DEC-11-ULKAA-A-D

DOS/BATCH
Linker (LINK)

Programmer's Manual

FOR THE DOS/BATCH OPERATING SYSTEM

Monitor Version VØ9

August 1973

For additional copies, order No. DEC-11-ULKAA-A-D from Digital Equipment Corporation, Software Distribution Center, Maynard, Massachusetts Ø1754.

Your attention is invited to the last two pages of this document. The "How to Obtain Software Information" page tells you how to keep up-to-date with DEC's software. The "Reader's Comments" page, when filled out and mailed, is beneficial to both you and DEC; all comments received are acknowledged and considered when documenting subsequent manuals.

Associated documents:

DOS/BATCH Monitor
Programmer's Manual, DEC-11-OMPMA-A-D

DOS/BATCH User's Guide, DEC-11-OBUGA-A-D

DOS/BATCH Assembler (MACRO-11)
Programmer's Manual, DEC-11-LASMA-A-D

DOS/BATCH FORTRAN Compiler and Object Time System
Programmer's Manual, DEC-11-LFRTA-A-D

DOS/BATCH System Manager's Guide, DEC-11-OSMGA-A-D

DOS/BATCH File Utility Package (PIP)
Programmer's Manual, DEC-11-UPPAA-A-D

DOS/BATCH Debugging Program (ODT-11R)
Programmer's Manual, DEC-11-UDEBA-A-D

DOS/BATCH Librarian (LIBR)
Programmer's Manual, DEC-11-ULBAA-A-D

DOS/BATCH Text Editor (EDIT-11)
Programmer's Manual, DEC-11-UEDAA-A-D

DOS/BATCH File Compare Program (FILCOM)
Programmer's Manual, DEC-11-UFCAA-A-D

DOS/BATCH File Dump Program (FILDMP)
Programmer's Manual, DEC-11-UFLDA-A-D

DOS/BATCH Verification Program (VERIFY)
Programmer's Manual, DEC-11-UVERA-A-D

DOS/BATCH Disk Initializer (DSKINT)
Programmer's Manual, DEC-11-UDKIA-A-D

Trademarks of Digital Equipment Corporation include:

DEC	PDP-11
DIGITAL (logo)	COMTEX-11
DECtape	RSTS-11
UNIBUS	RSX-11

PREFACE

This document describes the features and operations of the LINK-11 Linker, a system program for the PDP-11 DOS/BATCH Monitor System. The reader is expected to be familiar with the DOS/BATCH Monitor, Macro-11 Assembler, and PIP programs, as described in their respective documents listed on the previous page.

DOCUMENTATION CONVENTIONS

As shown in the examples herein, command strings are typed in response to the underlined . and $ and # characters.
All command strings are terminated with the RETURN key.

NOTE

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION TO LINK

The PDP-11 Disk Operating System (DOS/BATCH) software includes the Linker Program (LINK), which is a system program for linking and relocating user programs assembled by an assembler or generated by a compiler running under DOS/BATCH. LINK enables the user to assemble separately his main program and various subprograms without assigning an absolute address for each segment at assembly time.

LINK processes the binary output (object module) of an assembly as follows:

Relocates each object module and assigns absolute addresses.

Links the modules by correlating global symbols defined in one module and referenced in other modules.

Produces a load map, which displays the assigned absolute addresses.

Creates a load module that can be loaded subsequently (by the Monitor or the Absolute Loader) and executed.

The advantages of LINK include:

The source program can be divided into sections (usually subroutines) and assembled separately. If an error is discovered in one section, only that section needs to be reassembled. LINK can then link the newly reassembled object module with other object modules already existing. Similarly, a general-purpose module can be assembled and used within several different main programs.

Absolute addresses need not be assigned at assembly time; LINK automatically assigns them. This prevents programs from arbitrarily overlaying each other, and also allows subroutines to change size, thereby influencing the placement of other routines but not affecting their operation.

Separate assemblies allow the total number of symbols to exceed the number allowed in a single assembly.

Internal symbols (which are not global) need not be unique among object modules. Thus, naming rules are required for global symbols only, as when different programmers prepare separate subroutines for a single run-time system.

Subroutines may be provided for general use in object module form to be linked into the user's program.

LINK requires at least a PDP-11 capable of running DOS/BATCH with a disk and a keyboard. DECtape, high-speed paper tape reader and punch, a line printer and extra memory can be used if available. A line printer provides a fast display device for the load map listing.

1.1 GLOBAL SYMBOLS

Global symbols provide the links, or communication, between object modules. Symbols that are not global are called internal symbols. If a global symbol is defined (as a label or by direct assignment) in an object module, it is called an entry symbol and other object modules can reference it. If the global symbol is not defined in the object module, it is an external symbol and is assumed to be defined (as an entry symbol) in some other object module.

As LINK reads the object modules it keeps track of all global symbol definitions and references. It then modifies instructions and/or data that reference the global symbols.

1.2 RELINKING LINK

LINK is provided as a system program with the DOS/BATCH Operating System. Procedures that enable you to relink LINK using the LINK object modules can be found in the DOS/BATCH System Manager's Guide. The resulting LINK program assumes a top of memory corresponding to the system configuration; this can be overridden using the T (top) or B (bottom) switches (see Section 3.2.1.1).

The top address assumed by LINK is:

077460 for 16K
117460 for 20K
137460 for 24K
157460 for 28K

CHAPTER 2

INPUT AND OUTPUT

2.1 INPUT MODULES

Input to LINK is one or more object modules, which can be output from the DOS Assembler, FORTRAN compiler, or other system program.

On its first pass through a module, LINK reads the object module to gather enough information so that absolute addresses can be assigned to all relocatable sections and all globals can be assigned absolute values. This information appears in the global symbol directory (GSD) of the object module.

On its second pass, LINK reads all of each object module and produces a load module and link map. The data gathered during the first pass is used to guide the relocation and linking process of the second pass.

2.2 OUTPUT MODULE

The normal output of LINK is a load module, which can be loaded and run by the DOS/BATCH Monitor. A load module consists of formatted binary blocks of absolute load addresses and object data as specified for the Absolute Loader and the Monitor Loader. The first few words of data will be the communications directory (COMD), and will have an absolute load address equal to the lowest relocated address of the program (see Appendix C).

LINK can also produce a contiguous format file (see Appendix C); this format consists of an actual core image of the user program. This format is used mainly in the production of overlaid programs (see Chapter 4).

2.2.1 Absolute Loader

As described above, a communications directory (COMD) is included at the beginning of a load module. If the COMD is loaded by the Absolute Loader, it will be overlaid by normal code in the program, since the data in the COMD is not needed by the Absolute Loader. This overlaying of the COMD by the relocated program allows the Absolute Loader to handle load modules with a COMD. However, a problem arises if a load module is to be loaded by the Absolute Loader and either of the following conditions exists:

1. The object modules used to construct the load module contain no relocatable code, or

2. The total size of the relocatable code is less than the size of the COMD.

In either case, there would not be enough relocatable code to overlay the COMD, which means that the COMD will load into parts of core not intended by the user to be altered. LINK will select the COMD's load address such that the COMD will be against the current top of the area being linked (see T switch in Section 3.2). If the top is very low, LINK will not allow the COMD to be loaded below address 0; it will load it up from 0.

2.2.2 Program Transfer Address

If a transfer address is not specified by a switch, it is assumed by LINK to be the first even transfer address encountered in the object input. Thus, if four object modules are linked together and if the first and second have a .END statement without a transfer address, the third an .END A, and the fourth an .END B, the transfer address used would be A of module three.

2.3 LOAD MAP

The load map produced by LINK provides several types of information concerning the organization of the load module. The map begins with the load module filename and extension, time and date of creation, followed by the transfer address and the low and high limits of the relocatable code. At this point is placed a synopsis of Program Section arrangement describing the placement of each Program Section relative to the other. Then there is a section of the map for each oject module included in the linking process. Each of these sections begins with the module's name, identification if specified via the assembler .IDENT assembly directive, and the filename from which the module was obtained, followed by a list of the control sections and their entry points. For each control section, the base of the section (its low address, the top of the Section (its high address), and its size (in bytes) are printed to the right of the section name (enclosed in angle brackets). Following each Section name is an alphabetically ordered list of entry points and their addresses. The load map is concluded with a list of undefined symbols for each object module.

Note that modules are loaded such that the first specified in the command string to LINK is lowest in memory.

Appendix H describes load map formats in detail.

Note that an existing map file is deleted from a device before a new map file of the same name and extension is ouput to the device.

CHAPTER 3

OPERATING PROCEDURES

3.1 LOADING

LINK can be loaded into core from the disk by typing the following Monitor command.

.RUN LINK <CR>

NOTE

In the examples, typing the RETURN, LINE FEED, and SPACE keys are shown as <CR>, <LF>, and <SPACE>, respectively. Also, in the examples, program printout is underlined; user-typed input is not.

When LINK is loaded and ready to accept the user's command, it prints the following lines:

LINK Vxx (where xx is the LINK version number)
#

The user can now type a command string as described below.

3.2 COMMAND STRING

Commands are typed in response to the number sign, #, printed by LINK. The format of the command string adheres to the requirements of the DOS Command String Interpreter (CSI), as explained in the Disk Operating System Monitor Programmer's Handbook.

The Linker's file specifications must appear in the following order:

#load module, load map, symbol table < object modules <CR>

A null specification field signifies that the associated output is not desired. A complete file specification contains the following information:

dev:filnam.ext[uic]/s1:v/s2:v.../sn:v

The default values for each output specification are noted below.

	dev	filnam	ext	uic
Load Module	*	**	LDA	This user
Map Output	*	none	MAP	This user
Object Module	*	none	OBJ	This user
Symbol Table	*	none	STB	This user

*system device (SY:) or last device specified on this side of the < symbol
**the filename from the first input specification

If a syntax error is detected in a command string, LINK prints the command on the teleprinter up to and including the character in error, followed by a question mark, and then a line beginning with the input request character #. The user must re-type the entire command correctly.

If a command string to LINK requires more than one line at the keyboard (for example, when using the /IN or /EX switches), switch values can be continued on from one to three succeeding lines by typing a colon (:) at the end of each line to be continued. The colon can be used only to continue a series of switch values; the individual values cannot be broken up over two lines. See section 3.2.1.10 for an example of the proper usage of the colon to continue command strings.

Optionally, command input can be taken from a file. Such a file is called an "indirect command file", and can be specified anywhere in the command input stream. Normally, input is accepted from the keyboard; when a keyboard command line begins with an @ character, the subsequent characters are assumed to specify an indirect file.

Example:

```
@ INDIR.FIL
```

where INDIR.FIL is a DOS/BATCH filename and extension, causes commands to be obtained from the file INDIR.FIL.

NOTE

No file extension default exists for indirect files.

Upon encountering an indirect file, LINK stacks the current command file specification (i.e., the keyboard or another indirect file) and opens the specified indirect file. Commands are then read from the file until

1. another indirect file is specified, or
2. the end-of-file is reached.

Upon reaching end-of-file, the current command file is closed, and the next file (the one on the top of the file stack) is unstacked. Subsequent command lines are then read from this file until

1. another indirect file is specified, or
2. the end-of-file is reached.

LINK allows five such nested levels of indirect files. This should be adequate for most applications, but can be changed, if desired, through an assembly option.

The use of indirect command files relieves the user of the burden of typing repetitive commands at the keyboard, and makes possible

batching of commands.

3.2.1 Switches

The command switches associated with LINK are:

Input Switches

/T	Top
/B	Bottom
/OD	ODT
/CC	Concatenated File
/TR	Transfer Address
/E	End
/L	Library
/GO	Go
/MP	Overlay Mapping Description
/O	Options
/IN	Include
/EX	Exclude

Map Switches

/LG	Long map
/SH	Short map
/CR	Global Cross-Reference

Load Module Output Switches

/CO	Contiguous
/SQ	Control Section sequencing

The mnemonic representing each switch is always preceded by the slash symbol.

If a value is specified for a switch that does not require a value, the specified value is ignored.

3.2.1.1 Top and Bottom Switches

The T and B switches are used to control the placement or relocation of the object program. When neither switch is specified, LINK will link the object programs at the top of available core, i.e., immediately below the Absolute and Bootstrap Loaders.

The T switch (top) can be specified with any of the input file specifications. It must be in the following format:

```
/T:n
```

Where n is an unsigned octal number which defines the address of the object program.

The B switch (bottom) is specified in the same manner as the T switch. It must be in the following format:

```
/B:n
```

where n is an unsigned octal number which defines the bottom address of the object program.

If more than one T or B switch is specified during the creation of a load module, the value of the last T or B switch specification is used. When the load module creation is either finished or aborted, the default top value reverts to its original value, i.e., the top of core of the installation.

3.2.1.2 Concatenate Switch

The CC switch is used to indicate that the file was formed (for example, by PIP or the FORTRAN compiler) as a concatenation of several object modules. This switch may be used only with an input file specification. Its format is:

```
/CC
```

This switch does not have a value.

3.2.1.3 ODT Switch

The OD switch is used to link ODT with your object modules. It identifies the associated input file as ODT for Transfer address purposes. /OD appearing by itself in an input file specification is equivalent to

```
SY:ODT.OBJ[1,1]/OD
```

3.2.1.4 Transfer Address Switch

The TR switch can appear with any input file specification. It can be used with no value, or with an octal number or global symbol as its value.

When the TR switch has no value, it indicates that LINK should take the transfer address (even or odd) of the first object module in the file that has the /TR appended to it as the transfer address of the load module. Its format is:

```
/TR
```

When an octal number is specified as its value, it indicates that the value is the transfer address of the load module. Its format is:

```
/TR:n
```

When it has a global symbol as its value, it indicates that the value of the global symbol is the transfer address of the load module. Its format is:

```
/TR:xxxxxx
```

When the specified value is a nonexistent symbol or address, the transfer address is set to 1, and an error message is issued.

3.2.1.5 End Switch

The E switch should appear with the last input file specification. It indicates the end of input. Its format is:

```
/E
```

The /E switch should not be used with /GO.

3.2.1.6 Library Switch

The L switch is optionally used to indicate that the file is a library. It can appear in an input file specification only if the specification specifies a library. The L switch does not require a value. Its format is:

```
/L
```

Note that this switch is not necessary for correct functioning of libraries in LINK. This switch is supplied only for compatibility with the old linker.

3.2.1.7 Go Switch

The /GO switch should appear with the last input file specification when used. It indicates two things:

1. The end of input (in lieu of /E), and
2. When linking is complete, the load module is to be loaded and executed.

The GO switch should not be used with /E.

3.2.1.8 Overlay Mapping Description Switch

The /MP switch is used to specify that the file is an ASCII overlay description file as described in Chapter 4. No value is allowed on the switch. When specified, there must not be any other input files specified in this command string or any other input switches other than /E.

3.2.1.9 Options Switch

The /O switch is used in lieu of the /E switch to specify that the link options are required. The link options are described in detail in Chapter 8.

3.2.1.10 Include/Exclude Switches

The /IN and /EX switches are used on library files to cause the inclusion or exclusion of specific library modules. For example, the file specification

```
FTNLIB/IN:$PSH01
```

guarantees that when the library file FTNLIB is searched, the routine named $PSH01 within the library is linked. Conversely, the /EX switch guarantees that the specified module(s) are not loaded from the library. A typical specification might be

```
LIBRY/IN:ABC:DEF/EX:QKQ
```

which, when encountered, guarantees that ABD and DEF will be loaded from the library file LIBRY and QKQ will not be loaded.

If, for instance, the modules ABCTMP, DEFTMP, FILTMP, DATTMP, TSTTMP, and XPROPR (residing in a library named SPEC.LIB) are to be linked with a file named MASTER.OBJ, and the result is to be placed in a file named MASTER.LDA, the following command string can be used:

```
#MASTER<SPEC.LIB/IN:ABCTMP:DEFTMP:FILTMP:DATTMP:
#TSTTMP:XPROPR,MASTER.OBJ/E
```

Note the use of the colon (:) at the end of the first line of the command string; this serves to continue the switch value list from line 1 to line 2.

The /IN and /EX switches have no effect if specified for non-library files.

3.2.1.11 Long/Short Map Switches

The /LG switch is specified on the map file to cause the long map form to be produced. In addition to the normal entry points, a long map also prints out any external globals referenced by a module.

The /SH switch is specified to cause the short map to be printed. The short map consists only of the heading, program size description, and section allocation synopsis.

The /LG and /SH switches are mutually exclusive.

3.2.1.12 Global Cross-Reference Switch

The /CR switch is specified on the map file to cause a global cross-reference table to be produced on the map device when the link

is complete. See Appendix H for an example of a global cross-reference table. The /CR switch can be used with the /LG or /SH switches if desired.

3.2.1.13 Contiguous Output Switch

The /CO switch is used for a load module output file to specify that the file is to be contiguous, with an output format similar to that produced by the CILUS program (core- image file). When overlaid programs are generated by LINK, use of the /CO switch is automatically forced, since overlaid programs require a contiguous file.

The /CO switch can also be used with a value specifying that the contiguous file generated is to be built for a device with a block size that does not correspond to the block size of the output device actually used. For example, if LINK is run with load module output placed on an RF11 disk, the contiguous file produced will be formatted into 64-word blocks. Thus, the file produced will run only on disks with 64-word block sizes. If it is desired to produce a file on a 64-word block device to run on a 256-word block device, it can be done by specifying /CO:256 on the load module file specification to correctly generate the file.

Thus the /CO:n switch (where n must be a multiple of 64) can be used to allow contiguous output files to be generated on devices with block sizes other than that of the actual output device.

NOTE

A contiguous file generated by LINK will run correctly only on those devices with a block size equal to that for which the file was generated; a file generated for 256-word blocks will not run on a 64-word block device, and vice versa.

3.2.1.14 Program Section Sequencing Switch

Normally, program sections (.CSECT's and .PSECT's) are placed in memory in alphabetical order. The /SQ switch is used when it is desired to place program sections in memory in order of declaration (i.e., in the order they are encountered by LINK). The /SQ switch is useful mainly for programs that depend upon .CSECT ordering as implemented by previous versions of the Linker program.

3.2.1.15 General Notes on Switches

If a switch appears by itself as a specification (e.g., , ,/CC), it takes the default device and a null file name. Thus, the linking process will be aborted if the default device is file structured. The /OD switch is the only exception (see section 3.2.1.3).

3.3 LIBRARY SEARCHES

3.3.1 User Libraries

Object modules from the specified user libraries built by LINK will be relocated selectively and linked. The object modules in the Libraries must be ordered; only forward references are allowed.

The libraries are specified to LINK like any other input file.

For example, the user could type the following command string to the Linker:

```
#TASK01.LDA,LP:<MAIN.OBJ,MEASUR.LIB/E
```

Program MAIN.OBJ would be read in from the disk as the first input file. Any undefined symbols generated by program MAIN.OBJ can be satisfied by the library MEASUR.LIB specified in the second input file. The load module, TASK01.LDA would be put on the disk, and a load map would go to the line printer.

As described in section 3.2.1.6, the /L switch can be used in a library file specification. This switch is provided only for compatibility with the old linker, and does not affect proper processing of the library.

3.3.2 Monitor Library

At the end of pass 1, the Monitor library is searched for Monitor routines (EMT's) which were declared as globals in the user program. Satisfying these globals mean that the Linker passes the EMT trap number of the found routines (in the COMD) to the Monitor so that at load time the requested routines are made resident with the user program. Making EMT's core resident in a resident section can be accomplished by defining the appropriate EMT as a global before assembly with the .GLOBL assembly directive. Example:

```
.GLOBL  FOP.,LUK.,CKX.
```

Refer to Appendix C of the DOS Monitor Programmer's Manual for a description of globals associated with various EMT requests. Making a potentially swappable EMT routine core resident uses core space but saves swapping time for the routine. This tradeoff usually becomes important when an often-used subroutine uses one or more monitor functions that would normally be non-resident. For instance, this problem might arise from simultaneous use of the Block I/O routine and conversion routines within the same program.

The user libraries are searched first and the Monitor library is searched if any globals remain undefined.

NOTE

Although some undefined globals may be satisfied at the monitor level, they continue to be flagged as undefined globals. A message will be printed on the user's terminal stating that there are undefined globals, and a similar message is given in the load map listing. However, any undefined globals satisfied at the Monitor level are flagged in the LINK map undefined summary, with "**" following the name. See Appendix H for an example.

3.4 SAMPLE LINKS

3.4.1 FORTRAN

User is logged in under user identification code (UIC) of 200,200.

He wishes to link a FORTRAN program (FORT1.OBJ) to the FORTRAN library (FTNLIB) which is on the system disk under UIC 1,1. He wants a load map printed on the line printer. Input comes from and output goes to the disk.

The command string is:

```
#FORT1,LP:<FORT1,FTNLIB/E
```

(The default input extension is OBJ. Since both files FORT1 and FTNLIB had the extension OBJ there was no need to put this information in the command strings.)

3.4.2 Assembly Language

User is logged in under UIC 200,200. He has a DECtape, but he has no line printer at his installation. He wants his outputs, load module (LOAD.LDA) and load map (LOAD.MAP) on DECtape and his inputs to come from disk. (He has seven input files all with extension OBJ.)

