PART 9

THE DOS/BATCH LINKER

LINK

PART 9

CHAPTER 1

INTRODUCTION TO THE LINKER

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 subsequently loaded (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 need 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 accidentally 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, only global symbols need be unique, 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 Appendix J-3.1).

The top address assumed by LINK is

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

PART 9

CHAPTER 2

INPUT AND OUTPUT

2.1 INPUT MODULES

Input to LINK consists of one or more object modules, which can be output from the DOS assembler, FORTRAN compiler, or other system programs.

On an object module's first pass through the linker, LINK gathers 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 the second pass, LINK 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

LINK's main function is to produce a load module, which 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 Chapter 9-9).

LINK can also produce a contiguous format file (see Chapter 9-9); 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 9-4).

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 Appendix J-3.1). If the top is very low, LINK will not allow the COMD to be loaded below address Ø; it will load it up from Ø.

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 a .END A, and the fourth a .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. A synopsis of program section arrangement follows, describing the placement of each program section relative to the other. Then there is a section of the map for each object 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.

Modules are loaded in alphabetical order in memory.

Chapter 9-11 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 output to the device.

PART 9

CHAPTER 3

OPERATING PROCEDURES

3.1 LOADING

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

```
$RUN LINK
```

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 # symbol printed by LINK. The format of the command string adheres to the requirements of the DOS Command String Interpreter (CSI), as explained in Part 3 of this handbook, DOS/BATCH Monitor.

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

```
#load module, load map, symbol table < object modules
```

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

```
dev:filnam.ext[uic]/sl: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 terminal 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 retype 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 up 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 Appendix J-3.10 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. This line causes subsequent 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 up to five nested levels of indirect files. This should be adequate for most applications, but can be changed, if desired, through an assembly option. (See Appendix J-3.)

The use of indirect command files reduces typing repetitive commands at the keyboard, and provides for 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 9-4. 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.

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 ABC 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.

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. (Any module that makes reference to another module or entry point must appear before that referenced item in the library.)

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

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

```
#TASKØ1.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, TASKØ1.LDA would be put on the disk, and a load map would go to the line printer.

As described in Appendix J-3.6 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 means 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 Part 3-5 of this handbook, DOS/BATCH Monitor, 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 nonresident. 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. Input file look-up occurs in the following order, where #,# is the current UIC.

1. FILNAME.OBJ[#,#]
2. FILNAME[1,1]
3. FILNAME.OBJ[1,1]
4. FILNAME[#,#]

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 Chapter 9-11 for an example.

3.4 SAMPLE LINKS

3.4.1 FORTRAN

For this example assume that the user is logged in under user identification code (UIC) of [2ØØ,2ØØ]. 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 as follows:

```
#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

For this example assume that the user is logged in under UIC [2ØØ,2ØØ]. 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 as follows:

```
#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

For this example assume that the user is logged in under UIC [2ØØ,2ØØ]. He has an ODL file (overlay description language -- see Chapter 9-4) 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 as follows:

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

When using an ODL file no other input file may be specified.

3.5 PROGRAMMING NOTES

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.

PART 9

CHAPTER 4

OVERLAYS

An overlay capability is very important in computer systems such as DOS/BATCH where the size of a program is apt to be larger than the amount of 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 normal link 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 that may overlay each other.

Figure 9-1 illustrates a typical overlay structure. A is the root segment and B, C, D, E, and F are overlay segments.

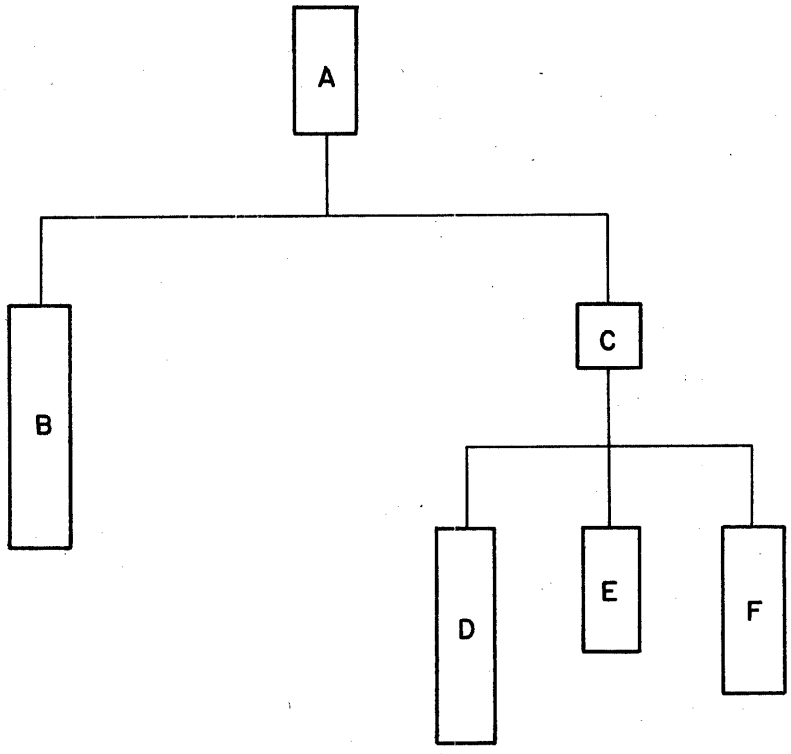


Figure 9-1

Overlay File 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 9-1, 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 path loading, and it 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.

4.1 TERMINOLOGY

AUTOLOAD - The process of automatically loading an overlay segment and subsequently transferring 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 GLOBL symbol defined in a source representation of a program and subsequently accessible to independently translated modules 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 overlay request.

4.2 LOADING OVERLAYS

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 that 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. Upon successful loading, the calling routine may then call entry points in the named segment via normal subroutine or transfer of control instructions. 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 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.

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.[1] Autoload allows a program to be separated into segments without reprogramming, while manual-load requires explicit calls to load overlay segments. Unlike manual-load, autoload calls perform path loading.

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.

4.2.1 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.

---

[1]However, condition codes are not passed on an autoload call.

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

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

where

strnam — is a 1- to 6-character ASCII module 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 Ø for a synchronous load, or to 1 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 Ø, no errors have occurred in the CALL LOAD. If error is nonzero, 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
```

Use JSR R5 for xxxMOD of OTS; JSR PC for xxxPC of OTS.

4.2.2 AUTOLOAD

During program creation from ODL, LINK records all autoload 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 9-2.

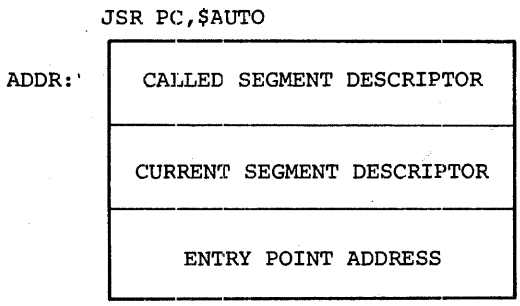


Figure 9-2

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 that 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 overlaid 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 technique similar to that described in Section 9-6.3.

For examples of autoload and manual load usage see Section 9-4.7 and Section 9-4.8.

4.3 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 Appendix J-3.8). 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 ODL comprises a number of directives that describe the overlay structure. In order to build tree-structured programs, LINK must interpret and carry out the directives provided by the 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 that LINK attempts to resolve during the linking process.

ODL consists of directives that specify a function and operands that 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 that are built into the overlays.

ODL provides the following:

- 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

.END

- Four operators:

-	concatenation
,	overlay
( )	overlay control point
*	autoload

The user specifies ODL operations via directives and operators.

The directives have the following general format:

```
label:   .direc   [oprnds]
```

where

label	is an alphanumeric label.
direc	is the directive name.
oprnd	is one or more optional operands.

4.3.1 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 that 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, consequently 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). Segments that occupy the same memory are not on a path, but they 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 9-3 are implied by enclosing operands in parentheses.

* The autoload operator (asterisk). See Section 9-4.4

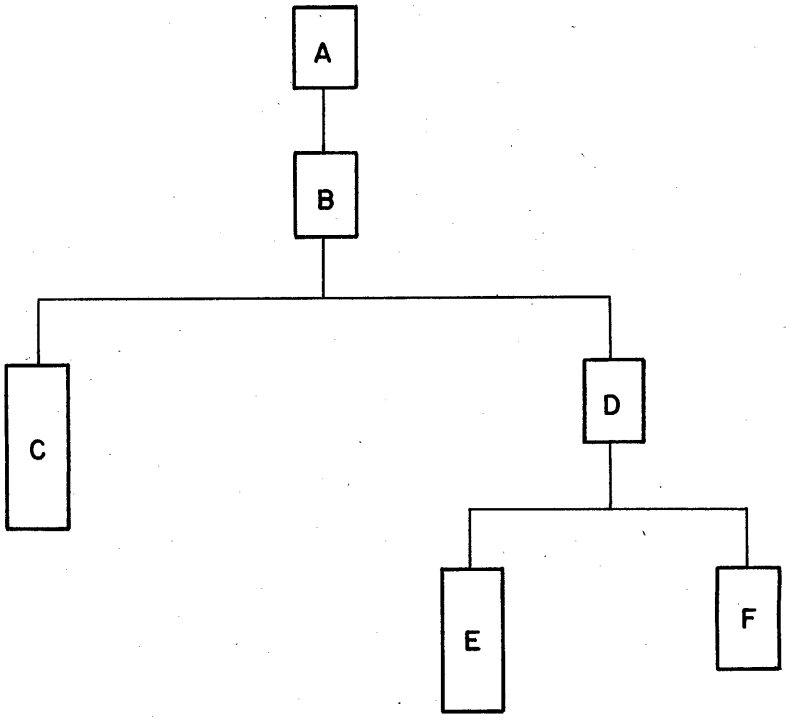


Figure 9-3

Sample Tree Structure

Figure 9-3 consists of six object files to be linked together by LINK. The input required by LINK to construct the program of Figure 9-3 should be specified. To sinplify the explanation, establish the structures below the branch point, X, as the leftbranch and rightbranch. Now Figure 9-3 can be described 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.)

The leftbranch can be represented 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. The ODL specification of Figure 9-3 is

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

The ODL syntax can now be completed with the .NAME, .FCTR, .PSECT, and .END directives and the * operator.

4.3.2 The .NAME Directive

.NAME declares 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; therefore this directive may be used to explicitly declare a segment name.

Directive syntax:

