FLOATING POINT
SYSTEMS, INC.

FPS-100 Loader (LOD100) Reference Manual

860-7423-000

by FPS Technical Publications Staff

FPS-100 Loader (LOD100) Reference Manual

860-7423-000

Publication No. 860-7423-000
September, 1979

NOTICE

This edition applies to Release A of FPS-100 software and all subsequent releases until superseded by a new edition.

The material in this manual is for informational purposes only and is subject to change without notice.

Floating Point Systems, Inc. assumes no responsibility for any errors which may appear in this publication.

Printed in USA

CONTENTS

		Page
CHAPTER 1	INTRODUCTION	
1.1	OVERVIEW	1-1
1.2	PURPOSE	1-1
1.3	SCOPE	1-1
1.4	CONVENTIONS	1-2
1.5	RELATED MANUALS	1-2
1.6	LOADER FUNCTIONS	1-3
1.6.1	FPS-100 Load Module	1-3
1.6.2	Host-FPS-100 Software Interface Mechanism (HASI)	1-4
1.7	FPS-100 JOBS	1-5
1.7.1	Overlay Segments	1-5
1.7.2	Single-Level Jobs	1-8
1.7.3	Multi-Level Jobs	1-8
1.7.4	Multiple Load Module Jobs	1-9
1.8	SUPERVISOR ENVIRONMENT	1-9
1.8.1	Tasks	1-9
1.8.2	Interrupt Service Routines	1-10
1.8.3	Task Mode	1-10
CHAPTER 2	LOD100 INPUT	
2.1	INTRODUCTION	2-1
2.2	CALLING LOD100	2-1
2.3	COMMANDS	2-2
2.3.1	INPUT	2-2
2.3.2	OUTPUT	2-2
2.3.3	RADIX	2-4
2.3.4	LMID	2-5
2.3.5	MODE	2-5
2.3.6	PRI	2-6
2.3.7	TASK	2-6
2.3.8	TREE	2-8
2.3.9	OVERLAY	2-9
2.3.10	CALL	2-10
2.3.11	LOAD	2-11
2.3.12	LIB	2-11
2.3.13	FORCE	2-12
2.3.14	NOLOAD	2-12
2.3.15	MDOFF	2-13
2.3.16	PSOFF	2-13
2.3.17	MMAX	2-14
2.3.18	PMAX	2-14
2.3.19	PPA	2-15
2.3.20	MARK	2-15

2.3.21	PURGE	2-15
2.3.22	MAP	2-16
2.3.23	LINK	2-22
2.3.24	INIT	2-23
2.3.25	EXIT	2-23
2.4	CREATING A SINGLE-LEVEL JOB	2-24
2.5	CREATING A MULTI-LEVEL JOB	2-26
2.6	CREATING A MULTIPLE LOAD MODULES	2-29
2.7	LOADING TASKS	2-30
2.7.1	Loading ASM100 Tasks	2-31
2.7.2	Loading FTN100 Tasks	2-32
2.8	OVERLAY TABLE AND PS PARTITION TABLE	2-34
2.9	TASK COMMUNICATION BLOCK (TCB)	2-36
2.10	READY QUEUE	2-39

CHAPTER 3	OBJECT MODULES	
3.1	INTRODUCTION	3-1
3.2	CODE BLOCK (0)	3-2
3.3	END BLOCK (1)	3-3
3.4	TITLE BLOCK (3)	3-4
3.5	ENTRY BLOCK (4)	3-4
3.6	EXTERNAL BLOCK (5)	3-5
3.7	LIBRARY START BLOCK (6)	3-5
3.8	LIBRARY END BLOCK (7)	3-5
3.9	DATA BLOCK DESCRIPTOR BLOCK (10)	3-6
3.10	DATA BLOCK INITIALIZATION BLOCK (11)	3-7
3.11	FORMAL PARAMETER BLOCK (12)	3-8
3.12	ALTERNATE ENTRY BLOCK (13)	3-9
3.13	TASK BLOCK (15)	3-10
3.14	ISR BLOCK (16)	3-10
3.15	SAMPLE OBJECT MODULE	3-11

CHAPTER 4	OUTPUT FROM LOD100	
4.1	INTRODUCTION	4-1
4.2	LOAD MODULE	4-1
4.2.1	Code/Overlay/32-Bit MD Data Block (0)	4-2
4.2.2	Data Block (1)	4-3
4.2.3	Information Block (2)	4-4
4.2.4	End Block (3)	4-5
4.2.5	Sample Load Module	4-5
4.3	HASI	4-7
4.3.1	FPS-100 Executive Routines	4-7
4.3.2	ADC HASI	4-12
4.3.3	UDC HASI	4-14
4.3.4	Common Blocks in HASI Routines	4-16

CHAPTER 5 ERROR MESSAGES

5.1 GENERAL INFORMATION 5-1
5.2 MESSAGES 5-2

ILLUSTRATIONS

Figure No.	Title	Page
1-1	Overlay Branch Structure	1-6
1-2	Overlay Memory Allocation	1-7
2-1	Overlays	2-8
2-2	Load Map	2-17
2-3	Binary Tree Structure	2-19
2-4	Load Map with Tasks	2-21
2-5	Overlay Structure Including Overlay Numbers	2-27
2-6	Overlay Segments	2-34
3-1	Sample Object Module	3-11
4-1	Sample Load Module	4-6
4-2	ADC Subroutine	4-12
4-3	ADC HASI	4-13
4-4	UDC Subroutine	4-14
4-5	UDC HASI	4-15

TABLES

Table No.	Title	Page
1-1	Related Manuals	1-2
2-1	Overlay Table Entry Format	2-35
2-2	TCB Format	2-37
5-1	Error Messages	5-2

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

The Floating Point Systems, Inc., FPS-100 is a peripheral device that operates independently but under the direction of a host processor. It contains its own internal memories and 38-bit floating-point arithmetic units which are interconnected with multiple data paths to allow parallel internal data transfers. Its arithmetic units, the floating adder and floating multiplier, are designed as pipelines (operations are performed in independent stages permitting new operations to begin before old operations are complete). This parallel processing capability and pipeline arithmetic permit the FPS-100 to perform high speed array processing.

Since the FPS-100 is under the direction of a host computer, programs are normally produced on the host computer and transferred over to the FPS-100 for execution. The FPS-100 loader (LOD100) uses the object modules produced by the ASM100 cross assembler and the FTN100 cross compiler and produces the load modules which can be transferred to the FPS-100 and executed. LOD100 also produces the routines which transfer the load modules to the FPS-100 and initiate FPS-100 operation.

1.2 PURPOSE

This manual documents the LOD100 loader. It is intended for programmers experienced in FORTRAN or assembly language programming. It assumes that the user can create the object modules necessary for input to LOD100 and can execute the load modules produced by LOD100. It does not contain detailed programming information.

1.3 SCOPE

This manual completely documents the commands available with LOD100. It also describes the format of the object module input and the HASI and load module output. Overlay structures are also discussed. Finally the error messages produced by LOD100 are described.

1.4 CONVENTIONS

Throughout this manual, the following conventions are used:

- In examples of dialogue at a terminal, user input is underlined to distinguish it from program or system output.
- All user input at a terminal is assumed to be terminated with a carriage return.
- In examples of statements or commands, uppercase characters must be entered exactly as shown; lowercase characters indicate that a value or name must be substituted for the characters. Optional parameters are surrounded by brackets (< >). A list of parameters surrounded by brackets (< >) indicates that the entire list is optional, but only one of the parameters can be entered. A list of parameters surrounded by braces ([ ]) indicates that one of the list must be entered, but no more than one can be entered.

1.5 RELATED MANUALS

The documents in Table 1-1 may also be useful:

Table 1-1 Related Manuals

MANUAL	PUBLICATION NO.
FTN100 Reference Manual	FPS 860-7422-000
FPS-100 Math Library Manual	FPS 860-7429-000
ASM100 Reference Manual	FPS 860-7428-000
SIM100/DBG100 Reference Manual	FPS 860-7424-000
FPS-100 Programmer's Reference Manual	FPS 860-7427-000
VFC100 Reference Manual	FPS 860-7447-000
APX100 Manual	FPS 860-7426-000
FPS-100 Supervisor Reference Manual	FPS 860-7445-000

1.6 LOADER FUNCTIONS

LOD100 is a host resident FPS-100 loader which uses the object modules generated by the FTN100 compiler and the ASM100 assembler and performs the following functions:

- allocates memory space in the FPS-100 for programs and data
- resolves symbolic references between object modules (links programs and subprograms together)
- adjusts address-dependent locations to correspond to the allocated space (relocates code)
- places machine instructions in a load module which is structured so that it can easily be placed in the FPS-100
- provides a mechanism for placing load modules in the FPS-100 and initiating FPS-100 processing
- provides a mechanism for sharing data between the host machine and the FPS-100
- initializes data structures for use by the FPS-100 supervisor

LOD100 performs these functions by producing two types of output: the FPS-100 load module and the host-FPS-100 software interface mechanism (HASI).

1.6.1 FPS-100 LOAD MODULE

The FPS-100 load module contains the machine code and the data that are actually transferred to the FPS-100. In addition to this code and data, the load module contains instructions for their correct placement in the FPS-100 at run time. These instructions ensure that the appropriate machine code is placed in program source memory and the data and any overlays in main data memory. The user controls the overlay structure and the machine code destination with LOD100 commands. An FPS-100 program may be large enough to require several load modules; however, only one load module can be resident in the FPS-100 at a given time.

LOD100 can produce two types of load modules, host resident load modules and disk resident binary formatted load modules. Host resident load modules are created as host FORTRAN subroutines. The information which is transferred to the FPS-100 is contained in DATA statements. If the size of the host resident load module exceeds available host memory size, disk resident binary formatted load modules can be used.

1.6.2 HOST-FPS-100 SOFTWARE INTERFACE MECHANISM (HASI)

The HASI consists of host FORTRAN subroutines which correspond by name and by formal parameters to host callable FPS-100 subroutines. Each subroutine of the HASI acts as the link between a host FORTRAN CALL statement and the execution of the FPS-100 subroutine on the FPS-100. The user can direct LOD100 to create two types of HASI subroutines; one type contains auto-directed calls (ADC), and the other contains user-directed calls (UDC).

The UDC type of subroutine is available for compatibility with LNK100. Programs which were previously loaded with LNK100 can be loaded with LOD100 and run with no modifications. With UDC subroutines, the user must separately transfer data from host to FPS-100 and back. The user must also use the actual main data memory address values as arguments of the subroutines. This is the same as LNK100. However, new programs can include common blocks which are transferred by the UDC subroutines. If a routine does include common blocks, simultaneous processing of the host program and the FPS-100 routine does not occur as it would without the presence of the common block.

The ADC type of HASI subroutine is designed for calling FTN100 routines, though ADC subroutines can be created for ASM100 routines. With ADC routines, all argument values and common blocks are passed automatically. The user merely calls the FTN100 routine in the standard call-by-reference manner. Simultaneous processing of host and FPS-100 routines is not possible with ADC subroutines. The type of HASI subroutine created is determined by the entry point of the FPS-100 routine. If the entry point is designated with an FTN100 SUBROUTINE statement or an ASM100 $SUBR pseudo-op, an ADC routine is created. If the entry point is determined by an ASM100 $ENTRY pseudo-op, a UDC routine is created.

In either case, when a host FORTRAN call is made to an FPS-100 subroutine, control is passed to the subroutine of the same name in the HASI. This routine transfers the load module containing the FPS-100 routine to the FPS-100 (if it is not already present there), passes the common block values, passes the formal parameter values (if it is an ADC subroutine), and transfers control to the FPS-100. Upon return, the HASI subroutine retrieves the common block values and, if it is an ADC subroutine, formal parameter values, which it in turn passes to the calling program.

Routines in the HASI must be compiled with the host FORTRAN compiler and linked to the calling program using the host loader.

1.7 FPS-100 JOBS

An FPS-100 job is that portion of a user's program that runs on the FPS-100. A job may be contained on one or more load modules. It may reside entirely in program source memory or contain overlays which may also reside in main data memory. It may contain host callable routines or an entire FPS-100 supervisor environment. The following sections discuss overlays, the three types of jobs (single level jobs, multi-level jobs, and multiple load module jobs), and the supervisor environment.

1.7.1 OVERLAY SEGMENTS

All FPS-100 jobs execute from FPS-100 program source memory. The size of program source is then a limiting factor on the size of jobs. To permit execution of jobs larger than program source memory or to permit the FPS-100 supervisor to function, overlay segments can be used. An overlay segment is a segment of code that must reside in program source memory only when it is actually executing. Otherwise, it can be stored in main data memory. This allows program source memory to be shared, permitting larger jobs to exist or permitting the supervisor to interrupt and restore tasks.

Before a routine contained in an overlay segment can be called, that overlay must be transferred from main data memory to program source memory. If there is not enough room in program source memory, a logically independent overlay segment (one not on the same branch) is transferred from program source memory to main data memory in order to make room for the first overlay.

An overlay structure is similar to a tree with branches. An overlay segment can be dependent on and call overlay segments in the same branch. It must be logically independent of and cannot call overlay segments on a different branch. Since separate branches are logically independent, they can at different times occupy the same portion of program source memory. This permits a large program, if it contains logically independent segments, to be broken up into an overlay structure and executed in a smaller amount of program source memory.

Figure 1-1 illustrates an overlay branch structure. The job in this figure contains the root portion (ROOT) which always resides in program source memory when the remaining overlay segments execute. The first level overlays, A_1, B_1, and C_1, each require ROOT to be present but are independent of each other. In the next level, A_{11} and A_{12} are dependent on A_1 but independent of each other. The same is true for B_{11}, B_{12}, and B_1. In the third level, A_{111} and A_{112} are independent of each other, but each requires A_{11} which in turn requires A_1 and which in turn requires ROOT.

