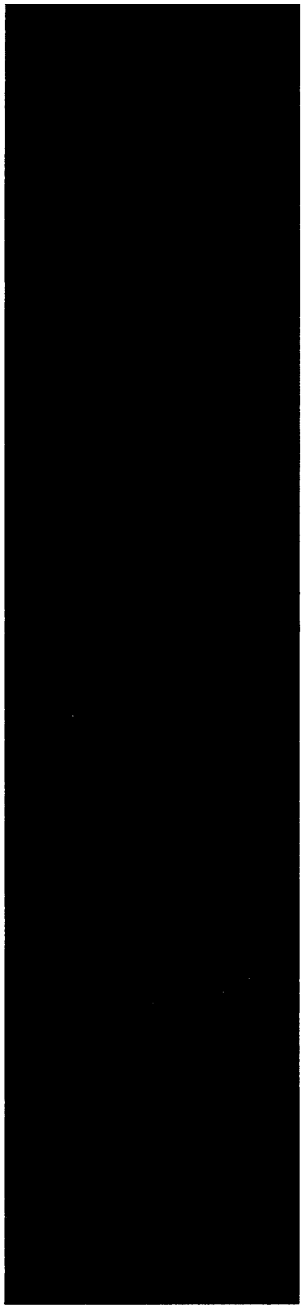
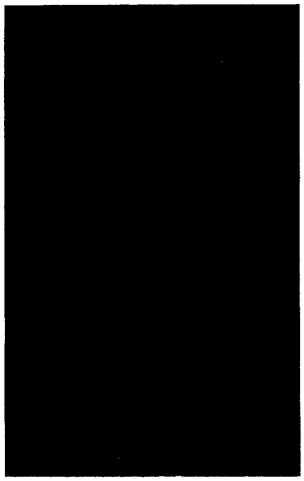


FLOATING POINT
SYSTEMS, INC.



Supervisor Reference Manual (MTS100)

860-7445-000

by FPS Technical Publications Staff

Supervisor Reference Manual (MTS100)

860-7445-000

Publication No. 860-7445-000
February, 1980

NOTICE

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

ILLUSTRATIONS

TABLES

CHAPTER 1

INTRODUCTION

1.1 PURPOSE

This manual documents the FPS-100 Multi-Tasking Supervisor (MTS100). It describes the supervisor operations to a programmer familiar with the FPS-100 assembly language and the LOD100 loader. It describes the formats of tasks and interrupt service routines used with the supervisor, explains how to send messages, and provides the instructions for installing the system. It does not attempt to teach assembly language programming to the user nor to describe the internal operation of the supervisor.

1.2 SCOPE

This manual documents the supervisor calls. It describes the software components in a supervisor environment and explains the message passing and task execution mechanisms. It also contains procedures for system installation. This manual is organized as follows:

Chapter	1	Introduction
Chapter	2	Supervisor Functions
Chapter	3	Supervisor Calls
Chapter	4	Communications
Chapter	5	Device Handlers
Chapter	6	MTS100 User's Guide
Appendix	A	System Standard Definitions
Appendix	B	I/O Device Configuration Table
Appendix	C	Alphabetical Index of Supervisor Calls (SVCs) and Host/FPS-100 Communication Routines

1.3 RELATED MANUALS

The documents in Table 1-1 also contains information useful to the MTS100 programmer.

Table 1-1 Related Manuals

MANUAL	PUBLICATION NO.
LOD100 Reference Manual	FPS 860-7423-000
ASM100 Reference Manual	FPS 860-7428-001
FTN100 Reference Manual	FPS 860-7422-000
APX100 Manual	FPS 860-7426-000
SIM100/DBG100 Reference Manual	FPS 860-7424-000
FPS-100 Maintenance Manual	FPS 860-7417-000
GPIOP Software Reference Manual	FPS 860-7430-000
IOP-16/38 User's Manual	FPS 860-7310-003

1.4 GENERAL DESCRIPTION

The MTS100 multi-tasking supervisor is a collection of FPS-100 resident programs that coordinate software activities within the FPS-100. Although no FPS-100 resident supervision is required when running single-activity programs in the FPS-100, some implementation systems require the additional speed achieved by processing multiple software activities concurrently. MTS100 provides this capability. It coordinates the multiple activities (called tasks) and provides the mechanism to pass control and data among them.

The supervisor coordinates activities in the FPS-100 by responding to interrupts. Interrupts can be generated by external devices, the host computer, or by tasks running in the FPS-100. The FPS-100 is interrupted to request a service (such as sending a message or reading from a disk). The supervisor responds to the interrupt by performing the service or transferring control from the currently executing task to the task which can perform the service. When the service has been performed, the supervisor returns control to the appropriate task, as determined by a priority ordered list of ready tasks that the supervisor maintains.

MTS100 is a passive/reactive supervisor rather than an active monitor. It remains inactive (using no processor time) until an interrupt occurs. When an interrupt occurs, the FPS-100 switches from user mode (the normal mode of operation) to supervisor mode (a mode in which the entire machine state can be saved and restored at a later time). The supervisor becomes active, performs the requested service, switches the FPS-100 back to user mode, and becomes inactive again.

MTS100 should not be confused with the other FPS-100 executive service, APX100. APX100 is a host resident executive which oversees the transfer of programs and data between the host computer and the FPS-100. The user can maintain direct control over this process by calling APX100 routines from the host. MTS100 manages the programs and data after they have been transferred to the FPS-100. MTS100 services are largely transparent to the host user program. Figure 1-1 illustrates the relationship between APX100 and MTS100. For a further description of APX100, refer to the APX100 Manual.

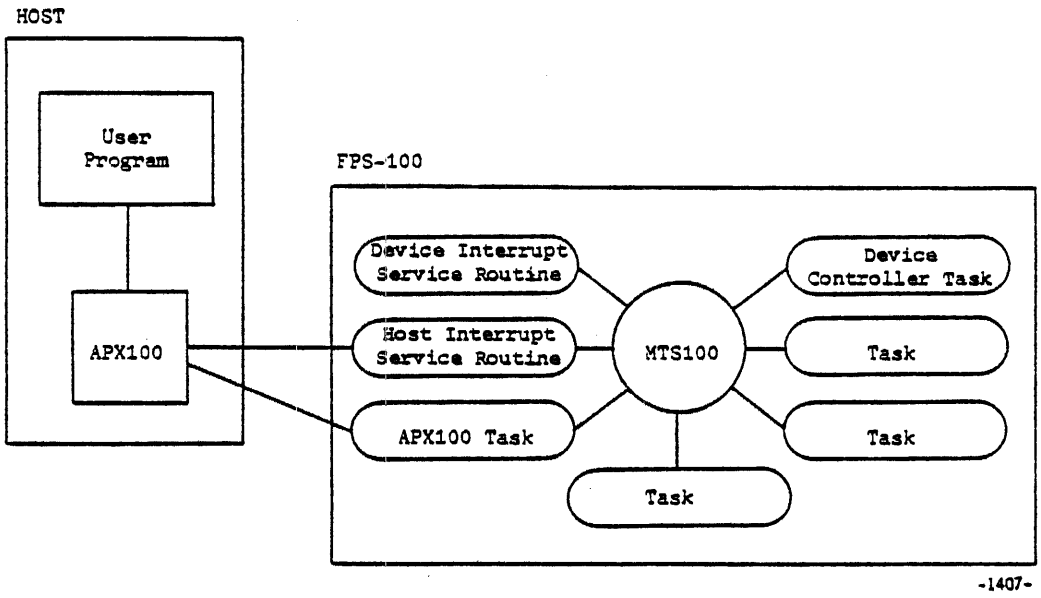


Figure 1-1 APX100 - MTS100 Relationship

1.5 SOFTWARE COMPONENTS

In a supervisor-controlled environment, there are three basic types of software routines: the task, the host-callable subroutine, and the device handler.

1.5.1 Tasks

The task is the fundamental unit of work performed in a supervisor-controlled FPS-100. Tasks consist of one or more ASM100 or FTN100 routines which perform a logically independent activity in the FPS-100. The task is a subprogram which is visible to the supervisor rather than to the host. Associated with each task is:

- A priority. This priority allows the supervisor to choose the highest priority task for execution when more than one are waiting.
- A task control block (TCB). The TCB contains status information and the execution context of the task. The status information includes such things as the run priority of the task, whether or not the task's priority is dependent on another task, and whether or not the task uses the floating-point unit. The execution context is a complete copy of all hardware registers and data paths used by the task when it executes. Each time an executing task is interrupted, the supervisor systematically copies the execution context into the TCB. When the task is resumed, it reloads the execution context into the registers. Thus, the task can resume at the exact point of interruption.

When the FPS-100 starts running, tasks begin execution immediately without being called from the host. The task with the highest priority executes first and executes continuously until the FPS-100 is interrupted. After the supervisor services the interrupt, it resumes execution of the highest priority ready task. So, for example, while one task is waiting for a message, another task can be executing.

1.5.2 Host-callable Subroutines

Subroutines are basic software elements of the FPS-100. The user may declare certain subroutines as host-callable. A host-callable subroutine is closely linked to the host. The host program provides parameters and data, initiates the subroutine execution, and receives the results.

Normal subroutines (not host-callable) are initiated by tasks or by the supervisor within the FPS-100. Communication between normal subroutines and the host user program is limited.

A special task, called the APX100 task, provides the mechanism for host-callable subroutines to run on the FPS-100 in the same manner as they run on an unsupervised FPS-100. Host-callable subroutines are processed sequentially and are under the control of a main program running on the host computer. They do not start running until called by the host and upon completion return control to the host. Data can be passed between the host program and a host-callable subroutine as parameters or in common blocks.

All host-callable subroutines and routines ultimately called by them must be part of the APX100 task. They cannot function as part of any other task.

1.5.3 Device Handlers

Device handlers are routines which control the I/O transfer between external devices and the FPS-100. A device handler consists of two parts, the device controller and the device servicer (also called the interrupt service routine or ISR).

The device controller handles control requests from other tasks. It performs control functions on the device such as rewind and skip file (if programmed for these functions). It also stages the buffers for the ISR. The device controller is a task itself and is the only communication between other tasks and the external device.

ISRs handle the interrupts generated by external devices (external devices interrupt the FPS-100 to indicate that the desired function or transfer is complete). The ISR provides notification that the requested function has been performed, returns buffers of data, and, if necessary, causes the external device to start transferring the next buffer. The ISR, though it may be created by the user, is considered part of the supervisor. There is no direct communication between tasks and ISRs except through the device controller. Chapter 5 contains further information concerning device handlers.

1.5.4 Differences Between Tasks and Subroutines

Although tasks consist of subroutines and all host-callable subroutines are part of a task (the APX100 task), it is legitimate to consider the differences between tasks and host-callable subroutines. Tasks and host-callable subroutines are alternative methods of processing on the FPS-100.

Tasks:

- start executing as soon as the FPS-100 begins operation, without being called from the host
- execute continuously except when waiting for some event, such as the arrival of a message
- are processed concurrently; when one task is waiting, another can be executing
- are not under the control of any host user program
- are limited in their communication with host user programs

Host-callable subroutines:

- do not start executing until explicitly called by a host user program
- either execute to completion and then return control to the host user program or execute concurrently with the host and send notification to the host upon completion.
- are processed sequentially; a second routine cannot start executing until the first has completed
- are under explicit control of a host user program
- can communicate with the host user program by means of parameters, common blocks, and APX100 routines such as APGET and APPUT.

Tasks are used when repetitive processing is required at unpredictable times (such as processing data read in from a disk). Host-callable subroutines are used for one-time or limited-time calculations which occur at known times. Host-callable subroutines can be used when more extensive communication between the host computer and the FPS-100 routine is desired.

1.6 COMMUNICATIONS

In a supervisor environment, several device handlers and tasks can be running (including the APX100 task which contains host-callable subroutines). Communications in this system can be broken into three types, host-FPS-100 communication, task-task communication, and task-external device communication.

1.6.1 Host/FPS-100 Communication

Three types of host-FPS-100 communications are possible:

- Communication with the FPS-100 using virtual front panel operations. From the user's point of view, this type of communication is similar to that on an unsupervised FPS-100; however, the actual method of accomplishment is quite different within the FPS-100 (refer to section 4.2).

- Communication with host-callable subroutines. The procedure for this type of communication is exactly the same as on an unsupervised FPS-100. APX100 routines are used to pass data and control subroutines.

- Communication with tasks using communication ports. The communication ports allow a host program to communicate with any task. A limited (13-bit) value can be sent or received from one of seven message exchanges.

Section 4.2 contains information on other types of host/FPS-100 communications.

1.6.2 Task/Task Communication

Messages are used when tasks communicate with each other in order to send data, perform control functions, or synchronize execution. Tasks use supervisor calls to send, receive, and answer messages. These messages are sent, received, and answered using message exchanges.

1.6.2.1 Messages

The message is the basic unit of communication in the supervisor environment. A message is actually an area of main data memory which is written or read by tasks. Messages consist of two parts, the message header and the text of the message. The message header contains such things as the length of the message, its type, and its priority. Messages are passed from one task to another not by transferring the message data from one task area to another but by transmitting the address of the message from one task to another.

1.6.2.2 Message Exchanges

A message exchange is a synchronizing mechanism between tasks that send messages and tasks that receive them. These processes seldom occur at the same time. Thus, either there are messages available and no tasks to receive them or there are tasks waiting and no messages. The message exchange is a place for either tasks or messages to wait until the other arrives; it is a meeting place.

A message exchange is actually a queue, either of tasks waiting for messages or messages waiting for tasks to receive them. The queue is implemented not as a table in the supervisor memory but as pointers in the TCBs of the tasks themselves (or in the case of messages, pointers in the message headers). This allows an unlimited number of tasks or messages to be waiting.

When both a task and a message are available at the same exchange, the supervisor gives the address of the message to the task, disassociates both the task and the message from the exchange, and reschedules the task.

1.6.2.3 Supervisor Calls

Supervisor calls (SVCs) are instructions that tasks give to the supervisor, directing it, for example, to adjust priorities, or to send, receive, or answer messages.

