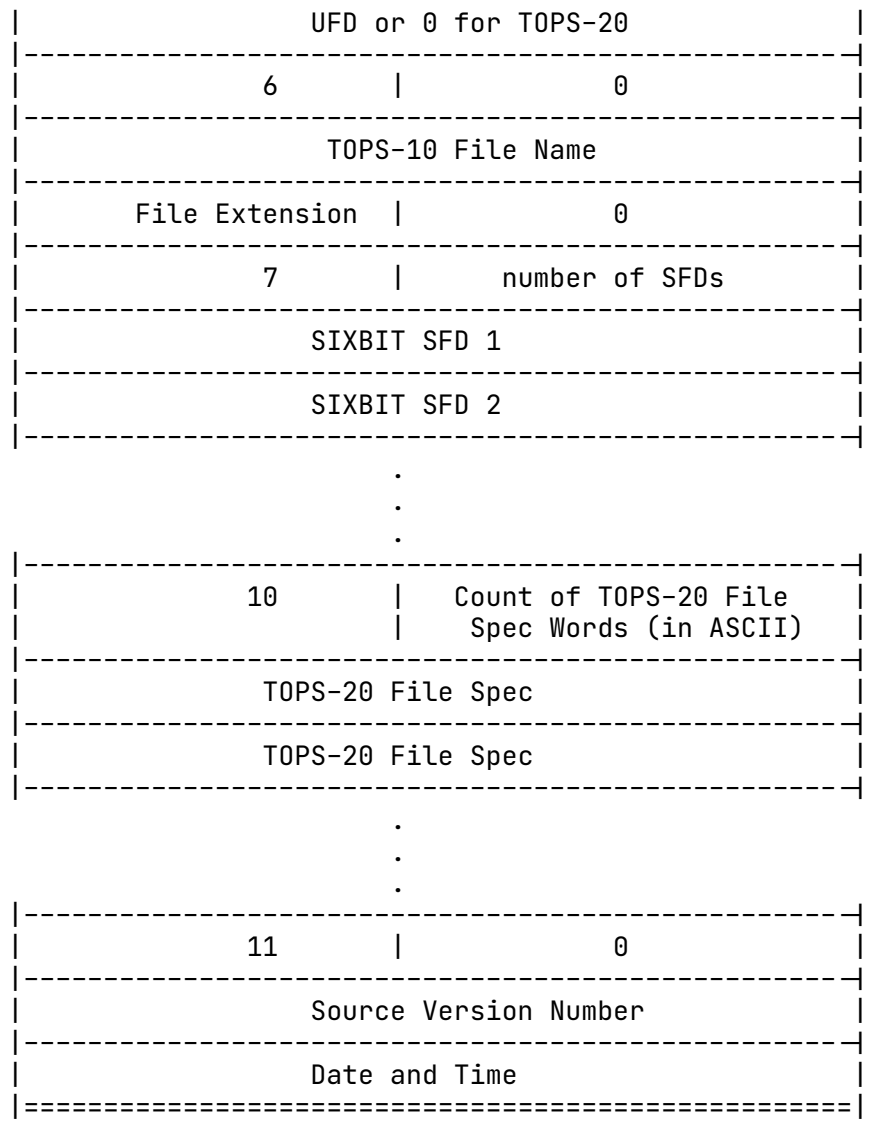
Block Type 1002 (Long Entry)



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)





Block type 1003 is used to declare long title symbols in SIXBIT and to furnish other information about the source module. This Block Type contains the information that LINK prints in the map file.

Block type 1003 consists of sub-blocks 1 through 11 (octal). The Title sub-block must be the first sub-block specified and cannot be omitted. You can omit other sub-blocks, but the sub-blocks must remain in numerical order.

The Program Title is a one word title from 1- to 72-SIXBIT characters long. You can specify a title of 0, and LINK defaults to .MAIN, but you may want to enter a more specific title.

For the compiler code and the CPU code, refer to the explanation of Block Type 6, where these codes are listed.

In sub-block 5, the Device name is where the source file resides and the UFD and SFD words are 0 for TOPS-20.

In sub-block 10, the TOPS-20 file specification must be specified in the following format:

<directory.subdirectories>filename.filetype.version number;attributes

This specification identifies the source file. LINK outputs this file specification to the map file in the order you enter it.

The Time and Date are in TOPS-10 format. The date is derived from a code. That is given by the following formula:

```
code =  31[12(year-1964)+(month-1)]+(day-1)
```

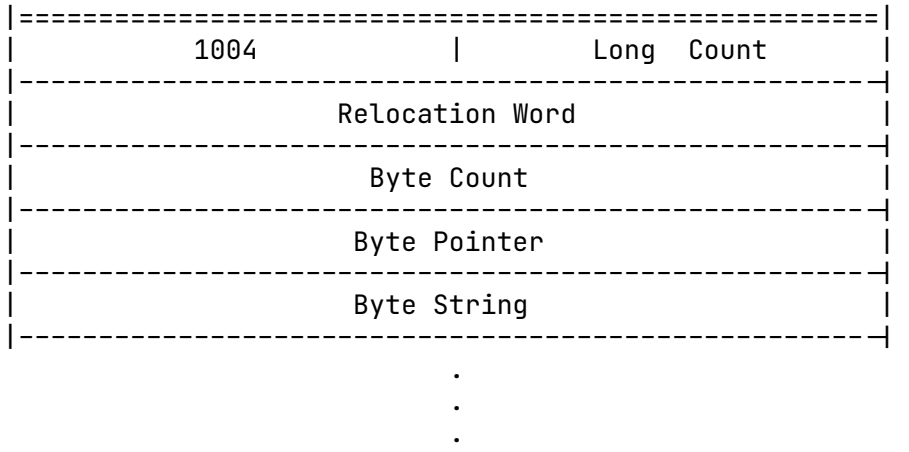
You can obtain the current day, month, and year using the formulas:

```
day = mod(code,31)+1
month = mod(code/31,12)+1
year = (code/372)+1964
```

The Time is the time in milliseconds that has elasped since midnight.

See the TOPS-10 Monitor Calls Reference Manual for additional information on date and time.

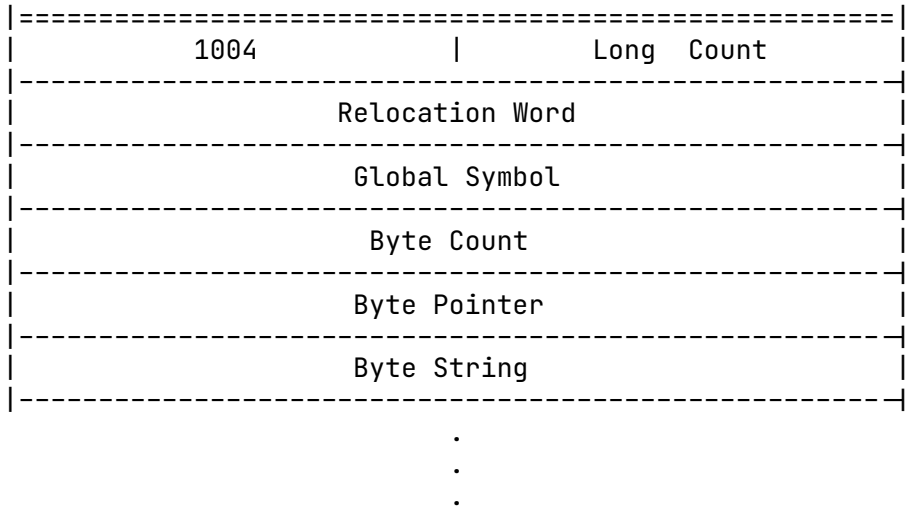
Block Type 1004 (Byte Initialization)



The above Block Type 1004 format is used to move a character string into static storage. This format uses old style relocation.

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 format for Block Type 1004 follows:



In this format, the global symbol (in SIXBIT) is used to relocate the byte pointer. The symbol must be defined when this REL block is encountered.

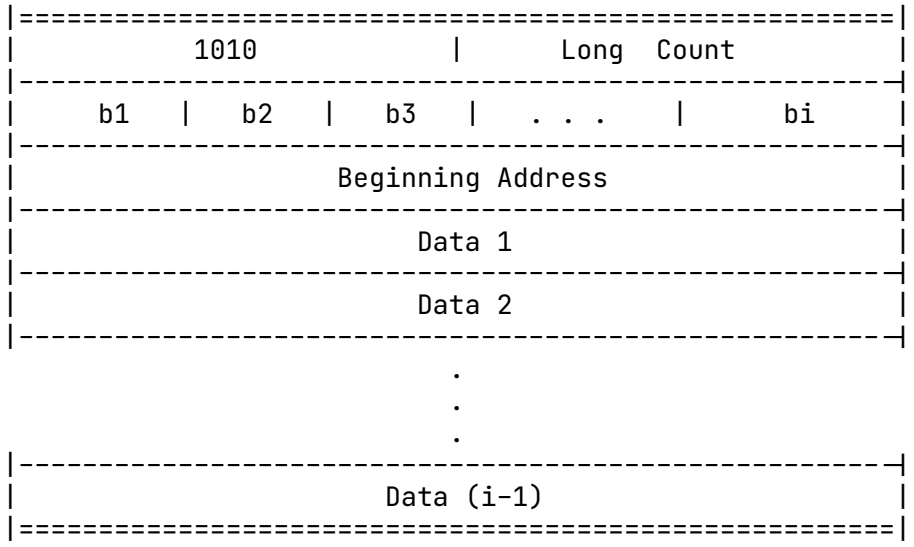
Block Types 1010 - 1037 (Code Blocks)

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)



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.

Long Count — is the length of the REL block, including all words in the block except the Header word.

b1,b2...bi — are the relocation bytes.