In other words, in order for a routine in an overlay to be executed, all overlays in the branch from that point to the root (in this case ROOT) must be present in program source memory.

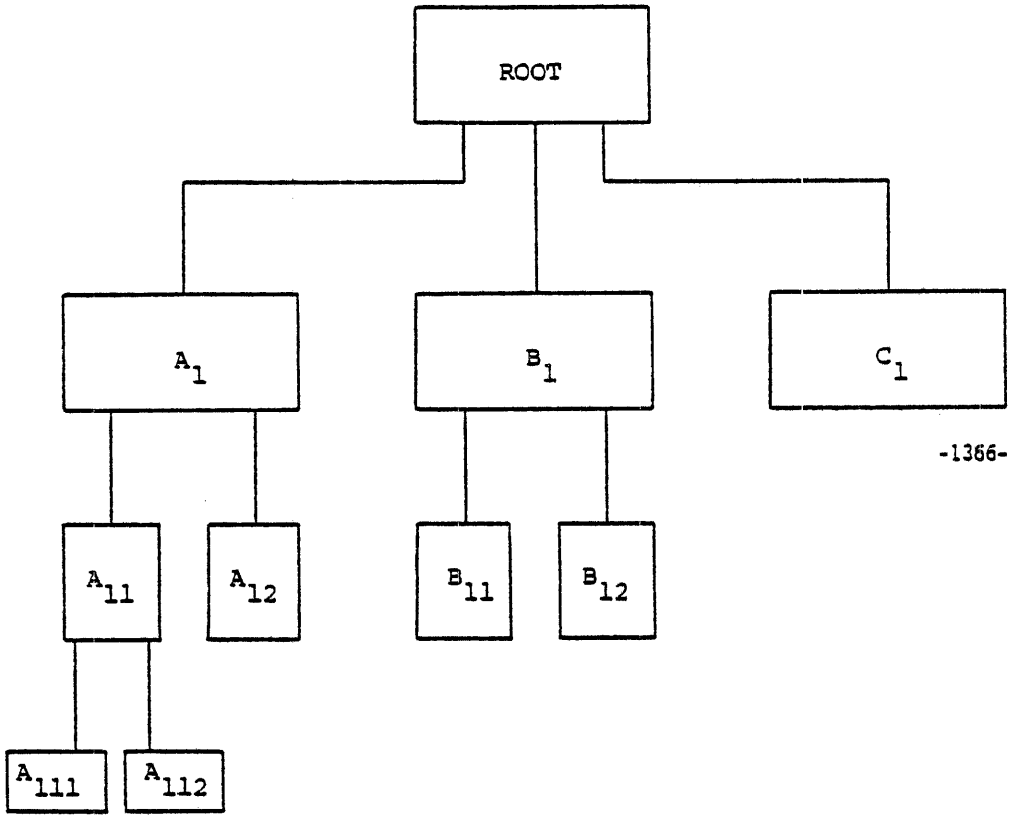


Figure 1-1 Overlay Branch Structure

Figure 1-2 illustrates the same overlay structure in terms of memory allocation. In this figure, overlay segments separated by vertical lines can occupy the same portion of program source memory. Overlay segments separated by horizontal lines are dependent on each other. The lower overlay segment requires the upper overlay segment to be present in program source memory in order for routines in the lower overlay segment to execute.

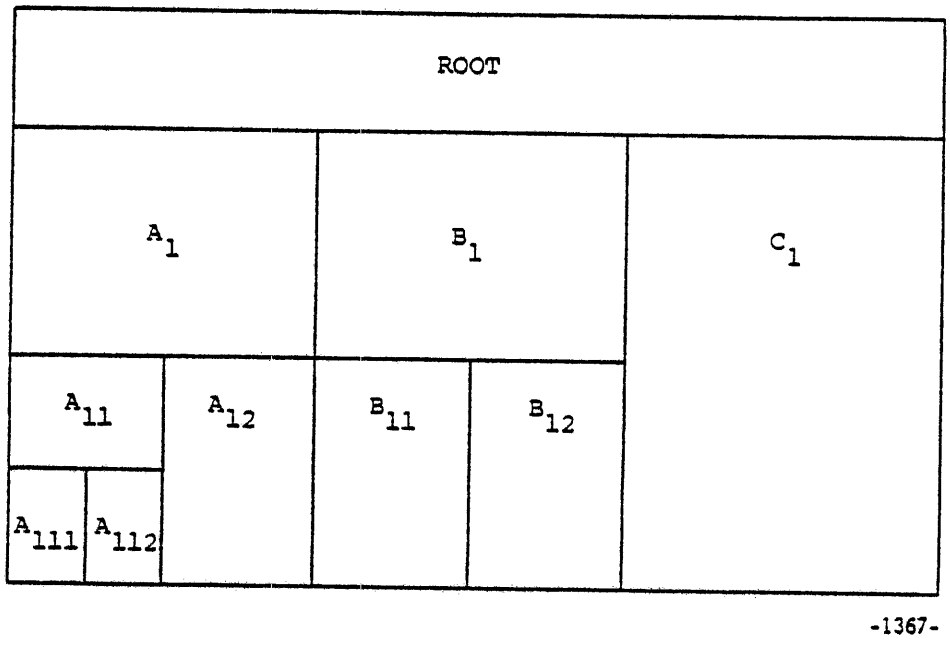


Figure 1-2 Overlay Memory Allocation

A job overlay structure is set up by the user with LOD100 commands described in Chapter 2. Each overlay segment resides in main data memory until it is copied into program source for execution. The number of main data memory words required for overlay storage is twice the number of program source memory words required for its execution.

1.7.2 SINGLE-LEVEL JOBS

The term single-level job implies a program without an overlay structure. A single-level job has no overlays and, therefore, must be small enough to fit in FPS-100 program source memory. Also, it is contained in a single load module. Due to this structure, any subroutine in a single-level job is capable of calling any other subroutine. In addition, all subroutines can be set up to be host callable.

Single level jobs cannot be used with the FPS-100 supervisor.

1.7.3 MULTI-LEVEL JOBS

A multi-level job is one in which an overlay structure is defined. This structure is defined by the user with LOD100 commands.

Any FPS-100 job that contains logically independent parts can be overlaid. Moreover, any job that is larger than program source memory must be structured into overlay segments (or multiple load modules, described in section 1.7.4). If the FPS-100 supervisor is used, all tasks must be structured into overlay segments. Only program code can be overlaid. Data areas (common blocks) always reside in main data memory and are accessible to all routines.

When the FPS-100 supervisor is not used, no restrictions exist on calling routines that reside in the same overlay segment. When calling a routine in a different overlay segment, the user must ensure that the overlay segment has been loaded into program source memory. An overlay segment can be loaded into program source memory by calling APOVLD, the overlay loader. APOVLD is described in section 4.3.1.5. If the overlay segment already resides in program source memory, APOVLD need not be called. When the FPS-100 supervisor is used, the previous discussion applies as long as all routines are in the same task. Routines in separate tasks cannot call each other.

When the FPS-100 supervisor is not used, all FPS-100 routines with one exception can be made host-callable. If multiple copies of a routine exist on the FPS-100, only one copy can be made host-callable. When the FPS-100 supervisor is used, only routines in the APX100 task can be made host-callable. Subroutines are declared host-callable with the LOD100 command CALL, which is described in section 2.3.11. With an option on the CALL command, the user can ensure that the overlay containing a host-callable subroutine is always loaded when the subroutine is called. This option causes the HASI routine for that subroutine to contain APOVLD calls. However, if the host-callable subroutine calls other routines not in the same overlay segment, it or the host calling program must contain APOVLD calls to load the additional overlay segment.

1.7.4 MULTIPLE LOAD MODULE JOBS

A multi-level job makes use of overlay segments to conserve program source memory. The savings in program source memory is counteracted by the increased use of main data memory and the overhead involved in maintaining the overlay segments at runtime. In certain large programs, there may not be enough space in main data memory to contain both data and overlay segments. If this is the case, a job that contains logically independent parts can be restructured into a multiple load module job, one that consists of two or more load modules. Each load module is then considered to be a logical job. Since only one load module can reside in the FPS-100 at a given time, this results in a possible savings of both program source memory and main data memory. However, if the FPS-100 supervisor is used, multiple load module jobs cannot be used.

When an FPS-100 subroutine is called from the host, the HASI associated with the subroutine makes certain that the load module containing the subroutine is loaded into the FPS-100. If an FPS-100 job includes multiple load modules, the user should try to avoid making calls to routines on alternate load modules since this results in the additional overhead of transferring load module data from host to FPS-100.

1.8 SUPERVISOR ENVIRONMENT

If an FPS-100 supervisor (MTS100 or RTS100) is used with the FPS-100, LOD100 not only loads host callable subroutines (and routines directly or indirectly called by them) but also loads the supervisor, its associated routines and common areas, and supervisor supplied and user created tasks and ISRs. When the supervisor is used, only single load module jobs are supported, since the load module consists not only of user routines but the supervisor as well. Also, all user routines must be loaded as part of a task or an ISR.

1.8.1 TASKS

Tasks, whether FPS-supplied or user written, are ASM100 or FTN100 routines which are not called or controlled by the host computer. Tasks are assigned priorities and execute on the basis of these priorities, under the direction of the supervisor. Task processing occurs in response to interrupts generated by peripheral devices, the host, or other tasks or ISRs.

Although some communication between the host and tasks is permitted (refer to the FPS-100 Supervisor Reference Manual for further information), all host callable routines must be loaded as a part of the FPS supplied APX100 task. The APX100 task is provided to allow user routines which run in an unsupervised FPS-100 to run under the supervisor without change. So, although the host does not call or control the APX100 task itself, it does control routines running as a part of it.

An overlay structure must be specified for each task, even if the task contains only one segment. In addition, the overlay structure for each task must be such that one and only one root (or top level overlay segment) can be specified.

1.8.2 INTERRUPT SERVICE ROUTINES

An ISR (interrupt service routine) services interrupts generated by external devices. This includes functions such as reading from or writing to the external device. A separate ISR must be provided for each external device connected to the FPS-100. ISRs are written in assembly code and must contain the $ISR pseudo-op.

1.8.3 TASK MODE

LOD100 is provided with a MODE command to facilitate the loading of FPS100 tasks and ISRs. The loader is placed in task mode by entering the command MODE TASK. In task mode, certain LOD100 commands also execute task building code whenever necessary. Although this task building is transparent to the user, the loader must be in task mode before tasks or ISRs can be loaded.

CHAPTER 2

LOD100 INPUT

2.1 INTRODUCTION

This chapter discusses communication with LOD100, describes the individual LOD100 commands, and outlines the procedure for creating single level, multi-level, and multiple load module jobs.

2.2 CALLING LOD100

To begin the load process, the user must call LOD100. This varies depending on the operating system, but usually involves entering:

```
LOD100
```

LOD100 responds by displaying:

```
LOD100  version date
*
```

In this case, version and date indicate the version of the loader and the date it was created. Additional information may also be displayed at this time. The asterisk (*) indicates that LOD100 is ready to accept user input; loader commands can now be entered. After LOD100 processes each command, it displays the asterisk to indicate that another command can be entered.

2.3 COMMANDS

The following sections describe the commands that are available with LOD100.

2.3.1 INPUT

This command indicates that a file, not input from the terminal, is used to specify LOD100 commands. LOD100 then processes the commands on that file. An end-of-file causes commands to be read from the terminal again. The format of this command is as follows:

```
INPUT filename

      or

INP filename
```

filename	This parameter specifies the file which contains LOD100 commands to be processed.

If an input file contains another INPUT command, commands are processed from the second input file. Subsequent input files can also be specified. However, when an end-of-file is encountered in any input file, control is transfered back to the terminal, not to any previously specified input file.

2.3.2 OUTPUT

This command specifies the LOD100 output files. The format of this command is as follows:

```
OUTPUT </size> hasifile lmfile-a </D
                                  lmfile-b/D>

                or

O </size> hasifile lmfile-a </D
                             lmfile-b/D>
```

/size — This optional parameter declares the size of the buffer used to transfer the load module at run time. This parameter should be a multiple of eight. If not, LOD100 uses the largest multiple of eight less than the specified number.

hasifile — This parameter specifies the file in which LOD100 writes the HASIs. A host FORTRAN HASI routine is created for each routine declared host-callable. Refer to sections 1.6.2 and 4.3 for a description of the HASI.

lmfile-i — This parameter specifies the file in which LOD100 writes the load module. A host resident load module, a binary formatted disk resident load module, or one of each can be specified. If one of each is specified, only the host resident load module can execute. The binary load module can only be used for debugging purposes. Refer to sections 1.6.1 and 4.2 for a description of the load module.

/D — If present, this parameter specifies that the associated load module is created as a disk resident binary load module. This is the only type of load module that can be used with SIM100/DBG100. If not specified, a host resident load module is created.

Unless an input file is being used, this command must be the first command entered in a session (except for the EXIT, HELP, INIT, MAP, MMAX, PMAX, MDOFF, PSOFF, and RADIX commands which can be entered at any time). If this command is entered more than once without reinitializing the loader, subsequent entries redefine hasifile only; lmfile-i and size cannot be redefined.

NOTE

On certain operating systems it is very difficult to use programmed file I/O. On these systems the user is required to assign files prior to calling LOD100. Then, after calling LOD100, all input and output files must be specified with logical unit numbers (in base 10) instead of file names.

Examples:

```
OUTPUT HASI LMOD1
```

In this example, a host resident load module LMOD1 is generated.

```
OUTPUT HASI LMOD2/D
```

In this example, a disk resident load module LMOD2 is generated.

```
OUTPUT HASI LMOD3 LMOD4/D
```

In this example, both a host resident load module LMOD3 and disk resident load module LMOD4 are generated. Normally, the HASI is set up to use either a host resident or a disk resident load module. However, when both types of load modules are generated with the same OUTPUT command, the HASI by default is set up to process the host resident load module. In this example then, load module LMOD3 executes when the HASI is called. LMOD4 can be used for debugging purposes.