The command strings are:

```
#DT1:LOAD,LOAD<IN1,IN2,IN3
#IN4,IN5,IN6,IN7/E
```

(Note that LINK accepts multiple command lines.) On the DECtape the load module has the extension LDA and the load map has extension MAP.

3.4.3 Overlays

User is logged in under UIC 200,200. He has an ODL file (Overlay Description Language -- See Chapter 4) named BUILD.ODL that describes

his overlaid program. He wishes to place the load module on the disk and the listing on the line printer.

The command string is:

```
#ABC,LP:<BUILD.ODL/MP/E
```

3.5 PROGRAMMING NOTES AND CAUTIONS

3.5.1 Programming Notes

1. No switch (except /OD) can appear alone in an input specification (i.e., /E is illegal, filename /E is legal).

2. There are several ways to link ODT-11R. Three examples follow.

   a. SY:<TEST,/OD/E

      (The OD switch is set)

      allows the Monitor command BE to start the test program, and the Monitor command OD to start ODT (assuming in both cases that the Monitor command GET TEST precedes).

   b. OUTFIL<ODT,TEST/E

      (no OD switch)

      allows BE to begin ODT. The command ODT will have no effect.

   c. SY:<ODT/E

      allows ODT to be run by itself with the RUN command.

3. When using the /OD switch:

default device (SY:)	--	as usual, system residence disk
default filename	--	ODT
default extension	--	as usual, OBJ
secondary UIC	--	1,1

   If ODT.OBJ is on either the system or user disk area, the input file specification for ODT in the CSI reduces to:

   ```
   /OD
   ```

   for example:

   ```
   <TEST,/OD
   ```

4. The /OD and /TR switches are mutually exclusive.

3.5.2 Cautions

If the user means to type:

```
PP:,LP:<PR:/E
```

but accidentally types:

```
PP:,LP<PR:/E
```

the load map (an ASCII file) will be punched on the paper tape followed by the load module (a binary file). The Linker will not detect the error since the erroneous string is a legal one (i.e., output file LP.MAP to default device, PP:), thus the load module cannot be loaded (since there is an ASCII file in front of it).

There is a fair amount of blank tape between the load map and the load module, so either separate them or relink with the correct command string.

CHAPTER 4

OVERLAYS

An overlay capability is very important in computer systems such as DOS/BATCH where the size of programs is apt to be larger than the amount of the memory available. Overlay support is an integral part of the design of LINK. The only difference, as far as LINK is concerned, between a normal link and an overlaid link is the fact that the former contains only one segment.

Overlays are defined in terms of a simple tree structure via a special Overlay Description Language (ODL) that is interpreted by LINK. The trunk of the tree is termed the root segment and always remains in memory. The branches of the tree represent overlay segments which may overlay each other.

Figure 4-1 illustrates a typical overlay structure. "A" is the root segment and "B", "C", "D", "E", and "F" are overlay segments.

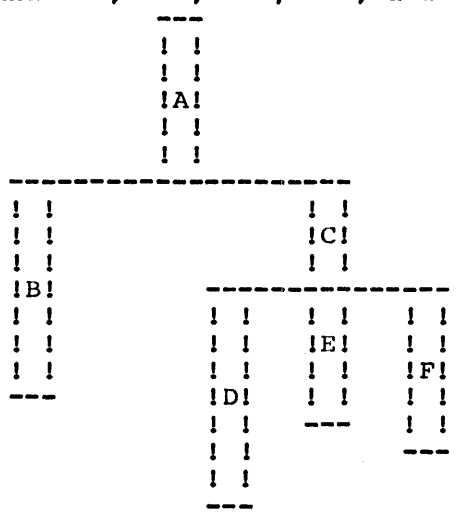


Figure 4-1 - Overlay Task Structure

A path is defined as a route that is traced from the root when following a series of branches to an outermost branch of the tree. In the above figure A-B, A-C-D, A-C-E, and A-C-F represent all possible paths.

Overlays may call other overlays if they occur on a common path. Thus in figure 4-1, the root segment may call overlays B, C, D, E, and F. On the calls to D, E, or F the overlay C is also normally loaded. This is termed "path loading" and occurs whenever a call is made from one segment to another segment that is more than one branch level up the tree (away from the trunk). Overlay C may call D, E, or F but B cannot call C, D, E, or F nor can C, D, E, or F call B.

Two methods are provided for loading overlay segments into memory. The first method is by an explicit call to the library routine LOAD to load a named segment. This is termed Manual-Load. Before a manual-load request is honored, LINK marks out-of-core all segments up the tree which emanate from the overlay control point where the request is to be loaded. The manual-load is then initiated and control is returned to the caller. The issuer of a manual-load request may, at his option, have the load performed either synchronously or asynchronously with program execution. Upon successful loading the calling routine may then call entry points in the named segment via normal subroutine or transfer of control instructions. When using manual-load, path loading is not performed; i.e., only the segment specified in the LOAD call is loaded into memory.

The second method of loading overlay segments is termed Autoload (also known as load-on-call or LOCAL). Autoload occurs whenever a transfer of control instruction is executed that references an autoload entry point in another segment that is further up the tree on a common path. Autoload causes the automatic loading of an overlay segment and subsequent transfer of control to the called entry point in a manner that is completely transparent to the caller. Unlike manual-load, path loading is performed on autoload calls.

Both methods of loading overlay segments have different merits that warrant their support. Autoload has the advantage of being completely transparent* while manual-load requires slightly less memory. Autoload allows a program to be separated into segments without reprogramming while manual-load requires explicit calls to load overlay segments.

The actual loading of all overlay segments is accomplished via the .TRAN request in the DOS Monitor. No extensive file open/read/close sequence need take place since the Monitor knows the disk address of the core image. The loading of an overlay segment thus requires a single disk access and can be very fast.

4.1 TERMINOLOGY

AUTOLOAD - The process of automatically loading an overlay segment and subsequently transfering control to a called entry point in a manner that is completely transparent to the caller. Also known as load-on-call or LOCAL.

AUTOLOAD ENTRY POINT - An entry point that has been defined such that a transfer of control to the entry point will cause the segment in which it is defined to be automatically loaded if it is not already in memory.

ENTRY POINT - A symbol defined in a source representation of a program and subsequently accessible to independently translated modules

---

*However, condition codes are not passed on an autoload call.

via the binding mechanism provided in the LINK. All such symbols must be established as globals.

LOAD-ON-CALL - See Autoload.

LOCAL - See Autoload.

MANUAL-LOAD - An explicit call to the library routine LOAD to load a named segment into memory.

PATH - A route that is traced when following a series of branches in an overlay structure.

PATH UP - The routes traced when following all paths from a branch segment away from the trunk to the outermost branches that lie on a common path.

PATH DOWN - The route traced when following a path from a branch segment toward the trunk.

ROOT SEGMENT - A group of modules and/or program sections that occupy memory simultaneously and are never overwritten. Every program has one and one only root segment (i.e., even a single segment program is considered to have a root segment).

SEGMENT - A group of modules and/or program sections that occupy memory simultaneously and may be loaded via a single call to the Executive.

4.2 Overlay Description Language

An Overlay Description Language (ODL) is provided to describe overlay structures. Rather than being a part of the command language itself, which would make it very complex, these descriptions are always read from a separate file.

An overlay description file is specified by including the /MP switch on the first input file specification (see section 3.2.1.8). This file contains all the object module input file specifications in addition to the overlay description. The /MP switch must appear on the first input file specification (ignored elsewhere) and no other input specifications to ODL may be given subsequently. Option input is accepted in the normal manner.

Example:

```
IMAGE,MAP,SYMBOL<OVERL/MP/E
```

specifies that the file OVERL contains a description of the overlaid program to be built.

The Overlay Description Language is composed of a number of directives that are used to describe the overlay structure.

In its role as the builder of tree-structured programs, LINK interprets and carries out the directives provided by the Overlay Description Language (ODL). Its inputs are directives written in ODL,

and object files produced by language translators. Its output is an overlaid program suitable for execution under DOS.

Object files result from a source to object transformation by a language translator. These object files consist of storage allocated under the three section types: .ASECT, .CSECT, and .PSECT. Individual files may contain unresolved global references which LINK attempts to resolve during the linking process.

ODL consists of directives which specify a function, and operands, which are either filenames, name strings reducible to filenames, or names appearing in PSECT directives. LINK uses the name-strings to locate or create object modules which are built into the overlays.

ODL provides for:

* Identification of the Root Segment
* Building overlays
* Naming overlays
* Strict placement within the overlay structure of globally referenced memory
* Establishing overlay control points
* Declaring autoload entry points
* Five directives:

  .ROOT

  .NAME

  .FCTR

  .PSECT, AND

  .END

* and four operators:

  "-" concatenation

  "," overlay;

  "(", ")" overlay control point, and

  "*" autoload

The directives and operators acting on name strings provide the semantics specified above.

The directives have the following general format

```
LABEL:  .DIREC  OPRNDS
```

Where:

LABEL is an alphanumeric label.

DIREC is the directive name.

OPRND is/are optional operands.

4.2.1 The .ROOT Directive

The .ROOT directive completely specifies the program tree structure and has the following format:

```
LABEL: .ROOT OPRNDS
```

The optional LABEL field, if present, is ignored.

The permissible operands of a .ROOT directive are

1. Filenames of the form:

   ```
   DEV:FILE.EXT[UIC]/SW
   ```

2. A name which appears in a .NAME directive

3. The label on a .FCTR directive

4. The name in a .PSECT directive

These operands are operated on by four operators.

"-" The concatenation operator (minus sign). A binary operator that specifies that its operands are to occupy memory simultaneously, which means, of course, they are part of a path.

"," The overlay operator (comma). A binary operator whose operands occupy memory starting at the same base address (a node in an overlay tree). Of course, segments that occupy the same memory are not on a path, and, indeed, overlay one another.

"(", ")" The overlay control point operator (parentheses). With an exception to be noted, a control point at which an overlay is to begin (the points x and y in Figure 4-2) are implied by enclosing operands in parentheses.

A fourth operator, "*", will be discussed after an example reinforces the use of .ROOT and the "-", "," and "(", ")" operators.

Actually, we now know enough of the ODL to describe to LINK a substantial percentage of the overlay structures that occur in practice. Consider the structure of Figure 4-2.

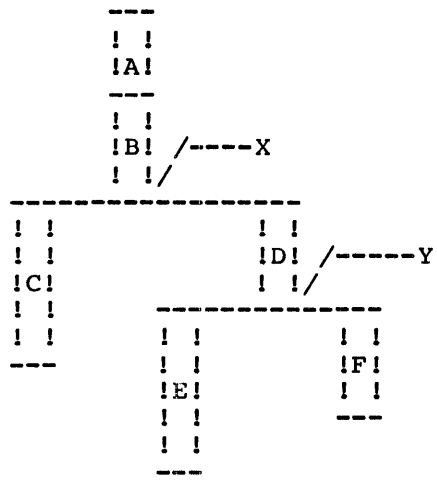


FIGURE 4-2 - SAMPLE STRUCTURE

Figure 4-2 consists of six object files to be linked together by LINK. We now want to specify the input required by LINK to construct the program of Figure 4-2, To simplify the explanation establish the structures below the branch point, X, as the leftbranch and rightbranch. Now we can describe Figure 4-2 as

```
.ROOT A-B-(leftbranch, rightbranch)
```

This statement instructs LINK to concatenate A and B, form a branch point, and cause the leftbranch and rightbranch to occupy memory starting at the same base address. (If the leftbranch is in memory, the rightbranch cannot be, and vice-versa.)

We can represent the leftbranch as,

```
C
```

and the right branch as,

```
D-(E,F)
```

specifying the concatenation of D with E and F; E and F overlay each other.

Thus complete ODL specification of Figure 4-2 is:

```
.ROOT A-B-(C,D-(E,F))
```

Using this brief example as an aid to familiarizing the reader with the overlay language we can now complete its specification. Specifically, the

.NAME,

.FCTR,

.PSECT, and

.END

Directives, and the

"*" operator

4.2.2 The .NAME Directive

Declare an alphanumeric name that may subsequently be used in a .ROOT or .FCTR directive to define the name of a segment.

Normally a segment is named according to the first file or P-section that is included in the segment, it is recognized that this may not be adequate in some cases and therefore this directive may be used to explicitly declare a segment name.

Directive syntax:

.NAME SNAME

Where:

.NAME is the directive NAME.

SNAME is an alphanumeric NAME of 1 to 6 characters.
(A-Z,1-9,$)

NOTE

If a label is present it is ignored. SNAME must be unique with respect to file names, P-section names, and other segment names that are declared in the description file.

If in Figure 4-2, we wanted to name the root segment in Figure 4-2, JIM, the directives:

```
.NAME JIM
.ROOT JIM-A-B-(C,D-(E,F))
```

would create a root segment with the name JIM.

4.2.3 The .FCTR Directive

The factor (.FCTR) directive has the same format as .ROOT:

LABEL: .FCTR OPRNDS

Its operands are the same as for ROOT. Unlike .ROOT the LABEL field is required. The .FCTR directive formalizes the pedagogical convenience we used earlier in presenting the development of the .ROOT directive used to describe the overlay structure of Figure 4-2. Recall that a factoring occurred by using the terms leftbranch and rightbranch. Using .FCTR this becomes a capability of the ODL itself.

Thus--

```
LEFTBR:  .FCTR  C
RHTBR:   .FCTR  D-(E,F)
         .ROOT A-B-(LEFTBR,RHTBR)
```

will describe to LINK the same overlay structure as--

```
         .ROOT A-B-(C,D-(E,F))
```

When expanding the .ROOT statement

```
         .ROOT A-B-(LEFTBR, RHTBR)
```

LINK will substitute the expressions equated in the .FCTR directives for LEFTBR and RHTBR.

.FCTR is a notational convenience for simplifying the process of representing complex overlay structures to LINK.

4.2.4 The .PSECT Directive

Often segments within an overlay structure have a requirement to access common storage. LINK allocates storage for referenced sections within the section in which it is defined (local reference) or in the branch on its path closest to the root (global reference). For example, if in Figure 4-3

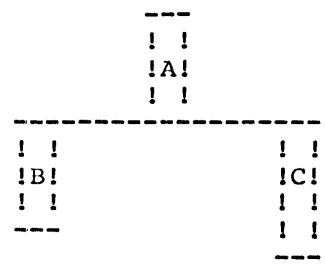


Figure 4-3 - Simple Global Reference

A, B, and C each reference a global storage area, D, then LINK will allocate storage in A. If, however, only B, and C reference a global storage area, D, then LINK will allocate storage in both B and C, a default decision, which may or may not coincide with the programmers wishes. It is the function of the .PSECT directive to permit the programmer to explicitly place the global area, overriding LINK's default.

.PSECT declares an alphanumeric P-section (program section) name that may be subsequently used in a .ROOT or .FCTR directive explicity placing a P-section in an overlay segment. A declared P-section may be placed anywhere in the overlay structure that is not ambiguous (i.e. not on a common path that already contains the specified P-section in another segment).* All actual references to the P-section from object modules must have exactly the same attributes as declared in the directive.

A good example of the use of this directive is the placement of a FORTRAN common area close to the root segment so that a number of branch segments that are not on a common path may share and communicate via the area. In this case, if the explicit placement were left out, the common area would be allocated in each branch segment and not shared.

Directive syntax:

.PSECT SNAME [,AT1,AT2,...,ATn]

Where:

.PSECT is the directive name.

SNAME is an alphanumeric control section name.(A-Z,1-9,$)

AT1 through ATn are optional control section attributes.

P-section attributes are specified exactly as they are for the .PSECT directive under MACRO-11. These attributes include:

RO or RW specify the access mode of the P-SECTION. RO means read only and RW read/write.

I or D specify the type of P-section. I means instruction and D data.**

GBL or LCL specify the scope over which the P-section is considered by LINK. GBL means global and the P-section will be considered across segment (overlay) boundaries. LCL means local and the P-section is considered only within the segment in which it is defined. If a single segment program is produced GBL and LCL have no effect on the core allocation in LINK (i.e. only one segment to consider P-sections over).

ABS or REL specify relocation of the P-section. ABS means absolute and no relocation is necessary. REL means

---

*The ambiquity is not detected in the ODL syntax check, but at the point where a reference to an ambiguous section is encountered during file processing.

**Not to be confused with the I and D space hardware on the PDP 11/45.

relocatable and a relocation bias must be added to all references to the P-section.

CON or OVR specify the allocation of the P-section. CON means that all allocation references to the P-section are concatenated to form the total allocation of the P-section. OVR means that all allocation references to the P-section from different modules overlay each other. The total allocation of the P-section is the largest request made by the individual modules that reference it.

HGH or LOW specify the speed of the memory that the P-section is to be loaded into. HGH means high speed and LOW means core.

NOTE

The HGH/LOW attribute is currently ignored by LINK.

Default attributes are applied to all .PSECT directives. These attributes may be subsequently overriden by an explicit attribute specification. The default attributes are:

```
.PSECT name,RW,I,LCL,REL,CON,LOW
```

4.2.5 The .END Directive

Declare the end of the overlay description file.

This directive is mandatory and must appear at the logical end of each overlay description file.

Directive syntax:

```
.END
```

Where:

.END is the directive name.

NOTE

If a label or operands are present they are ignored.

4.3 AUTOLOAD OPERATOR ASTERISK (*)

The asterisk (*) is a unary operator that specifies its operand as autoloadable. Any transfer of control to an entry point in a P-section with an "I" attribute will cause the segment in which the operand resides to be loaded unless the segment already exists in memory. If an "*" occurs on an open parenthesis "(", every operand within the parenthesis and its matching close parenthesis, ")", will

have the autoload attribute. The "*" operator applies only to P-sections with the "I" attribute.

As applied to specific operand types the "*" operator acts as follows:

1. For section names the section is made autoload.

2. For the name in a .NAME directive all the components in the segment to which the name applies are made autoload.

3. For name labeling a .FCTR statement, "*" applies to the first irreducible component of the factor. If the entire factor list is enclosed in parentheses, every file in the factor is made autoload.

4. For a Filename all components of the file are made autoload.

4.4 ODL USAGE SPECIFICATIONS

1. The directives may appear in the input file in any order, with the exception of .END which must always terminate the file.

2. Every ODL Task description must have one, and only one .ROOT directive.

3. A label must appear in a .FCTR directive .

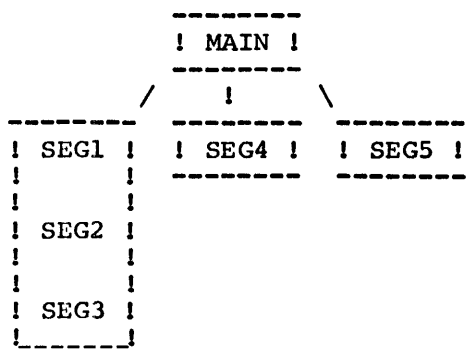
4. Labels in a .ROOT directive are ignored.

5. Redundant pairs of parentheses are permitted for notational clarity, but will not cause additional overlay control points.

6. A .FCTR directive label and SNAMEs must not contain periods.

4.5 EXAMPLE OF OVERLAID PROGRAM BUILD USING LINK

Given the following tree structure description of the desired overlay:



A file can be created with a name such as DESCR.ODL which will contain the ODL task description:

```
SEG123:    .FCTR    SEG1.OBJ-SEG2.OBJ/CC-SEG3.OBJ[27,63]/CC
ROOT:      .ROOT    MAIN.OBJ-(SEG123,SEG4.OBJ,SEG5.OBJ)
END:       .END
```

Then, when the following command string is type to LINK, the overlay load module will be named MAIN.LDA and the load map will be printed on the line printer.

```
$RUN LINK
LINK    Vxx
#MAIN,LP:<DESCR.ODL/MP/E
```

4.6 MANUAL LOAD OVERLAYS FROM FORTRAN

The following is an example of a synchronous overlay load using the FORTRAN callable routine LOAD(see section 6.1). The program requests that the overlay segment GAUSS (which contains the subroutine RANDOM) be loaded into core. Control will not be returned to the program until the load operation is complete. IERR is checked to assure that the segment was successfully loaded, then the program transfers control to a routine contained in the overlay segment that was just loaded.

```
      .
      .
      .
IF(ITEST.EQ.0) GO TO 990
CALL LOAD('GAUSS',1,IERR)
IF(IERR.NE.0) GO TO 500
CALL RANDOM(A,B,ITEST,2)
      .
      .
      .
```

The following is an example of an asynchronous overlay load. The program requests that the segment PRINT (which contains the subroutine ALPHIO) be loaded into core. The ENCODE operation following the CALL LOAD will be executed while the overlay is being loaded. Then the program performs a CALL WAIT to assure that the load operation is complete before transfering control to a routine in the overlaid segment.

```
      .
      .
      .
CALL LOAD('PRINT',0,IERR)
ENCODE(490,100,ALPHA) (VECT(I),I=1,70)
100   FORMAT(70I7)
CALL WAIT
IF(IERR.NE.0) GO TO 500
CALL ALPHIO(ALPHA,70)
      .
      .
      .
```

4.7 FORTRAN FORMAT CONVERSIONS AND I/O ROUTINES

Any format conversion not needed in a FORTRAN resident section but required by overlay sections must be forcibly loaded into the resident section.

This can be accomplished in any of three ways:

1. Declare the appropriate globals in an assembly language routine.

2. Insert dummy FORMAT statements in the resident main program for all format conversions that are required in the overlays but not in the resident section.