SVCs are made using the TRAP instruction. This instruction causes an internal interrupt of the FPS-100. When the FPS-100 is interrupted, the supervisor saves the state of the executing task, determines the cause of the interrupt, processes the SVC (which could mean sending a message to a message exchange, for example), and resumes execution of the highest priority ready task after restoring its state. Chapter 3 contains more specific information concerning SVCs.

For example, two tasks wish to communicate in a supervised FPS-100. Task 1 sends a message by issuing the SEND SVC via a TRAP instruction, which interrupts the FPS-100. The supervisor saves the state of task 1 and links the message in the message queue of the appropriate exchange if no task is waiting to receive the message. The supervisor then resumes the execution of the highest priority ready task.

In order to receive the message, task 2 issues the WAIT SVC via a TRAP instruction, which interrupts the FPS-100. The supervisor saves the state of task 2, and links the message to task 2 if a message exists. If no message is available in the message exchange, the supervisor removes task 2 from the ready state and places it in the wait state, links the TCB of task 2 to the waiting task queue of the message exchange, and resumes execution of the highest priority ready task.

A typical example of task communication consists of two tasks, a producer task and a consumer task. The function of the producer task is to take data (obtained perhaps from a device handler), process or reduce this data, and send it to the consumer task. The producer task is continually sending messages. The function of the consumer task is to receive messages consisting of reduced data and further process that data (perhaps sending it to another device handler for eventual output at a terminal). The consumer task is continually receiving messages.

These two tasks normally use two message exchanges to facilitate their communications. At initialization, the consumer task waits at the first exchange. The producer task uses the first exchange to send the message and uses the second exchange for the answer. When the producer task sends a message to the first exchange, the message contains the address of the second exchange to which the answer will be sent. The consumer task receives the message and sends an answer to the second exchange to confirm the receipt of the message. Figure 1-2 illustrates this communication.

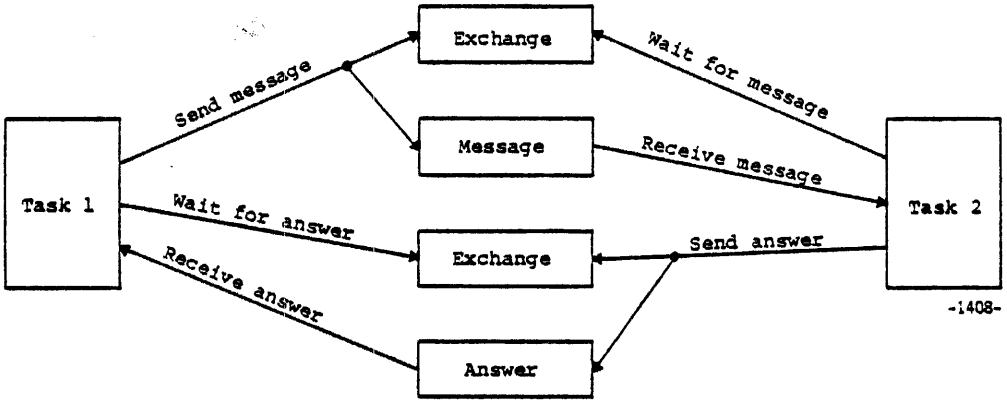


Figure 1-2 Sending Messages and Answers

1.6.3 Task/External Device Communication

Tasks communicate with external devices through each device's associated device handler. The controller portion of the device handler is itself a task, so this communication is basically the same as task-task communication. One difference is that when reading or writing to an external device, the ISR portion of the device handler and not the controller portion returns an answer to the original task.

1.7 INTERRUPTS

The MTS100 supervisor is able to function due to the ability of the FPS-100 to be interrupted. An FPS-100 interrupt is interpreted by the supervisor as a request for service. When an interrupt occurs, the FPS-100 switches from user mode to supervisor mode. The supervisor then becomes active and saves the state of the currently executing task. It determines the type of interrupt and performs the required service. The supervisor then restores the state and resumes execution of the highest priority ready task. (This may not be the same as the task just interrupted; a higher priority task may now have a message that it had been waiting for, or the currently executing task may be waiting for a message).

Two types of interrupts can occur, asynchronous and synchronous.

1.7.1 Asynchronous Interrupts

Asynchronous interrupts are caused by external devices which request FPS-100 service. Asynchronous interrupts can occur at any time, not just at instruction cycle intervals. When an external device interrupts the FPS-100, the supervisor determines which device it is, masks out interrupts from lower priority devices, and runs the appropriate interrupt service routine. It then restores the state of the highest priority ready task and resumes its execution.

1.7.2 Synchronous Interrupts

Synchronous interrupts are generated as a part of, or as a result of FPS-100 execution. Three types of synchronous interrupts can occur:

- fatal
- floating-point exception
- TRAP

Fatal interrupts are caused by the overflow of the subroutine return stack (SRS). The SRS can contain 15 entries. When an overflow (and thus a fatal interrupt) occurs, the task in which it occurs is terminated. If desired, a user-written routine can be inserted into the supervisor to handle the interrupt differently.

A floating-point exception interrupt is caused by an arithmetic overflow, underflow, or divide by zero. The supervisor terminates any task in which it occurs. However, in its initialized state, this interrupt is normally disabled. A user-written routine can be provided to perform error processing when this interrupt occurs. The interrupt itself can be enabled or disabled with the SETFPE SVC (described in section 2.3).

A TRAP interrupt occurs when a task requests supervisor service. It is caused by a task executing a TRAP instruction. When this type of interrupt occurs, the supervisor performs the requested service (refer to Chapter 3 for a description of all SVCs) and resumes the execution of the highest priority ready task.

CHAPTER 2

SUPERVISOR FUNCTIONS

2.1 INTRODUCTION

The supervisor consists of two components: the kernel, which is permanently resident in program source memory, and the interrupt service routines (ISRs), which can be overlaid from main data memory. The kernel performs the following basic supervisor funtions:

- executing tasks
- dispatching device interrupts
- saving and restoring task states
- handling supervisor service requests

ISRs service interrupts generated by external devices. Since ISRs are parts of device handlers, they are also discussed in Chapter 5.

2.2 EXECUTING TASKS

The Execute Task portion of the supervisor performs the following:

- selects a task to run
- ensures that the task is resident in program source memory
- restores the state of the chosen task
- starts the task running

2.2.1 Selecting Tasks to Run

The supervisor selects a task to run on the basis of its priority and its state. The supervisor uses the ready queue to make this selection.

2.2.1.1 Setting Task Priorities

A task priority is an octal number between 1 and 377 (255 decimal), with 1 being the lowest priority and 377 the highest. The user can set default task priorities either at assembly time or at load time. Priorities for ASM100 tasks can be set at assembly time with the $TASK pseudo-op and changed at load time with the PRI command. Priorities for FTN100 tasks can be set at load time with the TASK command. If the user does not set a task's priority, it is automatically set to 144 octal (100 decimal).

The user can also specify a task to be run first upon start-up of the supervisor, regardless of its priority, and thereafter retain its default priority. This is done with the /I option on the $TASK pseudo-op, TASK command, or PRI command. This option causes the supervisor initially to place the task at the top of the ready queue. This option is normally used only for device handler controller tasks. These tasks initially perform their startup functions and wait for requests before user tasks start processing. The controller tasks then assume their default priorities.

Another task priority option available to the user at assembly or load time is the /S option on the $TASK pseudo-op, the TASK command, and the PRI command. This option specifies that the task priority is slaved. A task whose priority is slaved acquires the priority of any other task that sends it a message (refer to the SEND SVC in section 3.5). This feature is useful for a task which performs a service for both high priority and low priority tasks. A high priority task then receives high priority service and a low priority tasks receives low priority service.

Additional information concerning the $TASK pseudo-op can be found in the ASM100 Reference Manual. Information concerning the TASK and PRI commands can be found in the LOD100 Reference Manual.

Tasks can also change their priorities during execution. The RUNPRI SVC can be used to change the priority of the currently executing task and SETPRI can be used to change the priority of other tasks. These SVCs are described in section 3.4.

2.1.1.2 Task States

A task in the FPS-100 can be in one of the following states:

- It can be ready. In this state a task has all the resources it needs in order to execute but is not executing because a task with a higher priority is currently executing. A ready task is actually waiting in the ready queue. It executes when it reaches the top of the queue. Section 2.1.1.3 describes the ready queue in more detail.

- It can be waiting. A task in this state is waiting at an exchange for a message from another task or device handler. The task cannot continue execution until it receives this message (or, in the case of timed waits, a certain time interval elapses). An exchange is a queue similar to the ready queue, except that tasks wait for messages instead of processor time and they are serviced on a first in, first out basis instead of of on a priority basis. Exchanges are further described in section 4.1.2.

- It can be running. Only one task in the system can actually be running at any one time.

2.1.1.3 Ready Queue

The ready queue is a priority ordering of ready tasks which is continually maintained by the supervisor. At the head of this queue is a system common area named READYQ. READYQ contains pointers RLINK and LLINK. RLINK points to the TCB of the highest priority ready task. LLINK points to the TCB of the lowest priority ready task. The TCB of each task also contains pointers RLINK and LLINK. These pointers continue the queue. In the TCB of the highest priority ready task, RLINK points to the TCB of the second highest priority ready task, and LLINK points to READYQ. In the TCB of the second highest priority ready task, RLINK points to the TCB of the third highest priority ready task and LLINK points to the TCB of the highest priority ready task. This linkage continues until finally in the TCB of the lowest priority task, RLINK points to READYQ, and LLINK points to the TCB of the second lowest priority task. Figure 2-1 illustrates this linkage.

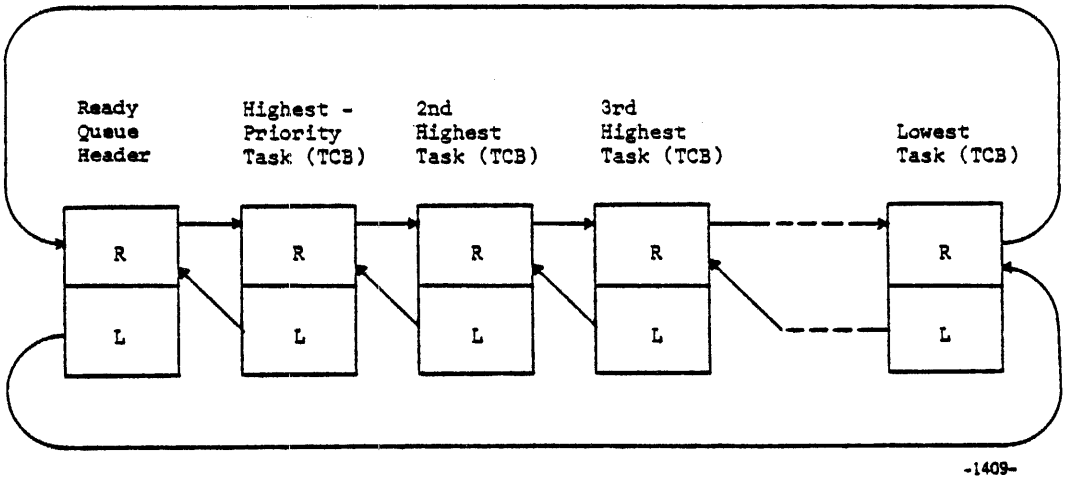


Figure 2-1 Ready Queue

At load time, LOD100 initializes the ready queue header and the TCBs of all tasks so that the ready queue contains all tasks in priority order (including tasks with the /I option specified, which are placed at the beginning of the queue regardless of their priorities). Thus, when FPS-100 execution begins, all tasks in the system are on the ready queue.

The supervisor uses this ready queue to select the appropriate task to run. It continually updates the queue to reflect the status of tasks in the FPS-100. Task execution begins with the highest priority task. This task continues execution until the FPS-100 is interrupted. If the interrupt is generated by the task itself and is a wait for a message which is not yet available, the supervisor places the task in a wait state by removing it from the ready queue and placing it in the waiting task queue of a message exchange (exchanges are described in section 4.1.2). The supervisor does this by adjusting the pointers of the task TCBs and READYQ.

The supervisor adjusts the pointers of the TCBs and READYQ whenever the state of a task changes. When a task waits for a message, it is removed from the queue; when a task becomes ready it is inserted into the queue according to its priority. When a task's priority changes, the supervisor changes its position in the queue.

2.2.2 Saving and Restoring States

Each time a running task is interrupted, the supervisor saves the condition (or state) of the task. The supervisor then processes the interrupt and restores the state, allowing the interrupted task to continue processing at the exact point of interruption. Whenever the supervisor selects a task to run, it restores the state of the task. Saving and restoring the state of a task is done by copying various values and registers into or from the task control block.

2.2.2.1 Task Control Block

Associated with each task running in the FPS-100 is a task control block (TCB). A TCB is an area in main data memory in which the supervisor stores the task's status information and execution context whenever the task is interrupted. Thus, whenever the task is restored to an executing state, the supervisor can copy the information in the TCB back into the actual machine registers, and the task can proceed at the point of interruption. The format of the TCB is shown in Table 2-1.

Table 2-1 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 queue link (RCLOCK)
14	Left clock queue 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 (MA)
48	Status register 3 (APSTAT3)
49-64	Subroutine return stack (SRS)

Table 2-1 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

As shown in Table 2-1, the TCB consists of four parts: a header area, the task data area, the minimum save area, and the maximum save area.

The header is a standard data item header, used for tasks, exchanges, and messages. The entries in the TCB header include:

RLINK and LLINK — Pointers that the supervisor updates to maintain the ready queue and the message exchanges. The ready queue is described in section 2.1.1.3. Message exchanges are described in section 4.1.2.

RPRI — Current run priority of the task.

TYPE — Type field which is unused for tasks but is used for messages or exchanges. Refer to section 4.1.1.

LENGTH — Length of the entire TCB in main data words (68 or 148).

ANSKEY — Answer exchange. Whenever a task sends a message, the address of the message's answer exchange is saved in this field so that the task can wait at that exchange for an answer.