2.3.3 RADIX

This command sets the radix for future user input. It can be entered at any point in the load sequence. The format of this command is as follows:

```
RADIX rad

    or

R rad
```

rad — This parameter specifies the radix for future input to LOD100. The following are acceptable values:

8	octal
10	decimal
16	hexadecimal

2.3.4 LMID

This command specifies the load module identification number of the current task. The format of this command is as follows:

LMID idnum

or

LM idum

idnum	This parameter specifies the load module identification number.

If specified, this command must be entered immediately after the OUTPUT command and not entered again unless the INIT command is specified. If this command is not specified, a load module identification number of one is implied.

2.3.5 MODE

This command notifies LOD100 that FPS-100 supervisor tasks are to be loaded during the session. LOD100 adjusts its internal flags so that the commands which follow execute task building code whenever necessary. The format of this command is as follows:

MODE TASK

or

MO TASK

This command should be issued before loading any tasks.

2.3.6 PRI

This command changes the priority parameters of an ASM100 task. It overrides the parameters specified with the $TASK pseudo-op in the ASM100 task. The format of this command is as follows:

PRI <priority> </I> </S>

priority — This parameter specifies the initial run priority and default priority of a task. Values between 1 and 255 can be specified with 255 the highest priority. If this parameter is not specified, a value of 100 is assumed.

/I — For the purpose of initializing the supervisor ready queue, this parameter indicates that the previously specified or default priority should be ignored and this task placed at the front of the ready queue. This optional parameter should normally be used only for I/O controller tasks, since it actually results in performing part of the system bootstrapping function (it causes the I/O controller tasks to be waiting for action before any user tasks start).

/S — This parameter, if entered, indicates that the priority of the task is slaved. Thus, when the task is activated, it acquires the priority of the activating task.

2.3.7 TASK

This command designates the next object module and the object modules which follow it to be loaded as a supervisor task. It provides the capacity to create FTN100 tasks and can only be used with FTN100 routines. If the next routine is an ASM100 object module which includes the $TASK pseudo-op, an error message is issued. The format of this command is as follows:

TASK idn </M> <priority> </I> </S>

idn — This is a 1- to 3-digit task identification number which LOD100 uses to create the task communication block (TCB) identifier. The TCB identifier is a common block with name TCBidn. So, for example, if a task is designated with an identification number of 5, the user can locate its TCB address by referencing the common block TCB005.

/M — If specified, the task uses minimal machine resources (only those saved in the minimum state save). If not specified, this task uses the full machine resources. This parameter is normally used for system tasks, such as I/O controller tasks, and should not be used for FTN100 tasks. This option can be used if the following registers are not needed:

```
s-pad registers 8-15
DPY write buffer
all DPX and DPY registers except DPX(0)-DPX(3)
DPA
floating adder
floating multiplier
flags
```

priority — This parameter specifies the initial run priority and default priority of a task. Values between 1 and 255 can be specified with 255 the highest priority. If this parameter is not specified value of 100 is assumed.

/I — For the purpose of initializing the supervisor ready queue, this parameter indicates that the previously specified or default priority should be ignored and this task placed at the front of the ready queue. This optional parameter should normally be used only for I/O controller tasks, since it actually results in performing part of the system bootstrapping function (it causes the I/O controller tasks to be waiting for action before any user tasks start).

/S — This parameter, if entered, indicates that the priority of the task is slaved. Thus, when the task is activated, it acquires the priority of the activating task.

2.3.8 TREE

This command is used to set up the overlay structure of the subsequent load module. This command must be entered before any of the OVERLAY commands are entered. The format of the TREE command is as follows:

TREE structure-spec

or

T structure-spec

structure-spec — This parameter specifies the complete overlay structure of the load module. Left parentheses, right parentheses, and overlay numbers can be entered. A left parenthesis indicates the start of a new overlay level; a right parenthesis rescinds a level. Overlay numbers between paretheses indicate the overlay segments that are contained in a particular overlay level. If structure-spec is too long to fit on one line, it may be continued on the next line.

Example:

A multi-level job consists of three overlay segments, one of which also contains two overlay segments. This is graphically described in Figure 2-1.

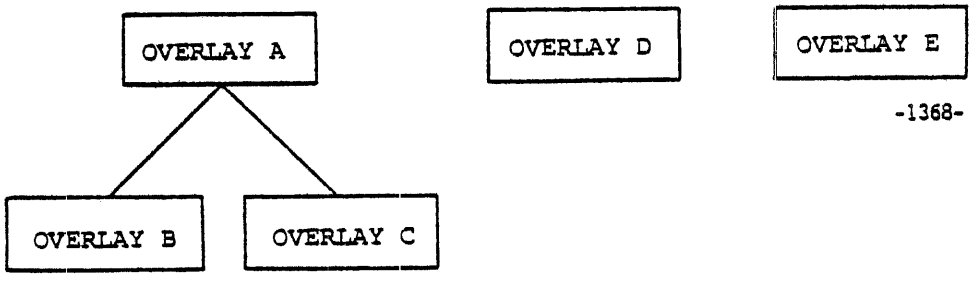


Figure 2-1 Overlays

A TREE command for this structure is as follows:

```
TREE ( (1 (2) (3) ) (4) (5) )
```

In order to create the load module with this overlay structure, the overlay segments have to be identified correctly (overlay segment A must be given an overlay number of 1, overlay segments B and C numbers 2 and 3, and so forth), and the overlay segments must be specified and loaded in the order indicated in the TREE command.

NOTE

The overlay structure shown in Figure 2-1 is illegal for tasks. Each task must have only one top level overlay segment.

2.3.9 OVERLAY

This command indicates that subsequently loaded code is part of an overlay segment. The format of this command is as follows:

```
OVERLAY ovnum

    or

OV ovnum
```

ovnum	This parameter specifies a number uniquely identifying the particular overlay segment.

OVERLAY commands are used to specify and load overlay segments in the order indicated in the TREE command. If the user attempts to load overlay segments in any other order, an error message is issued.

When the OVERLAY command is given and the overlay segment specified is not subordinate to the previous overlay segment specified (that is, it is on a different branch of the overlay tree), the user may notice a delay in the processing of the OVERLAY command. The delay occurs because, at this point, LOD100 links together the previous overlay branch up to the level of the new overlay segment. Also, any unsatisfied externals and NOLOAD designations that were entered with the previous overlay branch, up to the level of the new overlay segment, are cleared from the loader tables.

2.3.10 CALL

This command specifies routines as host-callable. It identifies the entry points of the routines which can be called from the host. (Entry points are declared in FTN100 routines with the SUBROUTINE statement and in ASM100 routines with the $ENTRY or $SUBR pseudo-ops.) LOD100 creates a HASI routine for each routine declared with this command. The format of this command is as follows:

```
CALL entry-a </>  entry-b </> ... entry-n </>

                    or

C  entry-a </>  entry-b </> ... entry-n </>
```

entry-i	This parameter specifies the entry point of a routine which is to be host-callable. In order to be host-callable, this entry point must be specified in the CALL command before being loaded with the LOAD command (refer to section 2.3.11). LOD100 creates host FORTRAN code for each routine declared in a CALL command and places it in the HASI.
/	If present, this indicates that the HASI subroutine created for the previous routine should contain an APOVLD call. (APOVLD calls are described in section 4.3.) This is done only if overlays are actually used in the load process.

If a routine contains multiple entry points and more than one is declared host-callable, an error message is issued.

If the FPS100 supervisor is used, only routines in the APX100 task can be specified as host-callable.

2.3.11 LOAD

This command identifies the object modules to be loaded. The format of this command is as follows:

LOAD filename

or

L filename

filename	This parameter specifies the file containing the object module or modules. Libraries or files containing embedded libraries can also be specified with this command. Routines in libraries are loaded only if they satisfy external references. Files containing embedded libraries can be loaded only if the routines that reference the embedded library occur in the file before the embedded library. Files containing embedded libraries must be structured in this manner because the LOAD command causes a one-pass load to be initiated. Also, since only one pass occurs, a library may have to be loaded more than once to ensure that all the proper externals are satisfied.

2.3.12 LIB

This command identifies libraries to be loaded. The LIB command causes as many load passes to occur on the library as are necessary to satisfy all externals that reference the library. Routines are loaded only if they satisfy external references. The format of this command is as follows:

LIB filename

filename	This parameter specifies the file containing a library of object modules. The file must start with a library start block and end with a library end block. Files containing embedded libraries cannot be loaded with the LIB command.

To minimize the number of passes needed to load a library, the library should be formatted so that routines which reference other library routines occur in the library before the referenced routines.

2.3.13 FORCE

This command forces certain routines to be loaded from a library if and when they are encountered during LOAD or LIB command execution, even if they are not needed to satisfy externals. This command does not affect libraries loaded previously. The format of this command is as follows:

```
FORCE routine-a   routine-b ... routine-n

                  or

F routine-a   routine-b ... routine-n
```

routine-i	This parameter specifies the routine which must be loaded when encountered in an object module.

2.3.14 NOLOAD

This command inhibits the loading of a routine when encountered in the load of a library. The specified routine is not loaded, even though it may satisfy an external reference. NOLOAD has no effect on routines loaded prior to the entry of this command. The NOLOAD function can be reversed later by entering the FORCE command. The format of this command is as follows:

```
NOLOAD routine-a   routine-b ... routine-n

                   or

NL routine-a   routine-b ... routine-n
```

routine-i	This parameter specifies a routine which is not loaded even if encountered in a library.

NOTE

If a library routine has more than one entry point, each must be declared with the NOLOAD command or the routine may still be loaded. In this case, the routine is loaded, but loader tables do not reflect those entry points declared with NOLOAD.

2.3.15 MDOFF

This command specifies the main data memory address at which LOD100 begins allocating memory space. This is reflected by a change in the value of DBBRK in the load map. It can be entered at any point during a load sequence. LOD100 issues a warning message if an overlay or data area extends across the specified address. The format of this command is as follows:

MDOFF address

or

MD address

address	This parameter specifies the main data memory address at which LOD100 begins allocation.

2.3.16 PSOFF

This command specifies the program source memory address at which LOD100 begins allocating memory space. This is reflected in a change in the value of PSBRK in the load map. It can be entered at any point during a load sequence. LOD100 issues a warning message if a program area extends across the specified address. The format of this command is as follows:

PSOFF address

or

PS address

address	This parameter specifies the program source memory address at which LOD100 begins allocation.

2.3.17 MMAX

This command defines the size of a main data memory page. It can be entered at any point during the load sequence. The format of this command is as follows:

MMAX address

or

MM address

address	This parameter specifies the highest main data memory address at which to load. A value from 0 through 65534 can be entered.

2.3.18 PMAX

This command defines the size of program source memory. It can be entered at any point in the load sequence. The format of this command is as follows:

PMAX address

or

PM address

address	This parameter specifies the highest program source memory address at which to load.

2.3.19 PPA

This command defines the size of the parameter passing area. The parameter passing area is used by the HASI for passing subroutine parameter values to the FPS-100. If this command is not specified, all main data memory which remains after the LINK command is specified is set aside for this use. The format of this command is as follows:

```
PPA size

    or

P size
```

size	This parameter specifies the size of the parameter passing area.

2.3.20 MARK

This command causes LOD100 to permanently define all presently defined common symbols and entry point symbols. Permanent symbols can be referenced by any user routine or task loaded during the session. If this command is not used, the definition of a new supervisor task causes these symbols to be lost. The format of this command is as follows:

```
MARK
```

2.3.21 PURGE

This command causes LOD100 to delete all common symbols and entry point symbols from the point of the last MARK command (or the beginning of the load session if no MARK command was issued). This command can be used to conserve space in the loader tables. The format of this command is as follows:

```
PURGE
```

2.3.22 MAP

This command causes LOD100 to generate a load map. It can be entered at any point in the load sequence. The format of this command is as follows:

```
MAP <option> <filename>

        or

M  <option> <filename>
```

option — This parameter indicates the type of map to be generated. One of the following can be specified for option.

option	description
0	complete load map
1	addresses of data blocks in main data memory
2	overlay addresses
3	undefined symbols
4	addresses of entry points in program source memory
5	current task map

If not specified, 0 (the complete map) is assumed.

filename — This parameter specifies the file on which LOD100 writes the load map. If this parameter is omitted, the map is displayed at the user terminal.

Figure 2-2 contains an example of a load map generated when MAP 0 is specified and tasks are not being loaded. Parts of this map can also be obtained by specifying options with the MAP command. The section entitled OVERLAY MAP is obtained by specifying option 2, DB MAP with option 1, PS ENTRIES with option 4, and UNDEFINED SYMBOLS with option 3.

```
OVERLAY MAP

    ID   PG  MDADDR     LEN    PSADDR  LEV BRL BRR
     1    0     531     400       144    0   0   2
     2    0    1162    1064       144    0   0   3
     3    0    4331     422       144    0   4   0
     4    0       0     350       355    1   0   0

DB MAP  0

    .APOVLD  000012   LBLCOM  000062   .SHOW    000463   .EXTRNL  000526
    .1       000531   DIST    001131   .SMOOTH  001133   .2       001162
    ARRAY    002246   IVECT   004216   .TEST    004312   .3       004331
    .BLANK.  004753   .ZER    006723   DBBRK    006731

PS TITLES

    PSLOW    000144   TEST    000144   ZER      000355   SPMUL    000523
    PSBRK    000541

PS ENTRIES

    PSLOW    000144   *TEST   000144   ZER      000355   SPMUL    000523
    PSBRK    000541

NO LOAD SYMBOLS

    DEF      000000

UNDEFINED SYMBOLS

    ABC      000000   JANSWR  000144
```

Figure 2-2 Load Map

The section of Figure 2-2 entitled OVERLAY MAP Lists and describes each overlay segment generated. The columns listed in the overlay map indicate the following:

ID	This indicates the identification number of the overlay segment. This numbering scheme was set up with the TREE command and assigned with the OVERLAY command.
PG	This indicates the page of main data memory on which the overlay segment is stored. Either the OUTPUT or OVERLAY command can be used to specify the page number.
MDADDR	This indicates the beginning address in main data memory where the overlay segment is stored.
LEN	This indicates the length of the overlay segment in main data memory words. This is twice the amount of program source words needed.
PSADDR	This is the beginning address in program source memory where the overlay segment is stored when transferred over from main data memory.
LEV	This indicates the overlay level. A 0 indicates that the overlay segment is not subordinate to any other overlays. A 1 indicates that it is on the second level of the overlay tree or subordinate to only one other overlay segment. Other numbers indicate levels in the same manner.