3. Specify in the root segment link the appropriate module names needed (through the /IN switch). Table 4-1 contains a detailed list of these names.

For example, assume I and L format conversions are needed for READ and I and E format conversions are needed for WRITE. An assembly language routine such as the following could be written:

```
.TITLE DUMMY
.GLOBL $LCI,$ICI,$ICO,$DCO
.END
```

where $LCI performs the L conversions for READ, $ICI performs the I conversions for READ, $ICO performs the I conversions for WRITE, and $DCO performs the E conversions for WRITE.

An alternate mode involves dummy FORMAT statements supplied in the resident main program to force linking of these routines. (If this is done, a message may be printed at compile time indicating that there is non-executable code in the program.

For example:

```
          LOGICAL L
          GO TO 1000
          READ (6,100) I,L
100       FORMAT (I1,L1)
          WRITE (6,101), I,E
101       FORMAT (I1,E6.0)
             .
             .
             .
1000      CONTINUE
```

Another alternative is use of the /IN switch as follows:

```
XXXLIB/IN:$LCI:$ICI:$DCI
```

where XXXLIB is the library specified in the ODL command file, and $LCI,$ICI, and $DCI are module names associated with the required globals (see Table 4-1).

Including global references in an assembly language routine (or specifying module names with the /IN switch) causes only the four

format conversion packages to be linked to the resident program section. Inserting dummy FORMAT and Input/Output statements causes the resident to carry the overhead of the four format conversion packages plus the FORTRAN READ/WRITE processor, FORMAT scanner, and associated routines.

Two other possibilities are either to perform all I/O in the resident program, or to perform all I/O in the overlay section. If there is no I/O in the resident section, each overlay includes only those modules needed to satisfy its own I/O requirements.

If those format conversion routines which are needed in the overlays and not required in the resident section are not forcibly loaded into the resident section, the FORTRAN system causes the linking of dummy routines. Global requests in the overlay files are then linked to the resident dummy routines and at execution time result in the fatal error message:

FORT008000 LINKAGE ERROR (MISSING FORMAT CONVERSION ROUTINE)

If it is essential to minimize the amount of memory used by the resident section, the technique of forced loading of modules by means of an assembly language routine or the /IN switch is recommended. The assembly language routine does not force all routines in the I/O package into the resident section, but rather causes the loading of some modules which would otherwise be blocked. The resulting resident section may be smaller than that produced by the inclusion of the dummy FORTRAN statements shown above.

Table 4-1 is useful in building overlay systems. If any module not needed by the resident is required in an overlay, then the corresponding global must be declared in the resident section.

Table 4-1

Format Conversion Packages and I/O Routines

Globals in Package	Module Name	Function Performed	Length of Package in Decimal Words*
$DCO $ECO $FCO $GCO	$DCO	Output Conversions D, E, F, G	469
$ICO $OCO	$ICO	Output Conversions, I, O	93
$LCO	$LCO	Output Conversion L	31
$DCI $RCI	$DCI	Input conversions D, E, F, G	384
$ICI $OCI	$ICI	Input conversion I, O	85
$LCI	$LCI	Input conversion L	31

---

*Includes certain associated modules.

CHAPTER 5

PROGRAM MEMORY ORGANIZATION

5.1 ALLOCATION FOR A NON-OVERLAID PROGRAM

A non-overlaid program is allocated to memory as shown in Figure 5-1.

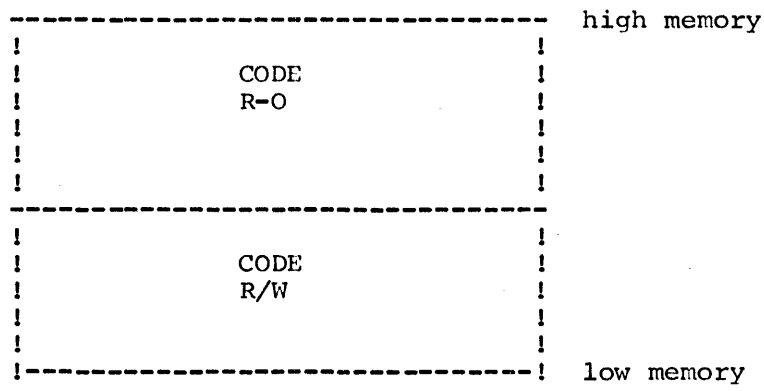


Figure 5-2 - Non-Overlaid Program

5.1.1 Read/Write Code (and Data) (R/W)

The program's read/write code and data are placed in the lowest memory allocated.

5.1.2 Read-Only Code (and Data) (R-O)

If the program has a read-only portion, LINK places it immediately above the area occupied by the read/write code.

5.2 ALLOCATION FOR AN OVERLAID PROGRAM

5.2.1 Root Segment Allocation

The allocation of real memory to the root segment is shown in Figure 5-2.

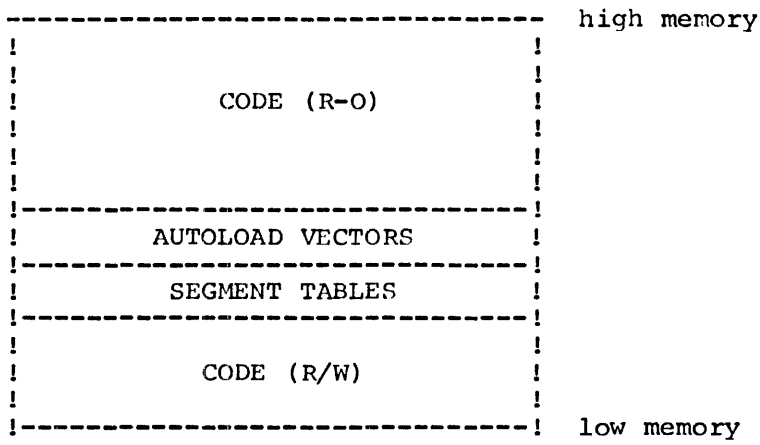


Figure 5-2 - Root Segment Overlaid Program

Code (R/W) and Code (R-O) are the same as for non-overlaid programs, and we will describe only the Segment Tables and Autoload vectors.

5.2.2 The Segment Tables

Each segment in an overlay structure has a 10-word Segment Descriptor formatted as shown in Figure 5-3.

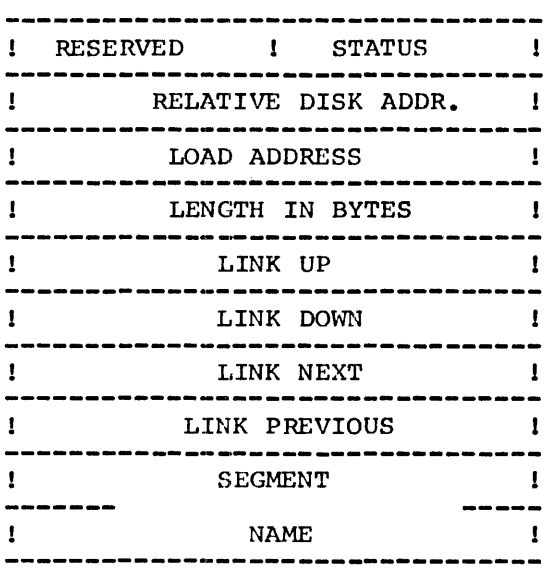


Figure 5-3 - Segment Descriptor

5.2.2.1 Status:

0 specifies the segment is in-core, 1 specifies not-in-core. The bit is used during path loading to eliminate unnecessary disk accesses.

5.2.2.2 Relative Disk Address of the Overlay Segment:

A program image occupies a contiguous disk area. Each overlay segment begins at a block boundary and this index is a relative block number from the start of the program disk image. This word enables loading of segments with a single disk access.

5.2.2.3 Load Address of The Segment:

The program relative address where this segment is to be loaded.

5.2.2.4 Length of the Segment:

The number of bytes in the segment; this number is used to constuct the disk read.

5.2.2.5 Link Fields

The function of the link fields is to permit, given the address of any descriptor, finding a path to the root and to develop from any segment the path to any other segment (if it exists) up the tree.

Link up:

A pointer to a Segment Descriptor away from the root. Such a segment emanates from an overlay control point which starts at the base of this descriptor. Since many segments may emanate from an overlay control point, this pointer does not point to a unique successor. In figure 5-4 the segment descriptor for the root segment may point to B C, or D depending on how the LINK algorithm makes its link-up pointer selection; once made, however, it is never altered.

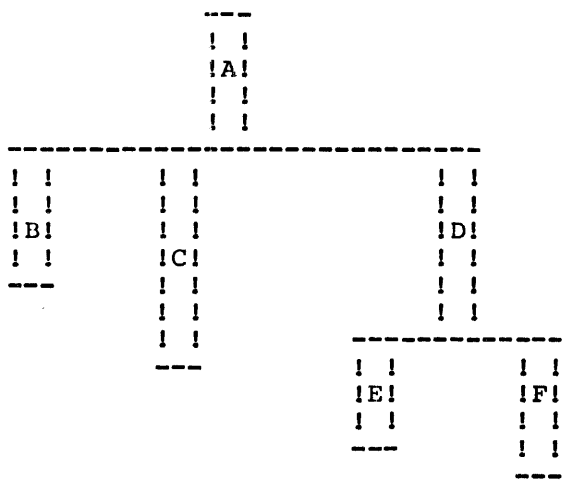


Figure 5-4 - Link Paths

Link Down:

A pointer to a segment nearer the root which is the immediate predecessor of the segment described by this descriptor. This pointer is always unique since paths moving toward the root always have unique predecessors.

Link Next and Link Previous

All segments emanating from an overlay control point are circularly linked forward and backward. This facilitates the search needed to mark in core segments out of core when they are overlayed. In Figure 5-4, B, C, and D are circularly linked as are E, and F. A has null link-next and link-previous pointers.

5.2.2.6 Segment Name

The six character (max) RAD50 representation of the segment name.

5.2.3 Autoload Vectors

Autoload vectors appear in every segment which references autoload entry points in segments farther away from the root than the referencing segment. Segments which reference autoload entry points toward the root are resolved directly. Autoload entry points occur in the segment making an autoload transfer. A discussion of the format of the autoload vector and the autoload machinery is discussed in Chapter 6, Run Time Support.

5.3 OVERLAY MEMORY ALLOCATION

Every overlay in a program has an allocation as shown in Figure 5-5. The construction of segments is identical to the root segment structure discussed in Section 5.2.

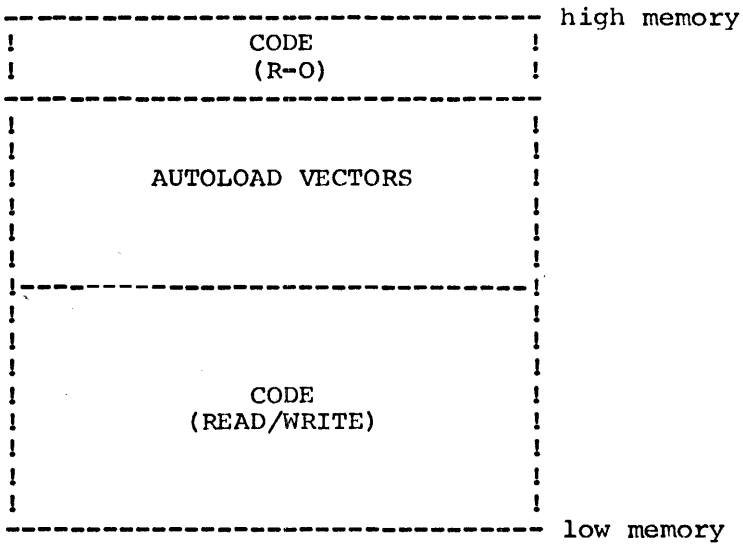


Figure 5-5 - Overlay Segment

5.4 OVERALL MEMORY ORGANIZATION

Figure 5-6 shows the overall memory allocation of an overlaid program running under DOS/BATCH.

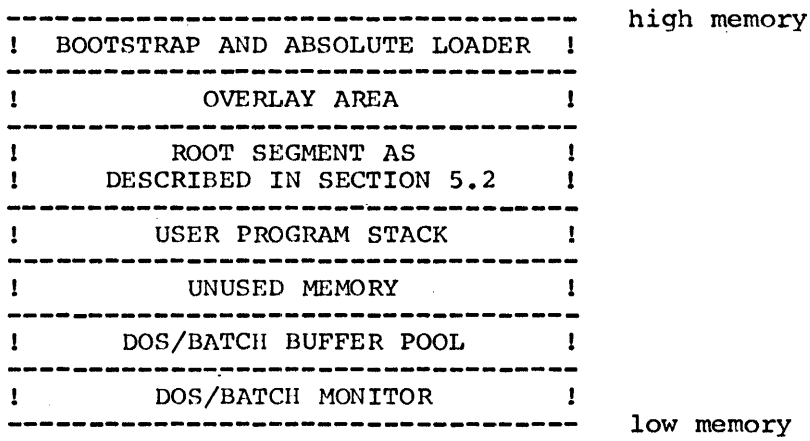


Figure 5-6
Overall Memory Allocation

Note that the size of the overlay area is just large enough to accommodate the largest possible combination of overlays that could exist simultaneously in memory.

CHAPTER 6

RUN TIME OVERLAY SUPPORT

Two methods of calling overlays exist in DOS/BATCH:

1. Manual Load, and
2. Autoload.

6.1 MANUAL LOAD

Manual load is initiated by a call to the LOAD routine. LOAD can operate either synchronously or asynchronously with program execution, and does not path load. LOAD marks as out-of-core any segment currently in-core and not along the path leading to the requested segment.

Calling Sequence: In FORTRAN, LOAD is referenced as shown below.

```
CALL LOAD ('strnam',sync,error)
```

where strnam is a 1- to 6-character ASCII name. If strnam is less than six characters long, it must be terminated by a blank or null character.

sync is a value set to 1 for a synchronous load, or to 0 for an asynchronous load. In a synchronous load, the requested segment is already in memory when control is returned to the program after the call. In an asynchronous load, the input transfer is initiated by the call; segment loading may proceed concurrently with the execution of the program issuing the call.

NOTE

When using asynchronous calls, the user must insure that the desired overlay is in memory before referencing values within the overlay or jumping to an entry point within the overlay. By using "CALL WAIT" the user can insure that the overlay transfer is complete before control is returned to the program.

error is a value returned to the calling program. If error is 0, no errors have occurred in the CALL LOAD. If error is non-zero, the requested segment has not been loaded (for example, if a nonexistent segment has been specified or a permanent read error has occurred).

The equivalent assembler calls for LOAD and wait are shown below.

```
JSR       R5,LOAD        ;CALL LOAD
BR        .+8.           ;SKIP AROUND PARAMETERS
.WORD     strnam         ;ADDRESS OF STRING NAME
.WORD     sync           ;ADDRESS OF SYNCHRONOUS
                         ;FLAG WORD
.WORD     error          ;ADDRESS OF ERROR FLAG WORD

JSR       R5,WAIT        ;WAIT FOR COMPLETION
BR        .+2
```

6.2 AUTOLOAD

During program creation from ODL, LINK records all auto-load entry points referenced by a segment. References toward the root are resolved absolutely. Those away from the root are replaced by a jump into the autoload vector table built by LINK for the segment. The autoload vector table consists of one entry per unique autoload entry point referenced by the segment. Each entry in the autoload vector table consists of one instruction and a three-word descriptor as shown in Figure 6-1.

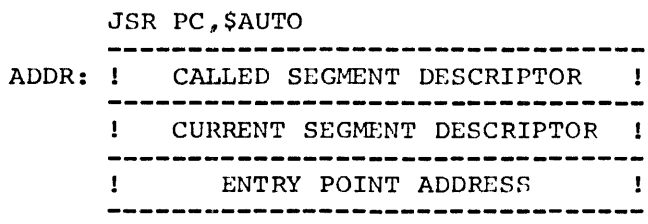


Figure 6-1 Autoload Vector Entry

The jump into the autoload vector table is made to transfer to the entry in the table describing the required autoload entry point.

$AUTO is a library routine which carries out the autoload process. $AUTO checks if the requested segment is in core (low order byte of the segment descriptor status word), and if it is in-core transfers to it. If the requested segment is not in core, $AUTO initiates a pre-emption scan, followed by a path load.

The function of the pre-emption scan is to mark out-of-core all segments currently in core that will be overlayed by the autoload call.

The path loading results in loading every segment along the path from the caller to the callee.

Both the pre-emption scan and path loading use a tree-walk techinique similar to that described in section 7.3.

For examples of autoload and manual load usage see sections 4.5 and 4.6

CHAPTER 7

MEMORY ALLOCATION

The allocation of memory occurs at the start of pass 2 of LINK. In the previous pass, LINK has established the memory requirements and attributes of every P-section in the program. It has also built the segment tables which completely define the structure described by the ODL, and has stored commands it must act upon during memory allocation. Using P-section memory requirements, P-section attributes, segment tables, autoload vector lists, and the command list, LINK can proceed to allocate memory.

7.1 MEMORY ALLOCATION PROCEDURES

7.1.1 Allocating Root Segment Memory

LINK begins by allocating the Read/Write portion of the root segment. It proceeds algorithmically as follows:

1. Allocate in alphabetical order all read/write P-sections of the root segment, accumulating the total memory required as the allocation proceeds. This implies that if in ODL a user described the root as

   A-C-B

   The actual allocation and placement would be as though he had specified

   A-B-C

   The placement of every P-section is clearly shown on the map listing produced by LINK. After a P-section is processed, a check is made of the extension list (created from EXTSCT commands) described in Chapter 8 and if a command is found for this P-section it is extended:

   1. If the CON attribute for the P-section is set, or

   2. If the OVR attribute is set and insufficient storage is currently allocated to the P-section to cover its extend request.

      Also, the processing of a P-section will result in proper boundary alignment. Presently the assembler only supports word alignment, but when it supports alignment requests to any specified boundary, LINK will place the P-section on the requested boundary, incrementing the virtual location counter appropriately.

   3. LINK now checks if it is building an overlaid program, and, if it is, it allocates the storage for the Segment Tables.

4. Finally, any storage needed to hold autoload entry points referred to up-the-tree by the root segment are allocated.

7.1.2 Allocating Overlay Segment Memory

The procedure follows closely the allocation of Read/write storage in the root, with the following exceptions:

1. If an overlay segment contains read-only P-sections, these sections are processed after the read/write sections of the same segment. Within each of the attribute types (read/write and read-only) allocation is alphabetical. If LINK encounters a read-only section in an overlay segment, it will issue a diagnostic and continue to process the read-only section as if it were read/write.

2. No Segment Tables are produced for overlay segments

3. Allocation for an overlay segment starts at address+1 of the bottom of the segment pointed to by the link-down of the segment being processed.

All memory allocation for a program described by ODL is now complete.

7.2 MEMORY ALLOCATION MAP

The listing of the memory map produced by LINK consists of a heading followed by detailed descriptions of each segment in the program. The data on each segment includes:

1. The statistics and attributes for each section.

2. The memory limits of every P-section in every segment.

3. File descriptions of the files used to build the program, and

4. Undefined references by file.

The segment description begins with the root segment, which begins on the same page as the heading. The overlay segments each start on a new page and their order is determined by a tree walk algorithm used in a number of contexts within LINK. See Appendix H for a detailed map description and example.

7.3 LINK TREE WALK ALGORITHM

In the map listing, LINK displays segment descriptions in a path order which results from a tree walk; the result of the walk is the segment list which appears following the root segment name on the heading page. The tree walk algorithm proceeds as follows:

1. After displaying the root segment's description, take the link-up.

1a. If a link-up exists,

THEN

1b. Display its description, try the next link-up, and return to 1a.

ELSE

1c. Try a link-next.

If an un-processed link-next is found

THEN

Go-To 1b.

ELSE

Try a link-down. If the link down is the root

THEN

Terminate the Walk.

ELSE

Go-To 1c.

Using this algorithm, Figure 13, and the ODL description:

```
LEFTBR   .FCTR B-(C,D,E)
RHTBR    .FCTR F-(G,H)
         .ROOT A-(LEFTBR,RHTBR)
```

Then LINK will walk the tree (thus producing segment descriptions) in the following order

```
A (root)
B (link-up)
C (link-up)
D (link-next)
E (link-next)
B (link-down):not re-displayed
F (link-next)
G (link-up)
H (link-next)
F (link-down):not re-displayed
A (link-down):not re-displayed
```

Note that the link-down is taken as the first filename in the ODL description following a new overlay control point.

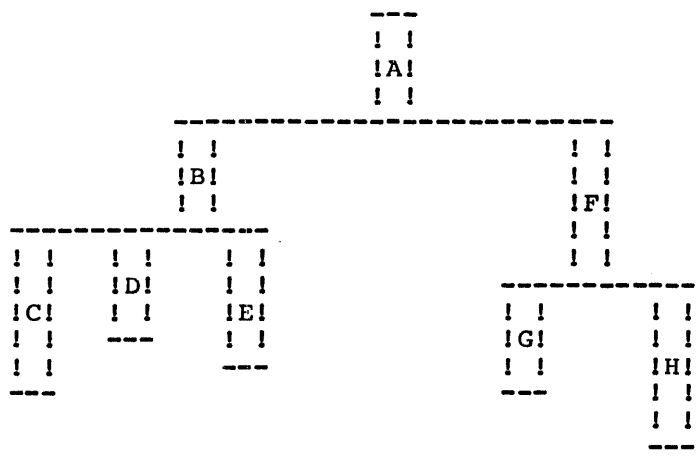


Figure 13 Tree Walk

CHAPTER 8

LINKING OPTIONS

8.0 OPTIONAL INPUT

Options input is accepted by LINK if the first command string was terminated by </O>. This input specifies options that are to be selected for the program being built.