The task data area contains additional information that the supervisor needs in order to load or restore tasks. The entries in this area include:

ID — Task identifier which the user specified in the $TASK pseudo-op or the LOD100 TASK command.

OVLPTR — Pointer to this task's first entry in the overlay table. The overlay table includes entries for all the overlay segments of this task in MD memory. The overlay table and overlay segments are described in section 2.2.3.

OVLCNT — Number of overlay segments in this task.

DPRI — Default priority of the task. This was set by the user with the $TASK pseudo-op or the LOD100 TASK or PRI commands.

STATUS — Task status bits. Any combination of the following can be set to indicate task status:

- 010000 — SLVBIT. This bit set indicates that the task priority is slaved. Slaved tasks are described in section 2.2.1.1.
- 004000 — CXTBIT. This bit set indicates that the task uses the full context of the machine. Maximum and minimum state saves are described in section 2.2.2.2.
- 001000 — RDYBIT. This bit set indicates that the task is ready to execute.
- 000400 — CLKBIT. This bit set indicates that the task is performing some sort of timed wait (TWAIT or TWAITA; refer to section 3.5) and is using the real-time clock queue for this timing.

LSTMSG Address of the last message received. The supervisor uses this address when answering a message of unspecified address.

RCLOCK Right pointer for this task in the clock queue. If this task is waiting at the clock queue, RCLOCK points to the next task in the queue. If this task is not waiting at the clock queue, RCLOCK points to itself. Refer to section 4.1.3 for more information concerning the clock queue.

LCLOCK Left pointer for this task in the clock queue. If this task is waiting at the clock queue, LCLOCK points to the previous task in the queue. If this task is not waiting at the clock queue, LCLOCK points to RCLOCK. Refer to section 4.1.3 for more information concerning the clock queue.

ICLOCK Increment of time that this task can wait in the clock queue which is greater than the amount of time all the previous tasks can wait in the clock queue. Refer to section 4.1.3 for more information concerning the clock queue.

TADDR This task's beginning address in main data memory.

The minimum save area contains copies of those registers used by the supervisor during the servicing of an interrupt. When a task is interrupted, the supervisor copies these registers into the minimum save area of the TCB before processing the interrupt.

The maximum save area contains copies of all other registers and flags not saved in the minimum save area.

2.2.2.2 Minimum and Maximum State Saves

As shown in Table 2-1, each TCB is set up with a minimum and maximum save area. An option exists at assembly or load time to declare that a task uses only the minimum machine resources. If this option is specified (the /M option on the $TASK pseudo-op or the LOD100 TASK command), the supervisor does not copy the maximum save area registers into the old task's TCB when a new task using full context is chosen to run. This results in a savings of processor time for those tasks which do not use the following registers:

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

Processor time is saved because these registers are not saved or restored if the task uses only minimum machine resources.

Device handler controller tasks are normally the only tasks which use only the minumum machine resources. Most other tasks use at least the floating adder or floating multiplier.

2.2.3 Loading Tasks

Once the supervisor selects the proper task to execute, using the ready queue, it may need to load the task into program source memory from main data memory if the task has been overlaid by another task or was never resident in PS. The supervisor uses the overlay table and the PS partition table to load the proper routines. When the task is resident in program source memory, the supervisor copies the TCB of the task into the actual machine registers and starts the task executing.

2.2.3.1 PS Partitions

When overlay segments are allocated space in PS memory (at load time), 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 2-2 illustrates this concept.

Overlay Segment A in Figure 2-2 fits in PS locations 100-200. Overlay Segment B fits into locations 150-200, and Overlay 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 covers three partitions. Overlay Segment B starts at location 150 and covers two partitions. Overlay Segment C starts at location 175 and covers one partition.

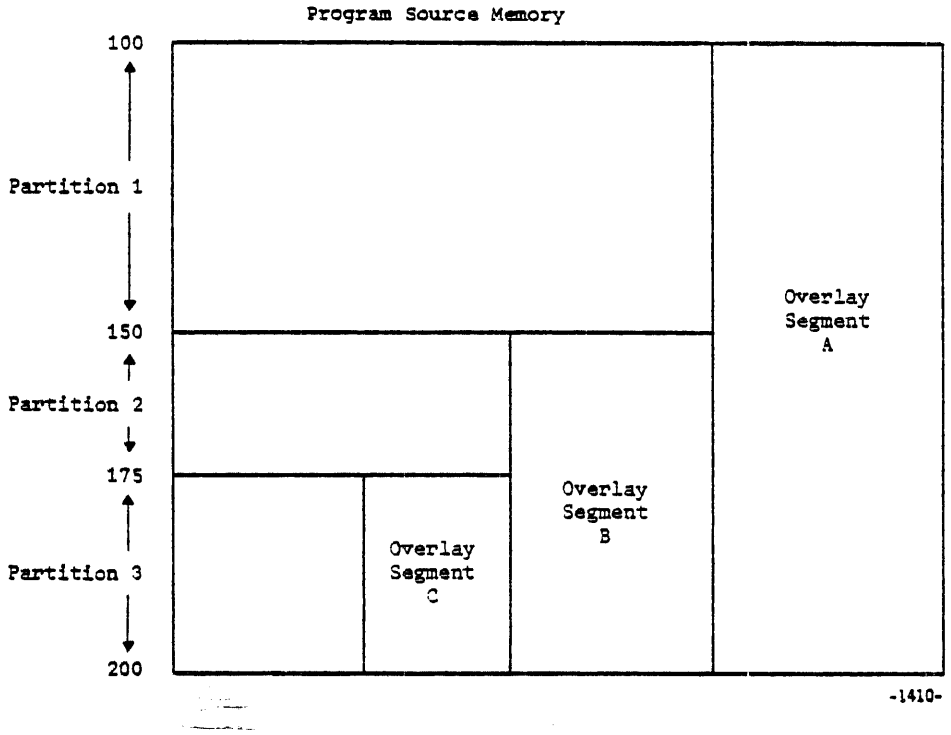


Figure 2-2 Overlay Segments in PS Memory

2.2.3.2 Overlay Table

An overlay table exists in main data memory for each task in the FPS-100 (identified as .MPnnn, where nnn is the task identifier). An additional overlay table (ISRMAP) exists for the ISRs. The supervisor uses this table when loading tasks into program surce memory from main data memory. An entry in the overlay table represents one overlay segment or ISR. Each entry consists of eight main data words. The format of an entry in the overlay table is shown in Table 2-2.

Table 2-2 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 refers to the high mantissa portion, and LM refers to the low mantissa portion of the main data word.

2.2.3.3 PS Partition Table

Although an overlay table exists for each task, only one PS partition table exists for the entire supervisor environment. The 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).

2.2.3.4 Loading Overlay Segments

When the supervisor loads a task from main data memory into program source memory, it first gets the address of the overlay table and the number of overlay segments in the task from the TCB. It then checks word 6 of each entry in the overlay table. Word 6 contains the should-be-resident bit and the currently-resident bit for that segment. If those bits agree (that is, a should-be-resident segment is actually resident or a should-not-be-resident segment is not) the supervisor does nothing. If however, a should-be-resident segment is not resident in program source memory, the supervisor does the following: it finds the PS partition(s) the segment will go into and determines if it (they) already contain other segments. If so, the other segments are marked non-resident. The new segment is marked resident, the partition table entries are updated to point to it, and it is overlaid into PS memory. Thus the overlay table and the partition table are used not only to load the correct routines from main data memory but also to maintain a record of the overlay segments resident in program source memory at any given time.

2.3 HANDLING INTERRUPTS

There are four types of interrupts which occur on the FPS-100:

- I/O interrupts
- TRAP interrupts
- fatal interrupts (subroutine stack overflow)
- floating-point exception interrupts

After an interrupt occurs, the supervisor performs a minimum state save and determines the type of the interrupt. It then processes the interrupt according to its type.

2.3.1 I/O Interrupts

When an I/O interrupt is received the supervisor executes an interrupt acknowledge instruction to determine which device caused the interrupt. The configuration table contains a five-word entry for each external device connected to the FPS-100. Table 2-3 shows the format of an entry in the configuration table.

Table 2-3 Configuration Table Entry Format

WORD	PORTION (see note)	CONTENTS
1	LM	Priority mask. This value is used to mask out lower priority devices and prevent them from interrupting.
2	LM	Bit mask. Identifies this device's bit in IMASK.
3	LM	Pointer to this device's ISRs entry in the overlay map.
4	EXP, LM	Device order number (EXP) and physical device address (LM).
5	LM	Save area for old IMASK value.

NOTE

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

After determining which device interrupted the FPS-100, the supervisor then uses the information in the configuration table to mask out interrupts from lower priority devices and find the ISRs entry in the overlay map, which is used to load the ISR into PS memory if it is not already there. After the ISR is resident, control is transferred to it. Refer to Chapter 5 for more information concerning ISRs.

2.3.2 TRAP Interrupts

A TRAP interrupt occurs when a task executes the following code:

```
TRAP;  DB=@svcname;  LDTMA
```

In the previous statement, svcname is the name of a supervisor call (SVC). SVCs are described in detail in Chapter 3.

When a TRAP interrupt occurs, the supervisor saves the state of the executing task, gets DPX(0) through DPX(3) from the task's TCB, and jumps to the appropriate SVC. Upon completion of the SVC, it stores DPX(0) through DPX(3) and s-pad 0 back into the task's TCB, loads the highest priority ready task, and starts it running.

2.3.3 Fatal and Floating-point Exception Interrupts

Fatal and floating-point interrupts result in the offending task being terminated. The task is removed from the ready queue and then linked to the MORGUE exchange. Refer to section 1.7.2 for other details.

CHAPTER 3

SUPERVISOR CALLS

3.1 INTRODUCTION

Tasks running in the FPS-100 communicate with the supervisor and other tasks via supervisor calls (SVCs). An SVC can be thought of as an extended machine instruction. SVCs in conjunction with the hardware instructions provide the virtual instruction set of the FPS-100 supervisor environment.

As an example, consider the following less than optimal assembly code which adds three and two:

```
DPX(0)<DB; DB=2.0              "FETCH FIRST ARGUMENT
DPY(0)<DB; DB=3.0              "FETCH SECOND ARGUMENT
FADD DPX(0),DPY(0)             "COMPUTE 2.0 + 3.0
FADD                           "PUSH THROUGH ADDER
DPX(0)<FA                      "SAVE THE RESULT
BFPE ERROR                     "IF ERROR OCCURS, PROCESS
```

The parameters (or operands) of the function FADD are prepared and stored in argument registers (DPX and DPY registers). The computation is initiated with the FADD instruction and the result is available some time later. The result is then saved in a register for future reference.

An SVC functions similarly. The operands are loaded into argument registers (DPX registers). The SVC is then invoked and the result is available some time later. The SVC is invoked by executing the following instruction:

```
TRAP; DB=@svcname; LTDMA
```

For example, consider the following code which changes the priority of the executing task:

```
DPX(X0)<DB; DB=100             "FETCH NEW PRIORITY (ARGUMENT)
TRAP; DB=@RUNPRI; LDTMA        "ESTABLISH NEW PRIORITY
```

In this code, the parameter for the SVC is loaded into DPX(X0) in the same manner as the parameters for the FADD instruction were loaded into DPX(0) and DPY(0). The SVC (RUNPRI) is then executed.

3.2 AFFECTED REGISTERS

Tasks which make supervisor calls should expect values in the registers DPX(0) through DPX(3), s-pad 0, and SPFN to change because they are used to pass parameters and status information between the supervisor and the tasks. These registers are also referred to as DPX(X0)-DPX(X3) and R0.

3.3 ERROR CHECKING

Some SVCs can fail to perform their intended functions under certain circumstances. Errors can be caused by invalid operands, for instance. When an SVC fails, it indicates this falure by returning an error code to the R0 register and SPFN. A negative number in R0 and SPFN indicates an error condition. Since all successful SVCs leave R0 zero or positive, a general error condition can be tested for by using the BLT instruction. For example:

```
.
.
.
TRAP; DB=@SEND; LDTMA          "ATTEMPT TO SEND A MESSAGE
MOV R0,R0
BLT ERROR                      "BRANCH IF ERROR OCCURRED
.
.
.
```

Although the type of error is generally clear from the context of the SVC, an error handler can be written to determine the exact error by comparing the error code against possible error codes. This can be done in a manner similar to the following:

```
ERROR:  LDSPI R1; DB=ERRBSY    "LOAD MESSAGE BUSY ERROR CODE
        SUB# R1,R0             "IS IT A MESSAGE BUSY ERROR?
        BNE NXTMSG             "IF NOT, TRY ANOTHER ERROR CODE
        .                      "IF SO, IT IS A MESSAGE BUSY ERROR
        .
        .
```

Specific error codes that can result from invoking particular SVCs are described in sections 3.4, 3.5, and 3.6. Currently there are two SVC error codes:

ERRBSY (177777 octal): message busy.

The user is trying to send a message or answer which is still linked to an exchange queue because it has not been received yet.

ERRMSG (177776 octal): invalid message.

The user is trying to answer the last message received, but there was no last message.

3.4 TASK HANDLING SVCS

Task handling SVCs are used to change the priority and run status of tasks in a supervisor environment. Task handling SVCs include:

RUNPRI
SETPRI
RESUME

```
**********                                              **********
*        *                                              *        *
* RUNPRI *          ----- CHANGE RUN PRIORITY ----      * RUNPRI *
*        *                                              *        *
**********                                              **********
```

PURPOSE: This SVC changes the run priority of the currently executing task. It can also reset the priority of the executing task to its default value without the calling task knowing this value.

Upon completion of this SVC, task rescheduling occurs if the changed priority of the calling task is less than the priority of another ready task.

ASM100 CALLING SEQUENCE: This SVC is invoked by loading the priority into DPX(0) and executing a TRAP instruction. For example:

```
DPX(X0)<DB; DB=priority
TRAP; DB=@RUNPRI; LDTMA
```

PARAMETERS:

Register	Contents
DPX(0)	New run priority of the currently executing task. This priority can be a value from 1 to 377 octal, with 377 being the highest priority (most important). If the priority is specified as ZERO, the default priority for this task is substituted. Although data pad registers contain 38 bits, only an eight-bit priority is guaranteed to be compatible with future versions of MTS-100.