The BRL and BRR columns are used to indicate the overlay structure. This is done by providing a binary tree representation of the overlay structure. A binary tree is a unique representation of a tree structure in which each node (in this case, overlay segment) has at most two subordinate nodes. Since there is a unique binary tree for each tree structure, it is possible to determine the actual tree structure from the binary tree representation.

BRL — This indicates the ID number of the given overlay segment's left subroutine branch in the binary tree structure. When translating this into the actual overlay structure, the overlay segment indicated in the BRL column is immediately subordinate to the given overlay segment and is the leftmost subordinate overlay segment.

BRR — This indicates the ID number of the given overlay segment's right subordinate branch in the binary tree structure. When translating into the actual overlay structure, the overlay segment indicated in the BRR column is subordinate to the same overlay segment as the given overlay segment (is on the same level) and is positioned immediately to the right of the given overlay segment.

Figure 2-3 shows the binary tree structure of the overlays whose actual structure is shown in Figure 1-1.

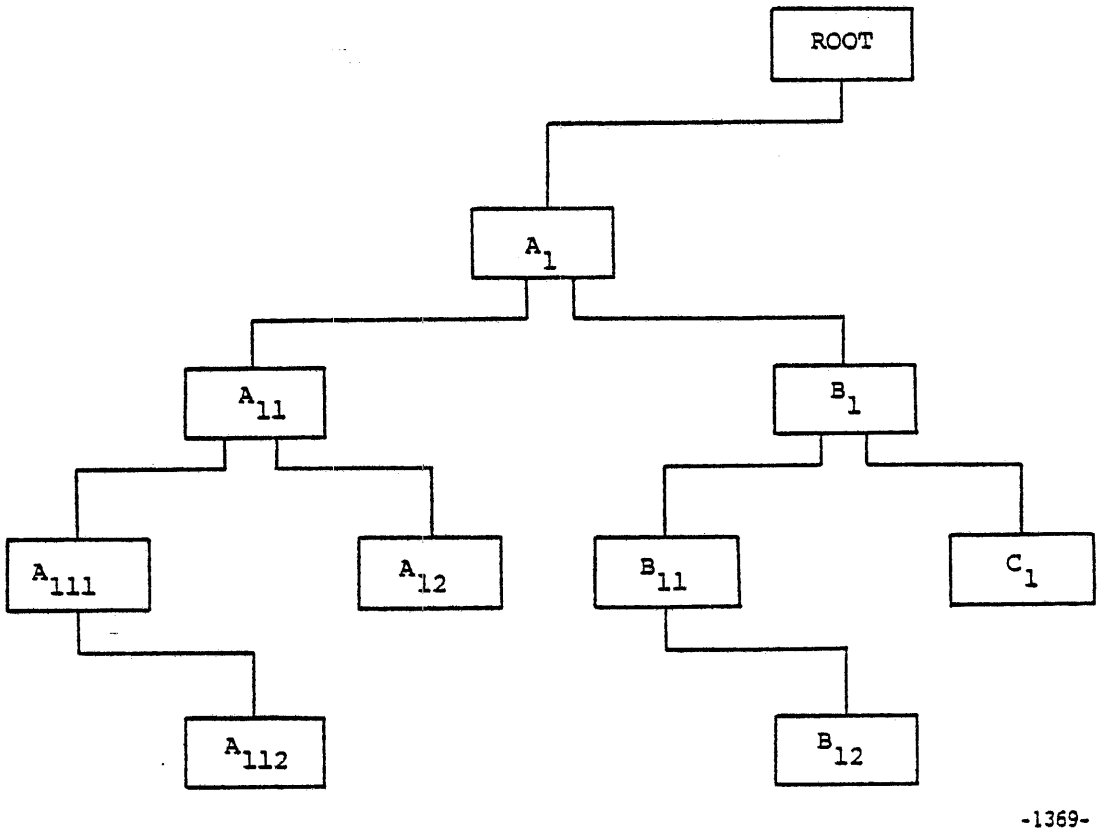


Figure 2-3 Binary Tree Structure

The section of the load map entitled DB MAP contains the names and the starting addresses in main data memory of the data blocks. Names beginning with a period have special meaning. For example, in Figure 2-2, the blocks .TEST, .ZERO, and .SMOOTH refer to three local data blocks in routines TEST, ZERO, and SMOOTH, respectively, each internally named .LOCAL. The block .BLANK. refers to a blank common area. The block .PPA. refers to the parameter passing area. The block .OVMAP. refers to the overlay map. Blocks such as .2 refer to overlays stored in main data memory. The number corresponds to the overlay number. The name DBBRK is always present in the data block map and indicates the next available location in main data memory.

The PS TITLES section of the load map is generated only when option 0 is specified. This section lists all routines loaded and their starting locations in program source memory. The names PSLOW and PSBRK are always present in the PS TITLES section. PSLOW indicates the first location in program source loaded, and PSBRK indicates the next available location.

The PS ENTRIES section of the load map lists the entry points of all routines loaded and their locations in program source memory. A routine's entry point may be the same as its starting location; however, for routines with multiple entry points, each entry point is listed in this section. Entry points preceded by an asterisk indicate host-callable routines. In Figure 2-2 entry point TEST is host-callable.

The NO LOAD section of the load map lists all symbols that were declared with the NOLOAD command.

The UNDEFINED SYMBOLS section of the load map lists all symbols which were referenced but not defined. The address associated with each symbol refers to the PS TITLES section and corresponds to the starting location of the routine which referenced the symbol. In Figure 2-2, the address associated with symbol JANSWR is the starting location of routine TEST as shown in the PS TITLES section. Routine TEST referenced JANSWR, but JANSWR is not defined.

If the user requires a final load map after loading all necessary routines, it should be generated after the LINK command has been entered. The final map then contains the address of the parameter passing area (.PPA.), the overlay map (.OVMAP.), and any necessary local data blocks which were not created by the routines themselves.

Figure 2-4 contains an example of a load map generated when MAP 0 is specified and tasks have been loaded. This map contains an additional section entitled TASK MAP. The task map can also be obtained by specifying option 5 of the MAP command.

```
OVERLAY MAP

    ID  PG  MDADDR     LEN  PSADDR  LEV BRL BRR

     1   0     536       2      22    0   2   0
     2   0     541      10      23    1   0   3
     3   0     551       2      23    1   0   0

TASK MAP

    ID  PRI  PSADDR     LEN  OPT RLINK LLINK

     6  255      21       1  M       7  1000
     7  144      22       5       1000     6

DB MAP  0

   READYQ   000001   ISRMAP   000003   VAL    000173   TCB006   000201
   TCB007   000312   .1       000536   A      000540   .2       000541
   .3       000551   .MP007   000553   .PPA.  000603   DBBRK    000603

PS TITLES

   PSLOW    000020   SYSCOM   000020   PSBRK  000027

PS ENTRIES

   PSLOW    000020   PSBRK    000027
```

Figure 2-4 Load Map with Tasks

The section of Figure 2-4 entitled TASK MAP lists and describes each task. The columns listed in the task map indicate the following:

ID	This indicates the identification number of the task. This number was assigned with the TASK command or the ASM100 pseudo-op $TASK.
PRI	This indicates the priority of the task. This priority was established with the TASK or PRI command or with the $TASK pseudo-op.
PSADDR	This indicates the beginning address of the task in program source memory.
LEN	This indicates the length of the task in program source words.

OPT — This indicates the option associated with the task. Possible values are M, I, and S which correspond to the /M, /I, and /S options of the TASK command (refer to section 2.3.7). If no value is listed, no option applies for the task.

RLINK — This indicates the right link pointer for the ready queue. This value is the task identifer of the next lower priority task. A value of 1000 indicates that this task is the highest priority task and points to the ready queue header (READYQ).

LLINK — This indicates the left link pointer for the ready queue. This value is the task identifier of the next higher priority task. A value of 1000 indicates that this task is the highest priority task and points to the ready queue header (READYQ).

The section of Figure 2-4 entitled DB MAP contains the same information described previously, except that only those symbols associated with the current task appear in the map unless a MARK command was issued earlier. Also, the block READYQ in Figure 2-4 indicates the ready queue header, and ISRMAP indicates the interrupt service routine map. Names such as TCB006 indicate task communication blocks of the corresponding tasks.

2.3.23 LINK

This command causes object code to be linked, relocated, and written to the load module. Space is also allocated for the TCB (if tasks are loaded), the overlay map, and any local data blocks that are needed by subroutines which did not declare them. The format of this command is as follows:

```
LINK

 or

LI
```

Object modules cannot be loaded after this command is entered unless task mode has been specified.

2.3.24 INIT

When not in task mode, this command re-initializes the loader. However, it does not affect the values set with the PMAX, MMAX, and RADIX commands. INIT can be entered at any point in the load sequence. In task mode this command performs the same functions as an EXIT and a call to LOD100. The format of this command is as follows:

```
INIT

 or

I
```

2.3.25 EXIT

This command closes all files and returns control to the operating system. It also initializes the remaining entries in the task overlay table, the partition table, and the ready queue and allocates space for the parameter passing area. It can be entered at any point in the load sequence. The format of this command is as follows:

```
EXIT

 or

EX
```

2.4 CREATING A SINGLE-LEVEL JOB

The following is the minimum sequence of commands needed to create a single-level job (the FPS-100 supervisor cannot be used in this job):

OUTPUT file1 file2	This defines the loader output files.
CALL routine1...	CALL commands indicate the routines that are to be host-callable. FORTRAN source code is written to the HASI for any routine specified on a CALL command.
LOAD object-module	This causes the user's object modules to be loaded.
LIB library	This loads necessary routines from libraries.
LINK	This causes LOD100 to link and relocate any object modules specified with LOAD commands and generate the resulting load module.
EXIT	This transfers control back to the operating system.

Example:

Assume that a load module is to be created containing routines CNTRL, A1, B1, and C1 and that only CNTRL is to be host-callable. The object modules of the routines reside on files of the same names. A library, APLIB, contains routines necessary to satisfy externals. The following command sequence can be used:

```
OUTPUT HSTFTN LM
 *
CALL CNTRL
 *
LOAD A1
 *
LOAD B1
 *
LOAD CONTRL
 *
LOAD C1
 *
LIB APLIB
 **LC
 *
LINK
 *
EXIT
```

The characters **LC displayed by LOD100 indicate that the load is complete and no more outstanding external references remain. The file HSTFTN now contains the host FORTRAN routine that is the HASI for CNTRL. The file LM contains the linked and relocated load module.

2.5 CREATING A MULTI-LEVEL JOB

The following is the general sequence of commands used to create a multi-level job without the FPS-100 supervisor:

OUTPUT file1 file2	This defines the loader output files.
TREE overlay-spec	This defines the overlay structure.
OVERLAY num	This indicates that the following routines are loaded as part of overlay segment num. Overlay segments must be loaded in the same sequence as specified on the TREE command.
CALL routine1...	
LOAD object-module	
FORCE entry1...	This forces the subsequent loading of any library routines that are needed by any of the lower level overlay segments.
LIB library1...	This loads any libraries needed.
.	
.	
.	
LINK	
EXIT	This transfers control back to the operating system.

Example:

In this example, assume that a load module is to be created with an overlay structure similar to the one described in section 1.7.1 and shown in Figure 1-1. The routines to be included in the load module are ROOT, the first level overlays (A1, B1, and C1), the second level overlays dependent on A1 (A11 and A12), the second level overlay dependent on B1 (B11 and B12), and the third level overlays dependent on A11 (A111 and A112). Each routine is an object module on a file of the same name. Only ROOT is host-callable. The Library APLIB contains routines necessary to satisfy externals. Figure 2-5 again shows the overlay structure and includes the overlay numbers which are assigned with the LOD100 commands.