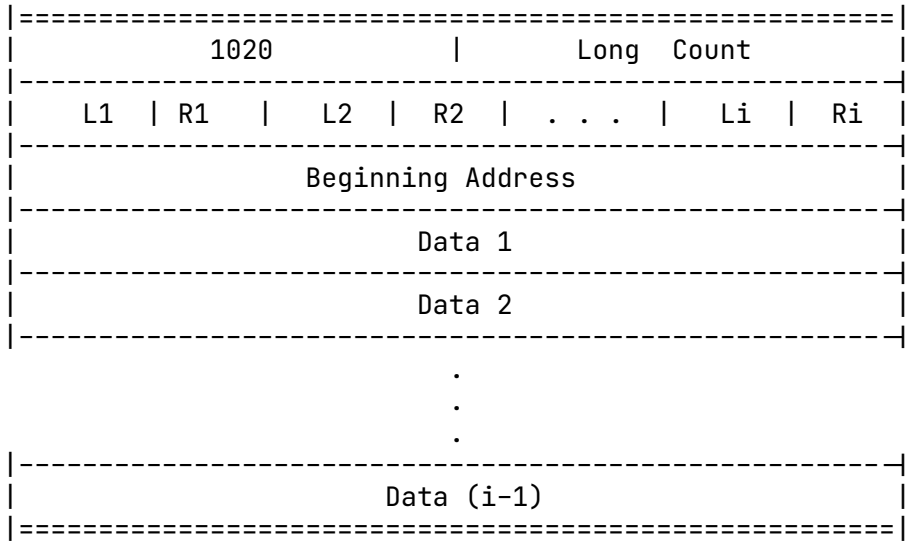
Each relocation byte contains a PSECT index number. A PSECT index must be explicitly specified. There is no default for PSECT index. A zero byte means no relocation (absolute data). All PSECT index numbers must reference predefined PSECTs. In the table below, I-value is the maximum number of PSECT indexes that can be referenced in a field.

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 30-bit address for the PSECT in "b1". 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)



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.

Long Count — 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. In the table below, I-value is the maximum number of PSECT indexes that can be referenced in a field.

Size	I-Value	Block Type
2	9	1020
3	6	1021
6	3	1022
9	2	1023

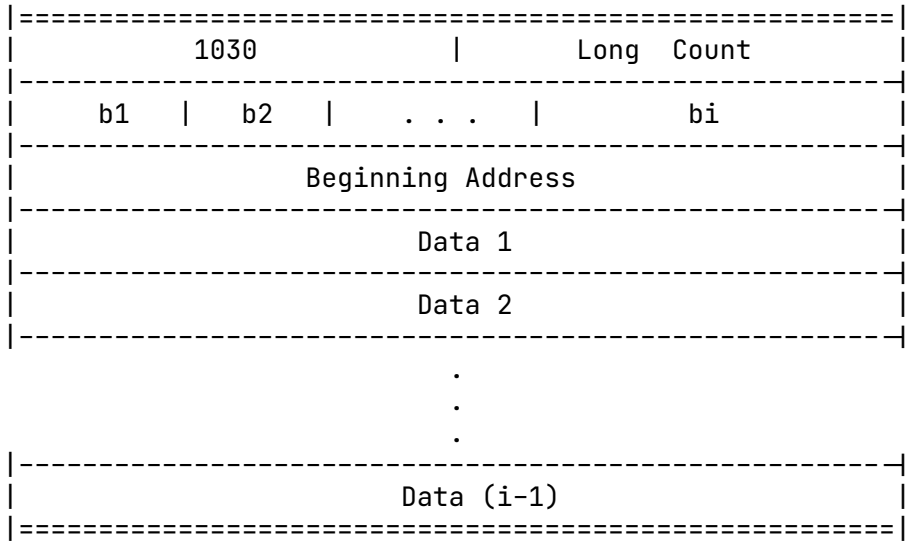
(Block Types 1024-1027 are reserved)

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 30-bit address for the PSECT in "R1". "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)



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

Long Count — 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. In the table below, I-value is the maximum number of PSECT indexes that can be referenced in a field.

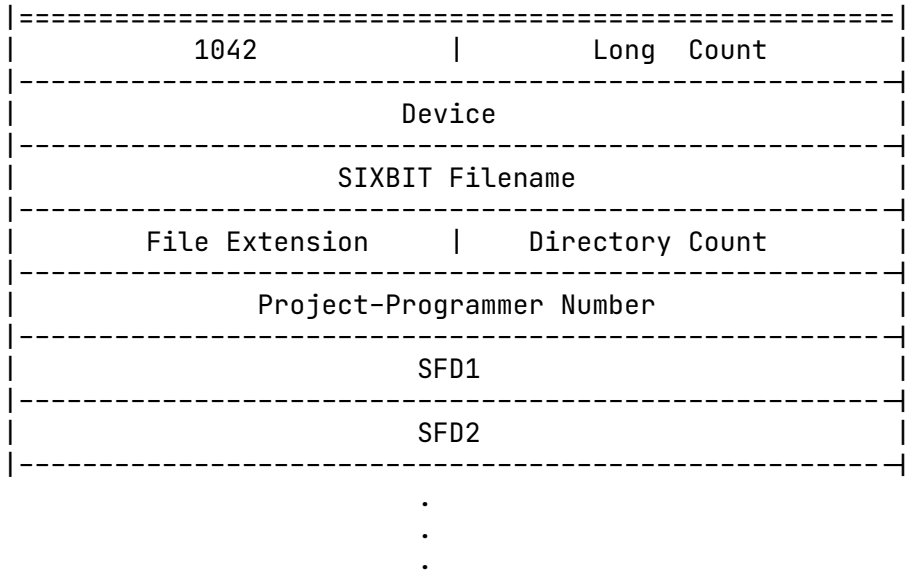
Size	I-Value	Block Type	Maximum No. of PSECTs
2	18	1030	3
3	12	1031	7
6	6	1032	63
9	4	1033	511
18	2	1034	More than 511

(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". 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.

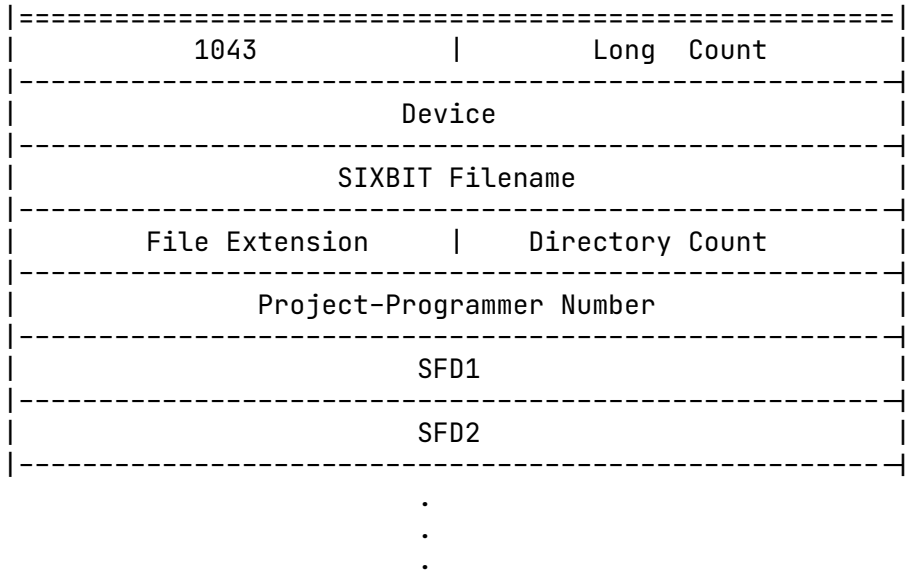
Block Type 1042 (Request Load for SFDs)



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.

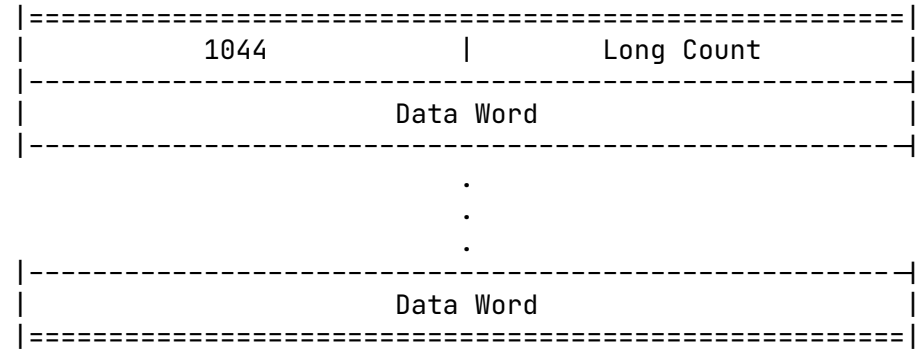
Block Type 1043 (Request Library for SFDs)



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)



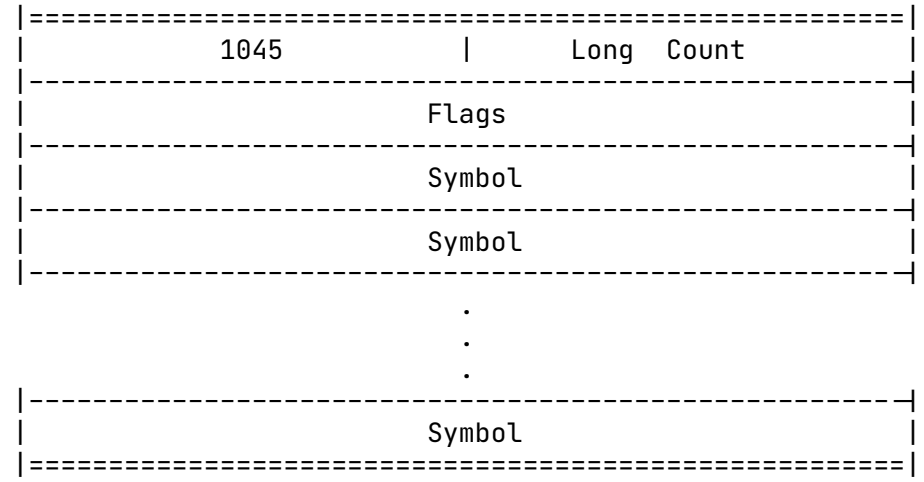
Block Type 1044 contains a debugging symbol table for ALGDDT, the ALGOL debugging program.