Input is solicited with a line of

ENTER OPTIONS:

followed by a line containing a leading hash mark.

Each option is specified by a keyword followed by one or more parameters. After each line of option input is processed, the next line is solicited with another hash mark.

The options input is terminated with the /E specification in the same manner as normal LINK commands.

Optional input lines have the following general format:

KW = P(1,1):P(1,2):...:P(1,N):P(2,1):P(2,2)!KW=P(1,1)...;COMMENT

Where

KW = an alphanumeric keyword identifier of 1 to 6 characters

<=> = a delimiter that delineates the keyword identifier from its parameters

P(1,1) P(1,2) P(2,2) = parameter values that are specified for the option. The construction P(N,M) is used for illustration purposes only and signifies the M(th) parameter of the N(th) set of parameters. The general format allows multiple sets of parameters for a single keyword. Actual parameters are specified as alphanumeric characters and/or octal/decimal numbers.

<:> = a delimiter that separates parameter values

<,> = a delimiter that separates multiple sets of parameters.

<!> = a delimiter that separates multiple keyword identifiers on a single line.

<;> = a delimiter indicating that a comment follows.

NOTE

At the end of a line an implied <;> is always performed. Thus, optional specifications must always fit on a single line.

Blank characters and horizontal tabs are ignored and may appear anywhere.

A brief description of each keyword option is given below followed by the keyword syntax. Parameter values are defined using the following abbreviations:

DEVNAM = a two character alphabetic device name followed by a one or two digit unit number

NAME = an alphanumeric name of 1 to 6 characters, using the RAD50 character set. (A-Z, 1-9, $ and .)

NOTE

Octal and decimal numbers may contain a sign (i.e., + or -).

Certain options require that global symbols or P-sections be defined in the object modules that are loaded into the program image. If the appropriate definitions are not found, the corresponding option input is treated as a no operation (i.e., it is not performed, and the user is not notified).

8.1 ABSOLUTE PATCH (ABSPAT)

This allows the user to declare a series of absolute patch values in a segment.

An absolute address is taken as the base address of where the patches are to be applied. All patch values must lie within the segment or a load address error is generated.

Keyword syntax:

ABSPAT = SGNAM:PADDR:VALUE:VALUE:...:VALUE

where:

SGNAM = the name of the segment in which the patches are to be applied

PADDR = the absolute patch address

VALUE = patch values.

NOTE

Three parameters are required by this command. A maximum of eight values (ten parameters total) can be specified.

Default:

None.

Example:

Declare a series of patches in segment PAY starting at the absolute address 100:

```
ABSPAT = PAY:100:-1:5:6
```

NOTE

Patch values are stored in consecutive locations as a byte string. Each patch value requires two bytes.

8.2 EXTEND CONTROL SECTION (EXTSCT)

Extend the length of a P-section.

If the P-section has the attribute CON then the section is extended by the specified length. If the attribute is OVR, the section is assured to be at least as large as the specified length. The extension occurs when the specified name is encountered in an input object file. If no such name is encountered, no extension occurs.

Keyword syntax:

```
EXTSCT = CNAME:LENGTH
```

where:

CNAME = control section name

LENGTH = length to extend the P-section in bytes (octal).

Default:

None.

Example:

Declare the P-sections ONE and TWO to both be eligible for extension by a length of 50 and 100 bytes, respectively.

```
EXTSCT=ONE:50,TWO:100
```

8.3 GLOBAL SYMBOL DEFINITION (GBLDEF)

Declare the definition of a global symbol.

The symbol definition is considered absolute. The symbol is entered in the root segment symbol table.

Keyword syntax:

```
GBLDEF = SNAME:VALUE
```

where:

```
SNAME = global symbol name

VALUE = absolute value (octal) to be assigned to the symbol.
```

Default:

None.

Example:

Declare the symbol SMART to have a value of 152525.

```
GBLDEF = SMART:152525
```

8.4 GLOBAL PATCH (GBLPAT)

Declare a series of patch values in a segment that are relative to a global symbol within a segment.

The value of the global symbol is taken as the base address of where the patches are to be applied. All patches must be within the segment or a load address error is generated.

Keyword syntax:

```
GBLPAT = SGNAM:SNAME:VALUE:VALUE:...:VALUE

         or

GBLPAT = SGNAM:SNAME+OFFSET:VALUE:VALUE:...:VALUE

         or

GBLPAT = SGNAM:SNAME-OFFSET:VALUE:VALUE:...:VALUE
```

where:

```
SGNAM =     the name of the segment in which the patches
            are to be applied
```

SNAME = global symbol name

OFFSET = relative offset (octal) from the global symbol to where patch values are to be applied

VALUE = patch values (octal)

NOTE

This command requires at least one patch value and can include a maximum of eight patch values.

Default:

None.

Example:

Declare a series of patches in the segment TELTAL relative to the global symbol PATCH.

```
GBLPAT = TELTAL:PATCH+10:177406:177344
```

NOTE

Patch values are stored in consecutive locations as a byte string. Each patch value requires two bytes.

APPENDIX A

ERROR HANDLING

The following error diagnostics are issued by LINK:

PREMATURE EOF COMMAND INPUT FILE

An end-of-file condition was encountered when LINK was expecting additional command input.

COMMAND SYNTAX ERROR

The command string last issued to LINK was not a valid command. It must be re-entered correctly.

REQUIRED INPUT FILE MISSING

At least one input file must be specified to LINK.

ILLEGAL SWITCH filnam.ext

The switch(es) specified for the file filnam.ext cannot be recognized or processed correctly.

NO DYNAMIC STORAGE AVAILABLE

LINK has no more memory available to complete a link. The link can be re-executed only if the memory requirement for linking is reduced.

"@" COMMAND FILE SYNTAX ERROR

An indirect command file has been incorrectly specified. The string following the @ character was not recognized.

INDIRECT FILE OPEN FAILURE

An indirect file that has been specified cannot be found.

INDIRECT COMMAND SYNTAX ERROR

An indirect command line has been specified incorrectly. Probably, no file name has been specified following the @ character.

INDIRECT FILE DEPTH EXCEEDED

An attempt has been made to nest more than five indirect files.

I/O FAILURE ON INPUT FILE filnam.ext

LINK cannot correctly read data from the file filenam.ext.

OPEN FAILURE ON FILE filnam.ext

LINK cannot find a specified file filnam.ext

SEARCH STACK OVERFLOW ON SEGMENT segnam

Too many overlay levels have been specified. Overlays must not be nested to a depth greater than 16 levels.

PASS CTRL STACK OVERFLOW ON SEGMENT segnam

Too many overlay levels have been specified. Overlays must not be nested to a depth greater than 16 levels.

FILE filnam.ext HAS ILLEGAL FORMAT

The file filnam.ext does not have the correct format for LINK.

MODULE modnam AMBIGUOUSLY DEFINES CTRL SECT secnam

LINK has found two or more P-section descriptions in the same segment whose attributes are not identical.

MODULE modnam MULTIPLY DEFINES CTRL SECT secnam

The P-section (secnam) described in a module (modnam) is not the original definition.

MODULE modnam ILLEGALLY DEFINES XFR ADDRESS transf

A transfer address (transf) is incorrectly defined in a module (modnam). Possibly, trasf was specified within an overlay segment.

CTRL SECTION secnam HAS OVERFLOWED

The control section secnam has overflowed machine address boundaries. No segment can exceed 32K words.

MODULE modnam AMBIGUOUSLY DEFINES SYMBOL symnam

The module modnam has defined a reference (symnam) that has been previously defined. Such a reference cannot be uniquely resolved.

MODULE modnam MULTIPLY DEFINES SYMBOL symnam

Two definitions for the same symbol (symnam) have occurred on the same path within a module (modnam).

SEGMENT segnam HAS RO CONTROL SECTION

Overlay segment segnam contains an RO control section. RO control sections can be specified for root segments only.

SEG segnam HAS ADDR OVERFLOW-ALLOCATION DELETED

The program has attempted to allocate more than 32K words within an overlay segment (segnam). This results in deletion of the program image file; a map is produced, but the program image file is not.

ALLOCATION FAILURE ON FILE filnam.ext

There was not sufficient space on the disk to allocate the output file (filnam.ext) contiguously.

I-O ERROR ON OUTPUT FILE filnam.ext

An unrecoverable output error has occurred on the file filnam.ext.

LOAD ADDR OUT OF RANGE IN MODULE modnam

An address has been specified within a segment of the module modnam that does not fall within the range specified for the segment.

TRUNCATION ERROR IN MODULE modnam

A byte value specified as relocatable in the module modnam exceeded 8 bits after relocation bias was added. The low-order eight bits are loaded into the byte.

cnt UNDEFINED SYMBOLS

Undefined symbols have been encountered during a link. The value cnt specifies the number of undefined symbols.

INVALID KEYWORD IDENTIFIER keynam

The name keynam has been specified and is not a legal options keyword. (See Chapter 8 for legal options.)

OPTION SYNTAX ERROR

The format of an option command is incorrect.

TOO MANY PARAMETERS

Too many parameters have been specified with an options keyword.

ILLEGAL MULTIPLE PARAMETER SETS

The specified option allows only one parameter set; more have been specified.

INSUFFICIENT PARAMETERS

Not enough parameters have been supplied for the option specified.

OVERLAY DIRECTIVE HAS NO OPERANDS

An overlay directive has been supplied without operands. The only directive that allows no operands is the .END directive.

ILLEGAL OVERLAY DIRECTIVE

An unrecognizable overlay directive has been encountered

OVERLAY DIRECTIVE SYNTAX ERROR

An overlay directive has been specified in an incorrect format.

ROOT SEGMENT IS MULTIPLY DEFINED

One .ROOT command (and no more than one) must be specified per program. This program has defined more than one .ROOT command.

LABEL OR NAME IS MULTIPLY DEFINED

A label or name has been defined more than once in an overlay description. This is illegal; labels and names must be uniquely defined.

NO ROOT SEGMENT SPECIFIED

An overlaid program does not have a root segment specified. This is illegal; overlaid programs must spcify a root segment.

BLANK CONTROL SECTION NAME IS NOT LEGAL

A .PSECT command has specified a blank name. This is illegal; all .PSECT commands, when used, must specify non- blank names.

ILLEGAL CONTROL SECTION ATTRIBUTE

An unrecognizable .PSECT attribute has been encountered.

ILLEGAL OVERLAY DESCRIPTION OPERATOR

An illegal ODL operator has been encountered.

TOO MANY NESTED .ROOT-.FCTR DIRECTIVES

An attempt has been made to nest .FCTR statements to a depth greater than 32 levels.

TOO MANY PARENTHESIS LEVELS

An attempt has been made to nest parentheses to a depth greater than 32 levels in an overlay description.

UNBALANCED PARENTHESIS

An overlay description contains mismatched parentheses (e.g., an odd number of parentheses).

ILLEGAL /B OR /T SWITCH VALUE value

The /B or /T switch value specified (value) was not a 1- to 6-digit octal constant.

ILLEGAL /TR SWITCH VALUE value

The /TR switch value specified (value) was one of the following: (1) an odd number, (2) an undefined symbol, or (3) an out-of-range symbol.

ILLEGAL /CO VALUE PARAMETER value

The /CO switch value specified (value) was not a decimal number that is an integral multiple of 64.

MISSING /B OR /T SWITCH VALUE

A /B or /T switch has been specified without a value. The /B and /T switches, when specified, must have an associated value.

APPENDIX B

LINK INPUT DATA FORMATS

An object module is the fundamental unit of input to LINK.

Object modules are created by any of the standard language processors(i.e. MACRO-11, FORTRAN, etc.) or LINK itself (symbol definition file). The librarian provides the capability to combine a number of object modules together into a single library file.

An object module consists of variable length records of information that describe the contents of the module. Six record (or block) types are included in the object language. These records guide LINK in the translation of the object language into a task image.

The six record types are:

Type 1 - Declare Global Symbol Directory (GSD)

Type 2 - End of Global Symbol Directory

Type 3 - Text Information (TXT)

Type 4 - Relocation Directory (RLD)

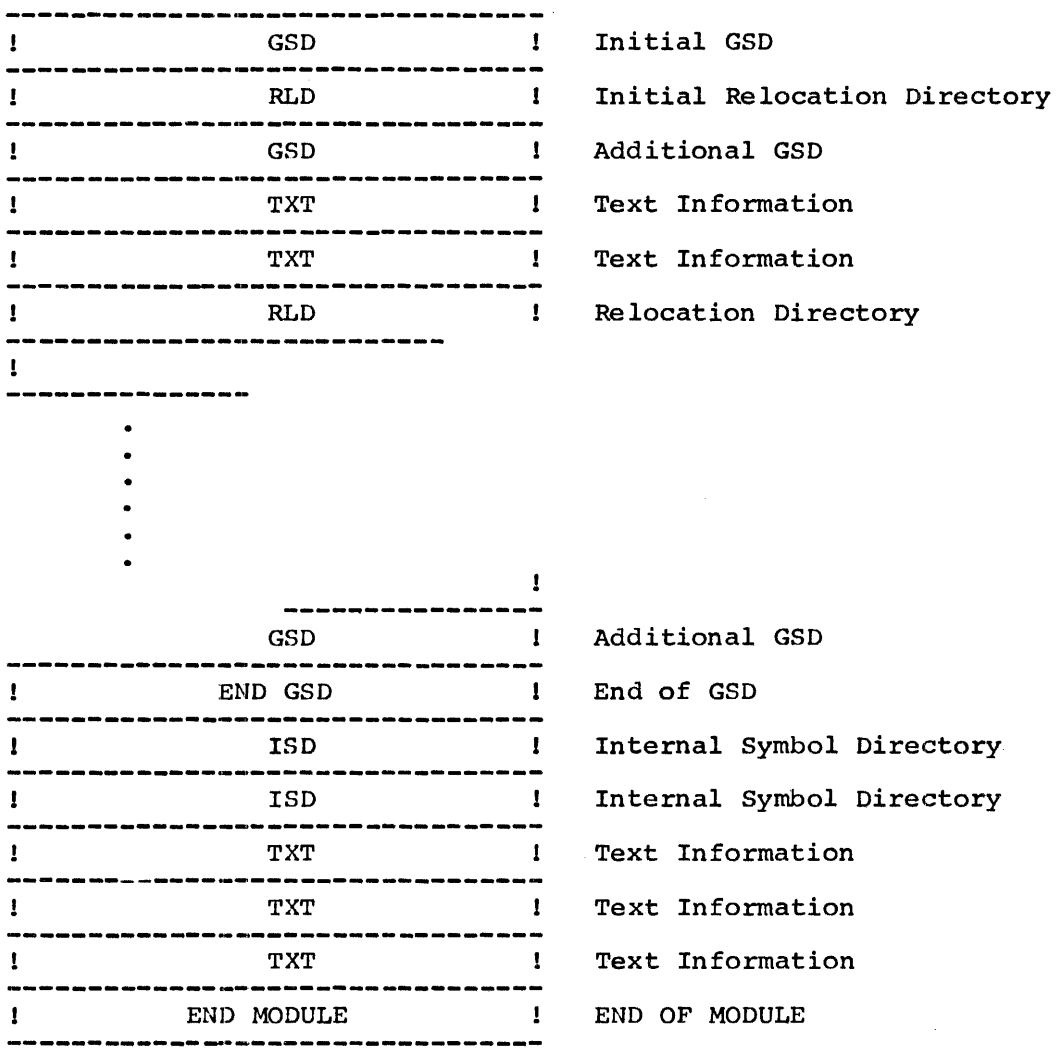
Type 5 - Internal Symbol Directory (ISD)

Type 6 - End of Module

Each object module must consist of at least five of the record types. The one record type that is not mandatory is the internal symbol directory. The appearance of the various record types in an object module follows a defined format.

An object module must begin with a declare GSD record and end with a end of module record. Additional declare GSD records may occur anywhere in the file but before an end of GSD record. An end of GSD record must appear before the end of module record. At least one relocation directory record must appear before the first text information record Additional relocation directory and text information records may appear anywhere in the file. The internal symbol directory records may appear anywhere in the file between the initial declare GSD and end of module records.

Object module records are variable length and are identified by a record type code in the first word of the record. The format of additional information in the record is dependent upon the record type.



GENERAL OBJECT MODULE FORMAT

B.1 GLOBAL SYMBOL DIRECTORY

Global symbol directory records contain all the information necessary to assign virtual addresses to global symbols and to allocate the virtual memory required by a program.

GSD records are the only records processed in the first pass and therefore significant time can be saved if all GSD records are placed at the beginning of a module (i.e., less of the file must be read in phase 3).

GSD records contain 7 types of entries:

Type 0 - Module Name

Type 1 - Control Section Name

Type 2 - Internal Symbol Name

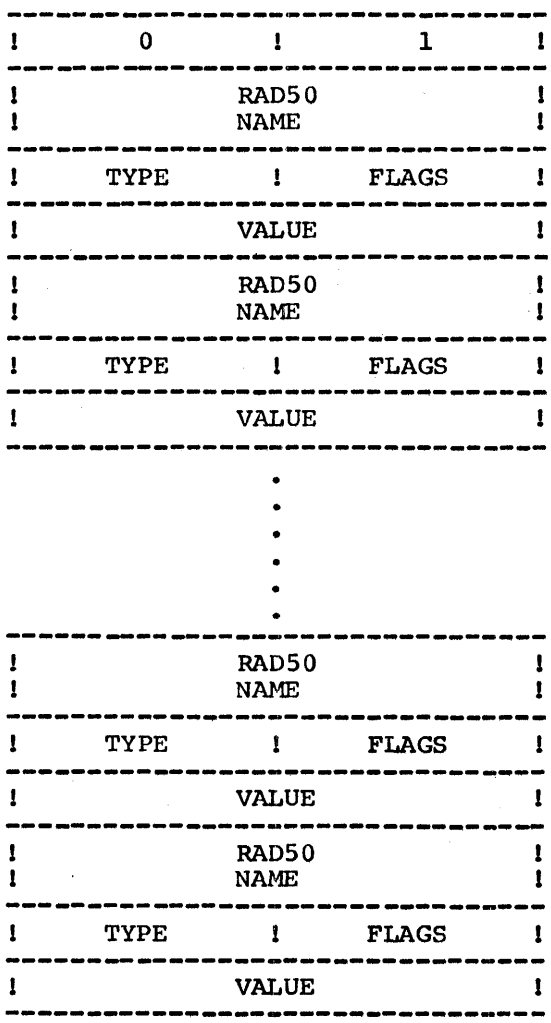
Type 3 - Transfer Address

Type 4 - Global Symbol Name

Type 5 - Program Section Name

Type 6 - Program Version Identification

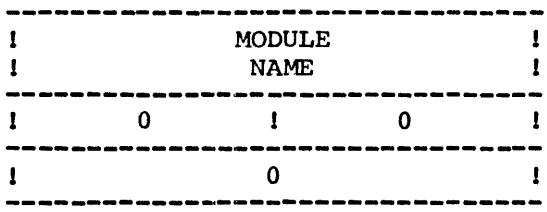
Each type of entry is represented by four words in the GSD record. The first two words contain six RAD50 characters. The third word contains a flag byte and the entry type identification. The fourth word contains additional information about the entry.



GSD RECORD
AND
ENTRY FORMATS

B.1.1 Module Name

The module name entry declares the name of the object module. The name need not be unique with respect to other object modules (i.e. modules are identified by file not module name) but only one such declaration may occur in any given object module.



MODULE NAME ENTRY FORMAT

B.1.2 Control Section Name

Control sections, which include ASECTs, blank-CSECTS, and named-CSECTs are obviated in DOS by PSECTs. For compatibility, LINK processes ASECTs and both forms of CSECTs. Section B.1.6 details the entry generated for for a PSECT statement. In terms of a PSECT statement we can define ASECT and CSECT statements as follows:

For a blank CSECT:

```
.PSECT ,LCL,REL,CON,RW,I,LOW
```

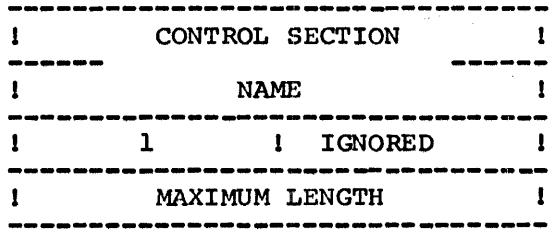
For a named CSECT:

```
.PSECT name, GBL,REL,OVR,RW,I,LOW
```

And for an ASECT:

```
.PSECT .  ABS.,GBL,ABS,I,OVR,RW,LOW
```

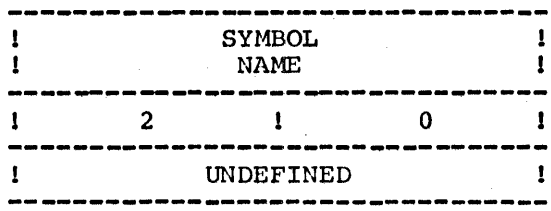
ASECTs and CSECTs are processed by LINK as PSECTs with the fixed attributes defined above. For a complete description of PSECT processing see B.1.6. The entry generated for a control section is shown in the Figure below.