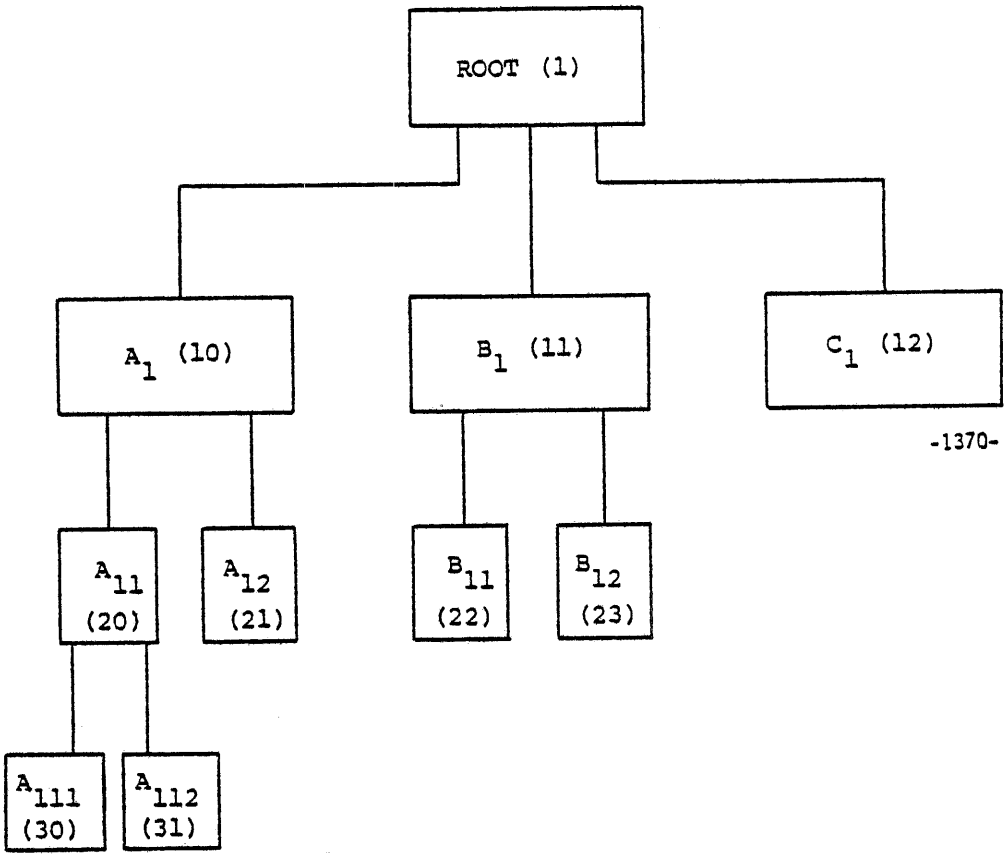


Figure 2-5 Overlay Structure Including Overlay Numbers

The following commands can be used to create the load module:

```
OUTPUT HASI LM
 *
TREE ((1 (10 (20 (30) (31)) (21)) (11 (22) (23)) (12)))
 *
OVERLAY 1
 *
CALL ROOT
 *
LOAD ROOT
 *
FORCE DIV SAVE
 *
```

```
LIB APLIB
 *
OVERLAY 10
 *
LOAD A1
 *
OVERLAY 20
 *
LOAD A11
 *
OVERLAY 30
 *
LOAD A111
 *
OVERLAY 31
 *
LOAD A112
 *
OVERLAY 21
 *
LOAD A12
 *
OVERLAY 11
 *
LOAD B1
 *
OVERLAY 22
 *
LOAD B11
 *
OVERLAY 23
 *
LOAD B12
 *
OVERLAY 12
 *
LOAD C1
 **LC
 *
LINK
 *
EXIT
```

At run time, either the HASI subroutine, the FPS-100 code, or the host FORTRAN calling program can load an overlay into program source memory by calling the overlay load routine (APOVLD) with an argument value corresponding to the overlay number. Refer to section 4.3.1.5 for a further description of APOVLD.

2.6 CREATING MULTIPLE LOAD MODULES

The following is the general sequence of commands used to create multiple load modules (multiple load modules cannot be used with the FPS-100 supervisor):

OUTPUT HASI1 LM1	This defines the loader output files for the first load module.
LMID 1	This identifies the load module.
load sequence	This defines a single-level or multi-level job.
LINK	This links the job.
INIT	This re-initializes the loader.
OUTPUT HASI2 LM2	This defines the loader output files for the second load module.
LMID 2	This identifies the second load module.
load sequence	
LINK	
.	
.	
.	
EXIT	

Any number of load modules can be defined. In the previous sequence, two are shown explicitly and assigned ID numbers 1 and 2. HASI1, LM1, HASI2, and LM2 contain their host FORTRAN source and load modules. The HASIs use these ID numbers to ensure that the load module associated with a given routine is loaded into the FPS-100. Therefore, the load modules should be assigned unique ID numbers.

2.7 LOADING TASKS

Tasks run in an FPS-100 supervisor environment continuously and without user or host direction. They respond to messages (interrupts generated by the host computer or external devices) by performing services. Due to the differences between tasks and other user routines, special load procedures are required for jobs containing tasks.

- Before tasks can be loaded, the FPS-100 supervisor and its associated system common areas and routines must be loaded. The MARK and PURGE commands can be used to ensure proper symbol definitions across task boundaries.

- Immediately before tasks are loaded during a session, the MODE TASK command should be entered to activate the special task building code in LOD100.

- A task can be a single routine or an entire overlay tree. However, even if the task is only a single routine, it must be loaded as an overlay segment. Also, tree structures with multiple top level overlay segments, such as one declared with the command TREE ((1) (2)), are illegal for tasks. A task must contain at least one overlay segment, but it can contain only one top level overlay segment.

- If any of the routines loaded are host callable, they must be loaded as part of the APX100 task. The root segment of this task is provided with the FPS-100 supervisor and permits routines which run on the unsupervised FPS-100 to run without change on the supervised FPS-100. The APX100 task should be set up with the supervisor-provided portion as the root segment of the overlay tree and the user routines as subordinate segments.

- Tasks may or may not share the same program source memory space. This is determined at load time with the PSOFF command. If two tasks are to share the same memory space, a PSOFF command should be given before loading each task with the same address specified.

Tasks can be written in ASM100 or in FTN100. Each type requires a slightly different loading sequence. These are described in the following sections.

2.7.1 LOADING ASM100 TASKS

Tasks written in ASM100 must be declared so with the $TASK pseudo-op. Therefore, the LOD100 command TASK cannot be specified when loading ASM100 tasks. However, the PRI command can be given to change the run priorities of the task. A typical sequence of commands to load a task is as follows:

OUTPUT filed file2	Define the loader output files.
PSOFF 0	Specify load location for supervisor.
LOAD system commons LOAD svc routines MARK LOAD remaining supervisor LINK	Begin system definition. Refer to the FPS-100 Supervisor Reference Manual for further information about loading supervisor routines.
MODE TASK	Inform loader that tasks are to be loaded.
PSOFF nnn	Specify load location for the task.
TREE (overlay structure)	Specify the overlay structure of the task (at least one overlay must be specified).
OV n	Indicate the first overlay segment to be loaded.
LOAD taskfile	Load the first overlay segment of the task.
PRI priority	Specify new task priorities, if required.
OV m LOAD file2 . . . LINK PSOFF nnn TREE (overlay structure) OV n	Load additional overlay segments, if necessary. Load APX100 task.

```
LOAD APX100 code                 Load the supervisor-provided APX100
                                 root segment.  Refer to the FPS-100
                                 Supervisor Reference Manual for
                                 further information.
PRI priority
OV m
LOAD host-callable routines      load host-callable routines (and
  .                              others, if necessary) in the user
  .                              specified overlay structure.
  .
LINK

  .                              Load additional tasks, if necessary.
  .
  .
EXIT
```

2.7.2 LOADING FTN100 TASKS

Unlike ASM100, there is no provision in the FTN100 language for declaring routines to be tasks. If tasks are written in FTN100, they can be declared as tasks in one of two ways.

- The user can write a one-line ASM100 program consisting of the pseudo-op $TASK, assemble it, and load it immediately before loading the file containing the FTN100 object code.
- The user can enter the LOD100 command, TASK, at load time to declare the task id and priority.

A typical sequence of commands to load an FTN100 task is as follows:

```
OUTPUT file1 file2               Define the loader output files.

PSOFF 0                          Specify load location for supervisor.

LOAD system commons              Begin system definition.  Refer to the
LOAD svc routines                FPS-100 Supervisor Reference Manual
MARK                             for further information about loading
LOAD remaining supervisor        supervisor routines.
LINK
MODE TASK                        Inform loader that tasks are to be
                                 loaded.
```

PSOFF nnn	Specify load location for the task.
TREE (overlay structure)	Specify the overlay structure of the task (at least one overlay segment must be specified).
OV n	Indicate the first overlay segment to be loaded.
TASK id options	Specify the task id and priority information. (This command is not necessary if the assembled one-line ASM100 program has been appended to the beginning of the FTN100 object module.
LOAD taskfile	Load the first overlay segment of the task.
OV m LOAD file2	Specify additional overlay segments if necessary.
. . .	
LINK PSOFF nnn TREE (overlay structure) OV n TASK id options LOAD file	Load additional tasks, if necessary.
. . .	
LINK TREE (overlay structure) OV n LOAD APX100 code	Load APX100 task. Load the supervisor-provided APX100 root segment. Refer to the FPS-100 Supervisor Reference Manual for further information.
LOAD host-callable routines . . .	Load host-callable routines (and others, if necessary) in the user specified overlay structure.
LINK . . .	
EXIT	

2.8 OVERLAY TABLE AND PS PARTITION TABLE

LOD100 creates overlay tables and a partition table which the FPS-100 supervisor uses to manage task usage of program source memory. Tasks consist of overlay segments which are transferred from main data memory to program source memory for execution. When the overlay segments are transferred to program source memory, the beginning address of each segment defines a new program source partition. A partition represents the smallest possible division of program source memory. An overlay segment may overlap several partitions, but there can be no more than one segment in a partition at any given time. Figure 2-6 illustrates this concept.

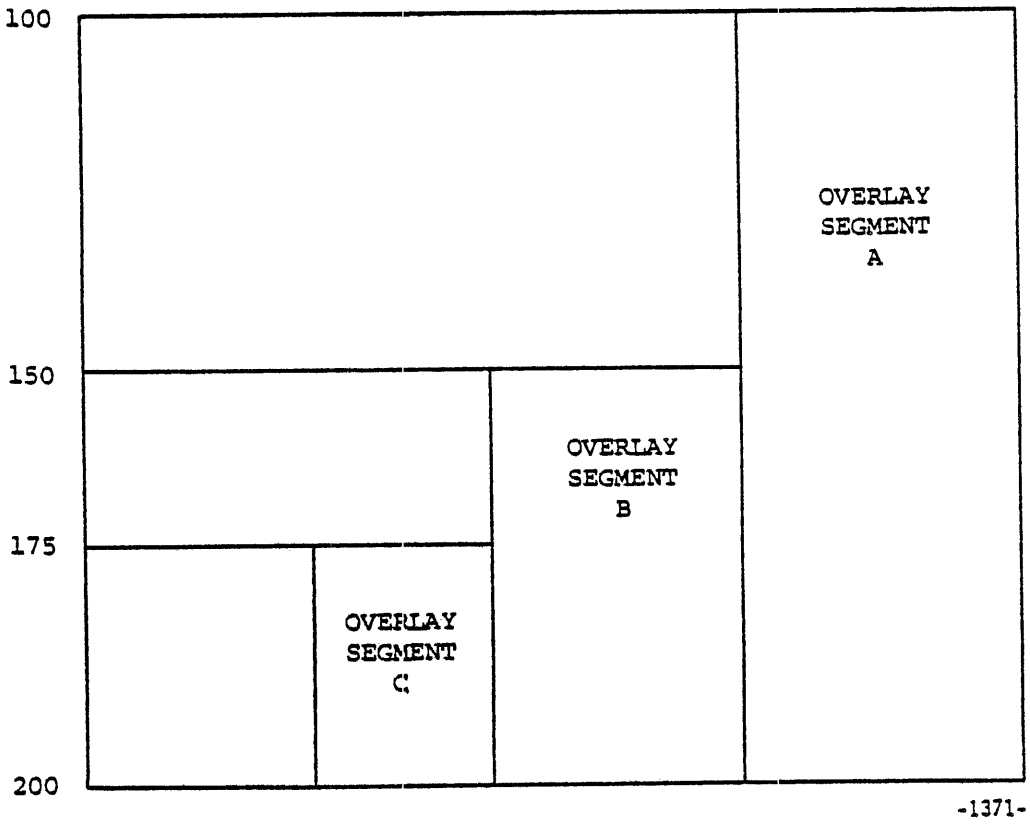


Figure 2-6 Overlay Segments

Overlay segment A in Figure 2-5 fits into PS locations 100-200. Segment B fits into locations 150-200, and segment C fits into locations 175-200. Therefore, PS partition 1 includes locations 100-150, partition 2 includes locations 150-175, and partition 3 includes locations 175-200. In terms of partitions of program source memory, overlay segment A starts at location 100 and contains 3 partitions. Overlay segment B starts at location 150 and contains 2 partitions. Overlay segment C starts at location 175 and contains 1 partition.

LOD100 creates an overlay table for each task in the system (.MPnnn where nnn is the task identifier) and one for the interrupt service routines (ISRMAP). An entry in the overlay table represents one overlay segment or ISR. Each entry consists of eight main data memory words. The format of an entry in the overlay table is shown in Table 2-1.

Table 2-1 Overlay Table Entry Format

WORD	PORTION (see NOTE)	CONTENTS
1	LM	overlay segment number
2	HM, LM	MD address
3	LM	PS address
4	LM	length (number of PS words)
5	LM	task id, name, or TCB address
6	EXP, LM	currently-resident bit (EXP) and should-be-resident-bit (LM)
7	LM	pointer to an entry in the PS partition table indicating the first partition this segment is loaded into
8	LM	Number of consecutive PS partitions this segment requires

NOTE

In this table, EXP refers to the exponent portion, HM the high mantissa portion, and LM the low mantissa portion of the main data word.

The PS partition table contains one entry for each PS partition. Each entry is one MD word long and contains either a zero (indicating that nothing currently resides in the partition) or a pointer to the entry in the overlay table representing the segment currently resident in the partition. (The pointer is actually the address of the residency word (word 6) in that segment's entry in the overlay table.)

LOD100 initializes the PS partition table to zeros. It initializes each word of the overlay table to the appropriate address, count, or id.

2.9 TASK COMMUNICATION BLOCK (TCB)

For each task the user defines, LOD100 creates a task communication block, a common block named TCBnnn (where nnn is the task identifier). A TCB contains status information and the execution context of the task which the FPS-100 supervisor updates and uses to interrupt and restore tasks during execution. Refer to the FPS-100 Supervisor Reference Manual for more information concerning the use of the TCB.

The TCB consists of a standard header, task relevant data, a minimum save area, and a maximum save area. Table 2-2 shows the format of a TCB. Each entry in the TCB consists of one MD word.