```
[label:] .NAME sname
```

where

.NAME is the directive name, and

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 9-3, the user wanted to name the root segment JIM, the directives

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

would create a root segment with the name JIM.

4.3.3 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 used earlier in presenting the development of the .ROOT directive used to describe the overlay structure of Figure 9-3. Recall that a factoring occurred by using the terms leftbranch and rightbranch. Using .FCTR this becomes a capability of the ODL itself.

The ODL string

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

describes the same overlay structure to LINK 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.

4.3.4 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 is is defined (local reference) or in the branch on its path closest to the root (global reference). For example, if in Figure 9-4

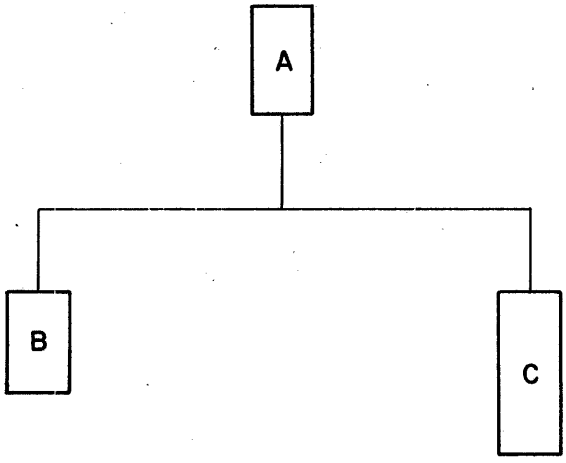


Figure 9-4

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 that may or may not coincide

with the programmer's wishes. The .PSECT directive permits 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 explicitly 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).[1] 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 this directive is a FORTRAN common area placed close to the root segment so that a number of branch segments that are not on a common path may share and communicate via this 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, and
(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. 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. Currently all PSECT's are I.
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

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

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 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 (MOS memory) and LOW means core.

NOTE

The HGH/LOW attribute is currently ignored by LINK.

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

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

4.3.5 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.

4.4 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 * occurs on an open parenthesis, every operand within the parenthesis and its matching close parenthesis will have the autoload attribute.

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

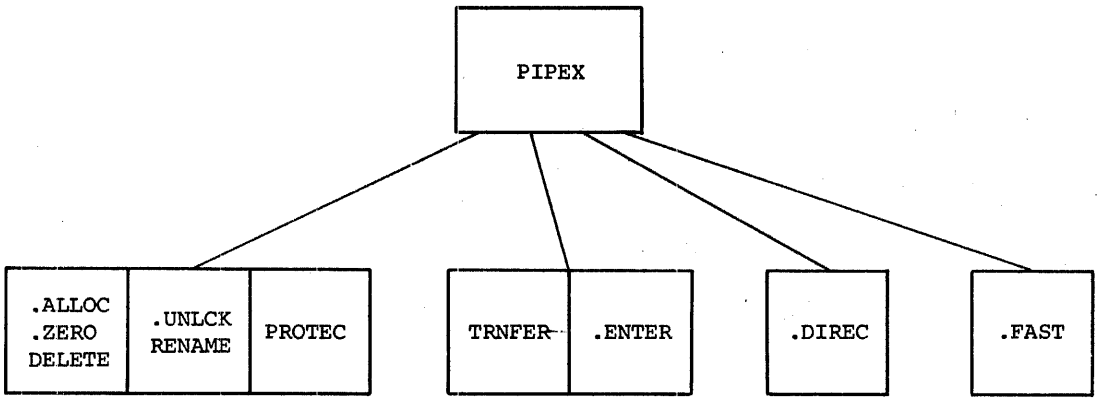
1. For section names the section is made autoload.
2. For the name in a .NAME directive all the components in the segment to which the name applies are made autoload.
3. For name labeling a .FCTR statement, * (asterisk) applies to the first irreducible component of the factor. If the entire factor list is enclosed in parenthesis, every file in the factor is made autoload.
4. For a Filename all components of the file are made autoload.

4.5 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 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.

4.6 EXAMPLES OF OVERLAID PROGRAM BUILDING USING LINK

Given the following tree structure of the desired overlay:



a file can be created to contain the following ODL task description.

```
          .NAME    PIP
          .ROOT    PIP-M1-F1
M1:       .FCTR    PIP.LIB/IN:PIPEX/EX:.ALLOC:.ENTER:.DIREC:
RENAME:.UNLOC:PROTEC:DELETE:.ZERO:TRNFER:.FAST
F1:       .FCTR    *(01,02,03,04)
01:       .FCTR    PIP.LIB/IN:.ALLOC:.ZERO:DELETE:.UNLOC:RENAME:PROTEC
02:       .FCTR    PIP.LIB/IN:TRNFER:.ENTER
03:       .FCTR    PIP.LIB/IN:.DIREC
04:       .FCTR    PIP.LIB/IN:.FAST
          .END
```

The following LINK command string builds the overlaid PIP module.

```
#PIP,LP:<PIP.ODL/MP/E
```

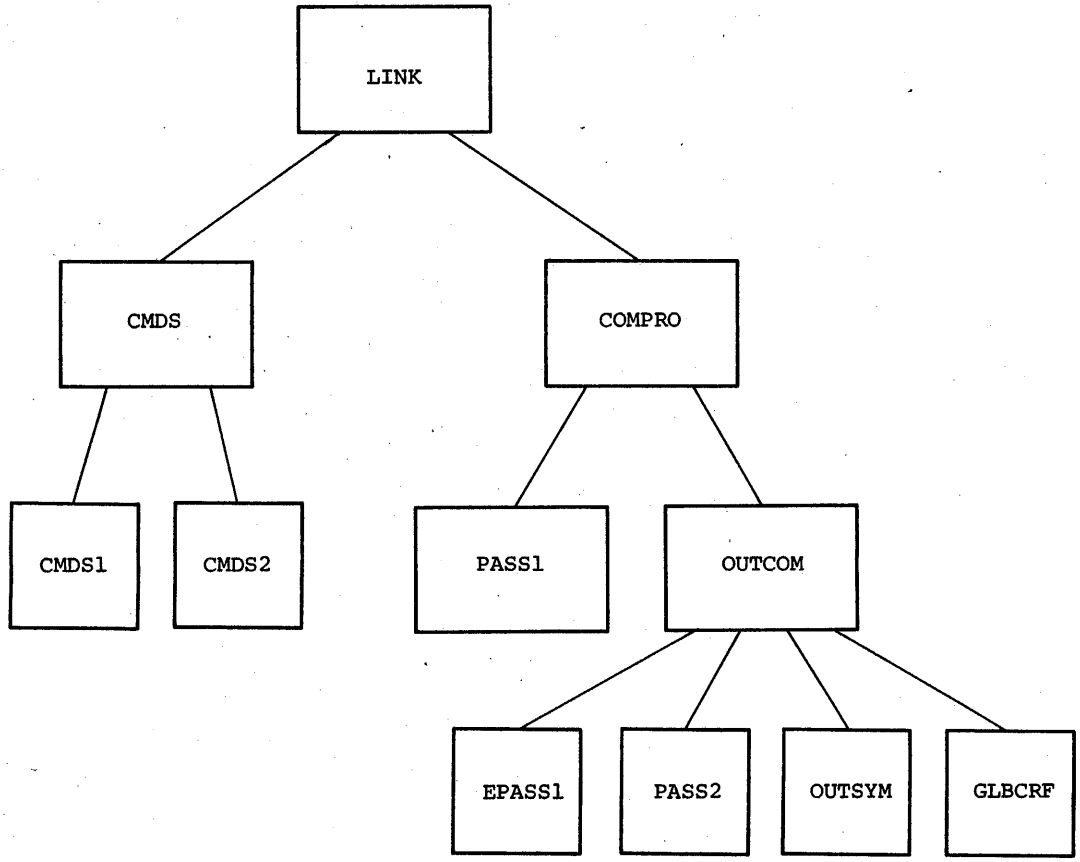
The following command builds a nonoverlaid version of the same program, using the Librarian.