DESCRIPTION: This SVC loads X1 with the TCB address of the currently executing task and executes the SETPRI SVC.

FTN100 CALL: This SVC is invoked from an FTN100 task with the following call:

```
CALL ZRUNPR (prty)
```

prty	New priority for the task. This priority can be a value from 1 to 377 octal. If prty is specified as zero or a negative, the default priority for this task is used.

```
**********                                              **********
*        *                                              *        *
* SETPRI *          ---- SET TASK PRIORITY ----         * SETPRI *
*        *                                              *        *
**********                                              **********
```

PURPOSE: This SVC sets or changes the run priority of a specified task within the system. It can also reset the specified task's priority to its default value without the calling task knowing this value.

Upon completion of the SVC, task rescheduling occurs if the specified task is in a ready state.

ASM100 CALLING SEQUENCE: This SVC is invoked by loading the priority into X0, the task control block (TCB) address of the specified task in X1, and executing a TRAP instruction. For example:

```
DPX(X0)<DB; DB=priority
DPX(X1)<DB; DB=tcb address
TRAP; DB=@SETPRI; LDTMA
```

PARAMETERS:

Register	Contents
DPX(X0)	New run priority of the specified task. This priority can be a value from 1 to 377 octal, with 377 being the highest priority. If the priority is specified as ZERO, the default priority for the task is substituted. Although data pad registers contain 38 bits, only an eight-bit priority is guaranteed to be compatible with future versions of MTS-100.
DPX(X1)	TCB address of the task whose priority is to be set. The address is typically expressed as the name of a labeled common; e.g., /TCB001/ for task 1.

DESCRIPTION: If the priority specified in X0 is less than or equal to zero, the default priority is placed in the run priority word of the TCB; otherwise the specified priority is placed there. If the specified task was in the ready state, it is replaced in the ready queue at its new priority.

FTN100 CALL: This SVC is invoked from an FTN100 task with the following call:

```
CALL ZSETPR (prty, taddr)
```

prty New priority for the task. This priority can be a value from 1 to 377 octal. If prty is specified as zero or a negative, the default priority for this task is used.

taddr The TCB address of the task to be changed.

```
**********                                              **********
*        *                                              *        *
* RESUME *              ---- RESUME TASK ----           * RESUME *
*        *                                              *        *
**********                                              **********
```

PURPOSE: This SVC causes a suspended (waiting) task to be resumed. The TCB of the task is linked to the ready queue.

Upon completion of the SVC, task scheduling occurs.

ASM100 CALLING SEQUENCE: This SVC is invoked by loading the TCB address into X1 and executing a TRAP instruction. For example:

```
DPX(X1)<DB; DB=tcb address
TRAP; DB=@RESUME; LDTMA
```

PARAMETERS:

Register	Contents
DPX(X1)	TCB address of the task to be resumed.

DESCRIPTION: The task is placed in the ready queue, in priority order.

3.5 MESSAGE FACILITY SVCS

Message facility SVCs are used by tasks to send, receive, and answer messages. Message facility SVCs include:

SEND
WAIT
TWAIT
ANSWER
MSGANS
WAITA
TWAITA

```
**********                                              **********
*        *                                              *        *
*  SEND  *              ---- SEND MESSAGE ----          *  SEND  *
*        *                                              *        *
**********                                              **********
```

PURPOSE: This SVC attempts to exchange a message with another task. If no other task is waiting to receive it, the message is queued.

ASM100 CALLING SEQUENCE: This SVC is invoked by loading the message address into X0, the exchange address into X1, and executing a TRAP instruction. For example:

```
DPX(X0)<DB; DB=message address
DPX(X1)<DB; DB=exchange address
TRAP; DB=@SEND; LDTMA
```

PARAMETERS:

Register	Contents
DPX(X0)	Address in main data memory of the message to be sent.
DPX(X1)	Address in main data memory of the exchange where message is to be sent.

DESCRIPTION: The run priority of the current task is placed in the priority field of the message. The message's answer exchange is saved in the current task's TCB. An attempt is made to pass the message to a waiting task. If unsuccessful, the message is queued at the exchange.

ERROR CONDITIONS: If the message specified in X0 is currently linked elsewhere in the system, a MESSAGE BUSY ERROR (ERRBSY) is returned to R0 and SPFN.

FTN100 CALL: This SVC is invoked from an FTN100 task with the following call:

```
CALL ZSEND (xaddr, maddr)
```

xaddr Location of the message exchange to which the message is sent.

maddr Name of an array containing the message to send.

```
**********                                                  **********
*        *                                                  *        *
*  WAIT  *            ---- WAIT FOR MESSAGE ----            *  WAIT  *
*        *                                                  *        *
**********                                                  **********
```

PURPOSE: This SVC causes the current task to attempt to receive a message. If no message is waiting, the task is suspended until a message is available.

ASM100 CALLING SEQUENCE: This SVC is invoked by loading the exchange key into X1 and executing a TRAP instruction. For example:

```
DPX(X1)<DB; DB=exchange address
TRAP; DB=@WAIT; LDTMA
```

PARAMETERS:

Register	Contents
DPX(X1)	Address of the exchange at which to wait for the message.

VALUES RETURNED:

Register	Contents
DPX(X0)	Address in main data memory of the message received.
R0	Type of message received. The following values are possible: 020000 Answer message (ANSBIT is set) 000000 Normal message 010000 Time-out message (for timed waits) If the Sign bit is set (R0 is negative), R0 contains an error code.

DESCRIPTION: An attempt is made to dequeue the message. If successful, the task receives the message. If unsuccessful, the task is placed in the exchange's queue.

ERROR CONDITIONS: Error codes and message types returned in R0 are set to facilitate fast return dispatch. An example of this follows:

```
DPX(X1)<DB; DB=exchange        "LOAD EXCHANGE
TRAP; DB=@WAIT; LDTMA          "WAIT FOR MESSAGE
MOV R0,R0
BLT ERROR                      "R0, SPFN<0 INDICATES ERROR
MOVL R0,R0
MOVL R0,R0                     "MOVE ANSWER TO SIGN BIT
BLT ANSMSG                     "BRANCH IF ANSWER MESSAGE
.....                          "PROCESS NORMAL MESSAGE
```

A normal message can be processed immediately by executing a BEQ instruction after the initial MOV R0,R0 .

FTN100 CALL: This SVC is invoked from an FTN100 task with the following call:

```
CALL ZWAIT (1, dummy, xaddr, maddr, type)
```

dummy Dummy variable which is not used but must be present.

xaddr Location of the message exchange at which to wait.

maddr Location of the returned message.

type Type of message returned (refer to section 3.7.1).

```
**********                                                  **********
*        *                                                  *        *
* TWAIT  *          ---- TIMED WAIT FOR MESSAGE ----        * TWAIT  *
*        *                                                  *        *
**********                                                  **********
```

PURPOSE: This SVC causes the current task to attempt to receive a message. If no message is waiting, the task is suspended. The task can specify the maximum number of clock ticks to be suspended. It that time expires, a TIMEOUT message is sent.

ASM100 CALLING SEQUENCE: This SVC is invoked by loading the time limit into X0 and the exchange address into X1 and executing a TRAP instruction. For example:

```
DPX(X0)<DB; DB=time limit
DPX(X1)<DB; DB=exchange address
TRAP; DB=@TWAIT; LDTMA
```

PARAMETERS:

Register	Contents
DPX(X0)	Maximum number of clock ticks to wait before a TIMEOUT message is sent. This value must be specified as a 16-bit unsigned integer (from 1 to 65535 decimal) in the low mantissa portion of the word. If 0 is specified, TWAIT is used as a poll for a message. An attempt is made to receive a message, but if none is available a TIMEOUT is returned immediately.
DPX(X1)	Address of the exchange at which to wait for the message.

VALUES RETURNED:

Register	Contents
DPX(X0)	Address in main data memory of the message received.
R0	Type of message received. The following values are possible. 020000 Answer message (ANSBIT is set) 010000 Timeout message (TIMBIT is set) 000000 Normal message If the sign bit is set (R0 is negative), R0 contains an error code.

DESCRIPTION: An attempt is made to dequeue the message. If successful, the task receives the message. If not, the time limit is checked, and a timeout message is returned if the time limit is 0. If the time limit was not 0, the task is placed on the exchange queue, and the timed request into the clock queue.

ERROR CONDITIONS: Error codes and message types returned in R0 are set to facilitate fast return dispatch. An example of this follows:

```
DPX(X0)<DB; DB=time limit      "LOAD MAXIMUM WAIT TIME
DPX(X1)<DB; DB=exchange        "LOAD EXCHANGE ADDRESS
TRAP; DB=@TWAIT; LDTMA         "WAIT FOR MESSAGE OR LIMIT
MOV R0,R0
BLT ERROR                      "R0, SPFN<0 INDICATES ERROR
MOVL R0,R0
MOVL R0,R0                     "MOVE ANSWER TO SIGN BIT
BLT ANSMSG                     "BRANCH IF ANSWER MESSAGE
MOVL R0,R0                     "MOVE TIMEOUT TO SIGN BIT
BLT TIMMSG                     "BRANCH IF TIMEOUT MESSAGE
........                       "PROCESS NORMAL MESSAGE
```

A normal message can be processed immediately by executing a BEQ instruction after the initial MOV R0,R0 .

FTN100 CALL: This SVC is invoked from an FTN100 task with the following call:

```
CALL ZWAIT (2, tlim, xaddr, maddr, type)
```

tlim Maximum number of clock cycles to wait.

xaddr Location of the message exchange at which to wait.

maddr Location of the returned message.

type Type of message returned.

```
**********                                              **********
*        *                                              *        *
* ANSWER *              ---- SEND ANSWER ----           * ANSWER *
*        *                                              *        *
**********                                              **********
```

PURPOSE: This SVC returns the last message received to its original sender as an acknowledgement and frees it. This is used to simplify task synchronization and sharing of physical messages. It further removes the bookkeeping associated with busy and free messages from the user.

ASM100 CALLING SEQUENCE: This SVC is invoked by executing the TRAP instruction. For example:

```
TRAP; LDTMA; DB=@ANSWER
```

PARAMETERS: No parameters are required for this SVC.

DESCRIPTION: The address of the last message received is obtained from the current task's TCB. The answer exchange is obtained from the message, the message type is set to ANSWER, and a SEND SVC is executed.

ERROR CONDITIONS: If no previous message has been received, an INVALID MESSAGE ERROR (ERRMSG) is returned to SPFN and R0.

If the message is currently linked elsewhere, a MESSAGE BUSY ERROR (ERRBSY) is returned to SPFN and R0.

FTN100 CALL: This SVC is invoked from an FTN100 task with the following call:

```
CALL ZANSR (1,0)
```

```
**********                                              **********
*        *                                              *        *
* MSGANS *          ---- SEND MESSAGE ANSWER ----       * MSGANS *
*        *                                              *        *
**********                                              **********
```

PURPOSE: This SVC returns a message to its original sender as an acknowledgement and frees it to continue execution. This is used to simplify task synchronization and sharing of physical messages. It further removes the bookkeeping associated with busy and free messages from the user.

ASM100 CALLING SEQUENCE: This SVC is invoked by loading the message address into X0 and executing a TRAP instruction. For example:

```
DPX(X0)<DB; DB=message address
TRAP; DB=@MSGANS; LDTMA
```

PARAMETERS:

Registers	Contents
X0	Address in main data memory of the message to be returned.

DESCRIPTION: The answer exchange is obtained from the message, the message type is set to ANSWER, and a SEND SVC is executed.

ERROR CONDITIONS: If the message is currently linked elsewhere, a MESSAGE BUSY ERROR (ERRBSY) is returned to SPFN and R0.

FTN100 CALL: This SVC is invoked from an FTN100 task with the following call:

```
CALL ZANSR (2, maddr)
```

maddr Name of an array containing the answer message.

```
**********                                                  **********
*        *                                                  *        *
* WAITA  *              ---- WAIT FOR ANSWER ----            * WAITA  *
*        *                                                  *        *
**********                                                  **********
```

PURPOSE: This SVC permits a task to wait for a response to the previously sent message.

ASM100 CALLING SEQUENCE: This SVC is invoked by executing a TRAP instruction. For example:

```
TRAP; DB=@WAITA; LDTMA
```

PARAMETERS: No parameters are required for this SVC.

DESCRIPTION: The answer exchange contained in the last message sent is loaded from the current task's TCB and is used as the parameter to the WAIT SVC.

FTN100 CALL: This SVC is invoked from an FTN100 task with the following call:

```
CALL ZWAIT (3, dummy, dummy, maddr, type)
```

dummy — Dummy variables which are not used but must be present.

maddr — Location of the returned message.

type — Type of message returned.

```
**********                                                  **********
*        *                                                  *        *
* TWAITA *          ---- TIMED WAIT FOR ANSWER ----         * TWAITA *
*        *                                                  *        *
**********                                                  **********
```

PURPOSE: This SVC permits a task to wait for a specified amount of time for a response to the previously sent message.

ASM100 CALLING SEQUENCE: This SVC is invoked by loading the maximum wait time into X0 and executing a TRAP instruction. For example:

```
DPX(X0)<DB; DB=wait time (in ticks)
TRAP; DB=@TWATA; LDTMA
```

PARAMETERS:

Register	Contents
DPX(X0)	Maximum number of clock cycles to wait before a TIMEOUT message is sent. This value must be specified as a 16-bit unsigned integer in the low mantissa portion of the word. If 0 is specified, either the pending message or a TIMEOUT is returned immediately.

VALUES RETURNED:

Register	Contents
DPX(X0)	Address of the message received.

DESCRIPTION: The answer exchange contained in the last message sent is loaded from the current task's TCB and is used as the parameter to the TWAIT SVC.

ERROR CONDITIONS: If the time limit expires before a message is received, the TIMEOUT message is returned to R0 and SPFN.