Table 2-2 TCB Format

WORD	CONTENTS
	HEADER
1	right link (RLINK)
2	left link (LLINK)
3	run priority (RPRI)
4	TYPE (unused for tasks)
5	TCB length (LENGTH)
6	answer exchange address (ANSKEY)
	TASK DATA
7	task identifier (ID, number, or name)
8	address in the overlay table of the entry of the first segment of this task (OVLPTR)
9	number of overlay segments in this task (OVLCNT)
10	default priority (DPRI)
11	task status bits (STATUS)
12	address of the last message received (LSTMSG)
13	right clock link (RCLOCK)
14	left clock link (LCLOCK)
15	delta time interval (ICLOCK)
16	this task's beginning MD address (TADDR)
17-20	reserved for future expansion
	MINIMUM SAVE AREA
21-23	MD FIFO
24	data pad bus (DPBS)
25	DPX write buffer
26	status register (APSTAT)
27-31	DPX(0) - DPX(3)
32	device address (DA)
33	s-pad destination (SPD)
34	s-pad 0
35	s-pad function (SPFN)
36-42	s-pad(1) - s-pad(7)
43	status register 2 (APSTAT2)
44	table memory address (TMA)
45	table memory register (TMREG)
46	FFT status bits
47	user memory address
48	status register 3 (APSTAT3)
49-64	subroutine return stack (SRS)

Table 2-2 TCB Format (cont.)

WORD	CONTENTS
	MAXIMUM SAVE AREA
65-73	s-pad(8) - s-pad (15)
74	DPY write buffer
75-78	DPY(0) - DPY(3)
79-134	remaining data pads
135	data pad address (DPA)
136-140	floating adder
141-146	floating multiplier
147-150	flags

Upon exit, LOD100 initializes each TCB as follows:

- RLINK and LLink are initialized as described in section 2.10 to set up the ready queue.
- RPRI and DPRI are set to the priority specified in the PRI command. If the PRI command was not specified, they are initialized to the value specified in the $TASK pseudo-op. If that was not specified, the values are obtained from the TASK command.
- OVLPTR is set to the address of the entry in the overlay table representing the first segment of this task.
- OVLCNT is set to the number of overlay segments in this task.
- TCBID is set to the task identification number obtained from the $TASK pseudo-op. If the $TASK pseudo-op was not specified, the value is obtained from the TASK command.

- STATUS is set as follows:

  004000 This bit is set unless the /M option was specified on the $TASK pseudo-op.

  010000 This bit is set if the /S option was specified in the PRI command. If the PRI command was not given, the $TASK pseudo-op is checked. If $TASK was not specified, the TASK command is checked.

  001000 This bit is set if the ready queue is implemented as described in section 2.10.

- RCLOCK and LCLOCK are set to the address of RCLOCK.
- APSTAT2 is set to disable floating-point exception interrupt.
- APSTAT3 is set to 55260 octal.
- SRS(0) is set to the address of the termination routine.
- SRS(1) is set to the program source address of the task.
- LENGTH is set to the number of main data words in the TCB.
- TADDR is set to the main data address of the task's first segment.
- All other items are initialized to zero.

2.10 READY QUEUE

In order for the ready queue to function, a system common area named READYQ must be defined as the header of this queue. In this case, RLINK of the ready queue header is set by LOD100 to point to the TCB address of the task with the highest priority, and LLINK of the ready queue header is set to point to the TCB address of the task with the lowest priority. RLINK and LLINK of the highest priority task are set to the point to the second highest task's TCB and the ready queue, respectively. RLINK and LLINK of the second highest priority task are set to point to the TCB of the third highest priority task and the TCB of the highest priority task, respectively. This pattern continues for all TCBs.

When READYQ is defined, LOD100 sets the ready queue bit in each TCB STATUS word as noted in section 2.9.

Tasks which have the /I option specified are placed at the front of the ready queue, ahead of the highest priority task, regardless of their own priorities.

CHAPTER 3

OBJECT MODULES

3.1 INTRODUCTION

The relocatable object modules produced by both the ASM100 assembler and the FTN100 compiler, which are used as input to LOD100, consist of numbers written as octal characters. Unlike most relocatable binary code, this code can be displayed at a terminal and edited with an ordinary text editor.

The relocatable object code is divided into a series of blocks. The order in which the blocks appear, if each type is present, is generally as follows (the octal block type number is in parentheses):

1. library start block (6)
2. task block (15)
3. ISR block (16)
4. title block (3)
5. data block descriptor blocks (10)
6. parameter block (12)
7. data block initialization blocks (11)
8. alternate entry block (13)
9. entry block (4)
10. code blocks (0)
11. external block (5)
12. end block (1)
13. library end block (7)

An object module must contain a title block and an end block. The presence and ordering of other types of blocks depends on the particular program.

The first line of each block is a block header, which describes the remainder of the block. The block header is easily identified because it contains the characters "***" followed by the name of the block. The remainder of the block contains data records.

Blocks are described in the following paragraphs, in order of their block type numbers (again, in octal).

3.2 CODE BLOCK (0)

Header:

0 count location ***CODE

count	This specifies the number of data records that follow.
locations	This specifies the address relative to the start of the routine where the code is loaded.

Data record:

* code-a code-b code-c code-d flddes-i type-i arg-i ...flddes-n type-n arg-n

*	The asterisk at the beginning of the line is optional, but is present if any field of the instruction is to be relocated or contains an external reference.
code-a through code-b	These are four 16-bit unsigned octal numbers. They make up the code for one FPS-100 instruction word.

The optional triples at the end of the data record are used to define the fields in the instruction word that are to be relocated.

flddes-i — This is the field designator, specifying which field to relocate. Possible values are:

0 value field

type-i — This specifies the type of relocation. Possible values are:

1 program source relocatable
2 external reference (absolute)
3 DB reference
4 relocation via the .LOCAL block of a subroutine
5 external reference (relative)

arg-i — The value of arg depends on the type specification. If type is 2, 4, or 5, arg-i specifies an external. If type is 3, arg-i specifies a data block. If type is 1, arg-i is ignored.

3.3 END BLOCK (1)

Header:

1 ***END

Data record:

title

title — This specifies the title of the routine (the same as appears in the title block).

3.4 TITLE BLOCK (3)

Header:

```
3    ***TITLE
```

Data record:

```
title
```

title	This specifies the title of the routine.

3.5 ENTRY BLOCK (4)

Header:

```
4   count   ***ENTRY
```

count	This specifies the number of data records that follow.

Data record:

```
symbol   value   type   paramnum
```

symbol	This is a six-character entry symbol.
value	This specifies the value of the symbol. If the symbol is relocatable, this value is relative to the start of this routine.
type	This indicates the type of symbol. Possible values are: 0 absolute (produced by the $GLOBAL pseudo-op in ASM100) 1 relocatable (produced by FTN100 or the $SUBR pseudo-op in ASM100)
paramnum	This indicates the number of parameters associated with the entry point. It is not present if type is 0.

3.6 EXTERNAL BLOCK (5)

Header:

```
5   count   ***EXT
```

count	This specifies the number of data records that follow.

Data record:

```
symbol
```

symbol	This is a six-character external symbol name.

3.7 LIBRARY START BLOCK (6)

Header:

```
6   ***LSB
```

3.8 LIBRARY END BLOCK (7)

Header:

```
7   ***LEB
```

3.9 DATA BLOCK DESCRIPTOR BLOCK (10)

Header:

```
10   count   symbol   dest    ***DBDB
```

count — This specifies the number of data records that follow.

symbol — This specifies the data block name, which corresponds to a labeled common block or .LOCAL for the local data block.

dest — If the subroutine is declared host callable, this defines how the data is transferred to and from the FPS-100. Possible values are:

1 data is transferred to the FPS-100 only
2 data is transferred from the FPS-100 only
3 data is transferred both to and from FPS-100

Data record:

```
type   number
```

type — This indicates the type of item contained in the data block. Possible values are:

1 integer
2 real

number — This specifies the number of elements in the item. For scalars, this value is 1, for arrays it is the number of elements in the array.

NOTE

If two or more consecutive items of the same type occur in a data block, they are combined in one data record.

3.10 DATA BLOCK INITIALIZATION BLOCK (11)

Header:

```
11   count   ***DBIB
```

count — This specifies the number of data records that follow.

Data record:

```
id  location  type  rc  value-a  value-b  value-c  flddes  type  arg
```

id — This specifies the number of the corresponding data block descriptor block where items are to be initialized.

location — This specifies the location (relative to the start of the data block) to be initialized.

type — This indicates the type of data to be initialized. Possible values are:

1 integer (if the data item referenced a relocatable symbol, 20 octal is added)
2 real
4 double precision integer (if the data item referenced a relocatable symbol, 20 octal is added)

rc — This is a repetition count. If greater than 1, it indicates that a number of locations are to be initialized to the same value.

value-a — This specifies the value to be initialized for data items of types 1 and 2. For type 4, this specifies bits 0 through 5 of the value to be initialized.

value-b — This appears only for type 4 and specifies the bits 6 through 21 of the value to be initialized.

value-c — This appears only for type 4 and specifies the bits 22 through 37 of the value to be initialized.

The parameters flddes, type, and arg are the same optional triple that was described in section 3.2. This optional triple appears when the data item references a relocatable symbol.

3.11 FORMAL PARAMETER BLOCK (12)

Header:

```
12   count   ***FPB
```

count — This specifes the number of data records that follow.

Data record:

```
type   dest   size
```

type — This indicates the type of the parameter. Possible values are:

1 integer
2 real

dest — If the subroutine is declared host-callable, this defines how the data is transferred to and from the FPS-100. Possible values are:

1 data is transferred to the FPS-100 only
2 data is transferred from the FPS-100 only
3 data is transferred both to and from the FPS-100

size — This specifies the dimensionality of the parameter (0 indicates a scalar, 3 indicates a three-dimensional array, and so on).

Data sub-record:

```
p-a     p-b
```

p-a — This indicates a static or dynamic dimension. Possible values are:

- 0 dimension is static
- 1 dimension is dynamic

p-b — If p-a is 0, this parameter indicates the static range. Otherwise, it specifies a formal parameter whose value dynamically defines the dimension at run time.

There is one data sub-record for each dimension.

3.12 ALTERNATE ENTRY BLOCK (13)

Header:

```
13   count   ***AENTRY
```

count — This specifies the number of data records that follow. (This block is produced by the $AENTRY pseudo-op in ASM100).

Data record:

```
symbol   value   type   spad
```

symbol — This is a six-character entry symbol.

value — This specifies the value of the symbol.

type — This indicates the type of symbol. Possible values are:

- 0 absolute
- 1 relocatable

spad — This specifies the number of s-pad parameters.

3.13 TASK BLOCK (15)

Header:

```
15   ***TASK
```

Data record:

```
id    priority    m-opt    i-opt    s-opt
```

id	This indicates the task identification number.
priority	This indicates the initial run priority or the default priority of the task.
m-opt	This indicates the amount of machine resources used. Possible values are: 0 full machine resources 1 minimum machine resources
i-opt	This indicates the initial priority option. Possible values are: 0 same as default priority 1 insert at front of ready queue
s-opt	This indicates the slave option. Possible values are: 0 task is not slaved 1 task is slaved

3.14 ISR BLOCK (16)

Header:

```
16   index   ***ISR
```

index	This indicates the number of the I/O device which the ISR services.

3.15 SAMPLE OBJECT MODULE

Figure 3-1 contains an object module which was generated by compiling an ASM100 subroutine.

```
     3       ***TITLE
ADCSFT
    12       1       ***PB
     1      1      0
    10       1 .LOCAL    7  ***DBDB
     1          1
    10       1 COMA      1  ***DBDB
     2          1
     4       1       ***ENTRY
ADCSFT       0       1       1
     0      17       0       ***CODE
*       3 102000    2000       0       0       3       1
        0       0       0       0
        0       0       0       0
        3 102000    5000       0
        0       0       0       0
        0       0       0       0
     1604       0    5000       0
     1670       0    2000      33
    40104       0   46004       0
    41671  156000     400       0
        1 100000       0       0
*    1664       0    2000       0       0       3       2
        0       0  100005       0
    41564       0    3500     360
        0     340       0       0
     1       ***END
ADCSFT
```

Figure 3-1 Sample Object Module

CHAPTER 4

OUTPUT FROM LOD100

4.1 INTRODUCTION

LOD100 generates two different output files: the load module and the HASI. The load module consists of the machine code and data that are actually loaded into the FPS-100, as well as some placement instructions. Depending on the OUTPUT command, the load module is stored in one of two forms, either as a host resident load module or as a disk resident, binary formatted load module. The HASI consists of host FORTRAN subroutines that correspond in name and in formal parameters to the FTN100 or ASM100 subroutines that were declared host-callable with LOD100. These subroutines control the placement of the load modules on the FPS-100 and the start of FPS-100 execution. This chapter describes the load module and the HASI.

4.2 LOAD MODULE

The load module is divided into a series of blocks. The following types of blocks can be present (block type numbers are in parentheses):

- code block or integer data (0)
- data block (1)
- information block (2)
- end block (3)

Each block consists of an eight-word header record followed by data records which can be of any size. Each word contains a decimal number evaluated as a 16-bit 2's complement integer or 32-bit 2's complement integer, depending on the host computer.

4.2.1 CODE/OVERLAY/32-BIT MD DATA BLOCK (0)

Header:

```
0   count   addr   pg   dest   0   0   0
```

count	This specifies the number of integers in the data record.
addr	This specifies the address in the FPS-100 where code should be loaded.
pg	If main data memory is loaded, this parameter specifies the page of main data to load.
dest	This indicates the memory destination. Possible values are: 0 program source memory 1 main data memory

Data record:

This record contains a block of integers (representing microcode to be placed in program source memory or main data memory) specified by count. If the host machine's word size is 32 bits or greater, each integer represents a 32-bit integer (1 PS word = 2 host words). Otherwise it represents a 16-bit integer (1 PS word = 4 host words).