```
#PIP,LP:<PIP.LIB/IN:PIPEX/E
```

the library listing:

```
LIBR  V05A

      PIP    .LIB       04-FEB-74  11:46:55

      SEQ.     NAME   VERSION

      00001    PIPEX    0809B
      00002    TRNFER   15
      00003    GLSUBS   009
      00004    .ALLOC   07B
      00005    .ENTER   06F
      00006    .ZERO    07
      00007    RENAME   08
      00008    .UNLOC   06F
      00009    .FAST    09
      00010    PROTEC   06F
      00011    .DIREC   20
      00012    DELETE   06F
      00013    CKSUM    01
```

Another example:



ODL task description:

```
          .NAME   LINK
          .ROOT   LINK-M1-F1
M1:       .FCTR   LINK,LIB/IN:LINK:TASKB:DEF:ERMSG:FILIO:IODAT:
ALBLK:CATB:ROLCB:SAVRG:SAVVR:SCVTR:MUL:CAT5:DTV:GTTXT:SRCINS:
TABLS/EX:BLDSG:GLOB:GTCML:P2OPT:P3MDS:P4MAL:P5MDS:P6STB:SSETUP:TEXT
F1:       .FCTR   (*012,*03456,TXT)
          .NAME   CMDS
012:      .FCTR   CMDS=LINK,LIB/IN:GETLN:GTCML:SCAN:
BLDSG/EX:MULSG:SNGSG=*(01,02)
01:       .FCTR   LINK,LIB/IN:MULSG
          .NAME   CMDS2
02:       .FCTR   CMDS2=LINK,LIB/IN:P2OPT:SNGSG:SSETUP
          .NAME   COMPRO
03456:    .FCTR   COMPRO=LINK,LIB/IN:GTBYT:STINP:WSRCH:PCTRL=*(03,0456)
          .NAME   PASS1
03:       .FCTR   PASS1=LINK,LIB/IN:P3MDS:GTVAL:GLOB
          .NAME   OUTCOM
0456:     .FCTR   OUTCOM=LINK,LIB/IN:P5IMG=*(04,05)
          .NAME   EPASS1
04:       .FCTR   EPASS1=LINK,LIB/IN:P4MAL:GTVAL
          .NAME   PASS2
05:       .FCTR   PASS2=LINK,LIB/IN:P5MDS:P6STB
          .NAME   TEXT
TXT:      .FCTR   TEXT=LINK,LIB/IN:TEXT
          .END
```

LINK command string for overlaid module:

```
#LINK,LP:<LINK.ODL/MP/E
```

Command string for nonoverlaid module, using Librarian

```
#LINK,LP:<LINK.LIB/IN:LINK:TASKB/E
```

the library listing:

```
LIBR  V05A

      LINK  .LIB      04-FEB-74  11:48:51

      SEQ.    NAME  VERSION

     00001   LINK    05
     00002   TASKB   02X26
     00003   BLDSG   015
     00004   P2OPT   02X10
     00005   P3MDS   02X22
     00006   P4MAL   02X02
     00007   P5MDS   02X11
     00008   P6STB   01X04
```

```
00009   NXTFL   02X06
00010   PCTRL   01
00011   P5ELM   02X05
00012   P5IMG   01X16
00013   ADRST   01
00014   ALALO   01
00015   ALBLK   01
00016   ALELO   01D01
00017   CATB    02
00018   DEF     01
00019   DIV     01
00020   DKALO   02X07
00021   DOSHED  02D03
00022   DOSLIB  02
00023   ERMSG   02X08
00024   GTTXT   D01
00025   ERROR   01
00026   FILIO   01X02
00027   GETLN   01X01
00028   GLGEN   01
00029   GLOB    D02
00030   GTBYT   01
00031   GTCMI   01X09
00032   GTVAL   01D04
00033   INITL   01X18
00034   $SETUP  D01
00035   IODAT   02X09
00036   LIBSWT  D05
00037   MPOUT   02X11
00038   MUL     01
00039   MULSG   02X25
00040   RQLCB   01X03
00041   SAVRG   01
00042   SAVVR   01
00043   SCAN    01X09
00044   SCVTR   02
00045   SGALO   02X15
00046   SGTBL   01X01
00047   SNGSG   01X01
00048   SRCINS  01
00049   STINP   01X03
00050   SWSCN   01D10
00051   SYALO   01X01
00052   TABLS   02X18
00053   WSRCH   02
00054   ALSGD   01X01
00055   CAT5    01
00056   EDTMG   01X01


00057   CBTA    01
00058   C5TA    01
00059   TEXT    D01
00060   $LOAD   03
00061   $AUTO   03
00062   $MARKS  02
00063   $RDSEG  04
00064   $RETA   01
```

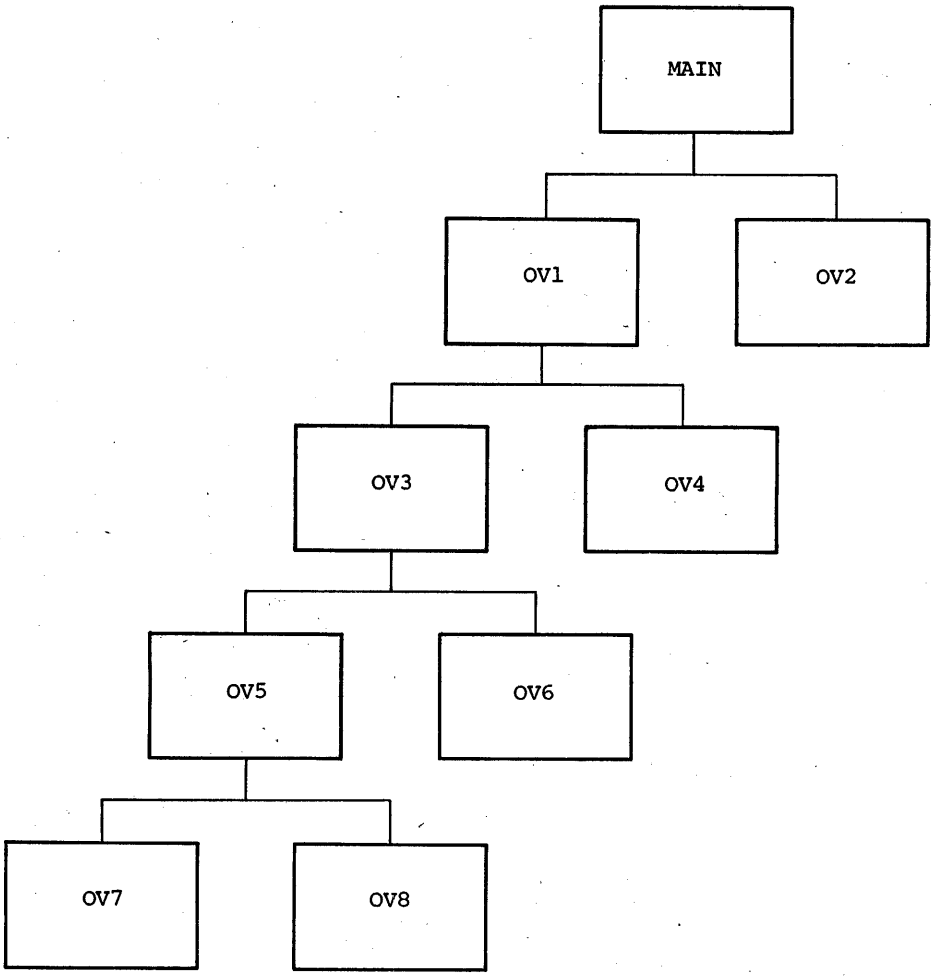
4.7 MANUAL LOAD OVERLAYS FROM FORTRAN

The following is an example of a synchronous overlay load using the FORTRAN callable routine LOAD (see Section 9-4.2.1). 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.Ø) GO TO 99Ø
      CALL LOAD ('GAUSS',1,IERR)
      IF(IERR.NE.Ø) GO TO 5ØØ
      CALL RANDOM(A,B,ITEST,2)
          .
          .
          .
99Ø
5ØØ
```

The following example illustrates the use of the asynchronous overlay load. Both the source coding and the ODL file are used as input to the linker.

In the main segment, the program requests that the segment OV1 (which contains the subroutine ADL1) be loaded into core. The two assignment statements following the CALL LOAD are executed while the overlay is being loaded. The program then performs a CALL WAIT to assure that the load operation is complete before transferring control to the subroutine ADL1 in the overlaid segment. This calling sequence is repeated through the tree structure until the CALL EXIT is encountered in the main segment and control is returned to the Monitor.