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)

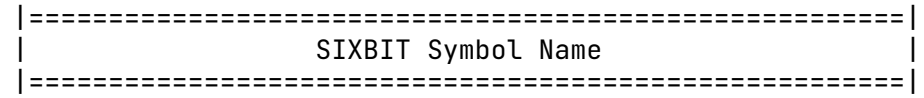


Block type 1045 declares as writable either the link containing the current module or the links containing the definitions of the specified symbols or both. This block type must follow any common block declarations (Types 20 or 6) in a module.

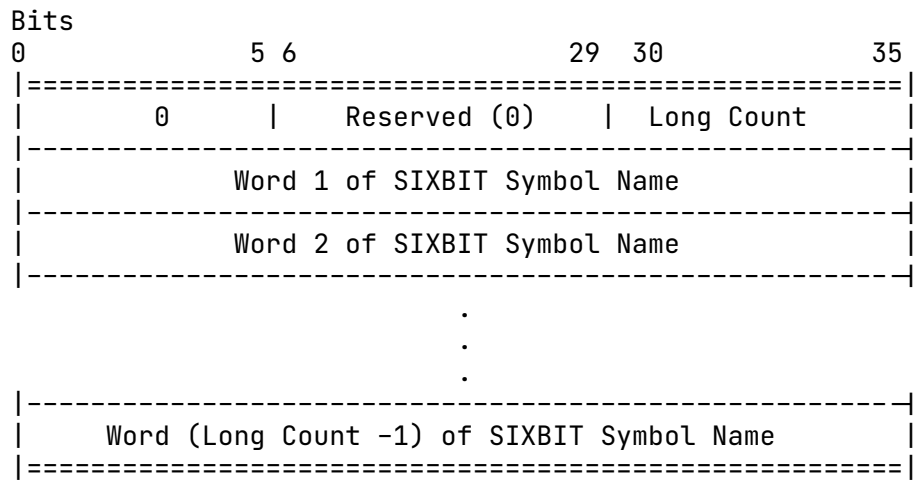
The flag word indicates which links are writable. If bit one is set then the link containing the current module and the links containing the definitions of the specified symbols are writable. If bit one of the flag word is not set then the link containing the current module is not writable, but the links 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:

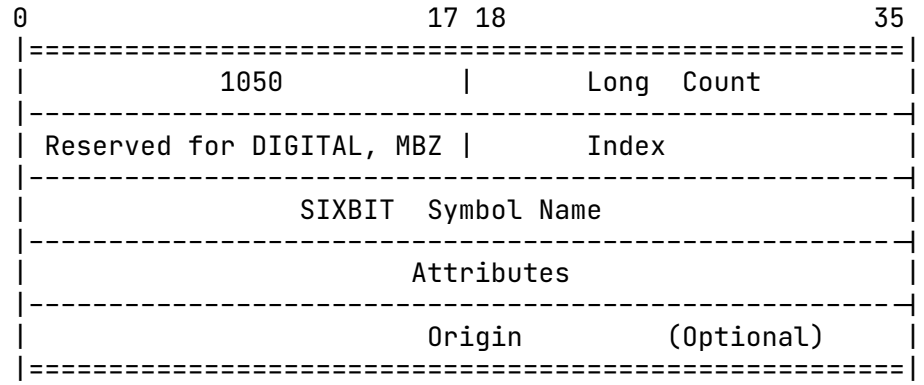


If the symbol name contains more than six characters it is represented in the following format:

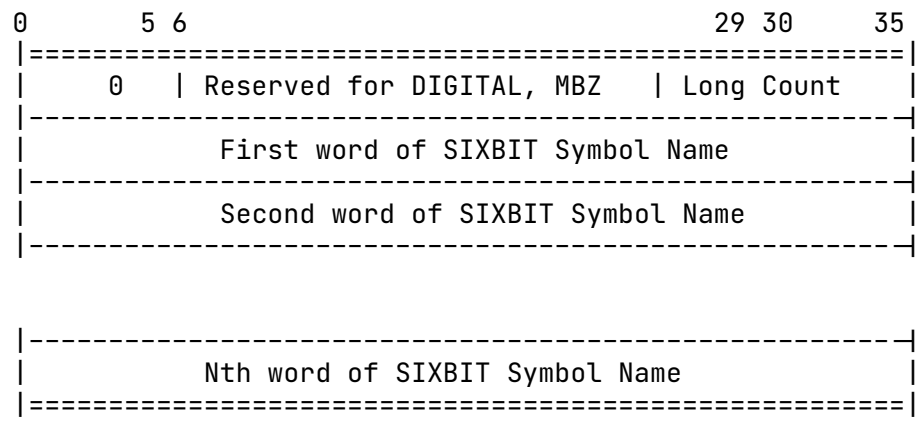


The first six bits of a long symbol are always 0. This distinguishes a long symbol name from a single word symbol name. N is the length of the symbol name including the header word. The remaining words contain the symbol name in SIXBIT, six characters to a word, left justified.

Block Type 1050 (Long PSECT Name Block)



where SIXBIT Symbol Name may be either a word of up to six SIXBIT characters, or the following block.



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. LINK clears PSECT indexes at the end of each module. PSECT indexes in any given module must be declared in consecutive order starting at index "1".

Blocks Type 1050 also assigns attributes to a PSECT and specifies the PSECT's origin address. The attributes that can be assigned are:

Bit	Description
11	PSECT is confined to one section. If this bit is set, LINK gives an error if the PSECT overflows. You can set Bit 11 or Bit 12, but not both. Bit 11 is the default. There is no equivalent MACRO .PSECT Keyword.
12	PSECT is in a nonzero section. If this bit is set, LINK gives a warning if the PSECT is placed in section zero. There is no equivalent MACRO .PSECT Keyword.

13 PSECT is PAGE-ALIGNED. PALIGNED is the equivalent MACRO .PSECT Keyword.

14 CONCATENATE parts of PSECTs seen in distinct modules.

You can set Bit 14 (CONCATENATED) or Bit 15 (OVERLAID), but not both. The CONCATENATE and OVERLAID (listed below) attributes are mutually exclusive. These attributes also span modules; so if one module sets an attribute and a later module sets a mutually exclusive attribute, LINK issues the warning:

%LNKCOE Both CONCATENATE and OVERLAY attributes specified for psect [name].

If neither is set, CONCATENATED is the default, and a warning message is not returned if subsequent pieces of the PSECT are marked OVERLAID.

CONCATENATED is the equivalent MACRO .PSECT Keyword.

15 OVERLAID parts of PSECTs seen in distinct modules. OVERLAID is the equivalent MACRO .PSECT Keyword.

16 This PSECT must be READ-ONLY.

You can set Bit 16 (READ-ONLY) or Bit 17 (WRITABLE), but not both. The READ-ONLY and WRITABLE (listed below) attributes are mutually exclusive. These attributes also span modules; so if one module sets an attribute and a later module sets a mutually exclusive attribute, LINK issues the warning:

%LNKRWA Both READ-ONLY and WRITABLE attributes specified for psect [name]. If neither is set, WRITABLE is the default, and a warning message is not returned if subsequent pieces of the PSECT are marked READ-ONLY.

RONLY is the equivalent MACRO .PSECT Keyword.

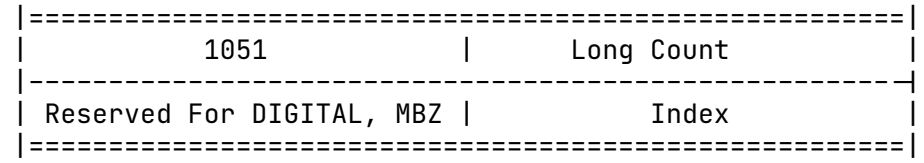
17 This PSECT must be WRITABLE. RWRITE is the equivalent MACRO .PSECT Keyword.

All other bits in the Attributes word must be 0.

The origin specified in this block is absolute.

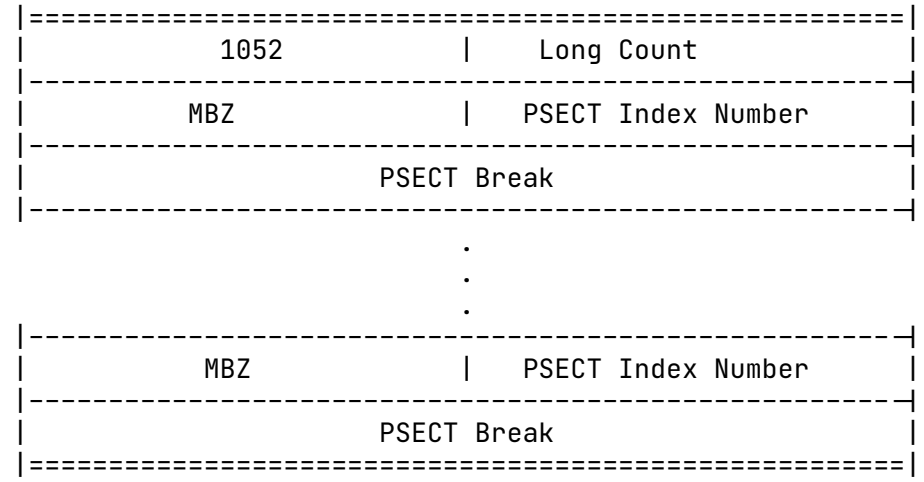
At least one block type 1050 (or the related block 24) is required for each PSECT being loaded, and this block must be loaded prior to any other blocks that reference its PSECT (that is, use the unique index number).