CONTROL SECTION NAME ENTRY FORMAT

B.1.3 Internal Symbol Name

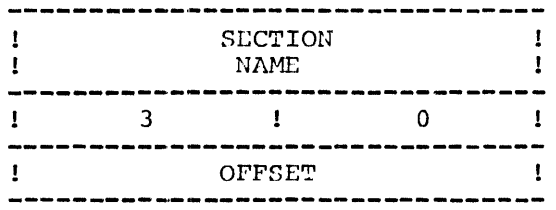
The internal symbol name entry declares the name of an internal symbol (with respect to the module). LINK does not yet support internal symbol tables; therefore the detailed format of this entry is not defined. If an internal symbol entry is encountered while reading the GSD, it is merely ignored.



INTERNAL SYMBOL NAME ENTRY FORMAT

B.1.4 Transfer Address

The transfer address entry declares the transfer address of a module relative to a P-section. The first two words of the entry define the name of the P-section and the fourth word the relative offset from the beginning of that P-section. If no transfer address is declared in a module, a transfer address entry must either not be included in the GSD or a transfer address of 000001 relative to the default absolute P-section (. ABS.) must be specified.



TRANSFER ADDRESS ENTRY FORMAT

NOTE

If the P-section is absolute, then OFFSET is the actual transfer address if not 000001.

B.1.5 Global Symbol Name

The global symbol name entry declares either a global reference or definition. All definition entries must appear after the declaration of the P-section under which they are defined and before the delcaration of another P-section. Global references may appear anywhere within the GSD.

The first two words of the entry define the name of the global symbol. The flag byte declares the attributes of the symbol and the fourth word the value of the symbol relative to the P-section under which it is defined.

The flag byte of the symbol declaration entry has the following bit assignments.

Bits 0 - 2 - Not used.

Bit 3 - Definition

0 = Global symbol references.

1 = Global symbol defintion.

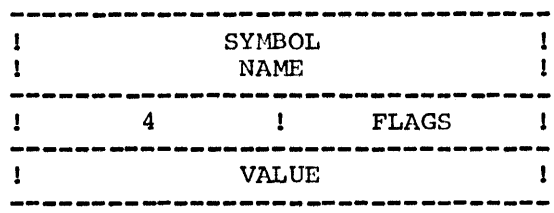
Bit 4 - Not used

Bit 5 - Relocation

0 = Absolue symbol value.

1 = Relative symbol value

Bit 6 - 7 - Not used.



GLOBAL SYMBOL NAME ENTRY FORMAT

B.1.6 Program Section Name

The P-section name entry declares the name of a P-section and its maximum length in the module. It also declares the attributes of the P-section via the flag byte.

GSD records must be constructed such that once a P-section name has been declared all global symbol definitions that pertain to that P-section must appear before another P-section name is declared. Global symbols are declared via symbol declaration entries. Thus the normal format is a P-section name followed by zero or more symbol declarations followed by another P-section name followed by zero or more symbol declarations and so on.

The flag byte of the P-section entry has the following bit assignments:

Bit 0 - Memory Speed

0 = P-section is to occupy low speed (core) memory.

1 = P-section is to occupy high speed (i.e. MOS/Bipolar) memory.

Bit 1 - Library P-section (not used by LINK)

0 = Normal P-section.

1 = Relocatable P-section that references a core resident library or common block.

Bit 2 - Allocation

0 = P-section references are to be concatenated with other references to the same P-section to form the total memory allocated to the section.

1 = P-section references are to be overlaid. The total memory allocated to the P-section is the largest request made by individual references to the same P-section.

Bit 3 - Not used but reserved.

Bit 4 - Access

0 = P-section has read/write access.

1 - P-section has read only access.

Bit 5 - Relocation

0 = P-section is absolute and requires no relocation.

1 = P-section is relocatable and references to the control section must have a relocation bias added before they become absolute.

Bit 6 - Scope

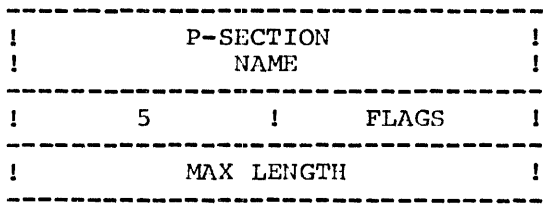
0 = The scope of the P-section is local. References to the same P-section will be collected only within the segment in which the P-section is defined.

1 = The scope of the P-section is global. References to the P-section are collected across segment boundries. The segment in which a global P-section is allocated storage is either determined by the first module that defines the P-section on a path or direct placement of a P-section in a segment via the segment description map.

Bit 7 - Type

0 = The P-section contains instruction (I) references.

1 = The P-section contains data (D) references.



P-SECTION NAME ENTRY FORMAT

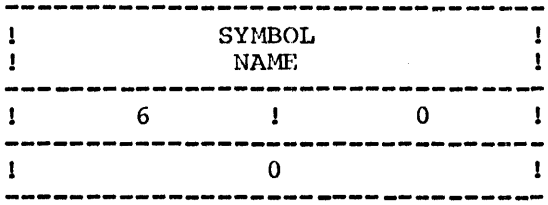
NOTE

The length of all absolute sections is zero.

B.1.7 Program Version Identification

The program version identification entry declares the version of the module. LINK saves the version identification of the first module that defines a nonblank version. This identification is then included on the memory allocation map and is written in the label block of the task image file.

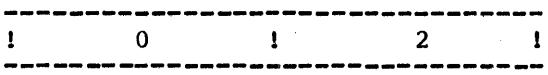
The first two words of the entry contains the version identification. The flag byte and fourth words are not used and contain no meaningful information.



PROGRAM VERSION IDENTIFICATION
ENTRY FORMAT

B.2 END OF GLOBAL SYMBOL DIRECTORY

The end of global symbol directory record declares that no other GSD records are contained further on in the file. Exactly one end of GSD record must appear in every object module and is one word in length.



END OF GSD RECORD FORMAT

B.3 TEXT INFORMATION

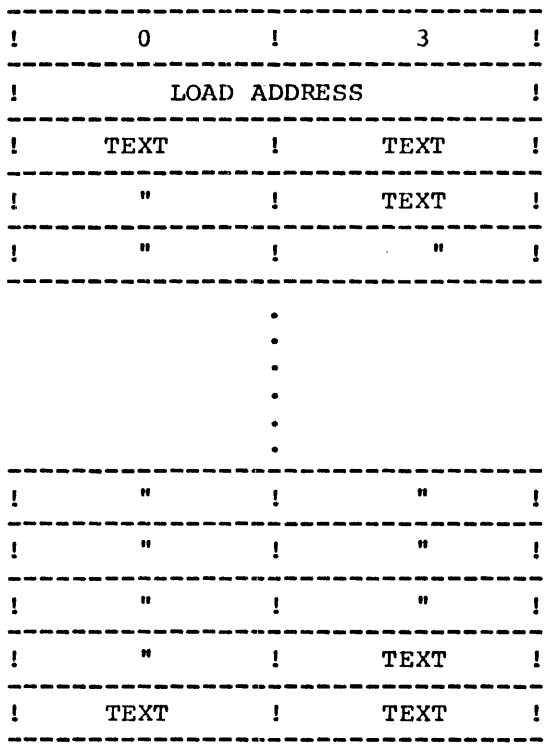
The text information record contains a byte string of information that is to be written directly into the task image file. The record consists of a load address followed by the byte string.

Text records may contain words and/or bytes of information whose final contents are yet to be determined. This information will be bound by a relocation directory record that immediately follows the text record (see B.4 below). If the text record does not need modification, then no relocation directory record is needed. Thus multiple text records may appear in sequence before a relocation directory record.

The load address of the text record is specified as an offset from the current P-section base. At least one relocation directory record must

precede the first text record. This directory must declare the current P-section.

LINK writes a text record directly into the program image file and computes the value of the load address minus four. This value is stored in anticipation of a subsequent relocation directory that modifies words and/or bytes that are contained in the text record. When added to a relocation directory displacement byte, this value yields the address of the word and/or byte to be modified in the task image.



TEXT INFORMATION RECORD FORMAT

B.4 RELOCATION DIRECTORY

Relocation directory records contain the information necessary to relocate and link a preceeding text information record. Every module must have at least one relocation directory record that precedes the first text information record. The first record does not modify a preceeding text record, but rather it defines the current P-section and location. Relocation directory records contain 13 types of entries. These entries are classified as relocation or location modification entries. The following type of entries are defined:

Type 1 - Internal Relocation

Type 2 - Global Relocation

Type 3 - Internal Displaced Relocation

Type 4 - Global Displaced Relocation

Type 5 - Global Additive Relocation

Type 6 - Global Additive Displaced Relocation

Type 7 - Location Counter Definition

Type 10 - Location Counter Modification

Type 11 - Program Limits

Type 12 - P-Section Relocation

Type 13 - Not used

Type 14 - P-Section Displaced Relocation

Type 15 - P-Section Additive Relocation

Type 16 - P-Section Additive Displaced Relocation

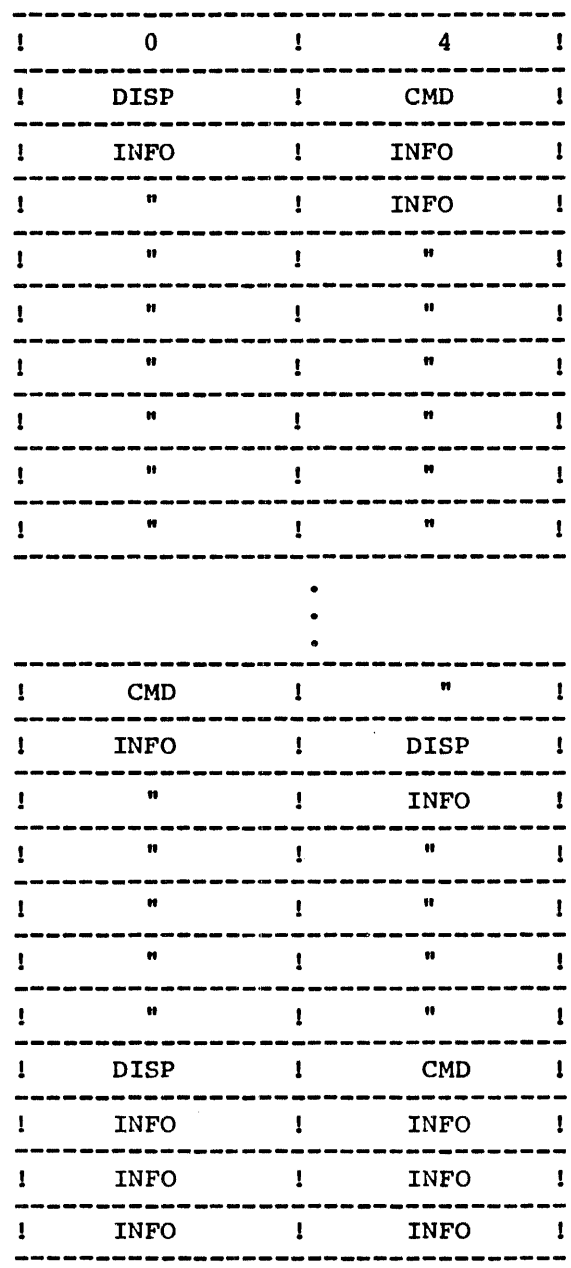
Each type of entry is represented by a command byte (specifies type or entry and word/byte modification)) followed by a displacement byte followed by the information required for the particular type of entry. The displacement byte, when added to the value calculated from the load address of the previous text information record, (see B.3 above) yields the virtual address in the image that is to be modified. The command byte of each entry has the following bit assignments.

Bits 0 - 6 Specify the type of entry. Potentially 128 command types may be specified although only 13 are implemented.

Bit - 7 Modification

0 = The command modifies an entire word.

1 = The command modifies only one byte. LINK checks for truncation errors in byte modification commands. If truncation is detected (i.e. the modification value has a magnitude greater than 255), then an error is produced.



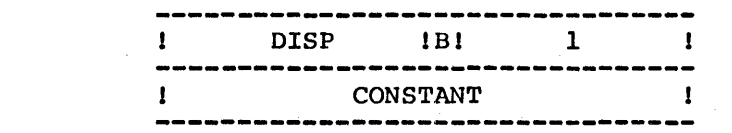
RELOCATION DIRECTORY RECORD FORMAT

B.4.1 Internal Relocation

This type of entry relocates a direct pointer to an address within a module. The current P-section base address is added to a specified constant and the result is written into the task image file at the calculated address (i.e. displacement byte added to value calculated from the load address of the previous text block).

Example:

```
A:      MOV     #A,R0

            or

        .WORD   A
```



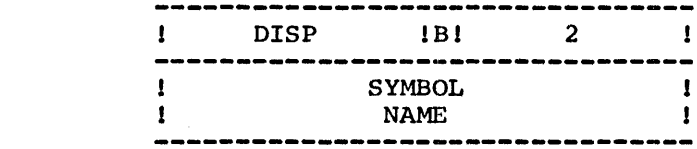
INTERNAL RELOCATION COMMAND FORMAT

B.4.2 Global Relocation

This type of entry relocates a direct pointer to a global symbol. The definition of the global symbol is obtained and the result is written into the task image file at the calculated address.

Example:

```
        MOV     #GLOBAL,R0

                   or

        .WORD   GLOBAL
```

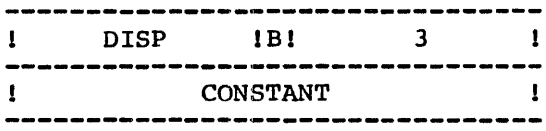


GLOBAL RELOCATION

B.4.3 Internal Displaced Relocation

This type of entry relocates a relative reference to an absolute address from within a relocatable control section. The address plus 2 that the relocated value is to be written into is subtracted from the specified constant. The result is then written into the task image file at the calculated address.

```
        CLR     177550

                   or

        MOV     177550.R0
```



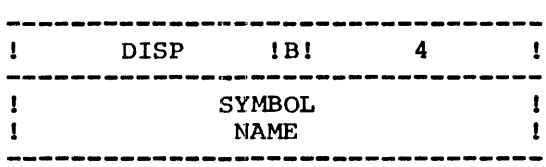
INTERNAL DISPLACED RELOCATION

B.4.4 Global Displaced Relocation

This type of entry relocates a relative reference to global symbol. The definition of the global symbol is obtained and the address plus 2 that the relocated value is to be written into is subtracted from the definition value. This value is then written into the task image file at the calculated address.

Example:

```
CLR     GLOBAL

           or

MOV     GLOBAL,R0
```



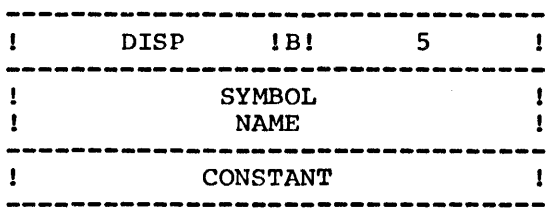
GLOBAL DISPLACED RELOCATION

B.4.5 Global Additive Relocation

This type of entry relocates a direct pointer to a global symbol with an additive constant. The definition of the global symbol is obtained, the specified constant is added, and the resultant value is then written into the task image file at the calculated address.

Example:

```
MOV     #GLOBAL+2,R0

        or

.WORD   GLOBAL-4
```



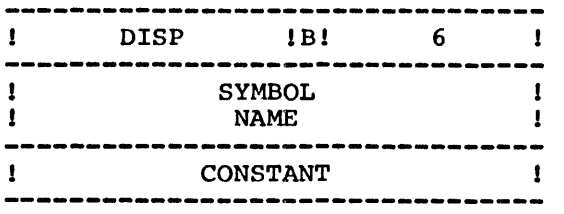
GLOBAL ADDITIVE RELOCATION

B.4.6 Global Additive Displaced Relocation

This type of entry relocates a relative reference to a global symbol with an additive constant. The definition of the global symbol is obtained and the specified constant is added to the definition value. The address plus 2 that the relocated value is to be written into is subtracted from the resultant additive value. The resultant value is then written into the task image file at the calculated address.

Example:

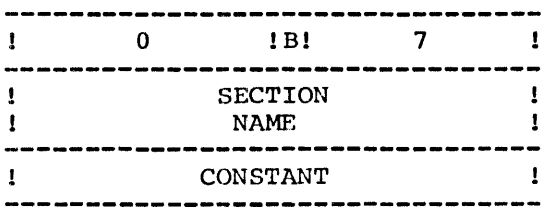
```
CLR     GLOBAL+2

        or

MOV     GLOBAL-5,R0
```



GLOBAL ADDITIVE DISPLACED RELOCATION

B.4.7 Location Counter Definition

This type of entry declares a current P-section and location counter value. The control base is stored as the current control section and the current control section base is added to the specified constant and stored as the current location counter value.



LOCATION COUNTER DEFINITION

B.4.8 Location Counter Modification

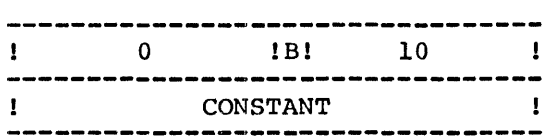
This type of entry modifies the current location counter. The current P-section base is added to the specified constant and the result is stored as the current location counter.

Example:

```
.=.+N
```

or

```
.BLKB   N
```



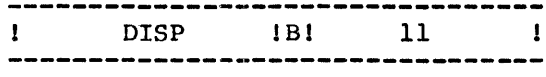
LOCATION COUNTER MODIFICATION

B.4.9 Program Limits

This type of entry is generated by the .LIMIT assembler directive. The lowest and highest virtual addresses allocated to the task are obtained and written into the task image file at the calculated address and at the calculated address plus 2 respectively.

Example:

```
.LIMIT
```



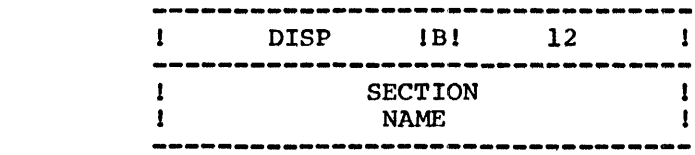
PROGRAM LIMITS

B.4.10 P-Section Relocation

This type of entry relocates a direct pointer to the beginning address of another P-section (other than the P-section in which the reference is made) within a module. The current base address of the specified P-section is obtained and written into the task image file at the calculated address.

Example:

```
        .PSECT  A
B:
        .
        .
        .
        .
        PSECT   C
        MOV     #B,R0

           or

        .WORD   B
```



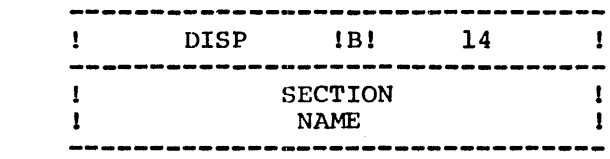
P-SECTION RELOCATION

B.4.11 P-Section Displaced Relocation

This type of entry relocates a relative reference to the beginning address of another P-section within a module. The current base address of the specified P-section is obtained and the address plus 2 that the relocated value is to be written into is subtracted from the base value. This value is then written into the task image file at the calculated address.

Example:

```
        .PSECT  A
B:
        .
        .
        .
        .
        .PSECT  C
        MOV     B,R0
```



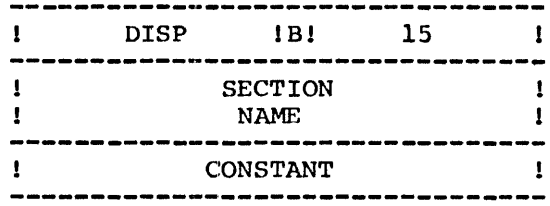
P-SECTION DISPLACED RELOCATION

B.4.12 P-Section Additive Relocation

The type of entry relocates a direct pointer to an address in another P-section within a module. The current base address of the specified P-section is obtained and added to the specified constant. The result is written into the task image file at the calculated address.

Example:

```
          .PSECT  A
B:
          .
          .
          .
          .
          .
C:        .
          .
          .
          .
          PSECT   D
          MOV     #B+10,R0
          MOV     #C,R0

          or

          .WORD   B+10
          .WORD   C
```



P-SECTION ADDITIVE RELOCATION

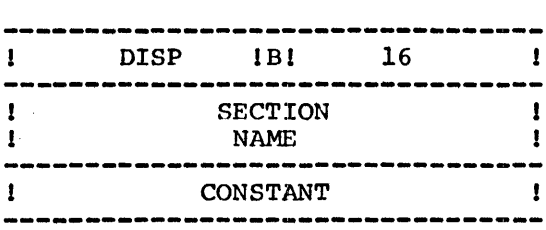
B.4.13 P-section Additive Displaced Relocation

This type of entry relocates a relative reference to address in another P-section within a module. The current base address of the specified P-section is obtained and added to the specified constant. The address plus 2 that the relocated value is to be written into is

subtracted from the resultant additive value. This value is then written into the task image file at the calculated address.