Sample Tree Structure

```
0001            SUBROUTINE ADL1
0002            DIMENSION A(10)
0003            A(1)=0.0
0004            WRITE(5,1)
0005    1       FORMAT(/,10X,'ADL1')
0006            CALL LOAD('OV3',1,IERR)
0007            X=1.
0008            Y=X/2.
0009            CALL WAIT
0010            IF(IERR.NE.0) GOTO 10
0011            CALL A2A

0012    2       CALL LOAD('OV4',1,IERR)
0013            X=2.
0014            Y=X/4.
0015            CALL WAIT
0016            IF(IERR.NE.0)GOTO 11
0017            CALL A2B
0018    3       RETURN
0019    10      CALL ERROUT(3)
0020            GO TO 2
0021    11      CALL ERROUT(4)
0022            GO TO 3
0023            END

0001            SUBROUTINE ADL2
0002            DIMENSION A(10)
0003            A(1)=0.0
0004            WRITE(5,1)
0005    1       FORMAT(/,10X,'ADL2')
0006            RETURN
0007            END


0001            SUBROUTINE A2A
0002            DIMENSION A(10)
0003            A(1)=0.0
0004            WRITE(5,1)
0005    1       FORMAT(/,10X,'A2A')
0006            CALL LOAD('OV5',1,IERR)
0007            X=1.
0008            Y=X/2.

0009            CALL WAIT
0010            IF(IERR.NE.0) GO TO 10
0011            CALL A3A
0012    2       CALL LOAD('OV6',1,IERR)
0013            X=2.
0014            Y=X/4.
0015            CALL WAIT
0016            IF(IERR.NE.0) GOTO 11
0017            CALL A3B
0018    3       RETURN
0019    10      CALL ERROUT(5)
0020            GO TO 2
0021    11      CALL ERROUT(6)
0022            GOTO 3
0023            END
```

```
0001            SUBROUTINE A2B
0002            DIMENSION A(10)
0003            A(1)=0.0
0004            WRITE(5,1)
0005    1       FORMAT(/,10X,'A2B')
0006            RETURN
0007            END


0001            SUBROUTINE A3A
0002            DIMENSION A(10)
0003            A(1)=0.0
0004            WRITE(5,1)
0005    1       FORMAT(/,10X,'A3A')
0006            CALL LOAD('OV7',1,IERR)
0007            X=1.
0008            Y=X/2.
0009            CALL WAIT
0010            IF(IERR.NE.0) GO TO 10
0011            CALL A4A
0012    2       CALL LOAD('OV8',1,IERR)
0013            X=2.
0014            Y=X/4.

0015            CALL WAIT
0016            IF(IERR.NE.0) GO TO 11
0017            CALL A4B
0018    3       RETURN
0019    10      CALL ERROUT(7)
0020            GOTO 2
0021    11      CALL ERROUT(8)
0022            GO TO 3
0023            END


0001            SUBROUTINE A3B
0002            DIMENSION A(10)
0003            A(1)=0.0
0004            WRITE(5,1)
0005    1       FORMAT(/,10X,'A3B')
0006            RETURN
0007            END


0001            SUBROUTINE A4A
0002            DIMENSION A(10)
0003            A(1)=0.0
0004            WRITE(5,1)
0005    1       FORMAT(/,10X,'A4A')
0006            RETURN
0007            END


0001            SUBROUTINE A4B
0002            DIMENSION A(10)
0003            A(1)=0.0
0004            WRITE(5,1)
0005    1       FORMAT(/,10X,'A4B')
0006            RETURN
0007            END
```

```
      C       RESIDENT MAIN
0001          DIMENSION A(10)
0002          WRITE(5,1)
0003    1     FORMAT(1H1,25X,'RESIDENT MAIN   ASYNCHRONOUS LOAD')
0004          A(1)=0.0
0005          CALL LOAD('OV1',1,IERR)
0006          X=1.
0007          Y=X/2.
0008          CALL WAIT
0009          IF(IERR.NE.0) GO TO 10
0010          CALL ADL1
0011    3     CALL LOAD('OV2',1,IERR)
0012          X=2.
0013          Y=X/4.
0014          CALL WAIT
0015          IF(IERR.NE.0) GO TO 11
0016          CALL ADL2
0017    4     CALL EXIT

0018    10    CALL ERROUT(1)
0019          GOTO 3
0020    11    CALL ERROUT(2)
0021          GO TO 4
0022          END

0001          SUBROUTINE ERROUT(OVLNUM)
0002          WRITE(5,2) OVLNUM
0003    2     FORMAT(1H ,25X,'FAILURE TO LOAD OV',I1)

0004          RETURN
0005          END


A:      .FCTR   OVRLY=ERROUT=FTNLIB/L
B:      .FCTR   OV1=ADL1=FTNLIB/L
C:      .FCTR   OV2=ADL2=FTNLIB/L
D:      .FCTR   OV3=A2A=FTNLIB/L
E:      .FCTR   OV4=A2B=FTNLIB/L
F:      .FCTR   OV5=A3A=FTNLIB/L
G:      .FCTR   OV6=A3B=FTNLIB/L
H:      .FCTR   OV7=A4A=FTNLIB/L
I:      .FCTR   OV8=A4B=FTNLIB/L
        .ROOT   A=(B=(D=(F=(H,I),G),E),C)
        .NAME   OV1
        .NAME   OV2
        .NAME   OV3
        .NAME   OV4
        .NAME   OV5
        .NAME   OV6
        .NAME   OV7
        .NAME   OV8
        .END
```

4.8 AUTOLOAD OVERLAYS FROM FORTRAN

This example of autoload overlays uses the same program and tree structure as the asynchronous manual load overlay example. The differences between the two load examples are:

1. The source code has a call to each subroutine with no explicit reference to loading each segment, and
2. The autoload operator ('*') is included in the ODL file.

```
0001            SUBROUTINE ADL1
0002            DIMENSION A(10)
0003            A(1)=0.0
0004            WRITE(5,1)
0005     1      FORMAT(/,10X,'ADL1')
0006            CALL A2A
0007            CALL A2B
0008            RETURN
0009            END

0001            SUBROUTINE ADL2
0002            DIMENSION A(10)
0003            A(1)=0.0
0004            WRITE(5,1)
0005     1       FORMAT(/,10X,'ADL2')
0006            RETURN
0007            END


0001            SUBROUTINE A2A
0002            DIMENSION A(10)
0003            A(1)=0.0
0004            WRITE(5,1)
0005     1       FORMAT(/,10X,'A2A')
0006            CALL A3A
0007            CALL A3B
0008            RETURN
0009            END

0001            SUBROUTINE A2B
0002            DIMENSION A(10)
0003            A(1)=0.0
0004            WRITE(5,1)
0005     1       FORMAT(/,10X,'A2B')
0006            RETURN
0007            END

0001            SUBROUTINE A3A
0002            DIMENSION A(10)
0003            A(1)=0.0
0004            WRITE(5,1)
0005     1       FORMAT(/,10X,'A3A')
0006            CALL A4A
0007            CALL A4B
0008            RETURN
0009            END
```

```
0001              SUBROUTINE A3B
0002              DIMENSION A(10)
0003              A(1)=0.0
0004              WRITE(5,1)
0005     1         FORMAT(/,10X,'A3B')
0006              RETURN
0007              END

0001              SUBROUTINE A4A
0002              DIMENSION A(10)
0003              A(1)=0.0
0004              WRITE(5,1)
0005     1         FORMAT(/,10X,'A4A')
0006              RETURN
0007              END

0001              SUBROUTINE A4B
0002              DIMENSION A(10)
0003              A(1)=0.0
0004              WRITE(5,1)
0005     1         FORMAT(/,10X,'A4B')
0006              RETURN
0007              END

         C        RESIDENT MAIN
0001              DIMENSION A(10)
0002              WRITE(5,1)
0003     1         FORMAT(1H1,25X,'RESIDENT MAIN')
0004              A(1)=0.0
0005              CALL  ADL1
0006              CALL ADL2
0007              CALL EXIT
0008              END


A;       .FCTR    OVRLY=FTNLIB/L
B;       .FCTR    ADL1=FTNLIB/L
C;       .FCTR    ADL2=FTNLIB/L
D;       .FCTR    A2A=FTNLIB/L
E;       .FCTR    A2B=FTNLIB/L
F;       .FCTR    A3A=FTNLIB/L
G;       .FCTR    A3B=FTNLIB/L
H;       .FCTR    A4A=FTNLIB/L
I;       .FCTR    A4B=FTNLIB/L
         .ROOT    A=*(B=(D=(F=(H,I),G),E),C)
         .END
```

4.9 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. (See Part 7, FORTRAN.)

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 9-5 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 nonexecutable code in the program. Also, this method may pull more routines into core than needed; the other two ways are more efficient.)

For example:

```
      LOGICAL L
      GO TO 1ØØØ
      READ (6,1ØØ) I,L
 1ØØ  FORMAT (I1,L1)
      WRITE (6,101), I,E
 1Ø1  FORMAT (I1, E6.0)
        .
        .
        .
1ØØØ  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 9-1).

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 that 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:

```
FORTØØ8ØØØ 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 9-1 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 9-1

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.

PART 9

CHAPTER 5

PROGRAM MEMORY ORGANIZATION

5.1 ALLOCATION FOR A NONOVERLAID PROGRAM

A nonoverlaid program is allocated to memory as shown in Figure 9-5.

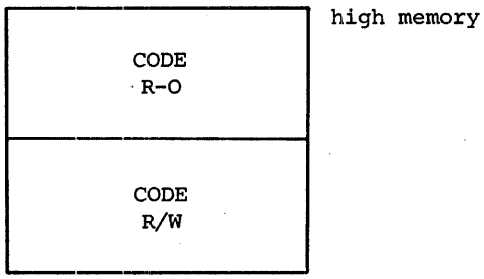


Figure 9-5

Nonoverlaid 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 9-6.

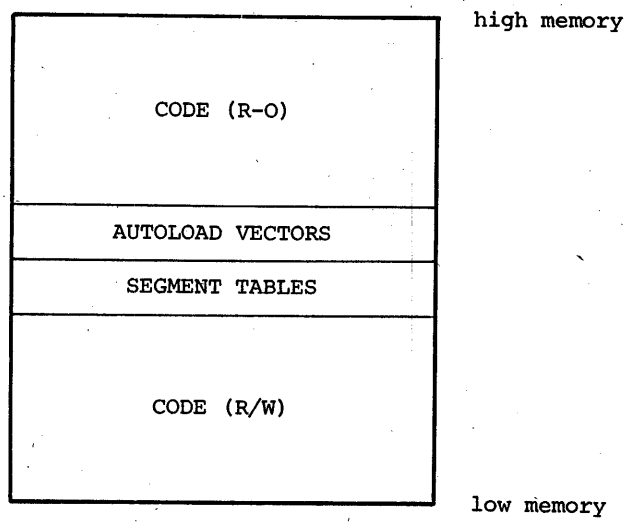


Figure 9-6

Root Segment Overlaid Program

Code (R/W) and code (R-O) are the same as for nonoverlaid programs, consequently, only the segment tables and autoload vectors are described.