Block Type 1051 (Set Current PSECT)



Block type 1051 resets the "current PSECT" against which LINK relocates subsequent REL blocks if no PSECT is explicitly specified.

Block Type 1052 (PSECT End)

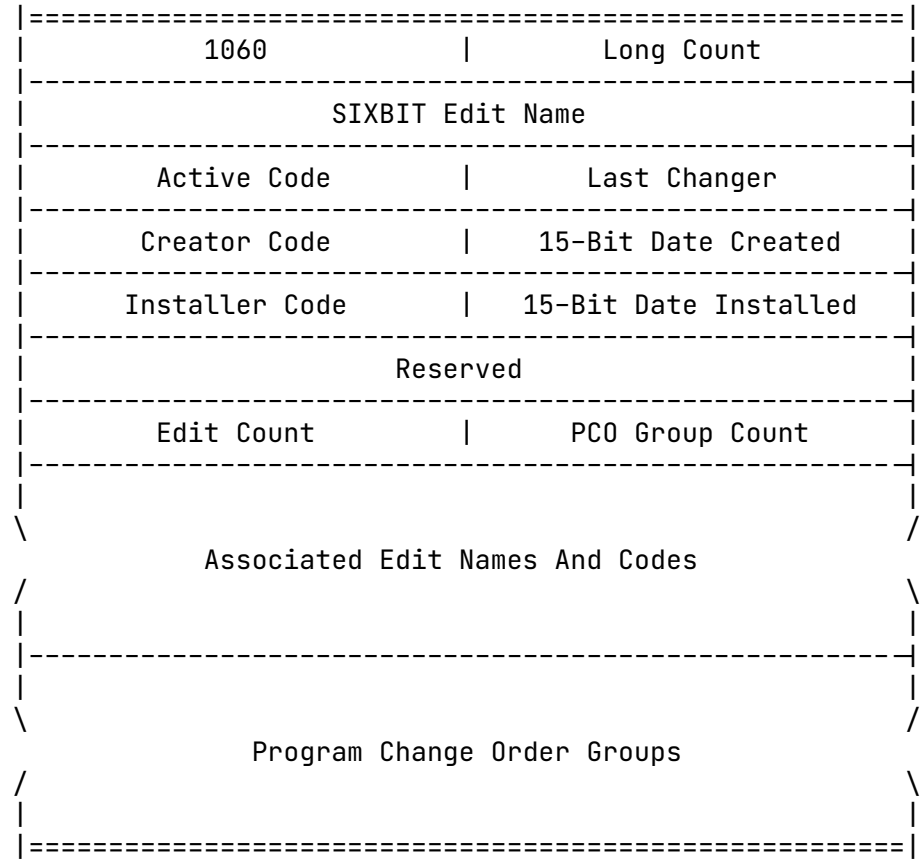


Block Type 1052 allocates additional space for a given PSECT. This space is located between the last address in the PSECT containing data and the address given by the PSECT break. A block of Type 1052 can contain more than one pair of PSECT indexes and breaks.

A module must contain a block of Type 24 (PSECT Name) or Type 1050 (Long PSECT Name) 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.

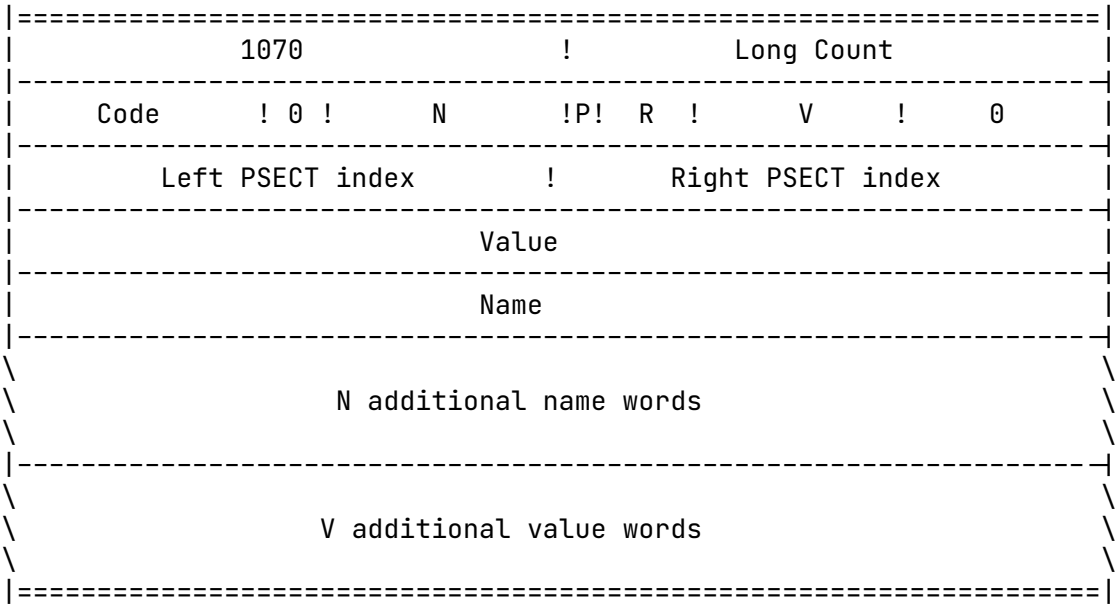
The break is interpreted as being relative to the PSECT's origin in the current module.

Block Type 1060 (Trace Block Data)



Block Type 1060 contains data used by the MAKLIB program. LINK ignores this block type.

Block Type 1070 (Long Symbol Names)



This block defines a long symbol. A symbol defined with this block can:

- be output to the DDT symbol table. Symbols longer than 6 characters are truncated when output to the DDT symbol table.
- be output to LINK MAP if requested.
- have its value relocated as specified.
- resolve global requests.

The Long Symbol Name Block is divided into two sections, the basic and the extension sections.

The basic section consists of four words: the flag word, an optional PSECT index word, the value word, and name word.

The Flags word contains information about the type of symbol, the length of the symbol name, and relocation. The optional word defines the PSECT index. The Value word contains the symbol's value. The Name word contains the symbol's 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 and value is stored in the Flag word and determines how many words are allocated for the long symbol name in the extension section. The maximum size for the symbol is 72 characters. In the case of a short symbol name only the basic section is used.

The following pages provide detailed information on the block. For each word, the field, bits, and description is given.

Field	Bits	
	Header Word	
	Description	
Block Type	0-17	1070
Block Length	18-35	Number of words used in this block
	Flag Word	
	Description	
Code	0-8	A nine-bit code field: bit 0 Must Be Zero 000 Program name 100 Local symbol definitions 110 Suppressed to DDT 120 MAP only 200 Global symbols completely defined by one word 202 Undefined 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 240 to 247 Global symbol request for chain fixup 240 Ignored (No fixup) 241 Undefined 242 Undefined 243 RH fixup 244 LH fixup 245 Undefined 246 30-bit fixup 247 Fullword fixup 250 to Global request for

257	additive fixups (the value of x has the same meaning as in 0-7 above)
260 to 267	Global request for additive symbol fixups (the value of x has the same meaning as in 0-7 above)
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
		Flag Word (Continued)
0	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.
P--PSECT Flag	18	If bit 18=0, relocate with respect to the current PSECT. No PSECT numbers are needed. If bit 18=1, relocate with respect to the PSECT specified in the next word.
R--Relocation Type	19-21	3-bit relocation type field. 0 Absolute 1 Right half 2 Left half 3 Both halves 4 30-bit 5 Fullword
V--Value field	22-28	Number of additional value words if value is a long symbol.
0	29-35	Not used
		PSECT Indexes
PSECT Indexes		Exists only if bit 18 equals 1 in the Flag word. Contains Left and Right PSECT numbers. Bit 0 and bit 18 of this word are zeros.
		Value
Value Word		Contains the symbol value, it may be relocated as specified by the relocation type and the PSECT numbers provided. Contains a symbol for 26x

codes.

Name

Name Word	Contains the symbol name in SIXBIT.

N Additional Name Words

Additional name field	Optional. It exists only if N > 0. It contains the additional characters when a long symbol name is used.

V Additional Value Words

Additional value field	Optional. It exist only if the V field is greater than 0. This field contains the additional characters when a long symbol name is being resolved. The first word contains the length of the extended field.

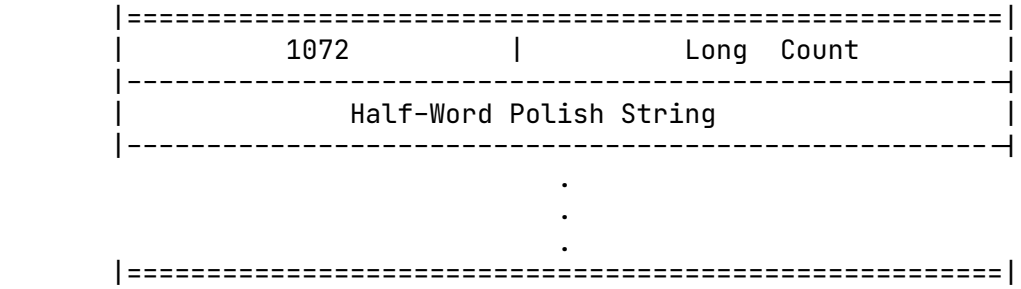
The following fixup rules apply to this block:

1. Only one fixup by a Type 2, 10, 11, 12, 15, 1070, 1072, or 1120 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.

3. A location must contain data before the location can be fixed up.

Block Type 1072 (Long Polish Block)



Long Polish Blocks of type 1072 define Polish fixups for operations on relocatable long external symbols. This Block Type is interpreted as a string of 18-bit operators and operands. The block is in Polish prefix format, with the store operator at the end of the block. Each halfword can contain one of the following:

- A halfword code in which the first 9 bits contain the data length (when applicable) and the second 9 bits contain the code telling LINK how to interpret the data that follows.