Example:

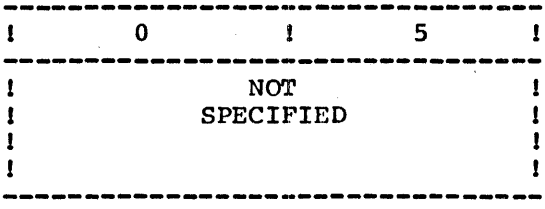
```
          .PSECT  A
B:
          .
          .
          .
C:        .
          .
          .
          .
          .
          .PSECT  D

          MOV     B+10,R0
          MOV     C,R0
```



P-SECTION ADDITIVE DISPLACED RELOCATION

B.5 INTERNAL SYMBOL DIRECTORY

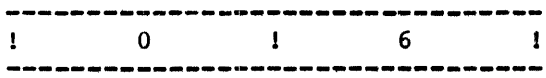
Internal symbol directory records declare definitions of symbols that are local to a module. This feature is not supported by LINK and therefore a detailed record format is not specified. If LINK encounters this type of record, it will ignore it.



INTERNAL SYMBOL DIRECTORY RECORD FORMAT

B.6 END OF MODULE

The end of module record declares the end of an object module. Exactly one end of module record must appear in one object module and is one word in length.



END OF MODULE RECORD FORMAT

APPENDIX C

PROGRAM LOAD MODULE FILE STRUCTURE

An Overlay image as it is recorded on disk appears in Figure C-1 (pertinent boundaries are shown).

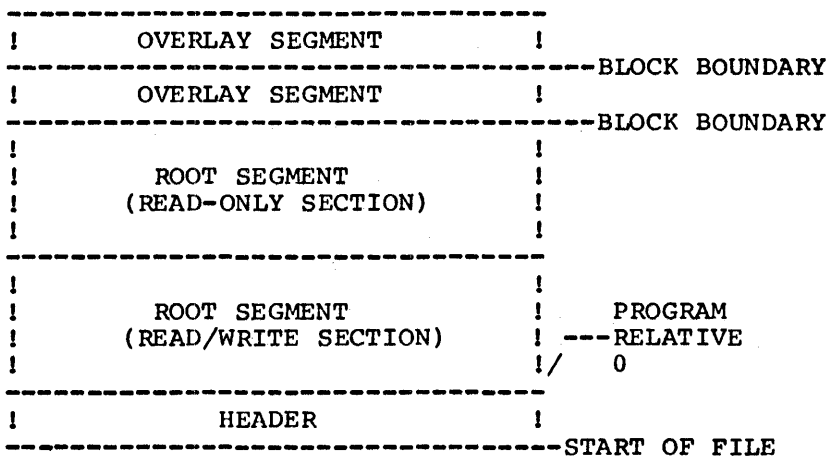


Figure C-1 Overlay Disk Format

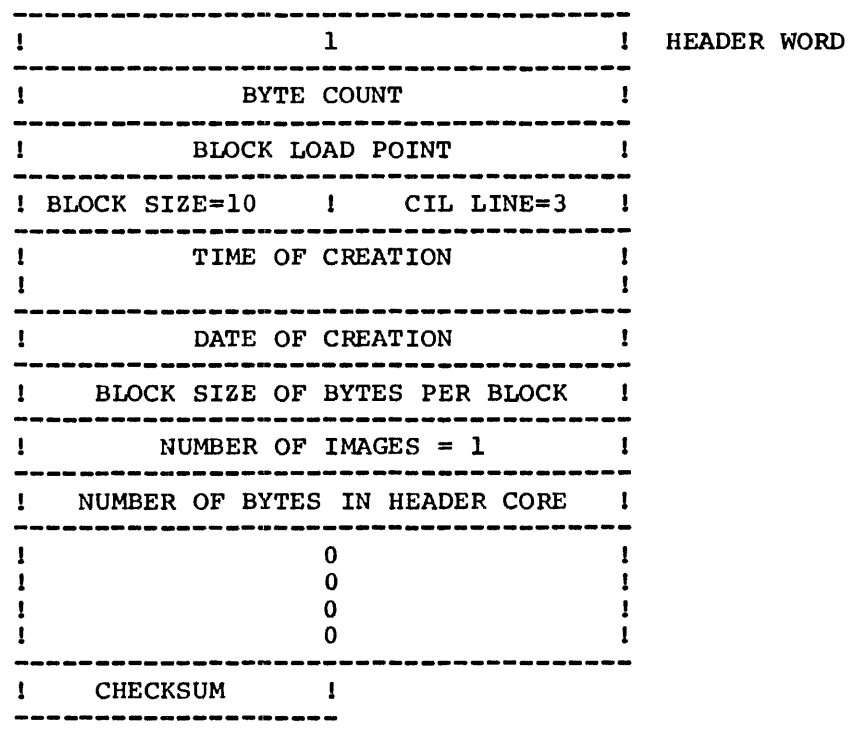
C.1 THE HEADER

The overlay header consists of two parts:

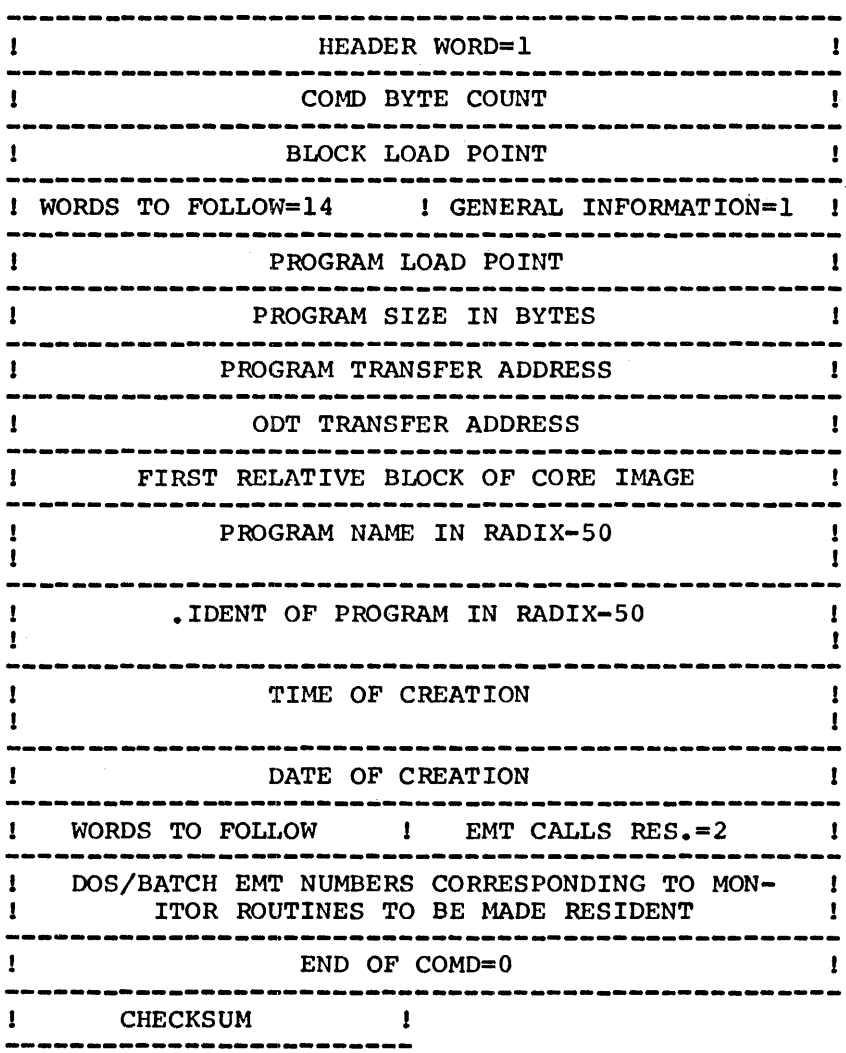
1. Core Image Descriptor
2. Communications Directory (COMD)

The core image descriptor is similar to a CIL line as produced by CILUS, except that certain items not needed by LINK are left out. The COMD describes the characteristics of the root segment so that it can be loaded and run using the DOS/BATCH RUN command.

The core image descriptor has the following format:



The COMD has the following format:


HEADER WORD=1
COMD BYTE COUNT
BLOCK LOAD POINT
WORDS TO FOLLOW=14
GENERAL INFORMATION=1
PROGRAM LOAD POINT
PROGRAM SIZE IN BYTES
PROGRAM TRANSFER ADDRESS
ODT TRANSFER ADDRESS
FIRST RELATIVE BLOCK OF CORE IMAGE
PROGRAM NAME IN RADIX-50
.IDENT OF PROGRAM IN RADIX-50
TIME OF CREATION
DATE OF CREATION
WORDS TO FOLLOW
EMT CALLS RES.=2
DOS/BATCH EMT NUMBERS CORRESPONDING TO MON-
ITOR ROUTINES TO BE MADE RESIDENT
END OF COMD=0
CHECKSUM

C.2 THE ROOT SEGMENT

The root segment is written as a contiguous number of blocks starting after the header.

C.3 OVERLAY SEGMENTS

Every overlay segment begins on a block boundary and is always read/write. The relative block number for the segment is placed in the segment table making it possible to load any overlay segment with a single read. Note that a given overlay segment occupies as many contiguous disk blocks as it needs to supply is space requests - the maximum size for any segment, including the root, is 32K words.

C.4 NON-OVERLAID PROGRAM FILE STRUCTURES

The file structure of a non-overlaid program normally consists of a formatted binary file beginning with a header whose COMD is similar to that described in section C.1, with the following exceptions:

1. "FIRST RELATIVE BLOCK OF IMAGE" entry is not included.

2. Byte count is correspondingly smaller.

The end of a non-overlaid program indicated by a formatted binary line with a byte count of 6. This line is the transfer address block.

If the /CO switch is used to produce a non-overlaid program, the file image will be as shown in Figure C.1, except for the omission of overlay segments.

APPENDIX D

COMPATIBILITY OF LINK-11 AND THE RSX-11D TASK BUILDER

LINK is not completely compatible with other PDP-11 linkers (e.g., the old LINK-11). Rather than define compatible areas, the following known incompatibilities exist.

Command language. The command language is basically that of DOS/BATCH, but some keyword options are different, and the overlay description language is new.

Memory allocation. The allocation of memory is in accordance with the enhanced P-section capability originally developed for RSX-11D. This allocation is not compatible in the handling of the blank P-sections. Named .PSECT's or .CSECT's are not interspersed in the blank .PSECT.

P-section names. P-section names are not treated as global symbols. Thus global symbols may have the same name as control sections without conflict.

Symbol table. The STB symbol table file format is not file formats compatible. The STB files created under the old Linker cannot be used with LINK.

Tapes switch. The /TA (tapes) switch is not implemented in LINK. If multisection paper tapes are to be linked, they must be specified individually; alternatively, they can be copied to the disk in advance using the PIP program.

Overlay switch. The /OV (overlay) switch is not implemented in LINK. See Appendix F for information on building overlays using the "CALL LINK" format.

Order of .PSECT's and .CSECT's. .PSECT's and .CSECT's are placed in memory in alphabetic order, not in order of declaration as in the old Linker. See Section 3.2.1.14.

Undefined globals switch. The /U (undefined globals) switch is not implemented in LINK. New map capabilities make it obsolete.

Map output. LINK map output is completely different from that of the old Linker (see Chapter 3 and Appendix H).

Error messages. All LINK error messages are incompatible with those issued by the old Linker. LINK error messages are textual rather than numeric (see Appendix A).

Library format. The LINK library format is slightly different from that of the old Linker. Use of old-format libraries with LINK causes map listing errors; however, new library formats are compatible with the old Linker.

Word boundaries. LINK does not automatically round addresses to a word boundary at the end of a module. If a module ends on an odd boundary, the following module will start on an odd boundary.

APPENDIX E

RESERVED SYMBOLS AND SPECIAL FILES

1. The symbol .NSTBL is reserved by LINK. Special handling occurs when the definition of this name is encountered in a program. Definition of this global symbol causes the word pointed to by this symbol to be modified with a value calculated by LINK. The value placed in this location is the address of the segment description tables. Note that this modification occurs only when the number of segments is greater than one.

2. If a global CREF is desired, the file GLOB.TMP is generated by LINK. After the global CREF is listed, GLOB.TMP is deleted. If a file named GLOB.TMP already exists when a CREF is specified, that file is deleted.

3. Overlay run-time support uses the following global symbols, which should not be accessed in any way by the user.

```
$AUTO
$MARKS
$RDSEG
$RETA
$SAVAL
$$WAIT
```

APPENDIX F

LINKING OVERLAYS USING "CALL LINK" FORMS

The "CALL LINK" capability, which was available with older versions of the Linker program, is still usable under LINK to maintain compatibility. However, the new LINK AUTOLOAD (LOCAL) overlays are much faster and generally require less memory overhead. The new overlay structures are described in Chapter 4. The contents of this appendix, in general, are not applicable to overlay structures described in Chapter 4.

Note that this Appendix does not describe AUTOLOAD or manual load overlays. The "CALL LINK" overlay capability is not compatible with those forms, and is not recommended in any case.

The run time supervisor (which is linked to the resident section of the user program) of the "CALL LINK" overlay facility is the subroutine LINK in the FORTRAN library, version V020A or later.

The resident portion of the run-time supervisor requires approximately 150(10) words. However, the Monitor requires an additional 768(10) words for buffers and approximately 150(10) words for the overlay stack at the time the overlay is loaded.

"CALL LINK" Overlays communicate through blank or labeled COMMON in the resident area or through the contents of general registers R0 through R4 and the stack. Overlay files can use any core resident routines (user generated or acquired from a Library).

F.2 COMMUNICATION AMONG RESIDENT AND OVERLAY ROUTINES

Overlays and the resident section communicate through blank and labeled COMMON areas (.CSECT's for assembly language programs) in the core resident area. Data can also be passed between overlays from assembly language programs in the general registers R0 through R4 and the stack.

F.3 LINK, THE RUN-TIME OVERLAY SUPERVISOR

The subroutine LINK is the run-time overlay supervisor. It has three entry points: LINK, RETURN and RUN (for RUN, see Section F.12).

The subroutine LINK performs the following functions for the user when entered at LINK:

a. Initializes the traceback origin to the new overlay file to allow FORTRAN error tracking via the traceback feature.

b. If called from the resident section saves general registers R0 through R5 and saves the return address to the resident section for use with the next CALL RETURN.

c. Brings the overlay file (designated by the argument to CALL LINK) into core and moves the stack below the new overlay file.

The subroutine LINK performs the following functions for the user when entered at RETURN:

a. restores general registers R0 through R5.

b. transfers control following the last CALL LINK executed from the resident section.

The LINK subroutine contains the global declaration:

```
.GLOBL  $OTSV
```

where $OTSV is the pointer to the FORTRAN impure area. This forces the impure area into the core resident section so that user I/O can be continued across overlays.

F.4 CALLING AN OVERLAY FILE

The resident section and overlay files are entered into the system as load modules. An overlay is requested by a

```
CALL LINK ('A')
```

statement, where A is the file specification of the overlay file to be called into core. All overlays are entered through their main program, not through any subroutines.

The argument of the overlay call is enclosed in single quotes since the Linker relies on a terminating null character inserted by the FORTRAN Compiler at the end of such a string. The maximum length of the argument is 25 decimal bytes (stored one character per byte).

The call can be written as follows:

```
CALL LINK ('dev:file.ext')
```

Due to the DOS default conventions, the call can normally be expressed as:

```
CALL LINK ('file')
```

with the system disk (SY:) the assumed device and .LDA the assumed extension for a load module.

Whenever the statement:

```
CALL RETURN
```

is encountered, control transfers to the instruction following the last CALL LINK executed from the resident section.

The first CALL LINK statement is issued from somewhere in the core resident section of the user program, bringing into core the first overlay file. Subsequent CALL LINK statements can be issued from them

main programs in the various overlays. Each CALL LINK brings into core the specified overlay file.

CALL LINK can be issued from anywhere in the core resident section (main program or subroutines). CALL LINK and CALL RETURN can be issued only from the main program in an overlay and not from a subroutine within an overlay.

There are four allowable control paths between resident and overlay load modules.

a. resident to overlay via CALL LINK

b. overlay to overlay via CALL LINK

c. overlay to resident via CALL RETURN

d. overlay utilizing resident subroutines

Figure F-1 illustrates these transfer paths.

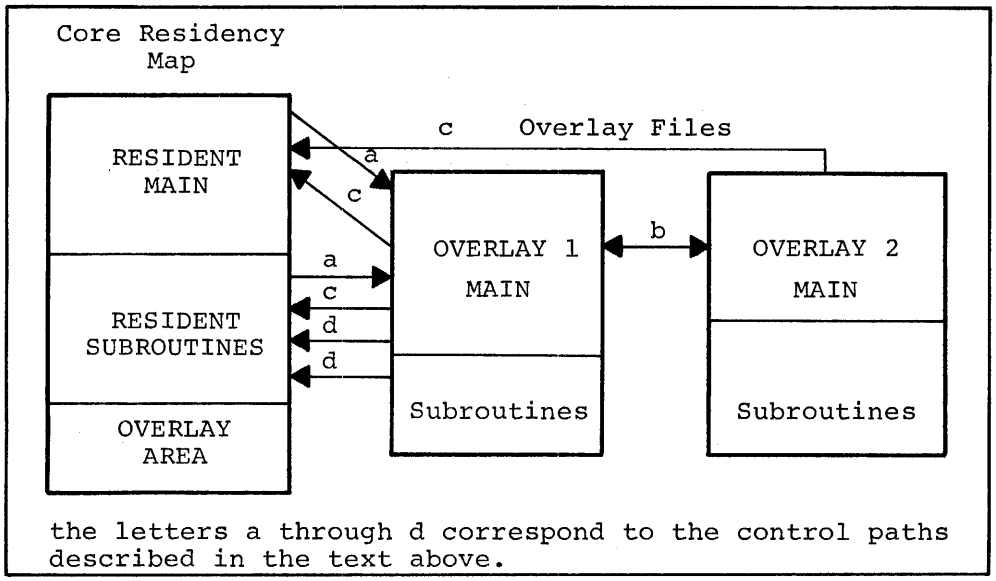


Figure F-1

TRANSFER PATHS BETWEEN RESIDENT AND OVERLAY PROGRAMS

F.4.1 Overlay Transfer Paths

CALL LINK may be issued from anywhere in the resident section. However, in an overlay, CALL LINK and CALL RETURN can be issued only from the main program.

Resident subroutines can be called from anywhere in an overlay file (main program or subroutine). However, there is a restriction that an overlay must not call any resident routine which contains (or calls another routine which contains) a CALL LINK.

Violation of these control path restrictions causes the overlay system to be corrupted, resulting in stack overflow or incorrect subroutine returns.

F.4.2 Search for Overlay Files

Depending upon the format of the overlay file specification, a CALL LINK statement searches for the overlay file under several possible, alternate, file specifications.

Where only the overlay filename was specified, for example:

```
CALL LINK('FILE')
```

the search for the overlay file proceeds as follows:

(1.) FILE.LDA, current UIC
(2.) FILE.LDA, 1,1 UIC
(3.) FILE, no extension, current UIC
(4.) FILE, no extension, 1,1 UIC

Where a filename and extension are specified for the overlay file, for example:

```
CALL LINK('FILE.EXT')
```

the search for the overlay file proceeds as follows

(1.) FILE.EXT, current UIC
(2.) FILE.EXT, 1,1 UIC

Where a UIC is specified and a file extension is not, for example:

```
CALL LINK('FILE[X,X]')
```

the search for the overlay file proceeds as follows:

(1.) FILE.LDA, X,X UIC

(2.) FILE, no extension, X,X UIC

where a complete file specification is given, only one attempt is made to find the file. For example:

```
CALL LINK('FILE.EXT[X,X]')
```

is searched under:

(1.) FILE.EXT, X,X UIC

If the search has failed, the DOS Monitor prints the message:

```
F012  XXXXXX
```

where the additional information word contains the request address (which is within the run-time supervisor).

F.5 OPERATING PROCEDURES

In order to create an overlay system, the FORTRAN main program and various overlay files are separately compiled and linked using LINK. One command string is passed to LINK for the main program and each overlay file. The overlays are built using the symbol table file created by the main. For example:

```
$RU LINK
LINK V01
#RES,LP:,SY:RES<RES,FTNLIB/E
#OVL1,LP:<RES.STB,OVL1,FTNLIB/E
#OVL1,LP:<RES.STB,OVL2,FTNLIB/E
```

F.5.1 Creating the Core Resident Module of an Overlay System

The format of the Linker command string which defines the core resident section is as follows:

```
dev:RES,dev:MAP,dev:ST<dev:RES,...subroutines,FTNLIB/E
```

where:

dev:RES	is the file specification for the user's core resident section load module.
dev:MAP	is the load map of the resident program, which is an optional output file.
dev:ST	is the global symbol table of the resident program module.
dev:RES	is the file specification for the user's core resident main program.
FTNLIB	the FORTRAN Library contains the overlay facility supervisor and other Library routines which must be linked to the user's resident section.
/E	indicates the end of the command string to the Linker. This is followed by the RETURN key.