5.2.2 The Segment Tables

Each segment in an overlay structure has a 10-word segment descriptor formatted as shown in Figure 9-7.

RESERVED	STATUS
RELATIVE DISK ADDRESS	
LOAD ADDRESS	
LENGTH IN BYTES	
LINK UP	
LINK DOWN	
LINK NEXT	
LINK PREVIOUS	
SEGMENT	
NAME	

Figure 9-7

Segment Descriptor

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

Relative Disk Address of the Overlay Segment: -- A program image occupies a contiguous disk area. Each overlay segment begins at a disk block boundary, and this word is an index to the relative block number from the start of the program disk image. This word enables loading of segments with a single disk access.

Load Address of The Segment: -- The program relative address where this segment is to be loaded.

Length of the Segment: -- The number of bytes in the segment; this number is used to construct the disk read.

Link Fields: -- The function of the link fields is, given the address of any descriptor, to find 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: -- This word is a pointer to a segment descriptor away from the root. Such a segment emanates from an overlay control point that 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 9-8 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.

Link Down: -- This word is a pointer to a segment, nearer the root, that 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 overlaid. In Figure 9-8, B, C, and D are circularly linked as are E and F. A has null link-next and link-previous pointers.

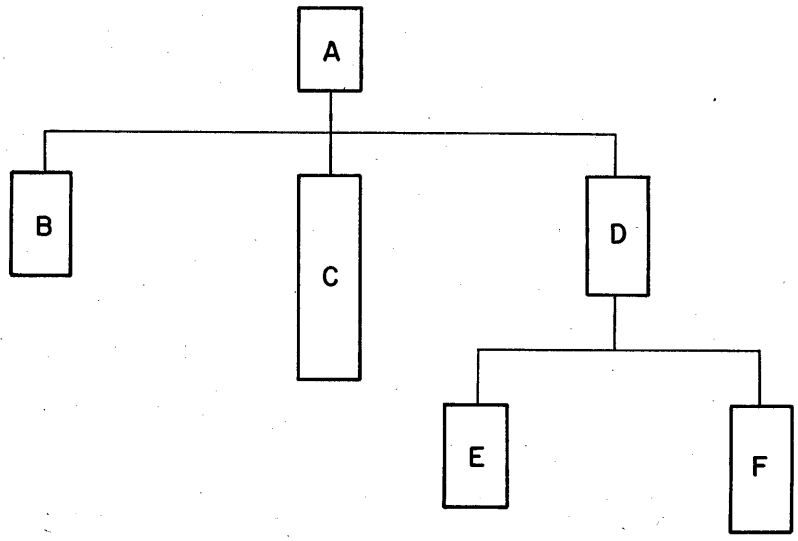


Figure 9-8

Link Paths

Segment Name: -- This two-word field is RAD5Ø representation of the segment name.

5.2.3 Autoload Vectors

Autoload vectors appear in every segment that references autoload entry points in segments farther away from the root than the referencing segment. Segments that 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 9-4.

5.3 OVERLAY MEMORY ALLOCATION

Every overlay in a program has an allocation as shown in Figure 9-9. The construction of segments is identical to the root segment structure discussed in Chapter 9-5.2.

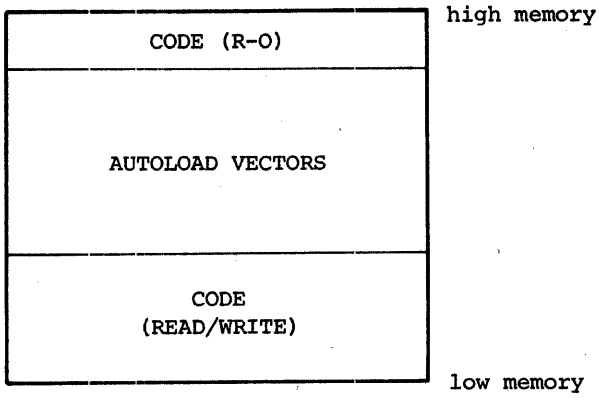


Figure 9-9

Overlay Segment

5.4 OVERALL MEMORY ORGANIZATION

Figure 9-10 shows the overall memory allocation of an overlaid program running under DOS/BATCH.

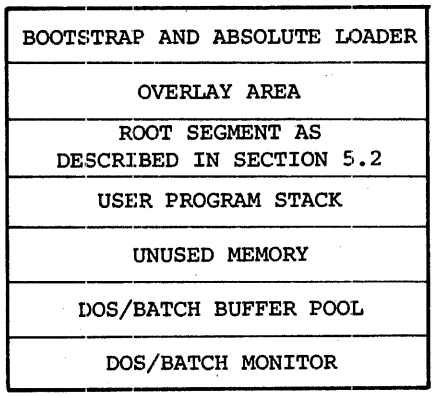


Figure 9-10

Overall Memory Allocation

The linker computes the size of the overlay area to accommodate the largest possible combination of overlays that could exist simultaneously in memory.

PART 9

CHAPTER 6

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.

6.1 MEMORY ALLOCATION PROCEDURES

6.1.1 Allocating Root Segment Memory

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

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 9-7; and if a command is found for this P-section it is extended under the following conditions:

1. If the CON attribute for the P-section is set.
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. Currently, 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 location counter appropriately.

LINK now checks if it is building an overlaid program, and, if it is, it allocates the storage for the Segment Tables. Finally, any storage needed to hold autoload entry points referred to up-the-tree by the root segment are allocated.

6.1.2 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.

6.2 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 the following:

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.
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 Chapter 9-11 for a detailed map description and example.

6.3 LINK TREE WALK ALGORITHM

In the map listing, LINK displays segment descriptions in a path order that results from a tree walk; the result of the walk is the segment list that 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 unprocessed 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 9-11, and the ODL description

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

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 redisplayed
F   (link-next)
G   (link-up)
H   (link-next)
F   (link-down):not redisplayed
A   (link-down):not redisplayed
```

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

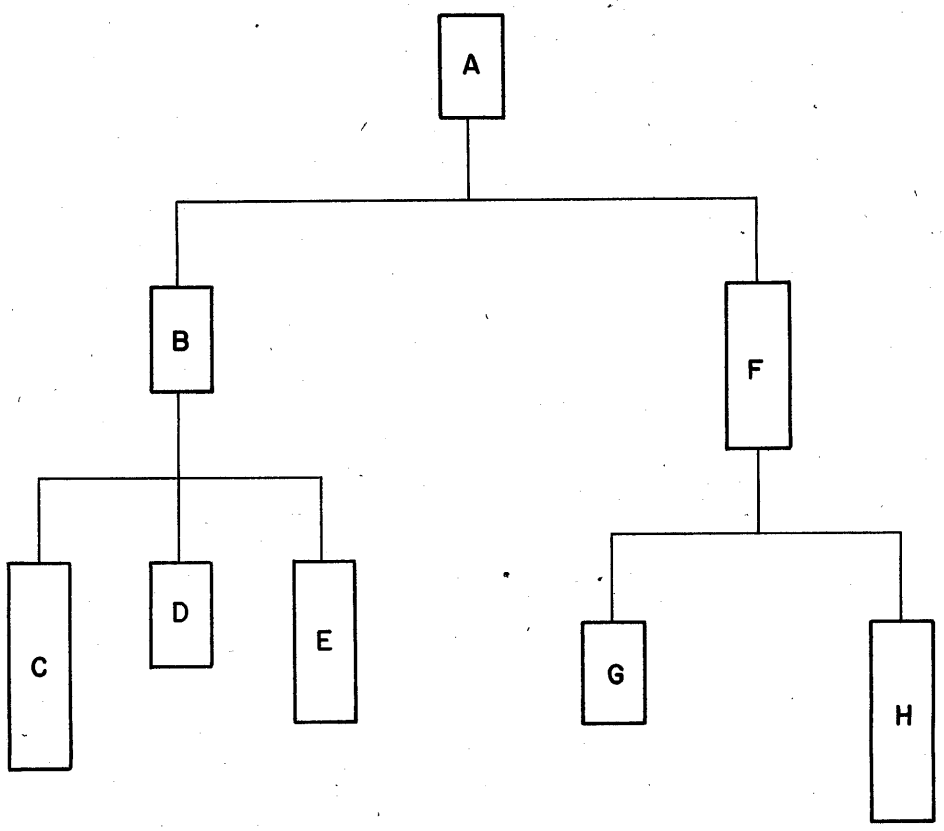


Figure 9-11

Tree Walk

PART 9

CHAPTER 7

LINKING OPTIONS

7.1 OPTIONAL INPUT

Optional 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 the line

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 optional 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 specifications must always fit on a single line. An optional input line has 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 Mth parameter of the Nth 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.

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 RAD5Ø 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).

7.2 ABSOLUTE PATCH (ABSPAT)

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

An absolute address is the physical core 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 value.

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 1ØØ:

```
ABSPAT = PAY:1ØØ:-1:5:6
```

NOTE

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

7.3 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 5Ø and 1ØØ bytes, respectively.