- 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 causes LINK to relocate addresses against the PSECT number specified in the "N" of the PSECT index 400000+N.

- A Polish operator.

NOTE

Operations are performed in the order in which they are encountered.

CODE DEFINITIONS

Data Packet Codes

Category	Code	Description
Operand	xxxyyy	next "xxx+1" 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+1 halfwords of symbol name in SIXBIT

NOTE

You cannot store a symbol in a single halfword. You must place the symbol in the first halfword and fill the second halfword with zeroes.

Polish Operator Codes

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 Codes

Store Operator	xxx=0 or 1	
	For xxx=0	Next two halfwords contain a Radix-50 symbol to be resolved.
	xxx777-xxx770	Chained fixup with relocatable addresses. Next xxx+1 halfwords contain the start address of the chain.
	000777	Right half chained fixup with relocatable address. Next halfword contains a relocatable address.
	000776	Left half chained fixup with relocatable address. Next halfword contains a relocatable address.
	000775	30-bit chained fixup with relocatable address. Next halfword contains a relocatable address.
	000774	Fullword chained fixup with relocatable address. Next halfword contains a relocatable address.
	001777	Right half chained fixup with relocatable address. Next fullword contains a relocatable address.
	001776	Left half chained fixup with relocatable address. Next fullword contains a relocatable address.
	001775	30-bit chained fixup with relocatable address. Next fullword contains a relocatable address.
	001774	Fullword chained fixup with relocatable address. Next halfword contains a relocatable address.
	xxx767-xxx764	Chained fixups with absolute addresses.
	000767-000764	Chained fixup with absolute address. Next halfword contains an absolute address.
	001767-001764	Chained fixup with absolute fullword

		address. Next two halfwords contain absolute address.
	xxx757-xxx754	Symbol fixup. For 1⩽xxx⩽377 the next xxx+1 halfwords contain a SIXBIT symbol name to be resolved.
	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.

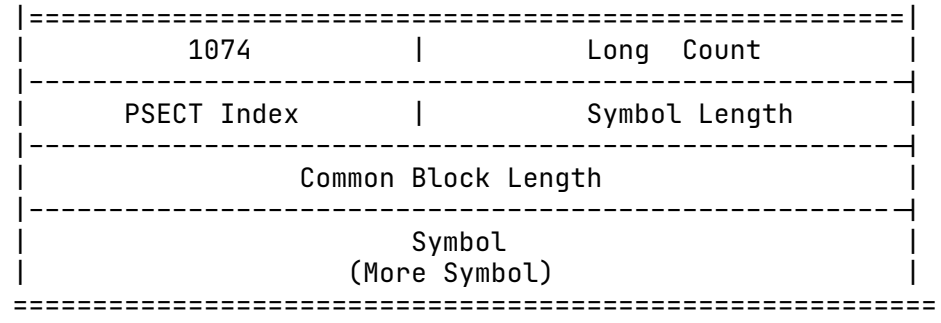
The following fixup rules apply to this block:

1. Only one fixup by a Type 2, 10, 11, 12, 15, 1070, 1072, or 1120 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.

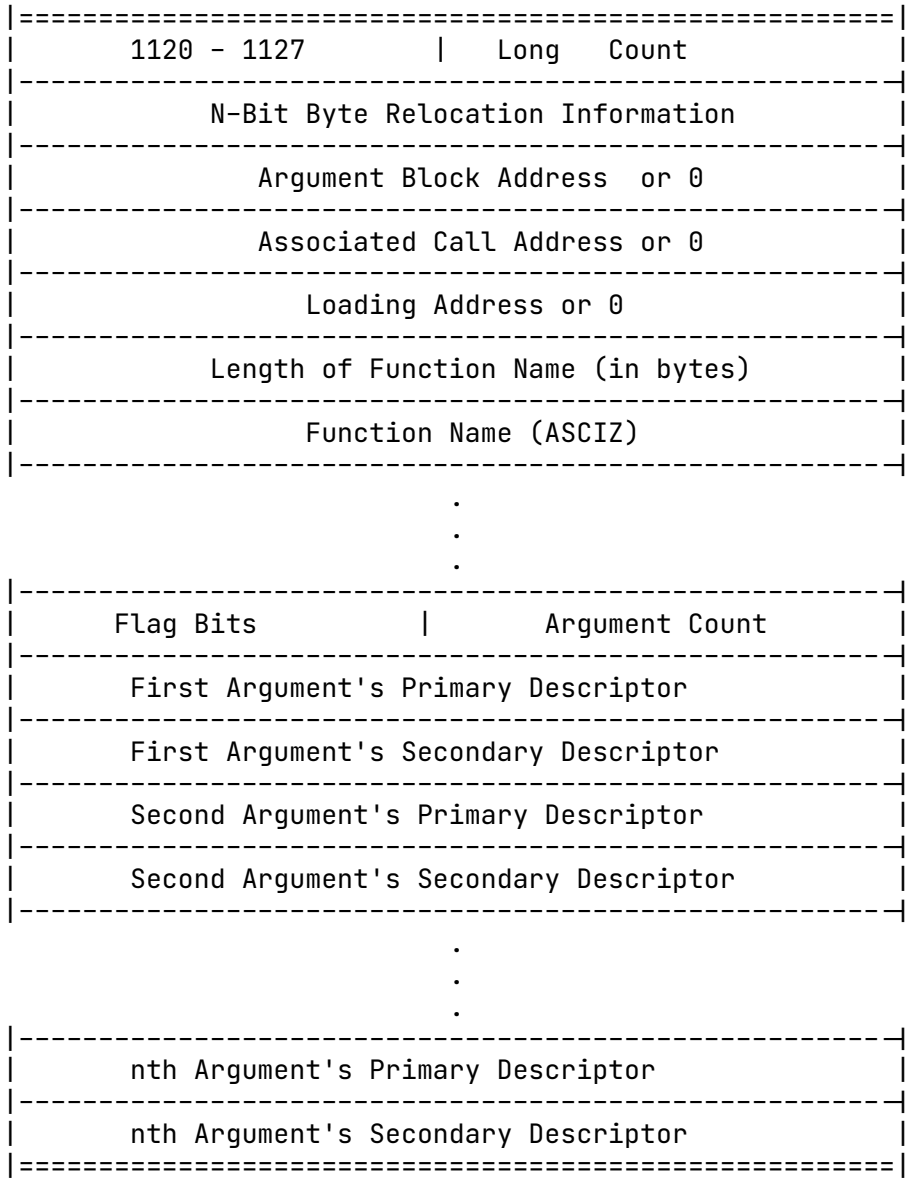
3. A location must contain data before the location can be fixed up.

Block Type 1074 (Long Common Name)



Block Type 1074 defines a long COMMON name.

Block types 1120-1127 (Argument Descriptor Blocks)



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 user memory at the loading address. This bit is ignored.
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)
12-18	(Reserved)
19-26	(Reserved)
27-35	Number of secondary descriptors

The argument type codes are as follows:

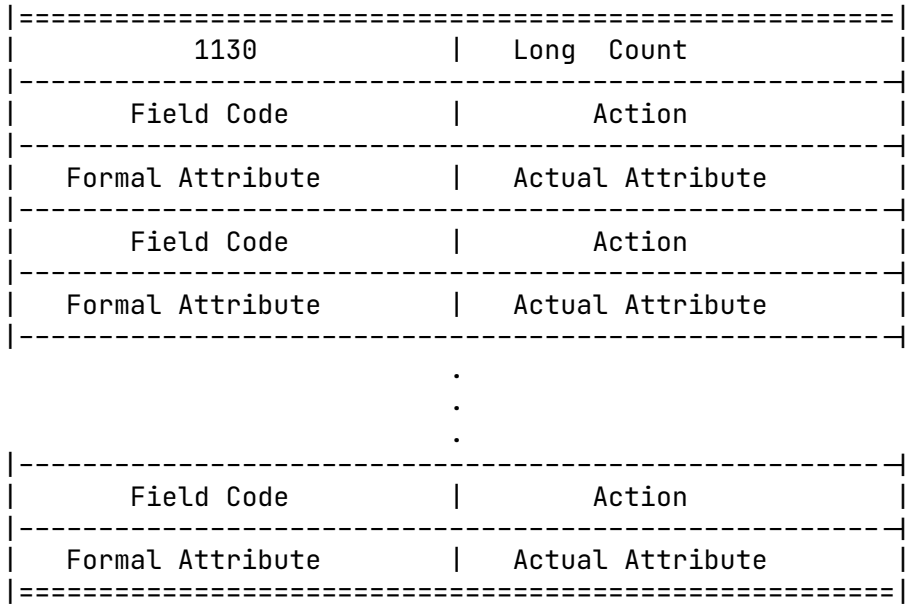
Type-Code	Usage
0	No use
1	FORTRAN logical
2	Integer
3	(Reserved)
4	Real
5	(Reserved)
6	36-bit string
7	Alternate return (label)
10	Double real
11	Double integer
12	Double octal
13	G-floating real
14	Complex
15	COBOL format byte string descriptor (for constant strings), or FORTRAN character for a program compiled with FORTRAN /NOEXTEND switch
16	BASIC shared string descriptor
17	ASCIZ string

20	Seven-bit ASCII string
21	FORTRAN character, one-word global byte pointer for a program compiled with /EXTEND

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 permissible 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)



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 last block seen is used for type checking.

If the description block and command strings are not in the same section, no error message is given.

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. However, if a byte description is not word-aligned, no 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:

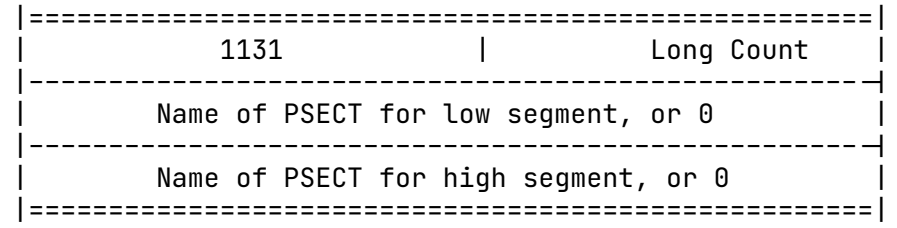
- If byte descriptor's P field is not word-aligned, issue a warning and continue.
- Pick up word address of start of string.
- If the string is not in the same section as the argument block, nonfatal error and continue.