Following the command string describing the resident module are the command strings describing the overlay files. The format of these command strings is as follows:

```
dev:OVER,dev:MAP<dev:ST,dev:OVER1,...subroutines,FTNLIB/E
```

where:

dev:OVER	is the file specification for the overlay load module. This specification must correspond exactly to the one used as the argument to CALL LINK when this overlay is wanted.
dev:MAP	is the load map of the overlay, which is an optional output file.
dev:ST	input global symbol table of the resident load module created above.
dev:OVER1	is the file specification for the overlay main program.
FTNLIB	the FORTRAN Library must be linked to each overlay.
/E	indicates the end of the command string to the Linker. This is followed by the RETURN key.

Notice that in the command strings describing the resident and overlay files the FORTRAN Library, FTNLIB, is linked to each load module.

Since the default device for load modules is always the system device and the default extension is .LDA, the load module file specification can be simplified for both resident and overlay file creation as follows:

```
RES,dev:MAP,dev:ST<dev:RES,...subroutines,FTNLIB/E

OVER,dev:MAP<dev;ST,dev:OVER1,...subroutines,FTNLIB/E
```

F.5.2 Creating the Overlay Files

The top of the overlay files is automatically set to the location directly below the bottom of the resident section. The /T:n switch can be used to change the default top linkage, if desired. The top of the overlay must never be set to a location above the bottom of the resident section.

In the example shown in Section F.5 the resident symbol table is discarded after the overlay system is linked. If the user can foresee additions or changes in the overlay files, he should save the resident symbol table so that future changes can be made without relinking the entire overlay system. For example:

```
$RU LINK
LINK Vxx
#RES,LP:,SY:RES<RES,FTNLIB/E
#OVL1,LP:<RES.STB,OVL1,FTNLIB/E
#OVL2,LP:<RES.STB,OVL2,FTNLIB/E
```

The first command string to the Linker causes the resident section symbol table (SY:RES.STB) to be maintained on the system disk. This symbol table file contains the global symbol of the resident section and the bottom address of the resident.

To relink a changed version of OVL2, the following command sequence is used:

```
$RU LINK
LINK  Vxx (xx is the version number)
#OVL2,LP:<RES.STB,OV2NEW,FTNLIB/E
```

Notice that in the example above, the symbol table of the resident section is the first input file specification for each overlay file and the FORTRAN Library, FTNLIB, is linked to the overlay.

Note, too, that the output filename cannot be altered since it is referenced in a call statement as OVL2 from either the resident and/or OVL1.

F.6 ERROR PROCEDURES AND MESSAGES

Error handling during the creation of the overlay system is performed by LINK. At run time there are four DOS fatal errors which are concerned with the "CALL LINK" overlay facility. These are as follows:

Error Code		Explanation
F275	xxxxxx	An argument to the LINK subroutine is syntacticaly incorrect or too long.
F276	xxxxxx	Transfer address of the overlay file was not specified. For FORTRAN overlays, no main program was included in the overlay file.
F277	xxxxxx	Overlay file could not be brought into core because it would overlay the resident section. This is the only error condition which generates the F277 code.

xxxxxx is the additional information word which contains the address in the user code following the offending CALL LINK.

After an F275 error message (where the additional information is the address following the offending CALL LINK), identify the offending

CALL LINK and search for one of the following error conditions in the argument of CALL LINK:

a. syntactically incorrect file specification,

b. multiple file specifications,

c. illegal switch specifications, or

d. file specification longer than 25 decimal characters.

When the error is isolated, correct it. Reassemble or recompile the object module in which the error occurred. If the error occurred in the resident section, relink the entire overlay system. If the error occurred in an overlay file, then relink only the particular overlay file with the aid of the Core Library symbol table of the resident section and the FORTRAN Library.

After a F276 error message, determine the offending CALL LINK as for F275. If the overlay file is a FORTRAN program the error was probably caused by not including a main program in the overlay file. If the overlay file was composed of assembly language modules, then the error was caused by not specifying a transfer address. Follow relinking instructions described under F275.

After an F277 error message, determine the offending CALL LINK as for F275. This error results when the overlay would have overlaid part of the resident section. To correct the error condition, the offending overlay must be relinked with a top of (low limit of the resident -2). See relinking insructions under F275. (This error does not occur if the correct symbol table (.STB) file is being used since the symbol table file includes a pointer to the bottom of the resident section.)

F.7 ASSEMBLY LANGUAGE OVERLAYS

The creation of the overlay system (resident and overlay load modules) is the same for assembly language programs as for FORTRAN programs.

The assembly language calling sequence for the LINK subroutine is as follows (the standard FORTRAN calling sequence):

```
        .GLOBL  LINK
        R5=%5
        JSR R5,LINK
        BR .+4              ;RETURN ADDRESS
        .WORD NAME          ;POINTER TO ARGUMENT
         .
         .
         .
NAME:   .ASCIZ  /DEV:FILE/ ;ARGUMENT
                            ;ASCII CHARACTER STRING OF OVERLAY
        .EVEN               ;LOAD MODULE FILE SPECIFICATION &
                            ;TERMINATING NULL BYTE
```

The assembly language calling sequence to return to the resident program is as follows (the standard FORTRAN calling sequence):

```
.GLOBL RETURN
JSR R5,RETURN
BR .+2
 .
 .
 .
```

F.7.1 Global Declarations

In all assembly language programs that call LINK or RETURN, the symbols LINK and RETURN must be declared as globals.

```
.GLOBL LINK, RETURN      ;EXTERNAL REFERENCES
```

F.7.2 Stack Usage

Assembly language resident code should not reset the stack at execution time to be at a lower address than set by the Monitor at loading time. The Monitor considers the stack as starting at the address set at loading time and ending at the contents of register 6 and uses this information to first decide if the stack must be moved and then to move it below an incoming overlay.

The stack should not be addressed absolutely (e.g., pointers to the stack should not be stored on the stack itself).

Exit from an overlay should be made only through the main program of that overlay, otherwise the return addresses from any nested routines are lost on the stack. (See Section 6.5.1.)

While executing FORTRAN programs, the stack is only used to store the return addresses from nested subroutines. (The latest entry on the stack is the old contents of register 5 and the return address to the last subroutine call is in register 5.)

With these restrictions in mind, the stack can be used by the assembly language program.

F.7.3 Register Usage

Before exiting from the resident program to an overlay, general registers R0 through R5 are saved by the LINK subroutine. On reentering the resident program from an overlay, registers R0 through R5 are restored by the RETURN subroutine.

The register contents are not disturbed by LINK; therefore, arguments can be passed between overlays through the registers R0 - R4.

With these points in mind, the general registers can be used by the assembly language program.

F.7.4 COMMON Communication

COMMON communication between a FORTRAN main program and an assembly language subroutine is shown below. In the example, the variable I of blank COMMON is input through the keyboard. First the two low order bits then the five low order bits of I are cleared by the mask in variable M of labeled COMMON through the assembly language subroutine MASK.

Note that two words are allocated for integers in the assembly language subroutine MASK in order to be compatible with FORTRAN storage allocation.

The following is the FORTRAN main program:

```
      COMMON I,J
      COMMON /X/L,M
5     WRITE(6,1)
1     FORMAT(1H ,7H INPUT)
C     INPUT VARIABLE I OF BLANK COMMON
C     FROM KEYBOARD

      READ(6,2) I
2     FORMAT(I5)

C     PREPARE THE MASK FOR
C     THE TWO LOW ORDER BITS
      M=3
      CALL MASK
      WRITE(6,3) I
3     FORMAT(1H ,2HI=,I5)
C     PREPARE THE MASK FOR BITS
C     0 THRU 4 (31(10) = 37(8)).
      M=31
      CALL MASK
      WRITE(6,4) I
4     FORMAT(1H ,2HI=,I5)
      GO TO 5
      END
```

The assembly language subroutine MASK follows:

```
          .TITLE MASK
          .GLOBL MASK

;CLEAR THE 1ST POSITION OF BLANK
;COMMON BY THE MASK OF THE 2ND POSITION
;OF NAMED COMMON, X.
MASK:

          R5=%5
          .CSECT  .$$$$.
          S=.+0
          T=.+4
          .=.+10
          .CSECT X
          A=.+0
          B=.+4
          .=.+10
          .CSECT
          BTC   B,S
          RTS   R5
          .END
```

APPENDIX G

.ASECTS, .CSECTS, AND .PSECTS

G.1 PROGRAM SECTION DIRECTIVES

G.1.1 .PSECT Directive

Program sections are defined by the .PSECT directive, which is formatted as:

```
.PSECT [NAME] [,RO/RW] [,I/D] [,GBL/LCL] [,ABS/REL] [,CON/OVR] [,HGH/LOW]
```

The brackets ([]) are for purposes of illustrating optional parameters, and are not included in the parameter specifications. The slash (/) indicates that a choice is to be made between the parameters. The program section attribute parameters are summarized in Table G-1.

Table G-1

.PSECT Directive Parameters

Parameter	Default	Meaning
NAME	Blank	Program section name, in Radix-50 format, specified as one to six characters. If omitted, a comma must appear in the first parameters position.
RO/RW	RW	Program section access mode; RO=Read Only RW=Read/Write
I/D	I	Program section type; I=Instruction D=Data
GBL/LCL	LCL	The scope of the program section, as interpreted by LINK; GBL=Global LCL=Local
ABS/REL	REL	Defines relocation of the program section; ABS=Absolute (no relocation) REL=Relocatable (a relocation bias is required)

CON/OVR	OVR	Program section allocation; CON=Concatenated OVR=Overlaid
HGH/LOW	LOW	Program section memory type; HGH=High-speed LOW=Core (Note:HGH/LOW is not supported in the current DOS/BATCH release.)

The only parameter that is position-dependent is NAME. If it is omitted, a comma must be used in its place. For example,

```
.PSECT ,RO
```

This example shows a PSECT with a blank name and the Read Only access parameter. Defaults are used for the remaining parameters.

LINK interprets the .PSECT directive's parameters as follows:

RO/RW — Defines the type of access to the program section permitted which is; Read Only, or Read/Write.

I/D — Allows LINK to differentiate global symbols that are entry points (I) from global symbols that are data values (D).

GBL/LCL — Defines the scope of a program section. A global program section's scope crosses segment (overlay) boundaries; a local program section's scope is within a single segment. In single-segment programs, the GBL/LCL parameter is ignored.

ABS/REL — When ABS is specified, the program section is absolute. No relocation is necessary (i.e., the program section is assembled starting at absolute 0). When REL is specified, a relocation bias is calculated by LINK, and added to all references in the section.

CON/OVR — CON causes LINK to collect all allocation references to the program section from different modules and concatenate them to form the total allocation for the program section. OVR indicates that all allocation references to the program section overlay one another. Thus, the total allocation of the program section is determined by the largest request made by a module that references it.

HGH/LOW — In future releases of DOS/BATCH, the user may be able to specify the kind of memory used to store the .PSECT (high or low speed); Currently, this parameter is ignored.

Once the attributes of a named .PSECT are declared in a module, the MACRO Assembler assumes that this .PSECT's attributes hold for all subsequent declarations of the named .PSECT in the same module. Thus, the attributes may be declared once, and later .PSECT's with the same name will have the same attributes, when specified within the same module.

The Assembler provides for 255(10) program sections: One absolute section, one blank relocatable section, and 253(10) named relocatable sections are permitted. The .PSECT directive enables the user to:

1. Create his program (object module) in sections; and,

2. Share code and data.

For each program section specified or implied, the Assembler maintains the following information:

1. Section name;

2. Contents of the program counter;

3. Maximum program counter value encountered; and,

4. Section attributes, (the six .PSECT attributes).

G.1.2 Creating Program Sections

A given program section is defined completely upon its first reference. Thereafter, the section can be referenced by completely specifying the section attributes or by specifying the name only. For example, a section can be specified as:

```
        .PSECT    ALPHA,ABS,OVR
```

and later referenced as:

```
        .PSECT    ALPHA
```

By maintaining separate location counters for each section, the Assembler allows the user to write statements which are not physically contiguous but are loaded contiguously, as shown in the following example:

```
        .PSECT  SEC1,REL        ;START A RELOCATABLE SECTION NAMED
A:      .WORD   0               ;SEC1 ASSEMBLED AT RELOCATABLE 0,
B:      .WORD   0               ;RELOCATABLE 2 AND
C:      .WORD   0               ;RELOCATABLE 4,
ST:     CLR A                   ;ASSEMBLE CODE AT
        CLR B                   ;RELOCATABLE ADDRESSES
        CLR C                   ;6 THROUGH 21
        .PSECT  SECA,ABS        ;START AN ABSOLUTE SECTION NAMED SECA
        .=4                     ;ASSEMBLE CODE AT
        .WORD   .+2,HALT        ;ABSOLUTE 4 THROUGH 7,
        .PSECT  SEC1            ;RESUME THE RELOCATABLE SECTION
```

```
        INC A                   ;ASSEMBLE CODE AT
        BR ST                   ;RELOCATABLE 22 THROUGH 27
        .END
```

The first appearance of a .PSECT directive with a given name assumes the location counter is at relocatable or absolute zero. The scope of each directive extends until a directive beginning a different section is given. Further occurrences of a section name in a subsequent .PSECT statement resume assembling where the section previously ended.

```
        .PSECT  COM1,REL        ;DECLARE RELOCATABLE SECTION COM1
A:      .WORD   0               ;ASSEMBLED AT RELOCATABLE 0,
B:      .WORD   0               ;ASSEMBLED AT RELOCATABLE 2,
C:      .WORD   0               ;ASSEMBLED AT RELOCATABLE 4,
        .PSECT  COM2,REL        ;DECLARE RELOCATABLE SECTION COM2
X:      .WORD   0               ;ASSEMBLED AT RELOCATABLE 0
Y:      .WORD   0               ;ASSEMBLED AT RELOCATABLE 2,
        .PSECT  COM1            ;RETURN TO COM1
D:      .WORD   0               ;ASSEMBLED AT RELOCATABLE 6,
        .END
```

All labels in an absolute section are absolute; all labels in a relocatable section are relocatable. The location counter symbol, ".", is relocatable or absolute when referenced in a relocatable or absolute section, respectively. An undefined internal symbol is a global reference. It essentially has no attributes except global reference. Any labels appearing on a .PSECT (or .ASECT or .CSECT) statement are assigned the value of the location counter before the .PSECT (or other) directive takes effect. Thus, if the first statement of a program is:

```
A:      .PSECT  ALT,REL
```

then A is assigned to relocatable zero and is associated with the relocatable section ALT.

Since it is not known at assembly time where the program sections are to be loaded, all references between sections in a single assembly are translated by the Assembler to references relative to the base of that section. The Assembler provides LINK with the necessary information to resolve the linkage.

Note that this is not necessary when making a reference to an absolute section (the Assembler knows all load addresses of an absolute section).

In the following example, references to X and Y are translated into references relative to the base of the relocatable section SEN.

```
        .PSECT  ENT,ABS
        .=1000
A:      CLR     X               ;ASSEMBLED AS CLR BASE OF
                                ;RELOCATABLE SECTION + 10
        JMP     Y               ;ASSEMBLED AS JMP BASE OF
                                ;RELOCATABLE SECTION + 6
        .PSECT  SEN,REL
        MOV     R0,R1
        JMP A                   ;ASSEMBLED AS JMP 1000
Y:      HALT
```

```
X:      WORD    0
        .END
```

Code or Data Sharing

Named relocatable program sections with the attribute OVR operate as FORTRAN labeled COMMON; that is, sections of the same name with the attribute OVR from different assemblies are all loaded at the same location by LINK. All other program sections (those with the attribute CON) are concatenated.

Note that there is no conflict between internal symbolic names and program section names; that is, it is legal to use the same symbolic name for both purposes. In fact, considering FORTRAN again, this is necessary to accommodate the FORTRAN statement:

```
COMMON              /X/A,B,C,X
```

where the symbol X represents the base of this program section and also the fourth element of this program section.

Program section names should not duplicate .GLOBL names. In FORTRAN language, COMMON block names and SUBROUTINE names should not be the same.

G.2 .ASECT and .CSECT Directives

DOS/BATCH assembly language programs may use the .PSECT directive exclusively, as it affords all the capabilities of the .ASECT and .CSECT directives defined for other PDP-11 assemblers. The DOS/BATCH Macro Assembler will accept .ASECT and .CSECT but assembles them as if they were .PSECT's with the default attributes listed below. Also, compatibility exists between old object programs and the LINK, because LINK recognizes .ASECT and .CSECT directives that appear in such programs. LINK accepts these directives from such object programs, and assigns default values as shown in Table G-2.

Table G-2

Non-RSX-11D Program Section Defaults

Attribute	Default Value		
	.ASECT	.CSECT (named)	.CSECT
Name	ABS	name	Blank
Access	RW	RW	RW
Type	I	I	I
Scope	GBL	GBL	LCL
Relocation	ABS	REL	REL

Allocation	OVR	OVR	CON
Memory	LOW	LOW	LOW

The allowable syntactical forms of .ASECT and .CSECT are:

```
.ASECT
.CSECT
.CSECT  symbol
```

Note that

```
.CSECT  JIM
```

is identical to

```
.PSECT  JIM,GBL,OVR
```

APPENDIX H

LOAD MAP EXAMPLES

H.1 MAP LISTING

The Map has a header followed by segment descriptions.

H.1.1 Map Header

The header consists of the following display: (lower case entries are self-explanatory variables filled in at runtime)

```
FILE  file-name  MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON   date
AT  time  LINK VERSION  ver-number
```

H.1.2 Segment Descriptions

Segment descriptions have four subsections

```
                                          --     ----
1. Attributes and Statistics             !-Short  !
2. Control Section Allocation Synopsis-- Map      !-Long
3. File Contents                                  ! Map
4. Undefined References                           !
                                                 ----
```

Segment title lines appear as:

***SEG: segname

H.2 ATTRIBUTES AND STATISTICS

LINK prints out the following data on segments. Only those items which apply to the segment being described will appear on the map.

H.2.1 Read/Write Memory Limits. Displayed as:

R/W MEM LIMITS: start end length

The addresses define the virtual storage allocated to the segments R/W section. The end address is an inclusive address.

APPENDIX H

LOAD MAP EXAMPLES

H.1 MAP LISTING

The Map has a header followed by segment descriptions.

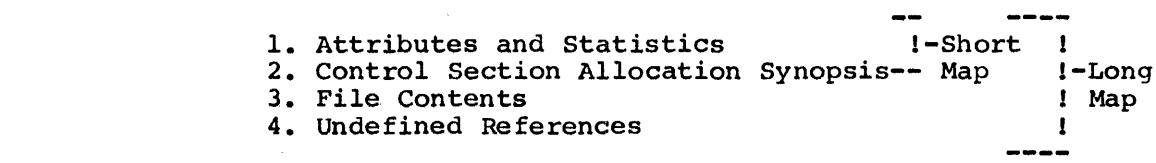
H.1.1 Map Header

The header consists of the following display: (lower case entries are self-explanatory variables filled in at runtime)

```
FILE  file-name  MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON   date
AT  time  LINK VERSION  ver-number
```

H.1.2 Segment Descriptions

Segment descriptions have four subsections



Segment title lines appear as:

***SEG: segname

H.2 ATTRIBUTES AND STATISTICS

LINK prints out the following data on segments. Only those items which apply to the segment being described will appear on the map.

H.2.1 Read/Write Memory Limits. Displayed as:

R/W MEM LIMITS: start end length

The addresses define the virtual storage allocated to the segments R/W section. The end address is an inclusive address.

H.2.2 Read-Only Memory Limits. Displayed as:

R-O MEM LIMITS: start end length

This entry can occur only for the root segment.

H.2.3 ODT Transfer Address. Displayed as:

ODT XFR ADDRESS: address

H.2.4 Program Transfer Address. Displayed as:

PRG XFR ADDRESS: address

H.2.5 Identification Displayed as:

IDENTIFICATION : name

The name is derived from the first non-blank .IDENT entry encounted during the processing of the segment's object files.

H.3 CONTROL SECTION ALLOCATION SYNOPSIS

The Control Section Allocation Synopsis lists all the p-sections comprising the segment. The sections are listed in alphabetical order. In segments other than the root, the read-only attribute is not honored. LINK processes R/W sections, then R-O sections, but declares any R-O Section R/W.

For each section encountered in building the segment LINK displays:

name: start end length.

Blank control sections are given the name
. BLK.

and collated lowest in the sort sequence. Absolute control sections are given the name . ABS.

Note that neither of these names is a legal assembler section name and thus cannot be user-generated.

H.4 FILE CONTENTS

This section of the map identifies by file every p-section contributed to the segment. And for each p-section it lists every global symbol defined in the section. The section begins with the display line:

***TITLE: t-name IDENT: i-name FILE: file-name

Where:

file-name = The name of the file specified in the ODL description of the Task.

t-name = The name of the first non-blank .TITLE entry encountered in module. If a file contains n modules then a complete FILE sections is displayed for each module.

i-name = The name of the first non-blank .IDENT entry encountered in this file.

Following the TITLE line, each section in the file, displayed in the order they were encountered, appear as:

name; start end length

Following a section identifier is a list of global symbols in the form:

name address

If the address is relocatable, -R is appended to the address.

If an undefined reference is encountered, the following line is displayed.

>>>>>>>>>>>>UNDEFINED REFERENCE: name

These undefined references will appear interspersed with the global symbol definitions.

UNDEFINED REFERENCES.