```
EXTSCT=ONE:5Ø,TWO:1ØØ
```

7.4 GLOBAL SYMBOL DEFINITION (GBLDEF)

Declare the definition of a global symbol.

The symbol definition is considered absolute (not relocatable). 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
```

7.5 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+1Ø:177Ø6:177344

NOTE

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

7.6 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
```

PART 9

CHAPTER 8

LINK INPUT DATA FORMATS

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

Object modules are created by any of the standard language processors (i.e., MACRO, 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 load module.

The six record types follow:

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 contain all of the record types except the ISD, which is optional. 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 an end of module record. Additional declared 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. Figure 9-12 illustrates this format.

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.

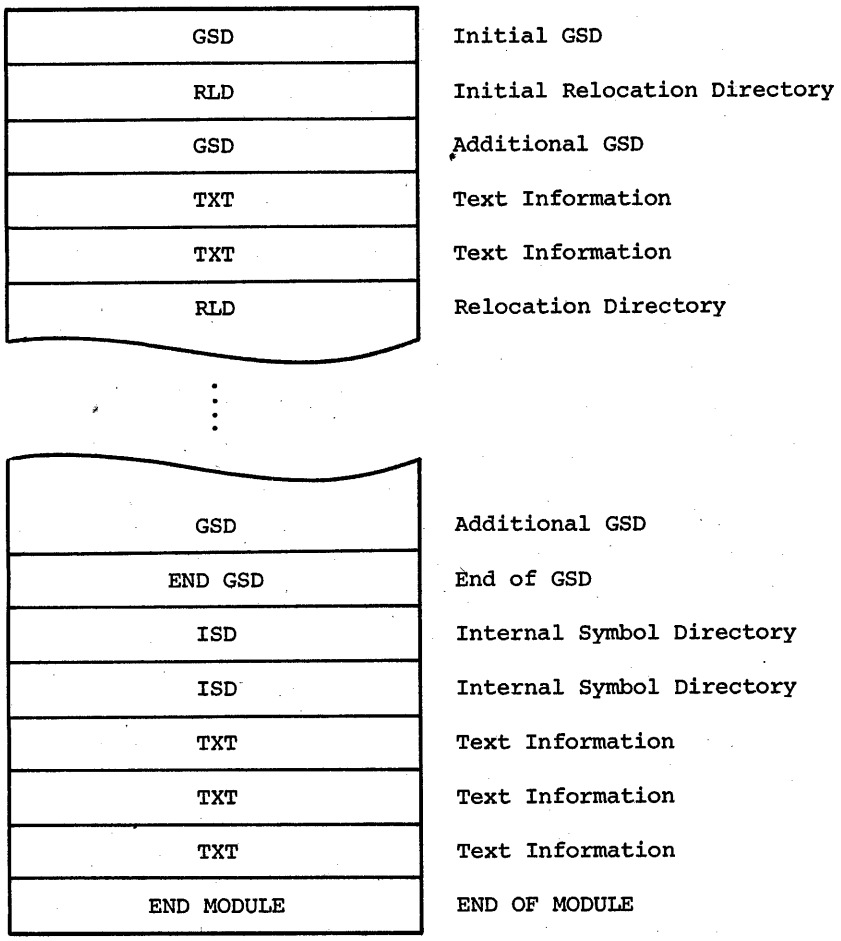


Figure 9-12

General Object Module Format

8.1 GLOBAL SYMBOL DIRECTORY

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

GSD records are the only records processed in the first pass. 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 Ø - 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 RAD5Ø characters. The third word contains a flag byte and the entry type identification. The fourth word contains additional information about the entry. Figure 9-13 illustrates the GSD record format.

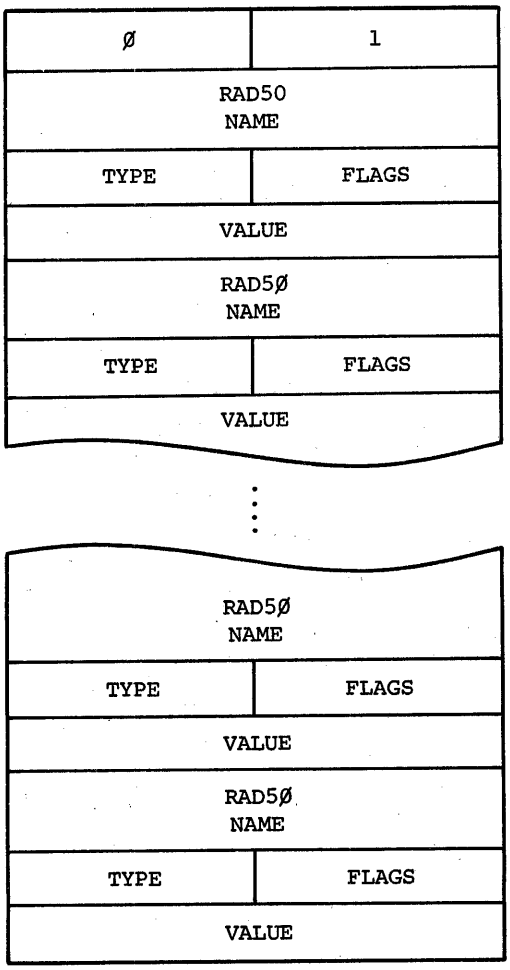
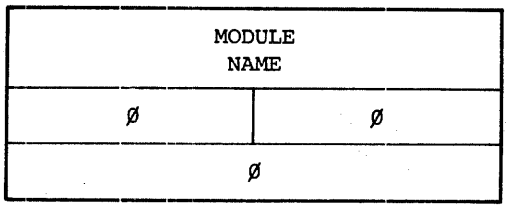


Figure 9-13

GSD Record and Entry Formats

8.1.1 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

8.1.2 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 9-8.1.6 details the entry generated 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
```

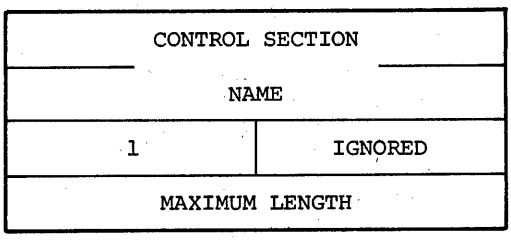
For a named CSECT:

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

And for an ASECT:

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

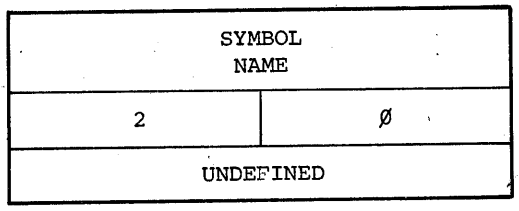
ASECTs and CSECTs are processed by LINK as PSECTs with the fixed attributes defined above (refer to Section 9-8.1.6.) The entry generated for a control section is illustrated on the following page.



Control Section Name Entry Format

8.1.3 Internal Symbol Name

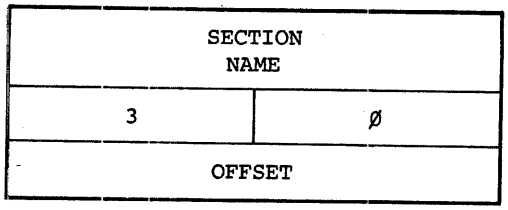
The internal symbol name entry declares the name of an internal symbol (with respect to the module). LINK does not yet support internal symbol tables; therefore the detailed format of this entry is not defined. If an internal symbol entry is encountered while reading the GSD, it is merely ignored.



Internal Symbol Name Entry Format

8.1.4 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 ØØØØØ1 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 ØØØØØ1.

8.1.5 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 declaration 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 Ø - 2 - Not used

Bit 3 - Definition

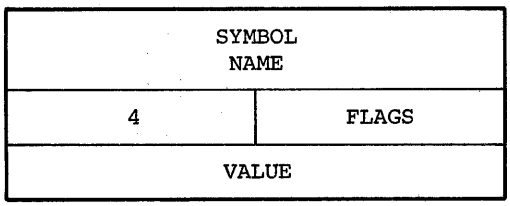
Ø = Global symbol references.
1 = Global symbol definition.

Bit 4 - Not used

Bit 5 - Relocation

Ø = Absolute symbol value.
1 = Relative symbol value.

Bit 6 - 7 - Not used



Global Symbol Name Entry Format

8.1.6 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 has the following bit assignments:

Bit Ø - Memory Speed

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

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

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

Ø = Normal P-section.

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

Bit 2 - Allocation

Ø = 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

Ø = P-section has read/write access.

1 - P-section has read only access.

Bit 5 - Relocation

Ø = 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

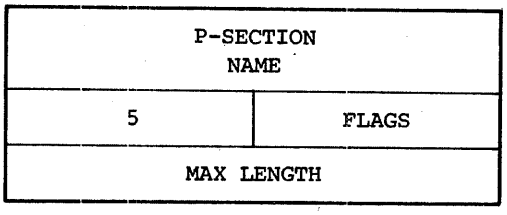
Ø = The scope of the P-section is local. References to the same P-section will be collected only within the segment in which the P-section is defined.