- Put the address of the string into the associated argument block in place of the address of the string descriptor.

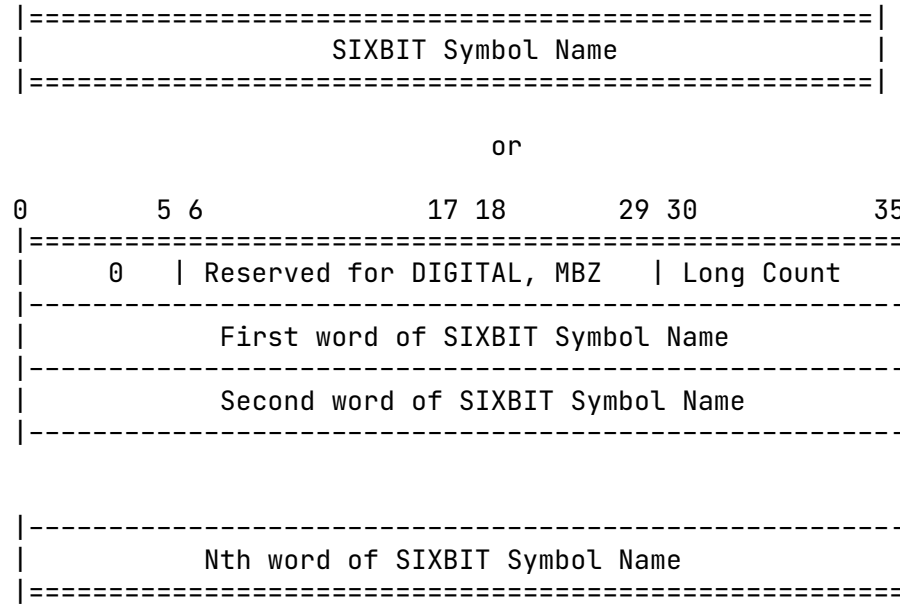
4	Suppress the message.
5-777776	Reserved
777777	Fatal error

These messages can be displayed or suppressed. Refer to the descriptions of the /ERRORLEVEL and /LOGLEVEL switches.

Block Type 1131 (TWOSEG Redirection Block)



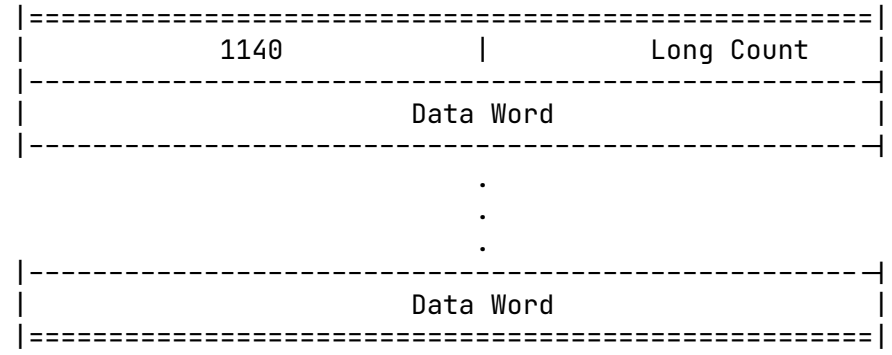
where each PSECT name has the form:



Block Type 1131 permits TWOSEG REL modules to be loaded into PSECTs by a compiler. You must redirect both the high and the low segment, you cannot redirect one or the other. Also, you cannot redirect both the high and low segment into the same PSECT.

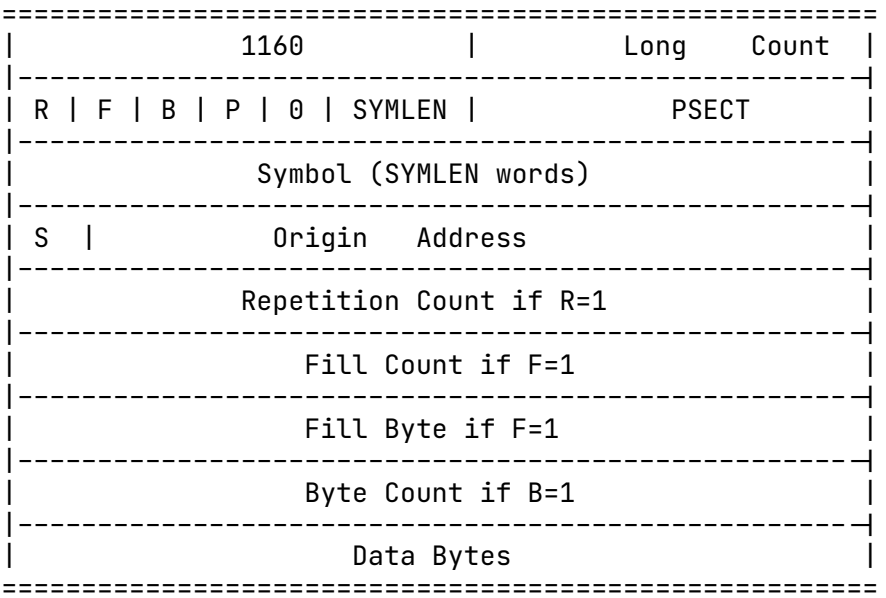
This block does not affect the current module, but all subsequent modules to be loaded.

Block Type 1140 (PL/1 debugger information)



Block Type 1140 is ignored by LINK.

Block Type 1160 (Extended Sparse Data Initialization Block)



Block Type 1160 supports the loading of data into different PSECTs and sections. This REL Block allow separate program units to load data into different bytes in the same word of memory at different times during the loading process.

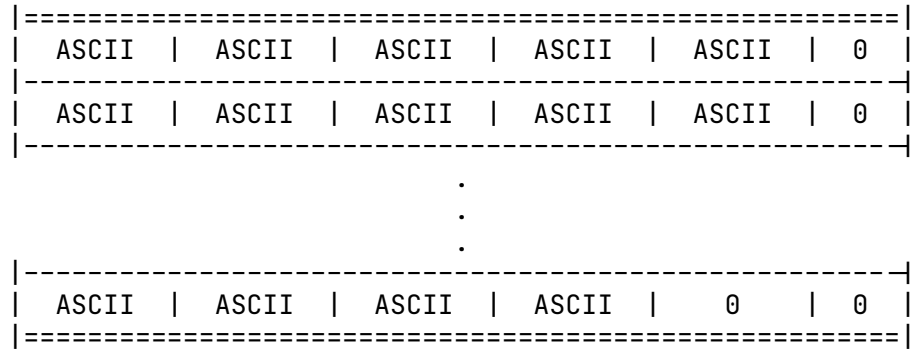
Block Type 1160 fields are described below.

Field Name	Position	Description
R	Bit 0	is a 1-bit field. If R is one, the Repetition Count word exists. If R is zero, the Repetition Count is assumed to be 1.
F	Bit 1	is a 1-bit field. If F is one, the Fill Count and Fill Byte words exist. If F is zero, no fill is used.
B	Bit 2	is a 1-bit field. If B is one, the Byte Count word exits. If B is zero, one Data Byte is assumed.
P	Bits 3-8	is a 6-bit field. This is the position within the word where the first byte is to stored.
Unused	Bit 9	is an unused bit that must be

		zero.
SYMLEN	Bits 10-17	is an 8-bit field. SYMLEN is the length in words of the global symbol to be used to calculate the address to store the byte string. If SYMLEN is zero, there is no global symbol. The value of the symbol is added to the origin address. The symbol must be completely defined before this addition occurs.
PSECT	Bits 18-35	is an 18-bit field. PSECT is the PSECT to relocate the Origin Address against. The relocation is 30-bit. If PSECT is zero, the Origin Address is absolute.
Symbol	Bits 0-35	is a SIXBIT symbol name of the length specified in SYMLEN. The value of this symbol is added to the Origin Address. The symbol must be defined when the block is seen, or a fatal error occurs.
S	Bits 0-5	is a 6-bit field. S is the size of the data bytes.
Origin Address	Bits 6-35	is a 30-bit field. Origin Address is the address where LINK begins to store Data Bytes.
Repetition Count	Bits 0-35	is a 36-bit field. If flag bit R is one, Repetition Count exists and contains the number of times to repeat the data store. The Data Bytes are stored and the fill operation is performed as many times as specified in the Repetition Count.
Fill Count	Bits 0-35	is a 36-bit field. If flag bit F is one, Fill Count exists and specifies how many times to store the Fill Byte after storing Data Bytes.

Fill Byte	Bits 0-35	is a 36-bit field. If flag bit F is one, Fill Byte exists and contains the right justified value to be used in the fill operation.
Byte Count	Bits 0-35	is a 36-bit field. If flag bit B is one, Byte Count exits and specifies the number of Data Bytes to be stored.
Data Bytes	Bits 0-35	are the data to be stored, of the length specified by the Byte Count, or 1 if flag bit B is not set. This data is stored left-justified, packed as many to a word as will fit without overlapping a word boundary.

Block Type Greater Than 3777 (ASCIZ)



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).

After loading the current REL file, LINK processes text statements in the reverse order in which they are encountered -- from the end to the beginning of the module. For example, the first, second, and third statements from the beginning of a module are processed third, second, and first. As a result, search requests in .TEXT blocks may be processed in the reverse order of entered /SEARCH switches. Keep this in mind when specifying the order the modules are to be searched.