4.2.2 DATA BLOCK (1)

Header:

```
1   count   0   pg   0   0   0   0
```

count	This specifies the number of data records that follow.
pg	This specifies the page of main data to contain the data values.

Data record:

```
type rc addr 0 value-a value-b value-c value-d
```

type	This indicates the type of number. Possible values are: 1 16-bit integer 2 single precision host real 4 double precision integer (38-bit)
rc	This specifies the number of repetitions of the data item (from the DATA statement in FTN100).
addr	This specifies the main data address where the value of the data item is to be placed.
value-a through value-d	This specifies the data value, which varies according to the type. For type = 1: value-a contains 16-bit integer value For type = 2: value-a contains 16-bit host real

For type = 4:

value-a contains bits 0-5 of double precision integer

value-b contains bits 6-21

value-c contains bits 22-37

4.2.3 INFORMATION BLOCK (2)

Header:

2 ppaad ppasz lmid ovlen ovaddr 0 0

ppaad — This specifies the address of the parameter passing area for this load module.

ppasz — This specifies the length of the parameter passing area.

lmid — This specifies the identification number of this load module.

ovlen — This specifies the length of the overlay map in main data memory.

ovaddr — This specifies the starting location in main data memory of the overlay map.

NOTE

Values appear for ovlen and ovaddr only if LOD100 is not operating in task mode. Also, there may be multiple occurrences of the information block header. Only the last one is meaningful.

4.2.4 END BLOCK (3)

Two types of end blocks are generated by LOD100, a logical end block and a terminating end block. The format of the logical end block is as follows:

Header:

```
3   0   0   0   0   0   0   0
```

This block indicates the logical end of the load module.

The format of terminating end block is as follows:

Header:

```
3   1   0   0   0   0   0   0
```

This block appears as the last record in the module.

4.2.5 SAMPLE LOAD MODULE

LOD100 can generate two types of load modules, a host resident load module and a binary formatted disk resident load module. The host resident load module is produced by default. Specifying the /D option on the OUTPUT command generates a binary load module.

A host resident load module is stored in the form of a FORTRAN subroutine consisting primarily of data statements. Figure 4-1 contains a sample. However, the exact format of the load module varies according to the host FORTRAN compiler. For example, some compliers put restrictions on the number of data statements. In this case, LOD100 creates more than one subroutine in the load module.

A binary formatted load module contains the information described previously in this chapter but is stored in binary format. An example of a binary formatted load module, because of its very nature, cannot be shown.

```
 SUBROUTINE L 101
 INTEGER CODE(   200)
   DATA CODE(    1),CODE(    2),CODE(    3),CODE(    4)/
*     0,    60,    16,     0/
   DATA CODE(    5),CODE(    6),CODE(    7),CODE(    8)/
*     0,     0,     0,     0/
   DATA CODE(    9),CODE(   10),CODE(   11),CODE(   12)/
*     3,-31744,  1024,     1/
   DATA CODE(   13),CODE(   14),CODE(   15),CODE(   16)/
*     0,     0,     0,     0/
   DATA CODE(   17),CODE(   18),CODE(   19),CODE(   20)/
*     0,     0,     0,     0/
   DATA CODE(   21),CODE(   22),CODE(   23),CODE(   24)/
*     3,-31744,  2560,     0/
   DATA CODE(   25),CODE(   26),CODE(   27),CODE(   28)/
*     0,     0,     0,     0/
   DATA CODE(   29),CODE(   30),CODE(   31),CODE(   32)/
*     0,     0,     0,     0/
   DATA CODE(   33),CODE(   34),CODE(   35),CODE(   36)/
*   900,     0,  2560,     0/
   DATA CODE(   37),CODE(   38),CODE(   39),CODE(   40)/
*   952,     0,  1024,    27/
   DATA CODE(   41),CODE(   42),CODE(   43),CODE(   44)/
* 16452,     0, 19460,     0/
   DATA CODE(   45),CODE(   46),CODE(   47),CODE(   48)/
* 17337, -9216,   256,     0/
   DATA CODE(   49),CODE(   50),CODE(   51),CODE(   52)/
*     1,-32768,     0,     0/
   DATA CODE(   53),CODE(   54),CODE(   55),CODE(   56)/
*   948,     0,  1024,     2/
   DATA CODE(   57),CODE(   58),CODE(   59),CODE(   60)/
*     0,     0,-32763,     0/
   DATA CODE(   61),CODE(   62),CODE(   63),CODE(   64)/
* 17268,     0,  1856,   240/
   DATA CODE(   65),CODE(   66),CODE(   67),CODE(   68)/
*     0,   224,     0,     0/
   DATA CODE(   69),CODE(   70),CODE(   71),CODE(   72)/
*     2,     3,    -5,     1/
   DATA CODE(   73),CODE(   74),CODE(   75),CODE(   76)/
*     0,     0,     0,     0/
   DATA CODE(   77),CODE(   78),CODE(   79),CODE(   80)/
*     3,     0,     0,     0/
   DATA CODE(   81),CODE(   82),CODE(   83),CODE(   84)/
*     0,     0,     0,     0/
 CALL FSLMLD(    1,CODE)
 RETURN
 END
```

Figure 4-1 Sample Load Module

4.3 HASI

Two types of HASI routines can be generated by LOD100. The ADC (auto directed calls) type of HASI routine is designed for calling FTN100 routines. All passing of parameter values and common blocks is handled automatically in this routine. The user merely calls the subroutine like any other FORTRAN subroutine.

The second type of routine, the UDC (user directed calls) routine, is designed to be compatible with previous programs. The basic difference between this and the ADC routine is that the sections controlling the passing of parameters are missing in the UDC routine. This method results in faster execution, but the user must place data in the FPS-100 and pass the actual main data address values as the subroutine arguments.

4.3.1 FPS-100 EXECUTIVE ROUTINES

HASI subroutines are written in host FORTRAN but contain calls to FPS-100 executive routines. These routines are described in the APX100 Manual; however, certain of these routines are especially important to the HASI subroutines. They are described in the following paragraphs.

4.3.1.1 APLMLD

This routine is called to load a given binary formatted load module into the FPS-100. This must be done prior to calling a routine in that load module. APLMLD is normally called automatically by the HASI subroutine; however, the user can also call this routine. The format of the call is as follows:

```
CALL APLMLD (id, array, size)
```

id	This parameter specifies the identification number of the load module to be moved. This identification number was originally established with the LOD100 LMID command.

array	This parameter specifies the name of an array which contains the binary formatted load module, read from disk.
size	This specifies the size of the load module array.

4.3.1.2 FSLMLD

This routine is called to load a given host resident load module into the FPS-100. This must be done prior to calling a routine in that load module. FSLMLD is normally called automatically by the host resident load module; however, the user can also call this routine. The format of the call is as follows:

```
CALL FSLMLD (id, array)
```

id	This parameter specifies the identification number of the load module to be moved. This number was originally established with the LOD100 LMID command.
array	This parameter specifies an array name. This array contains the load module to load into the FPS-100.

4.3.1.3 APRUN

This routine is called by the HASI subroutine to transfer control to a subroutine in the FPS-100. The format of the call is as follows:

```
CALL APRUN(psaddr,locadr,option,ovflg,brkloc)
```

psaddr — This parameter specifies the program source memory address of the subroutine's entry point.

locadr — This parameter specifies the main data memory address of the subroutine's local data block. (This parameter is used for ADC-type calls only.)

option — This parameter indicates the type of HASI routine. Possible values are:

1 ADC
2 UDC

ovflg — This parameter specifies whether or not overlays are used. Possible values are:

0 overlays are not used
1 overlays are used

brkloc — This parameter specifies the program source memory address at which a breakpoint should be set.

4.3.1.4 APLLI

This routine is called by the user to specify the file containing the binary formatted load modules. APLMLD uses this information when loading the load modules. The format of this call is as follows:

```
CALL APLLI('lmfile',n,logunit,1,lmid,dummy,dummy)
```

lmfile	This parameter specifies the name of the file that contains the load module and is specified in FORTRAN hollerith format.
n	This parameter specifies the number of characters in the file name lmfile.
logunit	This parameter specifies the host operating system logical unit number which may be used to assign the file lmfile. If multiple load modules are being used, and a logical unit appears more than once, the first logical unit is closed. That is, no subroutine in the first load module can be accessed until APLLI is called again for that load module.
lmid	This parameter specifies the load module identifier. This identifier was specified with the LOD100 command LMID. If not specified with the LMID command, LOD100 assigns a default value of 1. In this case, a 1 should be specified for lmid.
dummy	This is a dummy parameter which is not used but must be supplied.

4.3.1.5 APOVLD

This routine is called by the user's host FORTRAN program, the FTN100 or ASM100 subroutine, or the HASI to load an overlay into program source memory. The format of the FORTRAN call is as follows.

```
CALL APOVLD(id)
```

id	This parameter specifies the overlay number of the desired overlay. This number has been specified with the LOD100 OVERLAY command.

When this routine is called from an ASM100 subroutine, it is done with the following procedure.

1. set s-pad 10 to the value of the overlay number
2. perform the following: JSR OVLD

If the overlay with the indicated overlay number cannot be located, the overlay loader halts.

4.3.2 ADC HASI

Figure 4-2 shows an ASM100 subroutine for which an ADC HASI routine is created. An ADC HASI is created because the entry point is specified with a $SUBR pseudo-op.

```
        $TITLE ADCSFT
        $SUBR ADCSFT, 1
        $PARAM 1, I/I/IP
        $COMMON /.LOCAL/ I/I
        $COMIO COMA 1
        $COMMON /COMA/ A/R
        NM $EQU 1
        C27 $EQU 16
        ADDR $EQU 15
        NN $EQU 1
"       THIS IS AN ADC ROUTINE.  IT FLOATS THE VALUE OF ITS SINGLE
"       FORMAL PARAMETER AND PLACES IT IN LOCATION A IN THE COMMON
"       BLOCK CALLED /COMA/.  NOTE:  THE PARAMETER IS AN "INPUT
"       PARAMETER", AND THE COMMON BLOCK IS ONLY TRANSFERRED FROM
"       FPS-100 TO HOST AND NOT VICE-VERSA.

ADCSFT: LDMA; DB=I              "GET ADDRESS OF ARGUMENT
        NOP
        NOP
        LDMA; DB=MD             "GET ACTUAL ARGUMENT
        NOP
        NOP
        LDSPI NM; DB=MD         "SAVE I
        LDSPI C27; DB=27.       "THE NEXT FIVE LINES FLOAT SP(NM)
        MOV NM,NM; DPX<SPFN
        FADD ZERO,MDPX; MOV C27,C27
        FADD
        LDSPI ADDR; DB=A        "PREPARE TO WRITE FLOATED VALUE TO A
        DPX(NN)<FA
        MOV ADDR,ADDR; SETMA; MI<DPX(NN)        "WRITE THE VALUE
        RETURN
        $END
```

Figure 4-2 ADC Subroutine

Figure 4-3 shows a HASI subroutine generated for the routine in Figure 4-2. This HASI routine provides all parameter and common block passing from host to FPS-100. Since the HASI contains host FORTRAN, the HASI routine generated may vary from system to system.

When the user calls an ADC HASI routine, control does not return to the host program until the FPS-100 halts. No simultaneous processing on the host and the FPS-100 is permitted.

```
      SUBROUTINE ADCSFT
     * (P
     * 1)
      INTEGER P 1
       COMMON /APLDCM/ IPAV( 33),NU2,IDLM,NU1,IPPAAD,IPPAND,IOVS(33),
     *  LMT(10,3),LMTE
      COMMON /COMA   / C 2001(     1)
      IF (IDLM.NE.      1)CALL L 101
      IPAV(1)=  1
      IPA=IPPAAD
      IPAV(  2)=IPA
      IPA=IPA+     1 *  1
      IPAV(  3)=IPA
      CALL APPUT(P     1,IPAV(  2),IPAV(  3)-IPAV(  2),1)
      CALL APRUN(    16,      1,      1,      0,13)
      CALL APGET(C 2001,      2,      1,      2)
      CALL APWD
      CALL APEXC
      RETURN
      END
       BLOCK DATA
       COMMON /APLDCM/ IPAV( 33),NU2,IDLM,NU1,IPPAAD,IPPAND,IOVS(33),
     *  LMT(10,3),LMTE
       DATA NU2,IDLM,NU1,IPPAAD,IOVS(2),LMTE
     *  /0,0,0,0,0,0/
       END
```

Figure 4-3 ADC HASI

4.3.3 UDC HASI

Figure 4-4 contains an ASM100 subroutine for which a UDC HASI is generated. This program is the same as that shown in Figure 4-2, except that the entry point is declared with a $ENTRY pseudo-op.

```
        $TITLE SPFLT
        $ENTRY SPFLT,1
        NM $EQU 0
        C27 $EQU 16
        ADDR $EQU 15
        A $EQU 20
        NN $EQU 1

"       THIS IS AN EXAMPLE OF A UDC ROUTINE.  IT RECEIVES AS A
"       PARAMETER AN INTEGER, FLOATS IT, AND RETURNS THE FLOATED
"       VALUE IN MD LOCATION 20 (OCTAL).  THE PARAMETER IS S-PAD 0.

SPFLT:  LDSPI C27; DB=27.       "THE NEXT FIVE LINES FLOAT SP(NM)
        MOV NM,NM; DPX<SPFN
        FADD ZERO,MDPX; MOV C27,C27
        FADD
        LDSPI ADDR; DB=A        "PREPARE TO WRITE FLOATED VALUE TO MD
        DPX(NN)<FA
        MOV ADDR,ADDR; SETMA; MI<DPX(NN)        "WRITE THE VALUE
        RETURN
        $END
```

Figure 4-4 UDC Subroutine

Figure 4-5 shows a HASI subroutine generated for the routine in Figure 4-4. This routine does not contain as many APPUT and APGET calls as the HASI subroutine in Figure 4-3. This routine does not contain the parameter passing mechanism. The user must specify actual main data addresses as arguments.