1 = The scope of the P-section is global. References to the P-section are collected across segment boundaries. 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

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

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



P-Section Name Entry Format

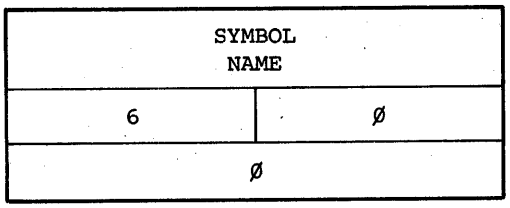
NOTE

The length of all absolute sections is zero.

8.1.7 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 contain the version identification. The flag byte and fourth words are not used and contain no meaningful information.



Program Version Identification Entry Format

8.2 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. Only one end of GSD record must appear in every object module. It is one word in length.

End of GSD Record Format

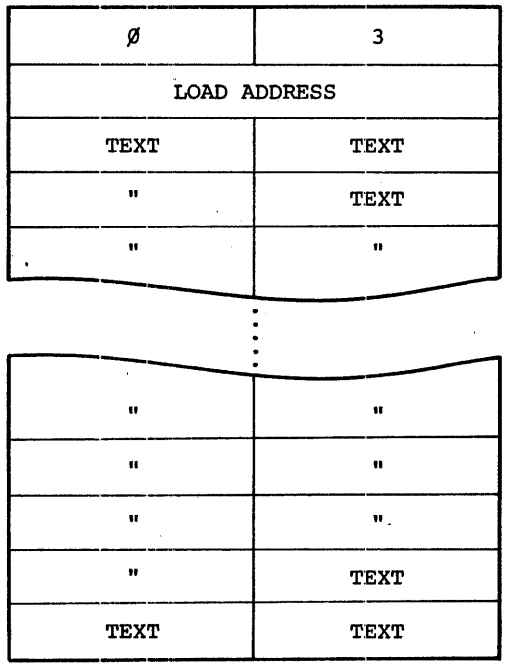
8.3 TEXT INFORMATION

The text information record contains a byte string of information that is to be written directly into load modules. 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 Section 9-8.4). 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

8.4 RELOCATION DIRECTORY

Relocation directory records contain the information necessary to relocate and link a preceding text information record. Every module must have at least one relocation directory record that precedes the first text information record. Figure 9-14 shows the relocation directory record format. The first record does not modify a preceding 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 1Ø	-	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 Section 9-8.3) yields the absolute address that is to be modified.

The command byte of each entry has the following bit assignments.

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

Bit - 7 Modification

Ø = 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.

ø	4
DISP	CMD
INFO	INFO
"	INFO
"	"
"	"
"	"
"	"
"	"
"	"
CMD	"
INFO	DISP
"	INFO
"	"
"	"
"	"
"	"
DISP	CMD
INFO	INFO
INFO	INFO
INFO	INFO

Figure 9-14

Relocation Directory Record Format

8.4.1 Internal Relocation

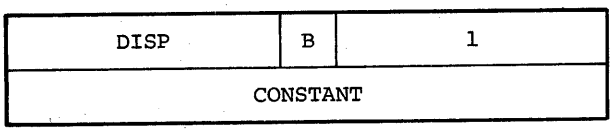
This type of entry relocates a direct pointer to an address within a module. The current P-section base address is added to a specified constant, and the result is written into the load module.

Example:

```
A:    MOV     #A,RØ
```

or

```
      .WORD   A
```



Internal Relocation Entry Format

8.4.2 Global Relocation

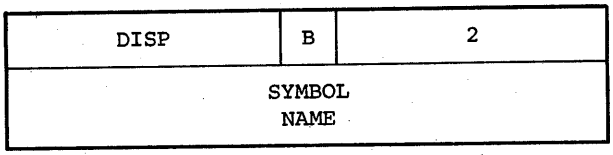
This type of entry relocates a direct pointer to a global symbol. The definition of the global symbol is obtained, and the result is written into the load module.

Example:

```
      MOV     #GLOBAL,RØ
```

or

```
      .WORD   GLOBAL
```

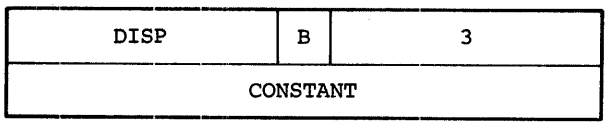


Global Relocation Entry Format

8.4.3 Internal Displaced Relocation

This type of entry relocates a relative reference to an absolute address from within a relocatable control section. The address +2 that the relocated value is to be written into is subtracted from the specified constant. The result is then written into the load module.

```
CLR @#17755Ø

    or

MOV @#177550,RØ
```



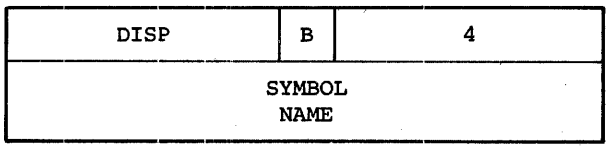
Internal Displaced Relocation Entry Format

8.4.4 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 +2 that the relocated value is to be written into is subtracted from the definition value. This value is then written into the load module.

Example:

```
CLR     GLOBAL

    or

MOV     GLOBAL,RØ
```



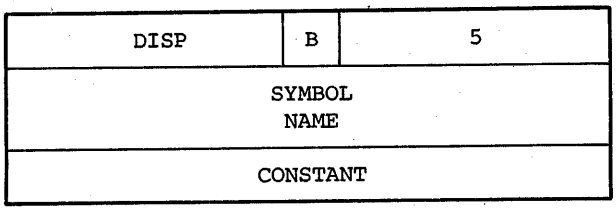
Global Displaced Relocation Entry Format

8.4.5 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 load module.

Example:

```
MOV     #GLOBAL+2,RØ

    or

.WORD   GLOBAL-4
```



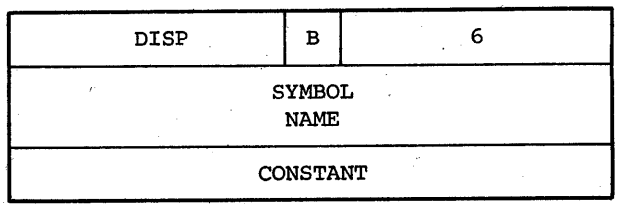
Global Additive Relocation Entry Format

8.4.6 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 +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 load module.

Example:

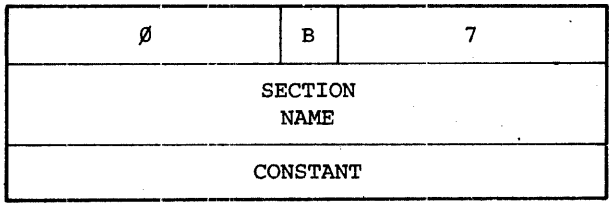
```
CLR     GLOBAL+2

    or

MOV     GLOBAL-5,RØ
```



Global Additive Displaced Relocation Entry Format

8.4.7 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.



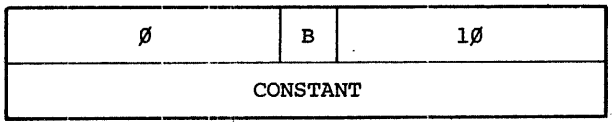
Location Counter Definition Entry Format

8.4.8 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
```



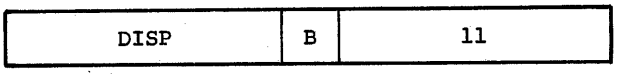
Location Counter Modification Entry Format

8.4.9 Program Limits

This type of entry is generated by the .LIMIT assembler directive. The lowest and highest addresses allocated to the task are obtained and written into the load module.

Example:

```
        .LIMIT
```



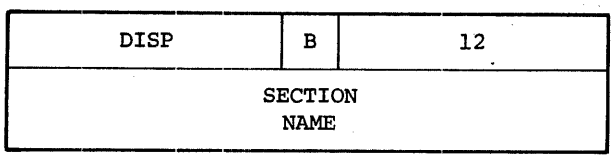
Program Limits Entry Format

8.4.10 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 load module.

Example:

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

            or

        .WORD   B
```



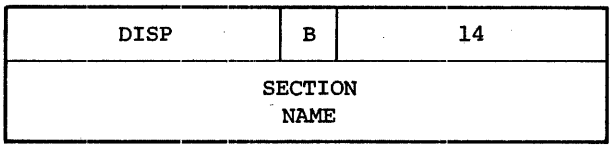
P-Section Relocation Entry Format

8.4.11 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 +2 that the relocated value is to be written into is subtracted from the base value. This value is then written into the load module.

Example

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



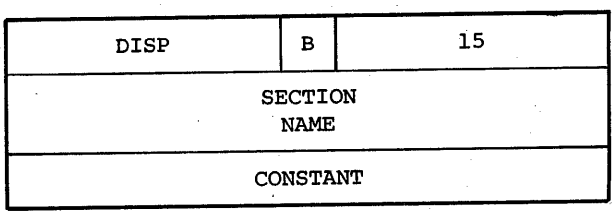
P-Section Displaced Relocation Entry Format

8.4.12 P-Section Additive Relocation

This 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 load module.

Example:

```
        .PSECT   A
B:
        .
        .
        .
        .
        .
C:      .
        .
        .
        .
        PSECT    D
        MOV      #B+1Ø,RØ
        MOV      #C,RØ

        or

        .WORD    B+10
        .WORD    C
```



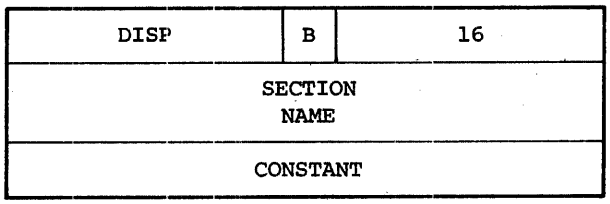
P-Section Additive Relocation Entry Format

8.4.13 P-Section Additive Displaced Relocation

This type of entry relocates a relative reference 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 address +2 that the relocated value is to be written into is subtracted from the resultant additive value. This value is then written into the load module.