APPENDIX B

LINK MESSAGES

This appendix lists all of LINK's messages except the messages from the overlay handler. Overlay handler messages have the OVL prefix and appear in Chapter 5. Section B.1 describes the format of messages and Section B.2 lists LINK's messages.

B.1 DESCRIPTION OF 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 **bold.** The long message follows, and appears beneath the medium message.

CODE LEV SEV MEDIUM MESSAGE

LONG MESSAGE

When a message is issued, the three letters are appended to the letters LNK, forming a 6-letter code of the form LNKxxx. For example, EXS is the 3-letter code for EXIT segment. EXS is appended to LNK to form the 6-letter code LNKEXS.

B.1.1 Message Levels

The level of a message determines whether it is returned to the terminal, the log file, or both. You can use /ERRORLEVEL to control message output to the terminal and /LOGLEVEL to control message output to the log file. Both switches suppress messages with a level of 8 or below. 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.

B.1.2 Message Severity

The severity of a message determines whether the load is terminated when the message is issued. Table B-1 lists the severity codes used in LINK, with their meanings. Severity codes are decimal. The /SEVERITY switch provides a means for lowering the severity that is considered fatal. For example, if the severity is set at 28 (the default for timesharing jobs) and you want to terminate the load if you receive a warning message, you can lower the severity to 16. This causes LINK to terminate the load if a message with a severity of 16 is issued.

The severity also determines the first character on the message line output to the terminal. This character can then be detected by the batch system. For all informational messages, the character is a square bracket ([). Informational messages also end in a square bracket (]). Warnings use percent sign (%), and fatal errors use question mark (?).

Table B-1: Severity Codes

Decimal 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.

B.1.3 Message Length

The /VERBOSITY switch determines whether the medium-length and long

messages are issued. If you use /VERBOSITY:SHORT, only the 6-letter code is issued. For example,

```
[LNKEXS]
```

If you use /VERBOSITY:MEDIUM, the medium-length message is also issued. For example,

```
[LNKEXS   EXIT segment]
```

If you use /VERBOSITY:LONG, the code, medium-length message, and long message are issued. For example,

```
[LNKEXS   EXIT segment

          LINK is in the last stages of loading your program
          (for example, creating .EXE and symbol files,
          preparing execution if requested).]
```

The /MESSAGE switch also specifies message length.

B.1.4 Message Conventions

{} indicate optional portions of the medium-length message, and are printed only in appropriate circumstances.

The JSYS error text optional message is a monitor call message. Refer to the TOPS-20 Monitor Calls Reference Manual for a description of these messages.

[] contain information pertinent to the particular error. This portion of the medium-length messages is filled in at runtime. Table B-2 describes each of these bracketed quantities.

file identifies the module or file where the error occurred. 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 last and latest redefinition and the module containing it.

Table B-2: Special Message Segments

Segment	Description
[area]	The name of one of LINK's internal memory management areas. See the /FRECOR switch in Chapter 3 for a list of these areas.
[date]	The date when LINK is running.
[decimal]	A decimal 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.
[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 attribute associated with a symbol.

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.

[1]

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

---

[1] + The level and severity of this message is determined by a compiler-generated coercion block. A coercion block specifies which data type associations are permissible and what action LINK should take if an illegal type association is attempted. See Block Type 1130 in Appendix A for more information.

two or more of its successors. You should revise your overlay structure to remove the ambiguity.

If you execute the current load, one of the following occurs 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.

CBO 31 31 Cannot build overlays outside section zero

You have tried to build an overlay structure for a program that is either too large to fit in section zero or which loads outside section zero by default. Check your LINK commands and also the assembly or compiler commands used to produce the program modules.

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.

CCE 8 8 Character constant not word aligned in call to routine [routine] called from module [module] at

location [address]

Link has detected a character constant that did not begin on a word boundary. This error is most likely the result of a language translator error. This error is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report

CCS 31 31 Cannot create section [octal]
{Detected in module [symbol] from file [file]}

LINK is unable to create the specified section. This could be because your system does not have extended addressing hardware, or because there are insufficient resources to create a section.

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.

CLD 31 28 Cannot load DDT
{JSYS error text}

DDT could not be loaded into memory with your program. The second line of the error message is the last TOPS-20 process error and indicates why the error occurred.> ;[2304]

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.

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.

CMP 31 28 Common [symbol] declared in multiple psects {Detected in module [symbol] from file [file]}

You have loaded a module which specifies that the named common block must be loaded in a PSECT which is not compatible with the psect in which it was originally loaded. Compile the module with the common in the same PSECT as the original.

CMX 8 8 Cannot mix GFloating FORTRAN compiled code with FORTRAN compiled code

You cannot load modules produced by FORTRAN with modules produced by GFLoating FORTRAN. Compile all FORTRAN modules the same way, then reload.

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.

COE 8 8 Both CONCATENATE and OVERLAY attributes specified for psect [name]

One of the modules you have already loaded explicitly sets an attribute for the named PSECT which conflicts with the declaration of PSECT attributes in the current module. Check the compiler switches or assembly language directives that were used in the generation of these modules.

COF + + Cannot open file [file] {JSYS error text}

LINK cannot open the specified file for input

CPU 31 31 Module incompatible with specified CPU {Detected in module [symbol] from file [file]}

The module you are attempting to load does not contain a .DIRECTIVE for any of the CPUs you specified with the /CPU switch. Recompile the module with the proper .DIRECTIVE, or use a different /CPU switch.

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.

CSP 31 28 Cannot setup UDDT symbol table pointers
{JSYS error text}

An error occurred while writing UDDT's symbol table pointers. Symbols may not be available for use now. This is probably because the symbol table pointers were on a write-protected memory page.

DEB 31 1 [name] execution

LINK is beginning program execution at the named debugger.

DLT 31 1 Execution deleted

Although 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 28 DDT not available
{JSYS error text}

SYS:UDDT.EXE could not be found.

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.

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 [file]
{JSYS error text}

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 [file]
{JSYS error text}

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.

EIF 31 31 Error for input file [file]
{JSYS error text}

A read error has occurred on the input file. Use of the file is terminated and the file is released.

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 [file] {JSYS error text}**

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] {JSYS error text}**

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.

ETP 31 31 **Error creating area TP overflow file {[reason]} [file] {JSYS error text}**

LINK could not make the typechecking area on the disk. You could be over your disk quota, or the disk could be full or have errors.

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).

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.

FEE * * ENTER error (octal) [reason] [file]

One of the following conditions occurred:

1. The specified file name was illegal.
2. When updating a file, the specified file name did not match the file being updated.
3. The RENAME monitor call following a LOOKUP monitor call failed.

FIN 1 1 LINK finished

LINK is finished. Control is passed to the monitor, or to the loaded program for execution.

FLE * * LOOKUP error (octal) [reason] [file]

One of the following conditions occurred:

1. The specified file name was illegal.
2. When updating a file, the specified file name did not match the file being updated.
3. The RENAME monitor call following a LOOKUP monitor call failed.

FRE * * RENAME error [reason] [file]

One of the following conditions occurred:

1. The specified file name was illegal.
2. When updating a file, the specified file name did not match the file being updated.
3. The RENAME monitor call following a LOOKUP monitor call failed.

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 [symbol] 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.

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.

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 reading area AS from file [file] {JSYS error text}

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 reading area HC
{JSYS error text}

An error occurred while reading in your high-segment code.

ILC 31 31 Error reading area LC
{JSYS error text}

An error occurred while reading in your low-segment code.

ILS 31 31 Error reading area LS from file [file]
{JSYS error text}

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 more memory than is available.

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.

INS 31 31 I/O data block not set

LINK attempted a monitor call (for example, LOOKUP, ENTER) for a channel that is not set up.

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.

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.

ISM 31 31 Incomplete symbol in store operator in Polish block (type 11 or 1072)
{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.

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.

ITP 31 31 Error reading area TP {Status [octal]} from file [file]
{JSYS error text}

An error occurred while reading in the typechecking area.

IUU 31 Illegal user UUO at PC [octal]

LINK's user UUO (LUUO) handler has detected an illegal UUO. 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.

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.
[2]

LMN 6 1 Loading module [symbol] from file [file]

LINK is loading the named 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

---

[2] + The level and severity of this message is determined by a compiler-generated coercion block. A coercion block specifies which data type associations are permissible and what action LINK should take if an illegal type association is attempted. See Block Type 1130 in Appendix A for more information.

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.

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.

MMF 31 31 Memory manager failure

The internal memory manager in LINK has failed a consistency check. This error is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.

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.

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.

NAP 31 31 No store address in polish block (type 11 or 1072)
{Detected in module [symbol] from file [file]}

The specified module contains an incorrectly formatted polish fixup block (type 11 or 1072). The store operator specifies a memory fixup, but the block ends before the address is 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.

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.

NFS 31 28 No free section for XDDT

There is no free nonzero section to load SYS:XDDT.EXE into. XDDT can only be loaded into

a section which is unused by your program. You must reconfigure your program to be able to use XDDT.

NHN 31 31 No high segment in nonzero section

You have attempted to load high segment code into a program that is being loaded into a nonzero section. Programs with high segments must be in section zero.

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.

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.

[3]

OAS 31 31 Error writing area AS from file [file] {JSYS error text}

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.

OFD 31 31 OPEN failure for device [device]

An OPEN or INIT monitor call for the specified device failed. The device may be under another user's control.>

OFN 31 31 Old FORTRAN (F40) module not available {Detected in module [symbol] from file [file]}

The standard released version of LINK does not support F40 code.

OFS 31 31 Overlay file must be created on a file structure

---

[3] + The level and severity of this message is determined by a compiler-generated coercion block. A coercion block specifies which data type associations are permissible and what action LINK should take if an illegal type association is attempted. See Block Type 1130 in Appendix A for more information.

Specify a disk device for the overlay file.

OHC 31 31 Error writing area HC
{JSYS error text}

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 writing area LC
{JSYS error text}

An error occurred while writing out your low-segment code.