FTN100 CALL: This SVC is invoked from an FTN100 task with the following call:

```
CALL ZWAIT (4, tlim, dummy, maddr, type)
```

tlim Maximum number of clock cycles to wait.

dummy Dummy variable which is not used, but must be present.

maddr Location of the returned message.

type Type of message returned.

3.6 MISCELLANEOUS SVCS

The only miscellaneous SVC currently provided is SETFPE.

```
**********                                                  **********
*        *                                                  *        *
* SETFPE *       ---- SET FLOATING-POINT EXCEPTION ----      * SETFPE *
*        *                                                  *        *
**********                                                  **********
```

PURPOSE: This SVC enables and disables the floating-point exception interrupt under task control. Although this function can be performed directly by writing the APSTAT2 register, user tasks should not manipulate that register directly. This interrupt is normally disabled under MTS100.

ASM100 CALLING SEQUENCE: This SVC is invoked by loading the logical enable/disable value into X0 and executing a TRAP instruction. For example:

```
DPX(X0)<DB; DB=logical enable/disable value
TRAP; DB=@SETFPE; LDTMA
```

PARAMETERS:

Register	Contents
DPX(X0)	Logical enable/disable value. A value of ZERO disables floating-point exception interrupts. Any non-ZERO value enables floating-point exception interrupts.

DESCRIPTION: The logical value of R0 (0 if ZERO, 1 if non-ZERO) is set in the INTE bit of APSTAT2 and in the saved APSTAT2 of the current task.

FTN100 CALL: This SVC is invoked from an FTN100 task with the following call:

```
CALL ZSETFX(opt)
```

opt — Logical enable/disable value. Possible values are:

- 0 — Disable floating-point exception interrupts
- 1 — Enable floating-point exception interrupts

3.7 FTN100 SVCS

Sections 3.4, 3.5, and 3.6 provided a description of each SVC and its calling sequences in ASM100 and FTN100. This section provides a complete description of the call to each of FTN100-callable routines.

3.7.1 ZWAIT

This subroutine is called in order to wait for a message or an answer. The format of the call is as follows:

```
CALL ZWAIT (opt, tlim, xaddr, maddr, mtype)
```

opt — Type of wait. One of the following can be specified:

1. Wait until a message arrives.
2. Timed wait for a message.
3. Wait until an answer arrives.
4. Timed wait for an answer.

tlim — Maximum number of clock ticks to wait when opt is specified as 2 or 4.

xaddr — Location of the message exchange at which to wait, when opt is specified as 1 or 2. When opt is specified as 3 or 4, the exchange location is obtained from the last message sent by this task.

maddr — Location of the returned message.

mtype — Type of message returned. The following are the message types:

1. Normal message
2. Answer message
3. Timeout message

3.7.2 ZSEND

This subroutine is called in order to send a message. The format of the call is as follows:

```
CALL ZSEND (xaddr, maddr)
```

xaddr — Location of the message exchange to which the message is sent.

maddr — Name of an array containing the message to send.

3.7.3 ZANSR

This subroutine is called to send an answer. The format of the call is as follows:

```
CALL ZANSR (opt, maddr)
```

opt — Type of answer to send. One of the following can be specified.

1. Send an answer using the last message received.
2. Send an answer using the message at location maddr.

The message exchange location is obtained from the message itself.

maddr — Name of an array containing the answer message.

3.7.4 ZRUNPR

This subroutine is called to change the run priority of the calling task. The format of the call is as follows:

```
CALL ZRUNPR (prty)
```

prty — New priority for this task. This priority can be a value from 1 to 377 octal. If prty is specified as zero or a negative number, the default priority for this task is used.

3.7.5 ZSETPR

This subroutine is called to change the run priority of another task. The format of the call is as follows:

```
CALL ZSETPR (prty, taddr)
```

prty — New priority for the task. This priority can be a value from 1 to 377 octal. If prty is specified as zero or a negative number, the default priority for this task is used.

taddr — The TCB address of the task to be changed.

3.7.6 ZSETFX

This subroutine is called to control floating-point exception interrupts. The format of the call is as follows:

```
CALL ZSETFX (opt)
```

opt — Logical enable/disable value. One of the following can be specified:

0 Disable floating-point exception interrupts.
1 Enable floating-point exception interrupts.

CHAPTER 4

COMMUNICATIONS

4.1 COMMUNICATION BETWEEN TASKS

In order to communicate, tasks transfer messages using SVCs. Message exchanges are the mechanisms used for the actual exchange of information. This section describes messages, message exchanges, and types of tasks which send or receive messages. SVCs were described in Chapter 3.

4.1.1 Messages

The message is the basic unit of data with which tasks communicate. The message is an area of main data memory which is written and read by tasks. It is not actually transferred from one area of memory to another when sent, but its address is passed from one task to another by means of SVCs and message exchanges.

A message consists of a standard header and the message body. The header is in the same format as the header of the TCB, which is described in section 2.2.2.1. The format of the message header is shown in Table 4-1.

Table 4-1 Message Header

WORD	CONTENTS
1	Right link (RLINK)
2	Left link (LLINK)
3	Run priority (RPRI)
4	Type of message (TYPE)
5	Message length (LENGTH)
6	Answer exchange address (ANSKEY)

The entries in the message header are used as follows:

RLINK and LLINK — Pointers that the supervisor uses to update and maintain exchanges. If multiple messages are available at an exchange, they are queued by the supervisor using these pointers. Refer to section 4.1.2 for a description of message exchanges.

RPRI — Run priority of the task which sends the message. This priority is used if the task which receives the message is a slaved task. Upon receipt of the message, the priority of a slaved task is changed to the priority specified in the RPRI field. The slaved task then executes at that new priority.

TYPE — Type of the message. The low mantissa portion of this word is set as follows to indicate the type:

000000	Normal user message
020000	Answer message
010000	Timeout message

Additional status information (for exchange headers) is also stored in the low mantissa portion, as follows:

002000 — Indicates that the message portion of the message exchange is currently in use. Refer to section 4.1.2 for a further description.

001000 — Indicates that the task queue portion of the message exchange is currently in use. Refer to section 4.1.2 for a further description.

The exponent portion of this word also contains status information, as follows:

000001 — Indicates that the message is currently busy (i.e., has been sent as a normal message but not returned as an answer).

LENGTH — Length of the entire message in main data words (at least six).

ANSKEY — Answer exchange. When a task sends a message, it may specify the exchange where the message may be returned as an answer. Thus, the sending task knows where to WAIT, and the receiving task need not know where to send the answer.

The body of the message consists of whatever data the sending task wishes to transmit to the receiving task. The message may consist of a buffer of data to be processed by the receiving task, an empty buffer to be filled by the receiving task, an indicator to transfer control to the receiving task, or any other information. The message body can be of any length.

In order to send a message to another task, the message must be in the proper format. The user normally defines messages and supplies all header information by assembly time. Once the message is created, it can be used any number of times.

4.1.2 Message Exchanges

Message exchanges are the mechanisms through which tasks send and receive messages. Conceptually, the message exchange is a pair of queues, the waiting task queue and the message queue. When a task sends a message (with the SEND, ANSWER, or MSGANS SVCs) it designates an exchange on which to send the message. If no task is waiting at the exchange when the message is sent, the message is placed on the message queue.

When a task wants to receive a message, it issues a WAIT, TWAIT, WAITA, or TWAITA SVC and includes an exchange from which to get the message (or one is obtained automatically as a part of the SVC.) If there are no messages available at the exchange when the receiving task issues one of the wait commands, this task is placed on the waiting task queue.

Actually, a message exchange consists of one queue which can be either a message queue or a waiting task queue. Only one queue is needed since either there are more messages available than waiting tasks, in which case the messages are queued, or there are more tasks waiting than messages, in which case the tasks are queued. At the head of this queue is a common data area which is similar to the header of the ready queue. This header contains pointers RLINK, LLINK, and a flag which indicates whether the queue is a message queue or a waiting task queue.

4.1.2.1 Message Queue

If more messages are available than tasks waiting to receive them, the message exchange is a message queue. In this case, RLINK of the header points to the message that has been waiting longest at the exchange (the first message sent which had no task waiting to receive it). LLINK of the header points to the message that has been waiting the least (the last message sent). Since each message has a standard header with pointers RLINK and LLINK (refer to section 4.1.1), these values are set in the messages waiting at the exchange to continue the queue. In the header of the first message sent, RLINK points to the second message sent, and LLINK points to the queue header. This continues throughout the queued messages until in the last message sent, RLINK points to the queue header, and LLINK points to the second to the last message sent. This is similar to the ready queue header and tasks.

Thus, whenever a task issues a WAIT, TWAIT, WAITA, or TWAITA SVC in order to receive a message, the supervisor supplies it with the address of the first message sent. The supervisor then removes the message from the message queue by adjusting the RLINK and LLINK values of both the header and the second message sent.

4.1.2.2 Waiting Task Queue

If there is an excess of tasks waiting for messages, the message exchange is a waiting task queue. In this case, RLINK of the header points to the TCB of the task which has been waiting the longest (first waiting task). LLINK of the header points to the task which has been waiting the least amount of time (last waiting task). In the TCB of the first waiting task, RLINK points to the TCB of the second waiting task and LLINK points to the header. This continues until in the TCB of the last waiting task, RLINK points to the header, and LLINK points to the TCB of the second to the last waiting task.

Thus, whenever a message is sent to the exchange, the supervisor supplies the address of that message to the first waiting task. The supervisor also removes the first waiting task from the waiting task queue by adjusting the RLINK and LLINK values of both the header and the second waiting task.

From the information in this section and the previous one, it can be seen that there is no limit to the number of messages or tasks that can be queued. No list or table is maintained, other than the pointers in the messages and TCBs themselves. Thus, if there is enough memory available for the message (which should be known at load time), it can be sent.

4.1.2.3 Relationship Between the Ready Queue and Message Exchanges

As described in section 2.1.1.3, tasks which wait for messages or answers (by issuing WAIT, TWAIT, WAITA, or TWAITA SVCs) are removed from the ready queue. The supervisor attempts to supply that task with the address of a message from the exchange. If a message is available, the supervisor then places the task back into the ready queue at its correct priority.

However, if, no message is available, the supervisor removes the task from the ready queue and places it in the waiting task queue of the appropriate exchange. This is done by adjusting the links in the ready and exchange queues (i.e., the links in the task's TCB and in the TCBs of the tasks it was linked to in the ready queue and the tasks it becomes linked to in the exchange queue). The task is now no longer in a ready state but in a wating state. When the task reaches the top of the queue and is supplied with a message, the supervisor again adjusts the links and places the task back into the ready queue at the proper priority.

4.1.3 Clock Queue

A task which performs timed waits for messages or answers (TWAIT or TWAITA) conceptually waits at two exchanges: the message exchange where it waits for the message to become available, and the clock exchange where it waits for the allotted time to expire.

The clock queue is similar to the ready queue and message exchanges. It is connected to the real time clock, and all tasks on this queue are performing timed waits and so are also waiting at message exchanges. Like the headers of the ready queue and the message exchanges, the header of the clock queue contains pointers to the TCBs of the first and last tasks on the queue. The tasks waiting at this exchange, however, are ordered by the length of their waits. Those waiting the least amount of time are placed at the beginning of the queue, and those waiting the longest are placed at the end. The pointers in the header of the clock exchange are RCLOCK and LCLOCK. RCLOCK points to the TCB of the task waiting for the least amount of time, and LCLOCK points to the TCB of the task waiting the longest.

The TCB of each task also contains pointers RCLOCK and LCLOCK (refer to Table 2-1). These pointers are used to link the clock queue in the same manner that the RLINK and LLINK pointers link the ready queue and message exchanges. In the TCB of the task waiting the shortest amount of time, RCLOCK points to the TCB of the task waiting the second shortest amount of time and LCLOCK points to the header. This continues throughout the queue until in the task that waits for the longest amount of time, RCLOCK points to the header, and LCLOCK points to the TCB of the task that waits for the second longest amount of time.

The TCB of each task also contains a field called ICLOCK, which, for tasks waiting in the clock queue, specifies an interval of time to wait. ICLOCK contains the difference between the number of clock ticks this task can wait and the number of clock ticks the task just ahead of it in the queue can wait. Thus for a task waiting in the clock queue, the sum of its ICLOCK value and the ICLOCK values of all the tasks ahead of it in the queue is equal to the amount of time that the task can wait.

For example, assume that four tasks are on the clock queue, each performing timed waits for messages. Task 1 waits 10 ticks, task 2 waits 40 ticks, task 3 waits 35 ticks, and task 4 waits 100 ticks. Table 4-2 shows the order of the queue and the relationship between the ICLOCK values and the timed wait values.

Table 4-2 ICLOCK Values

ORDER OF QUEUE	ICLOCK VALUE	MAXIMUM NUMBER OF CYCLES TO WAIT
Task 1	10	10
Task 3	25	10 + 25 = 35
Task 2	5	10 + 25 + 5 = 40
Task 4	60	10 + 25 + 5 + 60 = 100

If a number of clock ticks pass equal to the value of ICLOCK for the first task in the queue and that task has not received a message, that task times out. The supervisor removes it from the clock queue by adjusting RCLOCK and LCLOCK values, sends it a timeout message, removes it from the message exchange where it was waiting, and places it back in the ready queue. No adjustment is made to ICLOCK values of other tasks.

If, however, the first task in the message exchange receives the message it was waiting for before timing out, the supervisor removes that task from the clock queue and updates the ICLOCK value of the next task in the clock queue to reflect the difference in time. The supervisor also removes the task from the message exchange and places it back on the ready queue.

4.1.4 Types of Tasks

In general, tasks can be classified into two types, message-producing tasks and message-consuming tasks. The processing flow of each of these types is normally cyclic. The message-producing task performs a function, requests the services of another task by sending it a message, waits for an answer, and continues the whole procedure over. The message-consuming task waits until a message is available, performs a function, sends an answer indicating that the function was performed, and waits for the next message. Figure 4-1 illustrates the processing flow for each of these types.