When the user calls a UDC HASI, control returns to the host program immediately. If common blocks are not used, simultaneous processing on the host and the FPS-100 can occur.

```
      SUBROUTINE SPFLT
     * (P
     * 1)
      INTEGER P 1
       COMMON /APLDCM/ IPAV( 33),NU2,IDLM,NU1,IPPAAD,IPPAND,IOVS(33),
     *  LMT(10,3),LMTE
      IF (IDLM.NE.      1)CALL L 101
      IPAV(1)=  1
      IPAV(  2)=P      1
      CALL APRUN(     16,      0,      2,      0,13)
      CALL APEXC
      RETURN
      END
       BLOCK DATA
       COMMON /APLDCM/ IPAV( 33),NU2,IDLM,NU1,IPPAAD,IPPAND,IOVS(33),
     *  LMT(10,3),LMTE
       DATA NU2,IDLM,NU1,IPPAAD,IOVS(2),LMTE
     *  /0,0,0,0,0,0/
       END
```

Figure 4-5 UDC HASI

4.3.4 COMMON BLOCKS IN HASI ROUTINES

Each HASI subroutine contains a labeled common block, APLDCM (refer to Figure 4-3 and Figure 4-5). This common block is used to communicate run time information between the HASI routine and the FPS-100 support modules. Important elements of this block include the following:

IPAV(33)	This is the parameter address vector. The first element of this arrary contains the number of parameters. The remaining elements contain the actual addresses of parameters passed from host to FPS-100 (a maximum of 32 parameters can be passed). These addresses lie in the parameter passing area, if an ADC HASI is created.
IDLM	This is the load module identifier of module currently loaded in the FPS-100. If no module is currently loaded, this element has a value of 0.
IPPAAD	This is the starting address of the PPA (parameter passing area).
IPPAND	This is the ending address +1 of the PPA.
IOVS(33)	Each time the user calls routine APOVLD from the host, the overlay numbers are placed in this array. The first element of this array contains the number of overlays specified and is 0 if overlays are not used.
LMT(10,3)	This array contains information concerning the load modules specified in APLLI calls. Each first element contains the load module ID; each second element contains the APLLI option which is always 1 for this release; each third element contains the number of the logical unit assigned to the load module.
LMTE	This variable contains the number of load modules which are identified in array LMT.

CHAPTER 5

ERROR MESSAGES

5.1 GENERAL INFORMATION

When the user makes an error in an LOD100 command, LOD100 displays an error message at the user terminal. The messages are displayed in the following format:

```
message  (number  severity)
```

message	message describing the error
number	error number
severity	error severity; possibilities for this include:

F	A fatal error has occured. The user should restart the load.
W	This is a warning message. Loading may continue; however, the user may wish to restart the load.
I	The command is ignored.

The following is an example of an LOD100 error message:

```
COMMAND OUT OF ORDER (22 I)
```

NOTE

Some operating systems may have implemented LOD100 in such a way that only the number and severity fields are displayed.

5.2 MESSAGES

Table 5-1 contains a list of LOD100 error messages. Error numbers and descriptions are also included.

Table 5-1 Error Messages

NUMBER	SEVERITY	MESSAGE	DESCRIPTION
1	F	READ ERROR	An unrecoverable read error occurred. Start over.
2	F	BAD RECORD	A record did not correspond to the expectation of LOD100; usually a bad object module is the cause. This message is followed by the erroneous line.
3	F	CODE OVERLAP	Microcode must be placed in ascending PS locations. This message indicates an illegal usage of the ASM100 $LOC pseudo-op.
4	F	UNEXPECTED EOF	This is similar to error number 2. A bad object module was encountered.
5	F	LOADER BLOCKS OUT OF ORDER	The ordering of blocks within the object module was not conformed to.
6	F	TOO MANY ROUTINES	Too many subroutines encountered. Table overflow occurred.
7	F	TOO MANY ENTRIES	Too many entries were encountered. Table overflow occurred.
8	W	WARNING - DUPLICATE ENTRY:	A subroutine was loaded with an entry point identical to one encountered earlier. The new entry point is ignored.
9	F	TOO MANY EXTERNALS	Too many externals were declared. Table overflow occurred.
10	I	CAN NOT ASSIGN FILE	The requested file does not exist or cannot be assigned.
11	I	WHAT	An unrecognizable command was encountered.
12	I	BAD OR MISSING PARAMETER	A bad command line was encountered. It was ignored.

Table 5-1 Error Messages (cont.)

NUMBER	SEVERITY	MESSAGE	DESCRIPTION
13	F	DOUBLE TITLE BLOCK	A subroutine's title was encountered twice.
14	F	MISSING END BLOCK	A bad object module was encountered.
15	F	UNMATCHED COMMON BLOCK	Common blocks of the same name must correspond to each other with respect to size and item types.
16	F	TOO MANY DATA BLOCKS	A table overflow has occurred.
17	F	TOO MANY OVERLAYS	A table overflow has occurred.
18	F	ENTRY POINT DECLARED CALLABLE IS NOT RELOCATABLE	An entry point declared callable with the CALL command is not relocatable, as required (it is an absolute symbol).
19	F	TOO MANY PARAMETERS IN CALLED ROUTINES	A table overflow has occurred.
20	W	ARGUMENT ALREADY SPECIFIED	The indicated argument has already been specified.
22	I	COMMAND OUT OF ORDER	The required ordering of LOD100 commands has not been followed.
23	I	OUTPUT FILE(S) NOT ASSIGNED	A LOAD or LIB command was entered without specifying a file to be loaded.
25	I	NOT A LIBRARY	The file specified with the LIB command did not contain a library.
26	W	MD MEMORY OVERFLOW	Data has been loaded in main data memory beyond the point specified with the MMAX command.

Table 5-1 Error Messages (cont.)

NUMBER	SEVERITY	MESSAGE	DESCRIPTION
27	W	PS MEMORY OVERFLOW	Data has been loaded in program source memory beyond the point specified with the PMAX command.
28	W	MULTIPLE HOST CALLABLES	Only one entry point of a subroutine may be host callable.
29	I	COMMAND NOT IMPLEMENTED	This command is unknown to the loader.
30	F	RELOCATION ERROR	This item has been illegally relocated.
31	F	IMPROPER USE OF TRIPLE	A double precision integer cannot be used at this point.
32	F	PSDATA TABLE OVERFLOW	A partition table overflow has occurred indicating that too many PS partitions exist.
33	F	IMPROPER TREE STRUCTURE	The overlay tree must have a top node.
34	F	TASK SEGMENT ADDR MISSING FROM TABLE	This task does not have an address in the partition table.
35	F	RDYQUE NOT DEFINED	The user must define a common block called RDYQUE.
36	F	ISRMAP MISSING	The user must define a common block called ISRMAP.
37	F	ISRMAP WRONG SIZE	The ISRMAP common block must be 120 words long.
38	F	MULTIPLY DEFINED TASK NAME	This task name is already in use.
39	F	TSKDTER TABLE OVERFLOW	Too many tasks were specified.

Table 5-1 Error Messages (cont.)

NUMBER	SEVERITY	MESSAGE	DESCRIPTION
40	F	MULTIPLE ISR BLOCKS	Only one ISR block is allowed at one time.
41	F	OVPDTA TABLE OVERFLOW	Too many overlays were defined.
42	F	BOTH TASK AND ISR NOT ALLOWED	ISRs and tasks cannot be loaded in the same session.
43	F	TASK ALREADY SPECIFIED	This task is already defined.
44	F	WRONG TCB SIZE	A previously defined TCB does not have the size specified with the corresponding TASK command.

INDEX

ADC 4-7
ADC HASI 1-5; 4-12
Allocation of memory 2-13
Alternate entry block 3-9
APLDCM 4-16
APLLI 4-10
APLMLD 4-7
APOVLD 1-10; 2-10,29; 4-11
APRUN 4-9
APX100 task 1-12; 2-31
ASM100 tasks 2-32
Auto-directed calls 1-5; 4-7

Binary formatted load module 4-5
Binary tree representation 2-18
Block header 3-2
Block types 3-1
Blocks, load module 4-1

C command 2-10
CALL command 2-10
Callable routines 2-10
Calling LOD100 2-1
Code block 3-2
Code/overlay/32 bit MD data block 4-2
Commands 2-2
Common blocks 4-16
Conventions 1-2
Creating a multi-level job 2-27
Creating a single level task 2-25

Data block 4-3
Data block description block 3-6
Data block initialization block 3-7
DB map 2-21
DBBRK 2-21
Diagnotics 5-1
Disk resident load modules 1-5; 2-3

End block 3-3; 4-5
Entry block 3-4
Entry points 2-10
Error messages 5-1
EX command 2-24
Executive routines 4-7
EXIT command 2-24
External block 3-5

F command 2-12
FORCE command 2-12
Formal parameter block 3-8
FPS-100 jobs 1-6
FSLMLD 4-8
FTN100 tasks 2-33
Functions of the loader 1-4

HASI 1-5; 4-7
HASI file 2-3
HASI, ADC 4-12
HASI, UDC 4-14
Host - FPS-100 interface 1-5
Host callable routines 2-10
Host resident load module 1-5; 2-3; 4-5

I command 2-24
IDLM 4-16
Information block 4-4
INIT command 2-24
Initial priority 2-6,7
INP command 2-2
INPUT command 2-2
Input file 2-2
Input, LOD100 2-1
Interface mechanism 1-5
Interrupt Service routines 1-12
IOVS 4-16
IPAV 4-16
IPPAAD 4-16
IPPAND 4-16
ISR 1-12
ISR block 3-10

Job, multi-level 1-10; 2-27
Job, single level 2-25
Jobs 1-6
Jobs, multiple load module 1-11
Jobs, single level 1-10

L command 2-11
LI command 2-23
LIB command 2-11
Library end block 3-5
Library loading 2-11
Library start block 3-5
LINK command 2-23
LM command 2-5

INDEX

LMID command 2-5
LMT 4-16
LMTE 4-16
LOAD command 2-11
Load complete message 2-26
Load map 2-16
Load module 1-5; 2-3; 4-1
Load module identification number 2-5
Load module identifier 4-16
Load module loading 4-7
Load modules, multiple 2-30
Loader functions 1-4
Loading ASM100 tasks 2-32
Loading FTN100 tasks 2-33
Loading overlays 4-11
Loading tasks 2-31
LOD100 input 2-1
LOD100 output 4-1

M command 2-16
Machine resources 2-7
MAP command 2-16
MARK command 2-15,31
MD command 2-13
MDOFF command 2-13
Memory allocation 1-9; 2-13
MM command 2-14
MMAX command 2-14
MO command 2-5
MODE command 1-12; 2-5,31
Module, load 1-5
MTS100 1-11
Multi-level jobs 1-10; 2-27
Multiple load module jobs 1-11
Multiple load modules 2-30

NL command 2-12
No load map 2-21
NOLOAD command 2-12

O command 2-2
Object module 3-11
Object modules 3-1
OUTPUT command 2-2
Output files 2-2
Output from LOD100 4-1
OV command 2-9
OVERLAY command 2-9
Overlay loading 1-10; 4-11
Overlay map 2-18
Overlay numbers 2-9
Overlay segments 1-7
Overlay structure 1-7; 2-8,31
Overlay table 2-35

P command 2-15
Page size 2-14
Parameter address vector 4-16
Parameter passing area 2-15; 4-16
Partition table 2-35
PM command 2-14
PMAX command 2-14
PPA command 2-15
PRI command 2-6,32
Priority 2-6,7,40
Programmed file I/O 2-3
PS command 2-13,31
PS entries 2-21
PS partition table 2-35
PS titles map 2-21
PSBRK 2-21
PSOFF command 2-13,31
PURGE command 2-15,31

R command 2-4
RADIX command 2-4
Re-initializing the loader 2-24
Ready queue 2-23,41
Related manuals 1-3
Routines, executive 4-7
RTS100 1-11
Run Priority 2-7

Sample load module 4-5
Simultaneous processing 1-5
Single level jobs 1-10; 2-25
Slaved task 2-6,7
Structure, overlay 1-7
Supervisor environment 1-11

T command 2-8
Task block 3-10
TASK command 2-6,32,33
Task communication block 2-6,37
Task identification number 2-6
Task loading 2-31
Task map 2-22
Task mode 1-12
Task priority 2-6

INDEX

$TASK pseudo-op 2-32,33
Tasks 1-12
TCB 2-6,37
Title block 3-4
TREE command 2-8
UDC 4-7
UDC HASI 1-5; 4-14
Underfined symbols map 2-21
User directed calls 1-5; 4-7

READERS COMMENT FORM

Your comments will help us improve the quality and usefulness of our publications. To mail: fold the form in three parts so that Floating Point Systems mailing address is visible, then seal.

Title of document _____

Name/Title _____ Date _____

Firm _____ Department _____

Address _____

Telephone _____

I used this manual...

- ☐ as an introduction to the subject
- ☐ as an aid for advanced training
- ☐ to instruct a class
- ☐ to learn operating procedures
- ☐ as a reference manual
- ☐ other _____

I found this material...

	Yes	No
accurate/complete	☐	☐
written clearly	☐	☐
well illustrated	☐	☐
well indexed	☐	☐

Please indicate below, listing the pages, any errors you found in the manual. Also indicate if you would have liked more information on a certain subject.

First Class
Permit No.A-737
Portland,
Oregon

BUSINESS REPLY

No postage stamp necessary if mailed in the United States

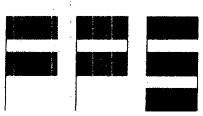
Postage will be paid by:

FLOATING POINT SYSTEMS, INC.

P.O. Box 23489

Portland, Oregon 97223

Attn: Technical Publications



FLOATING POINT
SYSTEMS, INC.

CALL TOLL FREE 800-547-1445
P.O. Box 23489, Portland, OR 97223
(503) 641-3151, TLX: 360470 FLOATPOINT PTL