This section is headed by:

****
UNDEFINED REFERENCES

This section separator is followed by an alphabetical list of all undefined symbols found in the segment. This list contains all undefined references that appeared in the FILES section.

At the end of this section the number of octal bytes used by LINK to complete the load, and the number of octal bytes remaining unused by LINK appear as:

SPACE USED xxxxxx SPACE FREE xxxxxx

The load map shown below was generated by the following LINK command string:

CILUS3,LP:/LG/CR<CILUS.ODL/MP/E

```
FILE CILUS3.LDA MEMORY ALLOCATION MAP
THIS ALLOCATION WAS DONE ON 27-JUN-73
AT 03:07:29 LINK VERSION V01-02


*** ROOT SEG: CILUS


R/W MEM  LIMITS: 142770 152651 007662
IDENTIFICATION : 71
PRG XFR ADDRESS: 142770


PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 142770 152265 007276
<COMMON>: 152266 152423 000136
<. ABS.>: 000000 000000 000000


*** TITLE: CILUS   IDENT: 71      FILE: CUSP  .LIB

<. ABS.>: 000000 000000 000000

    CLSICF ******-X  COMDEC ******-X  EDCOM  ******-X  EDIT   ******-X
    HOKBOT ******-X  LOAD   ******-X  P1     000001    P2     000000
    S1     000201    S2     000203

<. BLK.>: 142770 147701 004712

    CIL    143362-R  CILBDY 143406-R  CILBLP 147634-R  CILEND 147664-R
    CILFG  144036-R  CILFLB 144320-R  CILINE 147626-R  CIL1   143316-R
    CITMP  143470-R  CITMP1 143324-R  CITPFB 144264-R  CITPLK 144060-R
    CLSRLS 147174-R  CMDBUF 144636-R  COMBUF 144654-R  COMDBK 144212-R
    COMDLK 143720-R  COMIN  144466-R  COMIS  143346-R  COMOS  143354-R
    COPYDS 145336-R  CRLF   145142-R  CSIIS  143332-R  CSIOS  143340-R
    DELFIL 147404-R  DOSBS  147444-R  DOSFB  147442-R  DOSNB  147446-R
    ENDMOD 145656-R  FORPRP 147454-R  GETBUF 145416-R  GIVBUF 145570-R
    GNFBLN 146312-R  GNLBKS 146670-R  GN64BL 146714-R  IDSIZE 147652-R
    INSMOD 146014-R  LASCMD 146616-R  LCLFG  144006-R  LCLFL  144230-R
    LCLOT  143444-R  LCLTMP 145164-R  LIST   143432-R  MFIRCR 146224-R
    MNEXCR 146174-R  MPRECR 146146-R  NCICIL 147650-R  NULALL 146760-R
    N64NEC 147656-R  N64RES 147654-R  N64256 147662-R  N6464  147660-R
    OCILBS 147646-R  OMSGM  145156-R  PNAME  146232-R  PRCOMD 143514-R
    PRICT  143564-R  PRINTA 145040-R  PRINTF 145012-R  PRNTA  145030-R
    PRNTF  145002-R  RENMFL 147302-R  SAVRG  145212-R  SCCOMD 143540-R
    SECCI  143610-R  SEGBLP 147674-R  SEGLIN 147666-R  SEGNO  147700-R
    SYSLBS 143466-R  TERMFR 144374-R  TIME   147640-R  TLCLOT 143456-R
    TMPFL  144246-R  TTYOUT 145042-R  WRNING 144350-R

<COMMON>: 152266 152423 000136


*** TITLE: CILIO   IDENT: 70      FILE: CUSP  .LIB

<. ABS.>: 000000 000000 000000

    GETBUF ******-X  GIVBUF ******-X  SAVRG  ******-X  WRNING ******-X

<. BLK.>: 147702 151605 001704

    BLDBDY 151452-R  CLEAR  151542-R  CLOSE  151130-R  CLSICF 151372-R
    CLSOCF 151372-R  FLMCI  150220-R  INICF  151230-R  INOCF  151234-R
    OPEN   150510-R  PLMLCI 150374-R  RDCI   150620-R  RELOCI 151160-R
    RFBLCF 147702-R  TRANIN 151034-R  TRANOT 151042-R  WFBLCF 150116-R
    WTCI   150626-R
```

```
<COMMON>: 152266 152423 000136

*** TITLE: $AUTO    IDENT: 03       FILE: CUSP  .LIB
<. ABS.>: 000000 000000 000000
    $MARKS ******-X  $RDSEG ******-X  $SAVAL ******-X  $$WAIT ******-X
<. BLK.>: 151606 151715 000110
    $AUTO  151606-R

*** TITLE: $MARKS   IDENT: 02       FILE: CUSP  .LIB
<. BLK.>: 151716 152007 000072
    $MARKS 151716-R

*** TITLE: $RDSEG   IDENT: 03       FILE: CUSP  .LIB
<. ABS.>: 000000 000000 050000
    TRA.   ******-X
>>>>>>>>>>>> UNDEFINED REFERENCE: TRA.

<. BLK.>: 152010 152265 000256
    $RDSEG 152036-R  $SAVAL 152140-R  $$WAIT 152202-R


************
UNDEFINED REFERENCES:
    TRA.  **


*** SEG: COMDEC

R/W MEM  LIMITS: 152652 156735 004064

PROGRAM SECTION ALLOCATION SYNOPSIS:
<. BLK.>: 152652 156735 004064

*** TITLE: COMDEC   IDENT: 72       FILE: CUSP  .LIB
<. ABS.>: 000200 000000 000000
```

```
    CILFG  *******-X  CILFLB *******-X  CIL1   *******-X  CITMP  *******-X
    CITPLK *******-X  CMDBUF *******-X  COMBUF *******-X  COMDBK *******-X
    COMDLK *******-X  COMIN  *******-X  COPYDS *******-X  CRLF   *******-X
    CSIIS  *******-X  DELFIL *******-X  DOSBS  *******-X  DOSFB  *******-X
    DOSNB  *******-X  FORPRP *******-X  GETBUF *******-X  LCLFG  *******-X
    LCLFL  *******-X  LCLOT  *******-X  LCLTMP *******-X  LIST   *******-X
    NULALL *******-X  OCILBS *******-X  OMSGM  *******-X  PRCOMD *******-X
    PRNTA  *******-X  PRNTF  *******-X  RENMFL *******-X  SAVRG  *******-X
    SCCOMD *******-X  SYSLBS *******-X  TERMER *******-X  TLCLOT *******-X
    TMPFI  *******-X  WRNING *******-X

<. BLK.>: 152652 156735 004064

    COMDEC 152652-R

<COMMON>: 152266 152423 000136




*** SEG: EDCOM

R/W MEM  LIMITS: 152652 153677 001026

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 152652 153677 001026

*** TITLE: EDCOM   IDENT: 71      FILE: CUSP  .LIB

<. ABS.>: 000000 000000 000000

    COMBUF *******-X  COMIN  *******-X  COMIS  *******-X  COMOS  *******-X
    ENDMOD *******-X  INSMOD *******-X  PNAME  *******-X  PRINTF *******-X
    PRNTA  *******-X  PRNTF  *******-X  P1     *******-X  P2     *******-X
    SAVRG  *******-X  S1     *******-X  S2     *******-X

<. BLK.>: 152652 153677 001026

    EDCOM  152660-R

<COMMON>: 152266 152423 000136




*** SEG: EDIT

R/W MEM  LIMITS: 152652 157457 004606

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 152652 157457 004606
```

```
*** TITLE: EDIT     IDENT: 70       FILE: CUSP   .LIB

<. ABS.>: 000000 000000 000000

     BLDBDY *******-X  CILBLP *******-X  CILEND *******-X  CILINE *******-X
     CITMP  *******-X  CITMP1 *******-X  CITPFB *******-X  CLEAR  *******-X
     CLOSE  *******-X  CLSICF *******-X  CRLF   *******-X  CSIOS  *******-X
     DELFIL *******-X  FLMCI  *******-X  FNDMOD *******-X  GETBUF *******-X
     GIVBUF *******-X  GNFBLN *******-X  GNLBKS *******-X  IDSIZE *******-X
     INICF  *******-X  INSMOD *******-X  LASCMD *******-X  LCLOT  *******-X
     LIST   *******-X  NFIRCR *******-X  MNEXCR *******-X  MPRECR *******-X
     NCICIL *******-X  N64NEC *******-X  OCILBS *******-X  PLMLCI *******-X
     PNAME  *******-X  PRCOMD *******-X  PRICI  *******-X  PRINTA *******-X
     P1     *******-X  RFBLCF *******-X  SAVRG  *******-X  SCCOMD *******-X
     SECCI  *******-X  SEGBLP *******-X  SEGLIN *******-X  SEGNO  *******-X
     S1     *******-X  S2     *******-X  TERMER *******-X  TIME   *******-X
     TLCLOT *******-X  WRNING *******-X

<. BLK.>: 152652 157457 004606

     EDIT   152652-R

<COMMON>: 152266 152423 000136




*** SEG: LOAD

R/W MEM  LIMITS: 152652 156651 004000

PROGRAM SECTION ALLOCATION SYNOPSIS:

<. BLK.>: 152652 156651 004000

*** TITLE: LOAD     IDENT: 70       FILE: CUSP   .LIB

<. ABS.>: 000000 000000 000000

     BLDBDY *******-X  CIL    *******-X  CILBDY *******-X  CILEND *******-X
     CILINE *******-X  CIL1   *******-X  CLEAR  *******-X  CLOSE  *******-X
     CLSICF *******-X  CLSRLS *******-X  FLMCI  *******-X  GIVBUF *******-X
     GNFBLN *******-X  GNLBKS *******-X  GN64BL *******-X  IDSIZE *******-X
     INICF  *******-X  LASCMD *******-X  LCLOT  *******-X  NCICIL *******-X
     N64NEC *******-X  N64RES *******-X  N64256 *******-X  N6464  *******-X
     OCILBS *******-X  PLMLCI *******-X  PRCOMD *******-X  PRICI  *******-X
     RENMFL *******-X  RFBLCF *******-X  SAVRG  *******-X  TERMER *******-X
     TIME   *******-X  WFBLCF *******-X

<. BLK.>: 152652 154557 001730

     LOAD   152652-R  RCMDLN 153170-R  SAVHB  154346-R

<COMMON>: 152266 152423 000136
```

```
*** TITLE: HOKBOT  IDENT: 70       FILE: CUSP  .LIB

<. ABS.>: 000000 000000 000000

    CTL     *******-X  INICF  *******-X  MTSTRT *******-X  OCILBS *******-X
    RCMDLN *******-X  RCSTRT *******-X  RFSTRT *******-X  RKSTRT *******-X
    RPSTRT *******-X  SAVRB  *******-X  SAVRG  *******-X  TCSTRT *******-X
    TERMFD *******-X  TRANOT *******-X

<. BLK.>: 154560 155251 000472

    HOKBOT 154560-R

<COMMON>: 152266 152423 000136


*** TITLE: RPBOOT                  FILE: CUSP  .LIB

<. BLK.>: 155252 155451 000200

    RPSTRT 155252-R

*** TITLE: RFBOOT                  FILE: CUSP  .LIB

<. BLK.>: 155452 155651 000200

    RFSTRT 155452-R

*** TITLE: RKBOOT                  FILE: CUSP  .LIB

<. BLK.>: 155652 156051 000200

    RKSTRT 155652-R

*** TITLE: RCBOOT                  FILE: CUSP  .LIB

<. BLK.>: 156052 156251 000200

    RCSTRT 156052-R

*** TITLE: TCBOOT                  FILE: CUSP  .LIB

<. BLK.>: 156252 156451 000200

    TCSTRT 156252-R

*** TITLE: MTBOOT                  FILE: CUSP  .LIB

<. BLK.>: 156452 156651 000200

    MTSTRT 156452-R

SPACE USED 007240, SPACE FREE 050700
```

GLOBAL CROSS REFERENCE TABLE

SYMBOL	MODULE REFERENCES-----			
BLDBDY	CTLIO #	EDIT	LOAD	
CIL	CTLUS #	HOKBOT	LOAD	
CILBDY	CILUS #	LOAD		
CILBLP	CILUS #	EDIT		
CILEND	CILUS #	EDIT	LOAD	
CILFG	CTLUS #	COMDEC		
CILFLB	CTLUS #	COMDEC		
CILINE	CTLUS #	EDIT	LOAD	
CIL1	CILUS #	COMDEC	LOAD	
CITMP	CILUS #	COMDEC	EDIT	
CITMP1	CILUS #	EDIT		
CITPFB	CILUS #	EDIT		
CITPLK	CILUS #	COMDEC		
CLEAR	CTLIO #	EDIT	LOAD	
CLOSF	CTLIO #	EDIT	LOAD	
CLSICF	CTLIO #	CILUS	EDIT	LOAD
CLSOCF	CILIO #			
CLSRLS	CILUS #	LOAD		
CMDBUF	CILUS #	COMDEC		
COMBUF	CILUS #	COMDEC	EDCOM	
COMDBK	CILUS #	COMDEC		
COMDEC	CILUS	COMDEC#		
COMDLK	CILUS #	COMDEC		
COMIN	CILUS #	COMDEC	EDCOM	
COMIS	CILUS #	EDCOM		
COMOS	CTLUS #	EDCOM		
COPYDS	CTLUS #	COMDEC		
CRLF	CTLUS #	COMDEC	EDIT	
CSTIS	CTLUS #	COMDEC		
CSTOS	CILUS #	EDIT		
DELFIL	CILUS #	COMDEC	EDIT	
DOSBS	CILUS #	COMDEC		
DOSFB	CILUS #	COMDEC		
DOSNB	CILUS #	COMDEC		
EDCOM	CTLUS	EDCOM #		
EDIT	CILUS	EDIT #		
ELMCI	CTLIO #	EDIT	LOAD	
ENDMOD	CTLUS #	EDCOM	EDIT	
FORPRP	CILUS #	COMDEC		
GETBUF	CTLIO	CILUS #	COMDEC	EDIT
GIVBUF	CILIO	CILUS #	EDIT	LOAD
GNFBLN	CILUS #	EDIT	LOAD	
GNLBKS	CILUS #	EDIT	LOAD	
GN64BL	CTLUS #	LOAD		
HOKBOT	CILUS	HOKBOT#		
IDSIZE	CILUS #	EDIT	LOAD	
INTCF	CTLIO #	EDIT	HOKBOT	LOAD
INOCF	CILIO #			
INSMOD	CILUS #	EDCOM	EDIT	
LASCMD	CTLUS #	EDIT	LOAD	
LCLFG	CTLUS #	COMDEC		
LCLFL	CTLUS #	COMDEC		
LCLOT	CTLUS #	COMDEC	EDIT	LOAD
LCLTMP	CILUS #	COMDEC		
LIST	CILUS #	COMDEC	EDIT	
LOAD	CILUS	LOAD #		
MFIRCR	CTLUS #	EDIT		

```
MNEXCR    CILUS #  EDIT
MPRECR    CILUS #  EDIT
MTSTRT    HOKBOT   MTBOOT#
NCTCTL    CILUS #  EDIT      LOAD
NULALL    CILUS #  COMDEC
N64NEC    CILUS #  EDIT      LOAD
N64RES    CILUS #  LOAD
N64256    CILUS #  LOAD
N6464     CILUS #  LOAD
OCTLRS    CILUS #  COMDEC    EDIT      HOKBOT    LOAD
OMSGM     CILUS #  COMDEC
OPEN      CILIO #
PLMLCT    CILIO #  EDIT      LOAD
PNAME     CILUS #  EDCOM     EDIT
PRCOMD    CILUS #  COMDEC    EDIT      LOAD
PRICT     CILUS #  EDIT      LOAD
PRINTA    CILUS #  EDIT
PRINTF    CILUS #  EDCOM
PRNTA     CILUS #  COMDEC    EDCOM
PRNTF     CILUS #  COMDEC    EDCOM
P1        CILUS #  EDCOM     EDIT
P2        CILUS #  EDCOM
RCMDLN    HOKBOT   LOAD  #
RCSTRT    HOKBOT   RCBOOT#
RDCI      CILIO #
RELOCI    CILIO #
RENMFL    CILUS #  COMDEC    LOAD
RFBLCF    CILIO #  EDIT      LOAD
RFSTRT    HOKBOT   RFBOOT#
RKSTRT    HOKBOT   RKBOOT#
RPSTRT    HOKBOT   RPBOOT#
SAVHR     HOKBOT   LOAD  #
SAVRG     CILIO    CILUS #  COMDEC    EDCOM     EDIT      HOKBOT    LOAD
SCCOMD    CILUS #  COMDEC    EDIT
SECCT     CILUS #  EDIT
SEGBLP    CILUS #  EDIT
SEGLIN    CILUS #  EDIT
SEGNO     CILUS #  EDIT
SYSLRS    CILUS #  COMDEC
S1        CILUS #  EDCOM     EDIT
S2        CILUS #  EDCOM     EDIT
TCSTRT    HOKBOT   TCBOOT#
TERMFR    CILUS #  COMDEC    EDIT      HOKBOT    LOAD
TIME      CILUS #  EDIT      LOAD
TLCLOT    CILUS #  COMDEC    EDIT
TMPFL     CILUS #  COMDEC
TRANIN    CILIO #
TRANOT    CILIO #  HOKBOT
TRA.      $RDSEG
TTYOUT    CILUS #
WFBLCF    CILIO #  LOAD
WRNING    CILIO    CILUS #  COMDEC    EDIT
WTCI      CILIO #
$AUTO     $AUTO #
$MARKS    $AUTO    $MARKS#
$RDSEG    $AUTO    $RDSEG#
$SAVAL    $AUTO    $RDSEG#
$$WAIT    $AUTO    $RDSEG#
```

HOW TO OBTAIN SOFTWARE INFORMATION

Announcements for new and revised software, as well as programming notes, software problems, and documentation corrections, are published by Software Information Service in the following newsletters.

DIGITAL Software News for the PDP-8 and PDP-12
DIGITAL Software News for the PDP-11
DIGITAL Software News for 18-bit Computers

These newsletters contain information applicable to software available from DIGITAL'S Software Distribution Center. Articles in DIGITAL Software News update the cumulative Software Performance Summary which is included in each basic kit of system software for new computers. To assure that the monthly DIGITAL Software News is sent to the appropriate software contact at your installation, please check with the Software Specialist or Sales Engineer at your nearest DIGITAL office.

Questions or problems concerning DIGITAL'S software should be reported to the Software Specialist. If no Software Specialist is available, please send a Software Performance Report form with details of the problems to:

Digital Equipment Corporation
Software Information Service
Software Engineering and Services
Maynard, Massachusetts 01754

These forms, which are provided in the software kit, should be fully completed and accompanied by terminal output as well as listings or tapes of the user program to facilitate a complete investigation. An answer will be sent to the individual, and appropriate topics of general interest will be printed in the newsletter.

Orders for new and revised software manuals, additional Software Performance Report forms, and software price lists should be directed to the nearest DIGITAL field office or representative. USA customers may order directly from the Software Distribution Center in Maynard. When ordering, include the code number and a brief description of the software requested.

Digital Equipment Computer Users Society (DECUS) maintains a user library and publishes a catalog of programs as well as the DECUSCOPE magazine for its members and non-members who request it. For further information, please write to:

Digital Equipment Corporation
DECUS
Software Engineering and Services
Maynard, Massachusetts 01754

DOS/BATCH Linker (LINK)
Programmer's Manual
DEC-11-ULKAA-A-D

READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback--your critical evaluation of this document.

Did you find errors in this document? If so, please specify by page.

_____

_____

_____

_____

_____

How can this document be improved?

_____

_____

_____

_____

_____

How does this document compare with other technical documents you have read?

_____

_____

_____

_____

_____

Job Title_____Date:_____

Name:_____Organization:_____

Street:_____Department:_____

City:_____State:_____Zip or Country_____

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

)ESS REPLY MAIL
)STAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

e will be paid by:

digital

Digital Equipment Corporation
Software Information Service
Software Engineering and Services
Maynard, Massachusetts 01754

H.2.2 Read-Only Memory Limits. Displayed as:

R-O MEM LIMITS: start end length

This entry can occur only for the root segment.

H.2.3 ODT Transfer Address. Displayed as:

ODT XFR ADDRESS: address

H.2.4 Program Transfer Address. Displayed as:

PRG XFR ADDRESS: address

H.2.5 Identification Displayed as:

IDENTIFICATION : name

The name is derived from the first non-blank .IDENT entry encounted during the processing of the segment's object files.

H.3 CONTROL SECTION ALLOCATION SYNOPSIS

The Control Section Allocation Synopsis lists all the p-sections comprising the segment. The sections are listed in alphabetical order. In segments other than the root, the read-only attribute is not honored. LINK processes R/W sections, then R-O sections, but declares any R-O Section R/W.

For each section encountered in building the segment LINK displays:

name: start end length.

Blank control sections are given the name
. BLK.

and collated lowest in the sort sequence. Absolute control sections are given the name . ABS.

Note that neither of these names is a legal assembler section name and thus cannot be user-generated.

H.4 FILE CONTENTS

This section of the map identifies by file every p-section contributed to the segment. And for each p-section it lists every global symbol defined in the section. The section begins with the display line:

***TITLE: t-name IDENT: i-name FILE: file-name