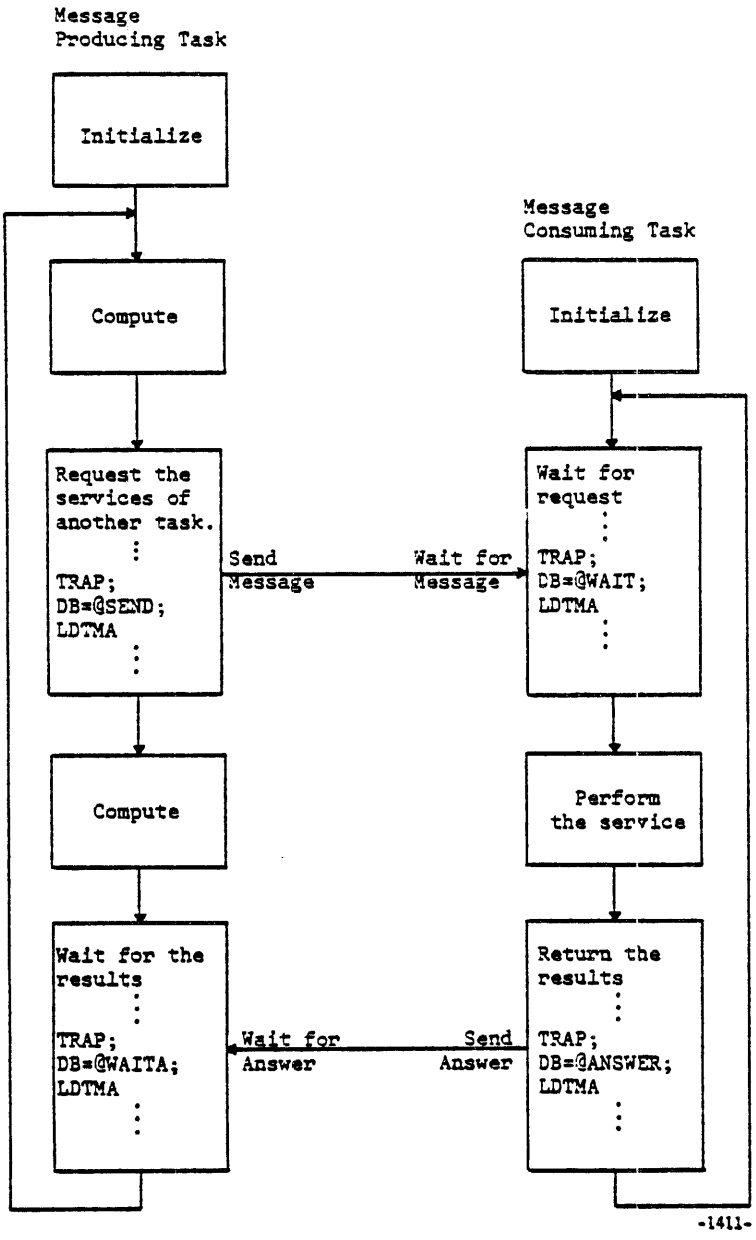


Figure 4-1 Message-producing and Message-consuming Tasks

A typical application of a message-producing and a message-consuming task is a real-time data-acquisition task and a data-reduction task. The data acquisition task must be at a high priority in order to operate on a real-time basis. This task collects the data, for example, from a device handler, and sends it in a message to the data-reduction task. Thus the data-acquisition task is the message-producing task. The data-reduction task waits until it receives a buffer full of data sent as a message and returns the empty buffer to the data-acquisition task in the form of an answer. The empty buffer can be filled again and sent back. The data-reduction task is the message-consuming task and can operate at a lower priority since the data-reduction operation is not as time critical as the data acquisition.

Notice that the distinction between message-producing tasks and message-consuming tasks is relative since both tasks produce and consume messages. The producing/consuming concept can, however, clarify the general structure of tasks. In general, a producer task executes the following steps:

1. performs a function
2. sends a message
3. waits for an answer
4. loops back to begin again

A consumer task executes the following steps:

1. waits for a message
2. performs a function
3. sends an answer
4. loops back to begin again

4.2 COMMUNICATION BETWEEN THE HOST AND THE FPS-100

Communication between a program running on the host computer and the FPS-100 is facilitated through the interaction of the host resident executive (APX100) and the FPS-100 resident supervisor (MTS100). APX100 transfers data and instructions from the host to the FPS-100. The supervisor interprets the instructions and causes the FPS-100 to perform the requested functions.

The FPS-100 can, however, be used with or without the supervisor. Therefore, like the FPS-100, APX100 has two modes of operation, supervisor mode and non-supervisor mode. The default mode for APX100 is non-supervisor mode. When APX100 and the FPS-100 are in non-supervisor mode, APX100 functions exactly like APEX funtions with an AP-120B/AP-190L array processor. All requests by APX100 routines to manipulate the FPS-100 are executed directly through the hardware registers. Non-supervisor mode is used to initialize the FPS-100 and load the supervisor into place. Non-supervisor mode can also be used to run the FPS-100 without the supervisor. The FPS-100 diagnostics and the hardware debugger can be used in this mode.

The other mode of operation for APX100 and the FPS-100 is supervisor mode. In supervisor mode APX100 assumes that the supervisor is running in the FPS-100. In this mode both APX100 and the supervisor add additional layers of software between the APX100 requests and the FPS-100 interface hardware. Instead of directly manipulating the FPS-100 front panel registers (SWR, LITES, and FN registers), APX100 routines communicate with software SWR, LITES, and FN registers on the host. Actual communication with the FPS-100 is initiated by interrupting the FPS-100. When this happens the interrupt from the host is serviced by a special ISR on the FPS-100, the host interrupt processor (HIRP), which obtains the data and processes the request. Thus in supervisor mode, the APX100 routines never actually manipulate FPS-100 registers; they only send messages to the supervisor, which does the manipulation.

The additional software layers present in supervisor mode are shown in Figure 4-2. The APX100 task, the function interpreter task, and the HIRP shown in this figure are discussed in the sections which follow.

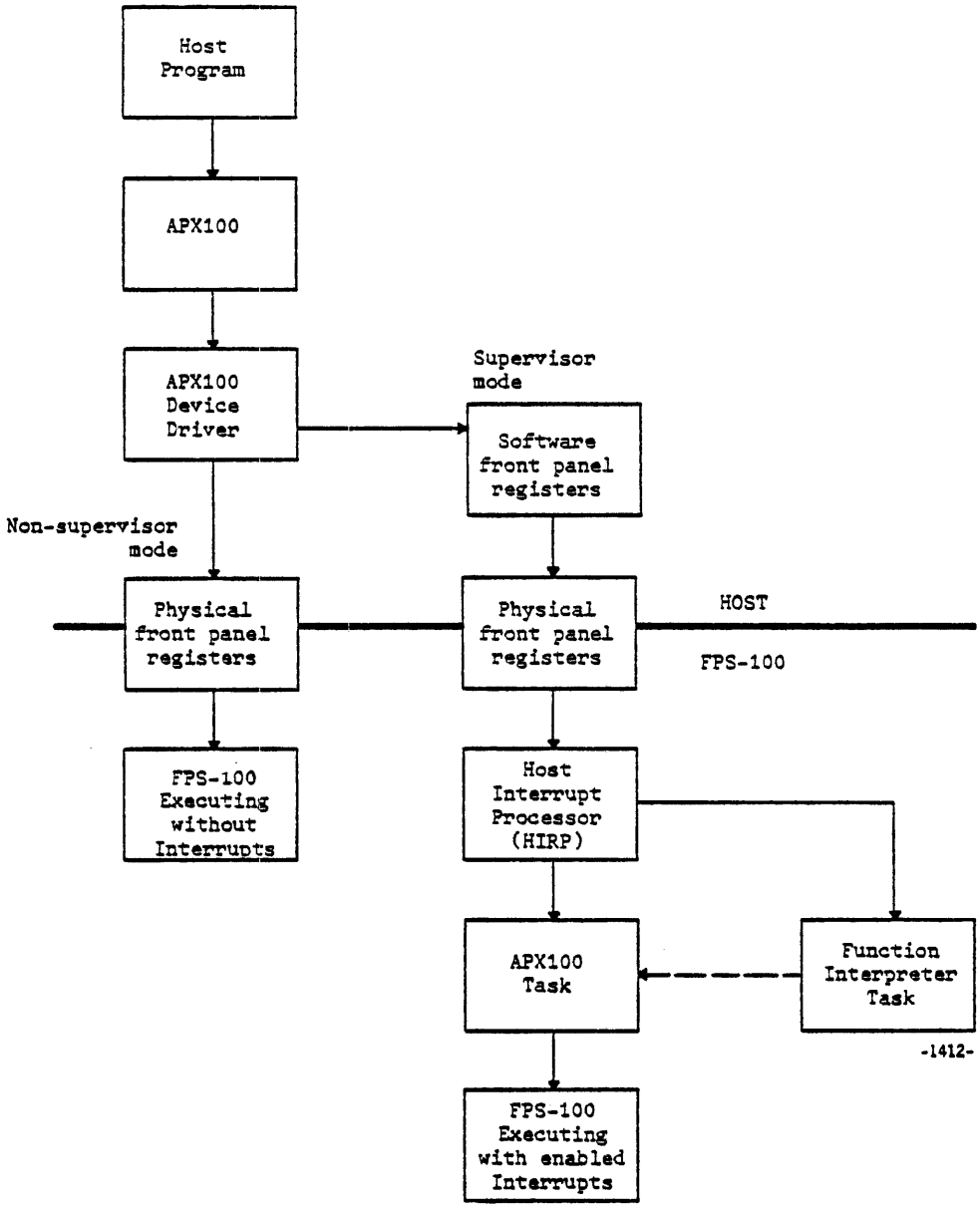


Figure 4-2 Supervisor Mode Software Layers

4.2.1 Interrupting the FPS-100

The host interrupts the FPS-100 for one of three reasons:

- to perform a virtual front panel operation
- to run a host-callable subroutine
- to communicate with a task using the communication ports

Each of these types of operations requires that the host and the FPS-100 become synchronized and pass data. This is done with the SWR register, the LITES register, and two hardware status bits (one associated with each register), SWRACK and LITACK.

The host can read the LITES register and, in so doing, automatically clears the LITACK bit. The host can write the SWR register. When it does this, SWRACK is automatically cleared. Conversely, the FPS-100 can write the LITES register. In doing this, it automatically sets LITACK. It can also read the SWR register and doing so automatically sets SWRACK.

Because of automatic setting of the status bits, host communication with the FPS-100 is performed in the following steps.

1. The host tests the FPS-100 to determine whether the SWR register is available. When it is, the host writes the first 16 bits of the message to the SWR register, thereby clearing SWRACK. It then interrupts the FPS-100 and waits until SWRACK is set by the FPS-100.

2. Upon receiving the interrupt from the host, the host interrupt processor (HIRP) is called. This FPS-supplied ISR, whose sole purpose is to process interrupts sent by the host, reads the SWR register (thereby setting SWRACK), saves the SWR value in a message buffer, and waits until the host clears SWRACK.

3. When the host determines that SWRACK has been set, it writes the second 16 bits of data into SWR (thereby clearing SWRACK) and waits again.

4. When the FPS-100 determines that SWRACK has been cleared, it reads SWR (thereby setting SWRACK), saves the second 16 bits of the message in the message buffer, and waits again.

5. Steps 3 and 4 are repeated until the entire message is transferred to the FPS-100. Both the host and the FPS-100 are aware of the number of transfers needed because this information is transmitted in the message.

6. Upon receipt of the entire message, HIRP sends the message to an exchange. For a virtual front panel operation, the message goes to the function interpreter task. For a host-callable subroutine, the message goes to the APX100 task. For a communication port, the message is placed on a port exchange. These procedures are described in the remainder of this section (4.2).

When the function interpreter task or APX100 task is finished, the FPS-100 sends a three-word message to the host acknowledging this fact. This message contains a status flag and the updated contents of the LITES register. This three-word message is sent by the FPS-100 in a manner analogous to the one previously described, with the FPS-100 writing the LITES register and the host reading it.

4.2.2 Virtual Front Panel Operations

Three registers in the virtual front panel are treated in a special manner when the FPS-100 and APX100 are in supervisor mode. These registers are the LITES register, the SWR register, and the FN register. APX100 routines do not communicate directly with the virtual front panel in supervisor mode; they communicate with simulated registers on the host.

4.2.2.1 LITES Register

An APX100 routine can read the LITES register. When this happens in supervisor mode, it actually reads the simulated LITES register in the host. The simulated LITES register contains the value of the LITES register which was sent in the last message from the FPS-100 (refer to section 4.2.1).

4.2.2.2 SWR Register

An APX100 routine can write the SWR register. When this happens in supervisor mode, it actually writes the simulated SWR register in the host. The simulated SWR register is sent to the FPS-100 with the next message (refer to section 4.2.1).

4.2.2.3 FN Register

APX100 routines can both read and write the FN register. Reading the FN register in supervisor mode involves nothing unusual. When a routine writes the FN register, however, it intends to perform a function in the FPS-100. Therefore, a message is sent from the host to the FPS-100 to indicate the function to be performed.

When an APX100 routine writes the function register, the APX100 driver interrupts the FPS-100 and sends the contents of the SWR register and the FN register to the FPS-100 using the method described in section 4.2.1. The HIRP in the FPS-100 processes the interrupt and sends the contents of the FN register and SWR register in a message to the FPS-supplied function interpreter task. The function interpreter task examines the contents of the FN and SWR registers and performs the requested function, not using the values in the actual machine registers but using the register values in the APX100 task's TCB. After the function has been performed, the function interpreter task sends a message back to the host. This message contains a status message and the updated LITES register.