OLS 31 31 Error writing area LS from file [file]
{JSYS error text}

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.

OTP 31 31 Error writing area TP {Status [octal]} from file [file]
{JSYS error text}

An error occurred while writing out the typechecking area.

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.

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:

1. There are page gaps between psects (except below the high segment).

2. There are psects above the origin of the high segment.

3. Your program will not fit in memory along with LINK's final placement code.

4. Your program's low segment has the read-only attribute, or the high segment has the read and write 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:

1. There are page gaps between psects (except below the high segment).

2. There are psects above the origin of the high segment.

3. Your program will not fit in memory along with LINK's final placement code.

4. Your program's low segment has the read-only attribute, or the high segment has the read and write 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.

PEL 31 8 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.

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.

[4]

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.

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.

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

---

[4] + The level and severity of this message is determined by a compiler-generated coercion block. A coercion block specifies which data type associations are permissible and what action LINK should take if an illegal type association is attempted. See Block Type 1130 in Appendix A for more information.

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.

RWA 8 8 Both READ-ONLY and WRITEABLE attributes specified for psect [name]

One of the modules you have already loaded explicitly sets an attribute for the named PSECT which conflicts with the declaration of PSECT attributes in the current module. Check the compiler switches or assembly language directives that were used in the generation of these modules.

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.

SPF 31 8 Splice fork failed {,saving on file [file]}

LINK was unable to replace itself with your program using the extended splice fork JSYS. The probable cause is your monitor not having the extended splice fork code installed. LINK will save your program if it has not already been saved, and will get or run it from the resulting EXE file.

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.

STL 31 31 Symbol too long

A symbol specified in a rel block is longer than the maximum allowed by LINK.

SUP 1 1 Loading suppressed

During the compilation process, errors occurred and a nonzero number was entered in JBERR. LINK does not attempt to load the REL files that result from this compile because they may be incorrect. Some compilers do not produce REL files if errors are encountered.

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.

TML 31 31 Too many links, use /MAXNODE

You have specified more overlay links than were allowed by the current value for the /MAXNODE switch. Reload your program with a larger /MAXNODE value.

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.

[5]

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.

UCB 8 8 Unknown COMMON [symbol] referenced

A reference was made to a common block, however the block does not exist.

UIE 31 31 Unexpected internal error during processing

An error occurred while processing a previous 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.

---

[5] + The level and severity of this message is determined by a compiler-generated coercion block. A coercion block specifies which data type associations are permissible and what action LINK should take if an illegal type association is attempted. See Block Type 1130 in Appendix A for more information.

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.

UMF 31 31 Unexpected monitor call failure for [JSYS] at PC [octal]
{JSYS error text}

A monitor call has unexpectedly failed. The second line of the error message is the last TOPS-20 process error and indicates why the error occurred. This may be either a system problem or a LINK internal error, depending on the message.

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.

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.

[6]

USA 8 8 Undefined start address [symbol]

You gave an undefined global symbol as the start address. Load a module that defines the symbol.

USB 8 8 Undefined symbol in byte array (type 1004) block {Detected in module [symbol] from file [file]}

LINK has detected an undefined global symbol in a type 1004 REL block. This global symbol is used to relocate a byte pointer and must be defined before the 1004 block that uses it is seen. This error is most likely the result of an error in the language translator used to generate the REL file.

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.

USD 31 31 Undefined symbol [symbol] used in loading code or data blocks

You have loaded a module that is loading code or data at a symbolic address and the symbol is currently unknown to LINK. Check the order in which you are loading modules.

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.

---

[6] + The level and severity of this message is determined by a compiler-generated coercion block. A coercion block specifies which data type associations are permissible and what action LINK should take if an illegal type association is attempted. See Block Type 1130 in Appendix A for more information.

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.

B.3 REASON EXPLANATION

The [reason] message segment of the LNKEOV, LNKETP, LNKFEE, LNKFLE, and LNKFRE messages may contain one of the following codes.

Code	Description
0	(LNKFEE and LNKFRE only) One of the following conditions occurred: 1. The specified file name was illegal. 2. When updating a file, the specified file name did not match the file to be updated. 3. The RENAME monitor call following a LOOKUP monitor call failed.
0	(LNKFLE only) The named file was not found. Specify an existing file.
1	The named directory does not exist on the named file structure, or the project-programmer number given was incorrect.
2	You do not have the sufficient access privileges to use the named file.

3 Another job is currently modifying the named file. Try accessing the file later.

4 The named file already exists, or a different file was specified on the ENTER monitor call following a LOOKUP monitor call.

6 One of the following conditions occurred:

1. A transmission, device or data error occurred while attempting to read the directory or the RIB of the named file.

2. A hardware-detected device or data error was detected while reading the named directory's RIB or data block.

3. A software-detected data inconsistency error was detected while reading the named directory's or file's RIB.

7 The named file is not saved a file. This message can never occur and is included only for completeness of the LOOKUP, ENTER and RENAME error codes. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software Performance Report (SPR) to DIGITAL.

14 You have exceeded the quota of the named directory, or the entire capacity of the file structure, Delete some files, or specify a directory or structure with sufficient space.

15 The named device is write-locked. Specify a write-enabled device or ask the operator to write-enable the named device.

24 A LOOKUP and ENTER monitor call was performed on generic device DSK: and the search list is empty.

30 A LOOKUP and ENTER monitor call was given to update a file, but the file cannot be updated for some reason For example, another user is superseding it or the file was deleted between the time of the LOOKUP and ENTER.

42 This message indicates that a LOOKUP, ENTER, or RENAME error occurred that was larger in number than the errors LINK knows about. This message is not expected to occur. If it does, please notify your Software Specialist or send a Software

Performance Report (SPR) to DIGITAL.

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 many languages and programs. In addition, two segment programs have a Vestigial Job Data Area of eight words following the high segment origin.

The /NOJOBDAT switch, described in Section 3.2.2, keeps LINK from filling in JOBDAT.

C.1 JOB DATA AREA

Address	Symbolic		Use
41	.JB41	HALT if not specified otherwise. Executes by LUUOs.	
42	.JBERR	Right:	Number of errors during loading.
74	.JBDDT	Left: Right:	Highest location occupied by DDT. Start address of DDT if loaded.
115	.JBHRL	Left: Right:	High segment length. Highest address in high segment.
116	.JBSYM	Left: Right:	Negative length of symbol table. Address of table.
117	.JBUSY	Left: Right:	Negative length of undefined symbol table. Address of undefined symbol table.
120	.JBSA	Left: Right:	First free location in low segment. Start address of program.

Address	Symbolic		Use
121	.JBFF		Right: First free location in low segment.
124	.JBREN		Right: 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:	Program version number (in octal) and flags, in the format shown below.

Bits	Meaning
0-2	Modifier flag:

Flag	Meaning
0	DIGITAL development group last modified the program.
1	Other DIGITAL employees last modified the program.
2-4	A customer last modified the program.
5-7	A customer's user last modified the program.

Bits	Meaning
3-11	DIGITAL's latest major revision number, usually incremented by 1 for each release.
12-17	DIGITAL's minor revision number, which is usually 0, unless the program has been modified since the last release.
18-35	The edit number, increased by 1 after each edit to the program. This value is never reset.

C.1.1 Vestigial Job Data Area

Offset	Symbolic	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.

GLOSSARY

absolute address	is a fixed location in user virtual address space which cannot be relocatable by the software.
ASCII	is the American Standard Code for Information Interchange. A 7-bit code in which textual information is recorded. ASCII code can represent 128 distinct characters. These characters are the upper and lower case letters, numbers, common punctuation marks, and special control characters.
assembler	translates assembly language into machine language.
relocatable binary file	contains machine language code corresponding to your source program.
chained fixup	is a linked list of locations that require a global symbol definition. The left-half, right-half, or whole words points to the next location in the list that requires the definition.
compiler	translates high-level language into machine language.
counted vector	is a region of memory whose size is in the first word of a block.
debugger	is a program that helps you locate programming errors or bugs.
entry name symbol	is a symbol in a module that contains an entry point name for other modules.
executable program	is the form of a program that is ready to be executed by the computer.

fixup	is a process LINK uses to resolve global symbol references. LINK stores the reference until it loads the module that contains the global symbol's definition. After LINK loads that module, it places the definition where the reference is stored.
global symbol	is a symbol defined in one module that can be referenced by other modules.
JOBDAT	contains information about a program in locations 20 decimal to 137 decimal such as its debugger symbol table pointer, version numbers, and memory use.
language translator	is a compiler or assembler that translates source code into machine executable format.
library	is a file that contains object modules that may be needed by programs to resolve global symbol references.
object module	contains a source program's relocatable code in machine-readable format.
PDV	stands for Program Data Vector and contains information about a program such as its debugger symbol table pointer, version number, and memory use. A PDV is used usually in place of JOBDAT for an extended addressing program.
Polish chained fixups	is a process that uses an algorithm to resolve global symbols.
PSECTs	stands for Program SECTions and are programmer or system defined regions of code or data that LINK relocates together in memory.
Radix-50	is a highly compressed code used to record textual information. The characters available in Radix-50 are uppercase letters, 0-9, ., %, $, and a space. A space is equal to 0, and cannot be used in names.
relocatable address	is an address within a module that is specified as an offset from the first location in that module.
REL blocks	are output from a language translator that make up and contain the information that LINK

	uses to load a program.
REL file	is the file produced by compilers or assemblers that contains the REL blocks used by LINK to load a program.
SIXBIT	is a 6-bit code in which textual information is recorded. It is a compressed form of the ASCII character set, therefore not all of the characters in ASCII are available in SIXBIT, notably the nonprinting characters and lower case letters are omitted.
symbol table	contains entries and values for each symbol defined or used within a program.
sharable save file	is the executable program that has been stored in a file using the LINK /SAVE switch or the TOPS-20 SAVE command. This file has an .EXE file extension.

INDEX