Example:

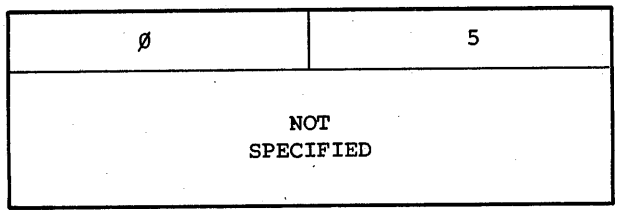
```
     .PSECT   A
B:
     .
     .
     .
C:   .
     .
     .
     .
     .
     .PSECT   D
     MOV      B+1Ø,RØ
     MOV      C,RØ
```



P-Section Additive Displaced Relocation Entry Format

8.5 INTERNAL SYMBOL DIRECTORY

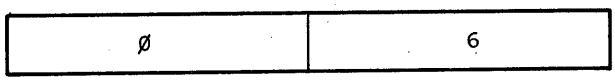
Internal symbol directory records declare definitions of symbols that are local to a module. This feature is not supported by LINK and therefore a detailed record format is not specified. If LINK encounters this type of record, it will ignore it.



Internal Symbol Directory Record Format

8.6 END OF MODULE

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



End of Module Format

PART 9

CHAPTER 9

PROGRAM LOAD MODULE FILE STRUCTURE

An overlay image as it is recorded on disk appears in Figure 9-15 (pertinent boundaries are shown).

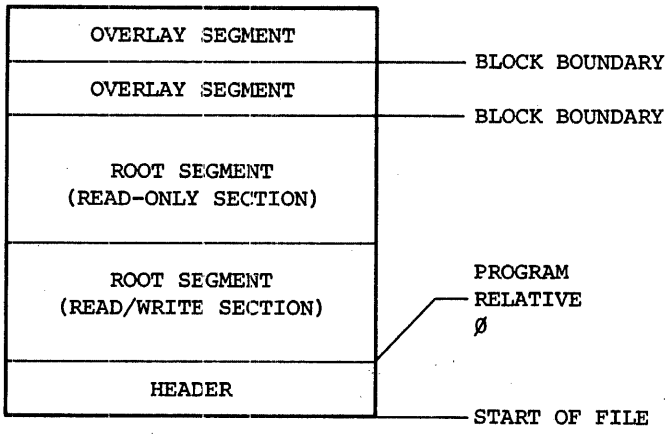


Figure 9-15

Overlay Disk Format

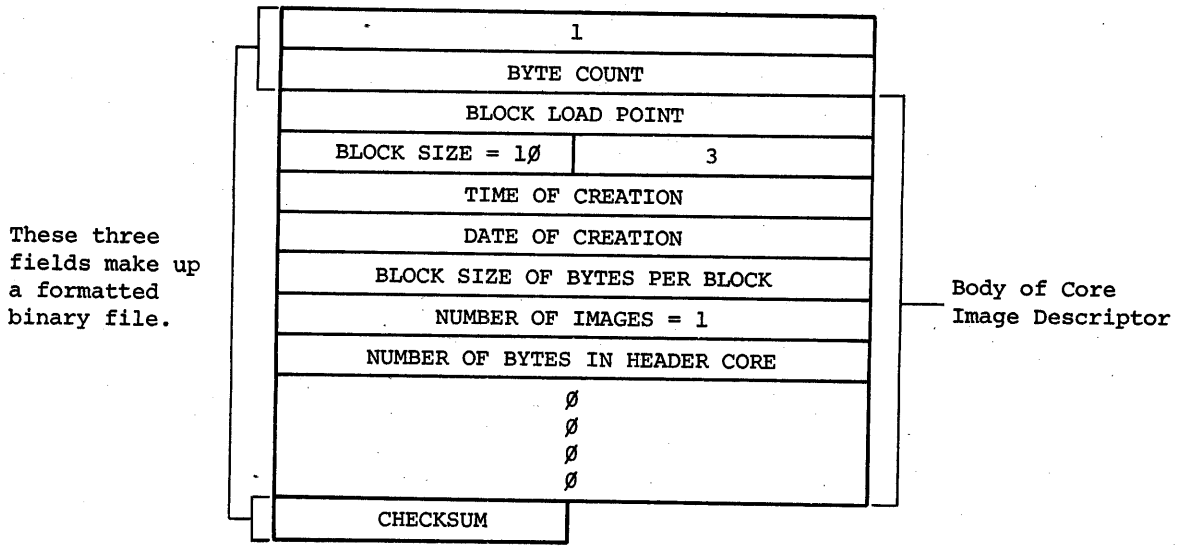
9.1 THE HEADER

The overlay header consists of two parts:

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

The core image descriptor contains information required for loading the overlay into core. 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:

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-5Ø	
.IDENT OF PROGRAM IN RADIX-5Ø	
TIME OF CREATION	
DATE OF CREATION	
WORDS TO FOLLOW	EMT CALLS RES. = 2
DOS/BATCH EMT NUMBERS CORRESPONDING TO MONITOR ROUTINES TO BE MADE RESIDENT	
END OF COMD = Ø	
CHECKSUM	

9.1.1 The Root Segment

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

9.1.2 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 its space requests. The maximum size for any segment, including the root, is 32K words.

9.2 NONOVERLAID PROGRAM FILE STRUCTURES

The file structure of a nonoverlaid program normally consists of a formatted binary file beginning with a header whose COMD is similar to that described in Section 9-9-1, with the following exceptions:

1. "FIRST RELATIVE BLOCK OF IMAGE" entry is not included.
2. Byte count is correspondingly smaller.

The end of a nonoverlaid 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 nonoverlaid program, the file image will be as shown in Figure 9-15, except for the omission of overlay segments.

PART 9

CHAPTER 10

.ASECTS, .CSECTS, AND .PSECTS

10.1 PROGRAM SECTION DIRECTIVES

10.1.1 .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 9-2.

Table 9-2

.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 (not implemented)
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)

Table 9-2 (Cont.)

.PSECT Directive Parameters

Parameter	Default	Meaning
CON/OVR	CON	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 Ø). 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(1Ø) program sections: one absolute section, one blank relocatable section, and 253(1Ø) named relocatable sections. The .PSECT directive enables the user to create his program (object module) in sections and 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
4. Section attributes (the six .PSECT attributes)

10.1.2 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 that are not physically contiguous but are loaded contiguously, as shown in the following example:

```
            .PSECT  SEC1,REL        ;START A RELOCATABLE SECTION NAMED
A:          .WORD   Ø               ;SEC1 ASSEMBLED AT RELOCATABLE Ø,
B:          .WORD   Ø               ;RELOCATABLE 2 AND
C:          .WORD   Ø               ;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   Ø               ;ASSEMBLED AT RELOCATABLE Ø,
B:          .WORD   Ø               ;ASSEMBLED AT RELOCATABLE 2,
C:          .WORD   Ø               ;ASSEMBLED AT RELOCATABLE 4,
            .PSECT  COM2,REL        ;DECLARE RELOCATABLE SECTION COM2
X:          .WORD   Ø               ;ASSEMBLED AT RELOCATABLE Ø
Y:          .WORD   Ø               ;ASSEMBLED AT RELOCATABLE 2,
            .PSECT  COM1            ;RETURN TO COM1
D:          .WORD   Ø               ;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 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
            .=1ØØØ
A:          CLR     X               ;ASSEMBLED AS CLR BASE OF
                                    ;RELOCATABLE SECTION + 1Ø
            JMP     Y               ;ASSEMBLED AS JMP BASE OF
                                    ;RELOCATABLE SECTION +6
            .PSECT  SEN,REL
            MOV     RØ,R1
            JMP A                   ;ASSEMBLED AS JMP 1ØØØ
Y:          HALT
X:          WORD    Ø
            .END
```

10.1.3 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; 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.

10.2 .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 9-3.

Table 9-3

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
```

PART 9

CHAPTER 11

LOAD MAP EXAMPLES

11.1 MAP LISTING

The Map has a header followed by segment descriptions.

11.1.1 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
```

11.1.2 Segment Descriptions

Segment descriptions have four subsections

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

Segment title lines appear as

***SEG: segname

11.2 ATTRIBUTES AND STATISTICS

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

11.2.1 Read/Write Memory Limits

Read/Write Memory limits are displayed as follows:

```
R/W MEM  LIMITS:  start end length
```

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

11.2.2 Read-Only Memory Limits

Read-Only memory limits are displayed as follows:

```
R-O MEM LIMITS:  start end length
```

This entry can occur only for the root segment.

11.2.3 ODT Transfer Address

ODT transfer address is displayed as follows:

```
ODT XFR ADDRESS:  address
```

11.2.4 Program Transfer Address

Program transfer address is displayed as follows:

```
PRG XFR ADDRESS:  address
```

11.2.5 Identification

Identification is displayed as follows:

```
IDENTIFICATION  :  name
```

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

11.3 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 valid. 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.

11.4 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 appears in the order it is encountered, formatted 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.

11.5 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
```