4.2.2.4 Supervisor Mode Front Panel Differences

When the FPS-100 and APX100 are operating in supervisor mode, some front panel functions are not performed in the same manner as in non-supervisor mode. Some front panel functions do not operate at all. The differences between supervisor mode and non-supervisor mode are outlined in the following paragraphs, along with the affected APX100 routines. The APX100 routines are described in detail in the APX100 Manual.

hardware breakpoint - SETBRK/CLRBRK

Although the hardware breakpoint mechanism functions in non-supervisor mode, it is not supported in supervisor mode. In supervisor mode the SWR register is used for message transfers. The SWR register is also required to contain the breakpoint address when performing a SETBRK or a CLRBRK. This conflict necessitates the non-support.

read s-pad register - APGSP

In non-supervisor mode this function operates on a halted FPS-100. In supervisor mode this function is not supported.

examine and deposit - APEXAM/APDEP

In supervisor mode not all registers in the FPS-100 are accessible. The accessible registers are associated with the APX100 task and include:

	SPD
TMA	DPA
SPFN	APSTAT
DA	
MA	SWR
FN	LITES
APMA	HMA
WC	CTRL
FMTH	FMTL
MASK	APMAE
MAE	

The inaccessible registers include all 38-bit registers and memory, program source memory, real time clock registers, IMASK, APSTAT2, APSTAT3, and SMA. If an inaccessible register access is attempted, an error routine is called.

PS functions of save, run, and restore - PSFUNC

In supervisor mode the use of PSFUNC is not supported. In non-supervisor mode this routine permits the execution of a small program in the FPS-100 while preserving the previous code located in those PS memory words.

4.2.3 Host-callable Subroutines

In order to run host-callable subroutines in a supervisor controlled FPS-100, these subroutines must be loaded as a part of the APX100 task. The root overlay segment of the APX100 task is supplied by FPS and permits host-callable subroutines to operate in the same manner as they do in a non-supervisor environment. All host-callable subroutines must be loaded by the user as additional parts of that task.

During FPS-100 execution, the APX100 task runs like any other task in the system. It runs at an assigned priority and waits at exchanges for messages. It normally waits at an exchange until it receives a message to start running a routine. This happens when the host HASI routine is called. This LOD100-generated subroutine contains a call to the APX100 routine APRUN, which in turn calls the SPLDGO routine. The SPLDGO routine interrupts the FPS-100 and initiates a message transfer described in section 4.2.1. When HIRP receives the message from the host and interprets it as a SPLDGO message, it puts the parameters into a common block in memory and sends the APX100 task a message which shifts it from the wait state to the ready state. The APX100 task resumes execution when it is the highest priority ready task, moves the parameters into the physical s-pads, and jumps to the entry point passed down from the HASI. This is actually the entry to the routine which ensures that all the overlays requested by the host are resident in PS memory and then jumps to the real user subroutine.

Both UDC (user-directed calls) routines and ADC (auto-directed calls) routines can be called in the supervisor environment. However, care must be taken when using UDC routines. With UDC routines, the user places data into the FPS-100 main data memory with APPUT calls. The user must not place data in any main data locations that are used by the supervisor. The user should study the load map produced by LOD100 to determine which main data locations are available or should use the BUFFER command at load time. Refer to the LOD100 Reference Manual for further information concerning UDC and ADC routines and the load map.

4.2.4 Communication Ports

In the supervisor environment, seven full-duplex communication ports are provided to allow limited communication between the host program and tasks running on the FPS-100. (Normally, the host program can communicate directly with only those routines running as part of the APX100 task). Using routines provided as part of APX100, the host program can send a small (13-bit) message to any one of seven communication ports in the FPS-100. The host can also receive a 13-bit message from any communication port in the FPS-100. Likewise, any task in the FPS-100 can send a 13-bit value to any one of the communication ports and receive a 13-bit value from any one of the communication ports.

On the host side, APX100 includes the routines necessary for sending and receiving messages through communication ports. On the FPS-100 side, the supervisor includes the routines necessary for tasks to send and receive the communication port messages.

4.2.4.1 Host Routines

Three APX100 routines, HPUT, HGET, and HTST, are provided to allow the host program to send and receive 13-bit data values.

The HPUT routine is used by the host program to send a message to the FPS-100. The format of the call to HPUT is as follows:

```
CALL HPUT(dest,value)
```

dest — Integer which identifies the communication port to which the message is sent. Values from 1 through 7 can be specified.

value — Value which is sent to the FPS-100. This value must not exceed 13 bits.

The HGET routine is used by the host program to get a message from the FPS-100. If a message is not available, this routine waits until a message becomes available. The format of the call to HGET is as follows:

```
CALL HGET(sors,flag,value)
```

sors Integer which identifies the communication port from which to get the message. Possible values include:

- 1-7 The message is obtained from the communication port of the same number. If no message is available at that port, HGET waits until one becomes available.
- 0 HGET gets a message from the lowest numbered communication port at which a message is available. If no message is available, HGET waits until one becomes available at some message port. When the message is received, sors is updated by HGET to indicate the number of the message port from which the message was received.

flag Status flag. HGET sets the value of this location, as follows:

- 0 A message was received and the FPS-100 continues processing.
- 1 The FPS-100 halted, and no message is available.
- 2 A message is available, but the FPS-100 has halted.

value Location in which HGET returns the message value. A value is returned when flag equals 0 or 2.

The HTST routine is used by the host program to get a message from the FPS-100, if one is available. HTST is similar to HGET except that HTST does not wait if a message is not available. The format of the call to HTST is as follows:

```
CALL HTST(sors,flag,value)
```

sors Integer which identifies the communication port from which to get the message. Possible values include:

- 1-7 The message is obtained from the communication port of the same number. If no message is available at that port, no message is returned.

0 HTST gets a message from the lowest numbered communication port at which a message is available. If no message is available none is returned. When a message is available, sors is updated by HTST to indicate the number of the message port from which the message was received.

flag Status flag. HTST sets the value of this location, as follows:

-1 No message is available and the FPS-100 is running.

0 A message was received and the FPS-100 is running.

1 No message is available and the FPS-100 is halted.

2 A message is available and the FPS-100 is halted.

value Location in which HTST returns the message value. A value is returned when flag equals 0 or 2.

4.2.4.2 FPS-100 Routines

Three supervisor routines, FPUT, FGET, and FTST, are provided to allow tasks to send and receive 13-bit data values. These routines are not called with the TRAP instruction like SVCs are, but are jumped to directly with the JSR instruction.

```
**********                                                  **********
*        *                                                  *        *
*  FPUT  *          ---- SEND DATA VALUE TO HOST ----       *  FPUT  *
*        *                                                  *        *
**********                                                  **********
```

PURPOSE: This routine permits a task to send a 13-bit value to the host.

ASM100 CALLING SEQUENCE: This routine is invoked by loading the destination value into s-pad 1, the data value into s-pad 2, and executing a JSR instruction. For example:

```
LDSPI R1; DB=dest
LDSPI R2; DB=value
JSR FPUT
```

PARAMETERS:

Register	Contents
s-pad 1	Integer which identifies the communication port to which the message is sent. Values from 1 through 7 can be specified.
s-pad 2	Value which is sent to the host. This value must not exceed 13 bits. Any extra bits are lost from the upper three bits.

REGISTERS USED: R0, R1, R2, DPX(0)

```
**********                                                  **********
*        *                                                  *        *
*  FGET  *        ---- GET DATA VALUE FROM HOST ----        *  FGET  *
*        *                                                  *        *
**********                                                  **********
```

PURPOSE: This routine permits a task to get a 13-bit value from any one of seven communication ports on the host. If no value is available, this routine waits until one is available. The availability of a value is indicated by a message on the associated exchange.

ASM100 CALLING SEQUENCE: This routine is invoked by loading the source value into s-pad 1 and executing a JSR instruction. For example:

```
LDSPI R1; DB=sors
JSR FGET
```

PARAMETERS:

Register	Contents
s-pad 1	Integer which identifies the communication port from which to get the message. Values from 1 through 7 can be specified.

VALUES RETURNED:

Register	Contents
s-pad 2	Status flag. This register is set as follows: -1 A 0 was specified for the source value in s-pad 1. Only values from 1 to 7 are allowed. 0 A 13-bit value is available in s-pad 3. 1 A value greater than 7 or less than 0 was specified for the source value in s-pad 1.
s-pad 3	The 13-bit data value received from the host.

REGISTERS USED: R0 through R3, DPX(0) through DPX(3)

```
**********                                                    **********
*        *                                                    *        *
*  FTST  *      ---- TEST IF DATA VALUE IS AVAILABLE ----      *  FTST  *
*        *                                                    *        *
**********                                                    **********
```

PURPOSE: This routine permits a task to get a 13-bit value from any one of seven communication ports on the host, if one is available. FTST is similar to FGET except that FTST never waits for a message. The availability of a message is indicated by the value returned in s-pad 2.

ASM100 CALLING SEQUENCE: This routine is invoked by loading the source value into s-pad 1 and executing a JSR instruction. For example:

```
LDSPI R1; DB=sors
JSR FTST
```

PARAMETERS:

Register	Contents
s-pad 1	Integer which identifies the communication port from which to get the message. Possible values include:
	1-7 The message is obtained from the communication port of the same number. If no message is available at that port, no message is returned.
	0 FTST gets a message from the lowest numbered communication port at which a message is available. If no message is available, none is returned.

VALUES RETURNED:

Register	Contents
s-pad 1	If s-pad 1 was specified as 0 and a message was returned, the number of the communication port from which the message was received is returned in s-pad 1.

s-pad 2 Status flag. This register is set as follows:

-1 No data value is available.

0 A 13-bit data value is available in s-pad 3.

1 A value greater than 7 or less than 0 was specified for the source value in s-pad 1.

s-pad 3 The 13-bit data value received from the host. This value is available when s-pad 2 is set to 0.

REGISTERS USED: R0 through R3, DPX(0) through DPX(3)

4.2.4.3 Communication Port Mechanism

Seven locations on the host are provided to store the 13-bit values received from the FPS-100, one for each communication port. Corresponding locations are available on the FPS-100 to receive the data values sent from the host. Also, on the FPS-100, for each communication port there exists a dedicated message exchange. Whenever a 13-bit value arrives in the FPS-100, the supervisor sends a message to the corresponding dedicated message exchange. This message notifies a task that a value is available.

The mechanism used to actually pass the 13-bit data values between the host and the FPS-100 is the same mechanism described in section 4.2.1. When the host program sends a 13-bit value to a communication port (using HPUT), the host actually writes into the SWR register the 13 bits of message (plus a 3-bit communication port identification number). The host then interrupts the FPS-100. In the FPS-100, the HIRP reads the SWR register, determines that the value is a message to a communication port, saves the 13-bit value in one of seven locations depending on the value of the 3-bit port number, and sends a message to the corresponding dedicated message exchange.

In order to receive the message, a task must call FGET or FTST and specify the same port number as the host routine specified in the HPUT call. The FGET or FTST call causes the task to retreive a message, if available, from the message exchange dedicated to the specified communication port. If no message is available, FGET causes the task to wait until a message arrives.

Going the other direction, a task sends a 13-bit value to the host using FPUT. This value is sent by the supervisor, along with the 3-bit port number, to the host using the LITES register (this method is described in section 4.2.1). In the host, the 13-bit value is placed in one of seven locations, depending on the port number. In order to receive this message, the host routine must call HGET or HTST with the proper port number specified. If a value is not available, HGET waits until a value arrives.

4.3 COMMUNICATION BETWEEN TASKS AND I/O DEVICES

Tasks communicate with I/O devices through device handlers. For the FPS interface products IOP and GPIOP, general purpose device handlers are provided with the supervisor. For other user devices attached to FPS-supplied interfaces, the user must create specific device handlers. Chapter 5 contains information that the user needs to create device handlers.

Communication between tasks and device handlers consists of a task sending a message to the device handler requesting a data transfer. The device handler performs the transfer and passes the message back to the requesting task.

In order for a task to obtain a buffer of data from an I/O device (a read operation), the requesting task sends to the handler a message containing the buffer to be filled. The task can then continue processing asynchronously from the data transfer, which is done using the DMA mechanism. At some later time, the task waits for an answer from the handler. The answer message is sent when the I/O device completes the requested operation. Upon completion, the device interrupts the FPS-100, causing the ISR portion of the device handler to be called, which in turn sends an answer message to the requesting task. This answer consists of the original message with the buffer now full of data.

If the requesting task sends more than one request at a time (with more than one empty buffer), the messages are queued, and the handler automatically performs multiple buffering. The number of requests originally sent determines the level of multiple buffering.

In order for a task to transmit a buffer of data to an I/O device (write operation), the task sends the buffer as a message to the handler. The handler performs the output operation. Upon completion the device interrupts the FPS-100, causing the ISR portion of the device handler to send an answer to the requesting task. The answer includes the original buffer which indicates to the task that the operation is complete and the buffer can be reused.

CHAPTER 5

DEVICE HANDLERS

5.1 INTRODUCTION

Device handlers are routines which control the I/O transfer between external devices and the FPS-100. A device handler must be provided for each I/O device connected to the FPS-100. Device handlers are structured into two parts, a device controller task which handles requests from the user and a device servicer ISR which handles interrupts from the device.

Since a great variety of I/O devices are available to the user, FPS cannot supply a full function device handler for each of these devices. Instead, FPS provides the software tools which allow the user to create special purpose device handlers. This chapter contains the information necessary for the user to create individual device handlers. FPS does, however, provide two complete special purpose device handlers, an IOP handler and a GPIOP handler. Instructions for their use are contained in this chapter.

5.2 DEVICE HANDLER FUNCTIONS

The user must provide three functions when creating device handlers. These include:

- initializing the I/O device
- controlling the I/O device
- transferring a buffer of data to (or from) the device

These functions are initiated by other tasks which send service requests to the device handlers. The service requests are in the form of messages, the standard communication mechanism in a supervisor environment. The messages specify the type of request (if the handler performs more than one type of request), the device (if the handler can support multiple devices), any parameters needed by the handler to perform the request, and the buffer of data (if needed).

With device handlers controlling the entire I/O software interface, the user does not have to worry about how individual I/O devices work. The user works with device handlers which perform the mechanics of the I/O transfers.

5.3 DEVICE HANDLER STRUCTURE

A device handler contains two main parts, the device servicer and the device controller. The device servicer is an interrupt service routine (ISR) and is called as a result of an interrupt by an I/O device. Its primary function is to respond to the immediate demands of the interrupting device. The device controller is itself a task. It is called by tasks which send messages to it in order to request service. The controller performs functions such as initializing the device, providing successive buffer information to the device driver, providing device status to the calling task, and shutting down the device.

When a task calls a device handler to perform a function on an I/O device, it sends a message to the device controller. The controller determines the type of function to be performed and performs the function on the I/O device. If the function is an I/O transfer function, the controller queues the request on an internal queue and executes it as soon as the device is available. The device performs the I/O transfer; when the transfer is complete, it interrupts the FPS-100. When the device interrupts the FPS-100, the device servicer sends an answer back to the requesting task, indicating the completion of the function.

5.3.1 Device Controller

The device controller processes tasks I/O requests. It is a task itself and is scheduled to run according to its priority, just as any other task in the system. When a device controller is started, the first thing it does is wait for a message. When a message is received, it processes the request and waits for another message. The basic functions performed by each device controller are:

- Device initialization. The controller initializes the device table and, if necessary, initializes the device in some manner. A GPIOP device is loaded with the required PPAL code at this time.

- Data transfer. The controller must queue requests to read or write buffers of data. The I/O device performs the operations as soon as possible. The notification of the completion of the transfer comes from the device servicer, not from the controller.

- Other functions. If there are any other control functions required for the user I/O device, they must be included in the controller portion of the device handler.

The processing flow for a device controller is contained in the following steps.

1. Initialize the device and the handler tables.

2. Wait for a message request.

3. Determine the type of request. If it is a control request, go to step 4; if it is an I/O transfer request, go to step 5.

4. For a control request:

   (a) Perform the requested function.

   (b) Send an answer back to the user.

   Go directly to step 6.

5. For I/O transfer requests:

   (a) Select the proper device execute queue.

   (b) Place the buffer address in the queue.

   (c) If it is the first buffer in the queue:

   - start the I/O transfer
   - enable the device interrupt

6. Go to step 2 and wait for the next message.

Figure 5-1 illustrates the logic flow.

5.3.2 Device Servicer

The device servicer responds to interrupts generated by I/O devices. It is an interrupt service routine (ISR) and is treated as an overlayable subroutine of the supervisor.

When the I/O device interrupts the FPS-100 to indicate the completion of a data transfer, the appropriate device servicer is called. The device servicer sends the just-transferred buffer back to the requesting task as an answer. If another I/O request has been queued by the controller, the servicer starts the I/O device processing the next request.

The user can make the device servicer as intelligent or as simple as desired. The trade-off is in interrupt latency. The smaller the servicer, the faster the response time.

Since the servicer is an ISR (a subroutine called by the supervisor itself), only the minimum state can be used (minus s-pad 7).

The following is the typical flow logic of the device servicer:

1. Select the proper device execute queue. This is done via the device order number.

2. Get the first buffer from the queue and send it to the requesting task as an answer. (When the servicer is called because of an interrupt, the first address out of the device execute queue is the address of the completed buffer.)

If the execute queue is not empty, perform steps 3, 4, and 5.

3. Get the next address in the queue (but do not send it to a task).

4. Restart the I/O device with the next transfer, using that address as the buffer address.

5. Zero the mask bit parameter, signifying that the associated interrupt is to be re-enabled.

6. Return to the supervisor.

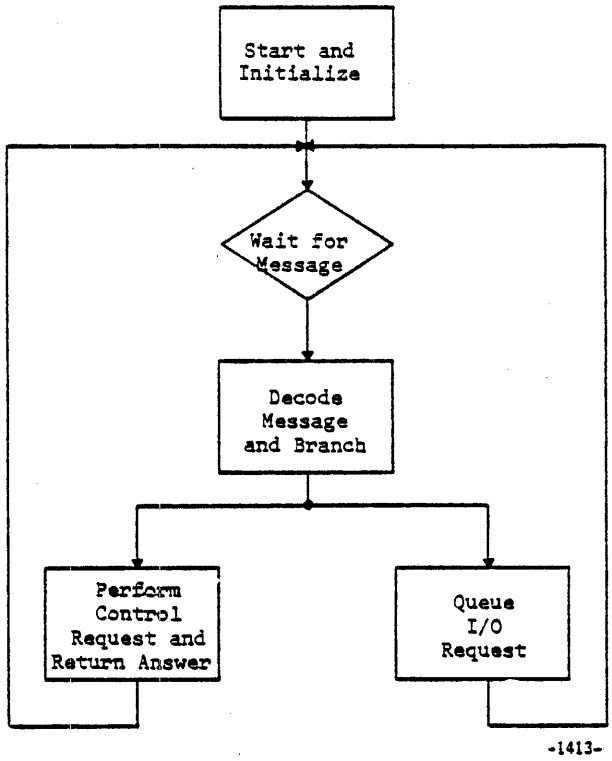


Figure 5-1 Device Controller Structure

5.4 IOP HANDLER

The IOP handler provided by FPS can perform only I/O transfer requests. To initiate I/O transfers using the IOP handler, a requesting task must send a message to the IOP controller task. The format of this message is shown in Table 5-1.

Table 5-1 IOP Handler Message Format

DESCRIPTION	WORD	CONTENTS
Standard message header data	1	RLINK
	2	LLINK
	3	RPRI
	4	TYPE
	5	LENGTH
	6	ANSKEY
Continuation of header for IOP message	7	STATUS (transfer status)
	8	IOP address
	9	Buffer word count
	10	External device address (part 1)
	11	External device address (part 2)
	12	Command function
Data buffer	13	Buffer data
	.	.
	.	.
	.	.

In Table 5-1, the standard message header is the same as described in section 4.1.1. The status returned by the IOP handler is 0 if no error occurred in the transfer and -1 if a transfer error occurred. The rest of the IOP header information is described in the IOP Manual.

5.5 GPIOP HANDLER

The GPIOP handler provided by FPS can perform several types of requests. Two types of message formats can be used when sending messages to the GPIOP handler.

Table 5-2 contains the format of the execute IOC message. This message causes the handler to start the GPIOP and perform a function.

Table 5-2 GPIOP Execute IOC Message Format

DESCRIPTION	WORD	CONTENTS
Standard message header data	1	RLINK
	2	LLINK
	3	RPRI
	4	TYPE
	5	LENGTH
	6	ANSKEY
	7	1
Continuation of header for GPIOP message	8	GPIOP number
	9	Function value
	10	Control packet address

In Table 5-2, the standard message header is the same as described in section 4.1.1. The rest of the GPIOP header information is similar to the parameters of GPEXEC, which are described in the GPIOP Software Reference Manual.

Table 5-3 contains the format of the start new buffer GPIOP message. This message is used to initiate new buffer transfers if the GPIOP is already running.

Table 5-3 Start New Buffer GPIOP Message Format

DESCRIPTION	WORD	CONTENTS
Standard message header data	1	RLINK
	2	LLINK
	3	RPRI
	4	TYPE
	5	LENGTH
	6	ANSKEY
	7	2
Continuation of header for GPIOP message	8	GPIOP number
	9	Remaining buffer count
	10	Return status
Data buffer	11	Buffer data
	.	.
	.	.
	.	.

In Table 5-3, the standard message header is the same as described in section 4.1.1. The status returned by the GPIOP handler is 0 if no errors occurred and -1 if an error occurred in the GPIOP. The rest of the GPIOP header information is similar to GPSYNC parameters, which are described in the GPIOP Software Reference Manual.

CHAPTER 6

USING MTS-100

6.1 CONCEPTS

The FPS-100 running under the MTS-100 supervisor has two basic units of work: the task and the host-callable subroutine (HCS). The HCS, as its name implies, is called directly from the host. That is, if the statement "CALL SUBR" (where SUBR is an FPS-100 subroutine) appears in the host FORTRAN mainline, then SUBR runs on the FPS-100 and comes to a virtual halt. At the same time, the multi-tasking supervisor coordinates the running of the user's tasks, which are typically self-contained routines running semi-independently of each other and of the host. The term "semi-independently" is used because although the tasks can be independent, they can also communicate for such purposes as control and data transfer.

Tasks communicate with each other by calling various supervisor services to send or wait for messages. Messages are sent to a predefined exchange, where a waiting task may pick them up. If no task is waiting, the message remains at the exchange until a task asks the exchange for a message. The routines which manage this communication system are referred to as supervisor calls (SVCs). A complete description of these and other types of SVCs are found in Chapter 3.

Tasks may communicate with the host in two ways. One way is via the HCSs, which are under host control but can send and receive messages to and from tasks. The advantage of this method is its flexibility, since an HCS can have a number of parameters which can also be arrays. The other way is via direct host/FPS-100 communication routines which provide limited bi-directional access. The advantages of this method are lower overhead and a greater measure of control for the FPS-100 than an HCS master/slave relationship normally allows. These methods can also be used to control the timing and placement of data transfers to and from the host.

6.2 MECHANICS

This section sets forth the basics of writing tasks, making load modules, and writing the host FORTRAN mainline to communicate with the FPS-100.

6.2.1 Writing a Task

At its simplest, a task is merely a routine which executes after other tasks of higher priority have suspended execution. It is defined as a task via the $TASK pseudo-op, which includes the task ID number and, optionally, a priority level (refer to the ASM100 Reference Manual for format and parameters). A task should terminate or suspend via the WAIT SVC. In other words, it should either wait for a message to start it up again, or it should wait at an exchange where no messages are received, such as the supervisor's NEVER exchange. For example, a task which executes once and terminates follows:

```
$TASK 1                     (task ID=1, default priority=100)
$TITLE PROC
$ENTRY PROC
$INSERT COMSYS              (defines system commons, including the
                            NEVER exchange)
PROC: .                     (process data in MD)
      .
      .
      .
      .
      .
   DPX(1)<NEVER             (parameter for WAIT: exchange name)
   TRAP; DB=@WAIT; LDTMA    (exit via the wait SVC)
$END
```

WARNING

A task should never terminate by executing a HALT instruction or simple RETURN, because this will cause the system to go down.

Exchanges and messages are data structures in main data (MD) memory that are expressed in the form of labeled commons. The user who wishes to send messages between tasks must define exchanges and messages in the following manner: for each message or exchange, create a common to be referenced by the tasks and define the contents of the common in a "block data" with the following format (refer to the ASM100 Reference Manual for $COMMON and $DATA).

For an exchange:

```
$COMMON /exchname/  dummy(6)   /I
$DATA     dummy (1)  exchname
$DATA     dummy (2)  exchname
$DATA     dummy (3)     0
$DATA     dummy (4)     0
$DATA     dummy (5)     6
$DATA     dummy (6)     0
```

For a message of length 'len':

```
$COMMON  /msgname/  dummy(len)   /I
$DATA     dummy (1) msgname
$DATA     dummy (2) msgname
$DATA     dummy (3)   0
$DATA     dummy (4)   0
$DATA     dummy (5)  len
$DATA     dummy (6)   0
$DATA     dummy (7)  value
  .          .         .
  .          .         .
  .          .         .
  .          .         .
$DATA     dummy (len) value
```

For example, the following task processes data, sends a message to another task telling it to start working on the result, and suspends itself until more data is available to process.

```
$TASK 1
$TITLE PROC
$ENTRY PROC
$COMMON  /EXCHA/ A(6)  /I
$COMMON  /EXCHB/ B(6)  /I
$COMMON  /MSGA/  MA(6) /I
PROC:
     ---
     ---
     ---
     DPX(0)<MSGA              (set up parameters
     DPX(1)<EXCHA             for the SEND SVC)
     TRAP; DB=@SEND; LDTMA    (send MSGA to EXCHA)
     DPX(1)<EXCHB             (exchange at which to wait for more
                               data)
     TRAP; DB=@WAIT; LDTMA    (wait there until a msg is received)
     JMP PROC                 (start over)
$END


$TITLE STRUCT
$COMMON /EXCHA/ A(6)  /I
$COMMON /EXCHB/ B(6)  /I
$COMMON /MSGA/ MA(6)  /I

$DATA  A(1)  EXCHA
$DATA  A(2)  EXCHA
$DATA  A(3)  0
$DATA  A(4)  0
$DATA  A(5)  6
$DATA  A(6)  0

$DATA  B(1)  EXCHB
$DATA  B(2)  EXCHB
$DATA  B(3)  0
$DATA  B(4)  0
$DATA  B(5)  6
$DATA  B(6)  0

$DATA  MA(1) MSGA
$DATA  MA(2) MSGA
$DATA  MA(3) 0
$DATA  MA(4) 0
$DATA  MA(5) 6
$DATA  MA(6) 0

$END
```

A task can also start with a WAIT for data from an I/O device, process it, send the result elsewhere, and then loop back to the start (which is the call to WAIT).

6.2.1.1 SVC Return Codes

Some SVCs return error codes in s-pad 0 for optional checking on return to the user program. Specific error codes are listed elsewhere, but any negative value indicates an error; zero indicates a normal return. The SVCs currently returning error codes are SEND, ANSWER, and MSGANS.

For other SVCs (in particular, the various WAITs), s-pad 0 returns a different sort of value (in this case, the type of message received by the waiting task).

6.2.1.2 Overlays within Tasks

Some tasks (particularly AP-FORTRAN tasks) may be too large to fit conveniently into PS memory and therefore may have to be split into separate overlays. The loader employs a tree-structure method for defining overlays at load time, in which each overlay is given an overlay segment number when it is loaded. The user uses this number when calling one overlay segment from another. The user writing tasks in FPS-100 assembly language (ASM100) loads in a new segment by executing the following:

```
LDSPI 10; DB=n
JSR OVLD
```

where "n" is the new overlay segment's number, and OVLD is declared an external symbol.

The call to OVLD places the new overlay in PS memory and then returns to the point of call. Subroutines in the new overlay are executed by calling with a JSR in the normal manner.

NOTE

The segment numbers assigned at load time are interpreted as octal, which is also the default at assembly time.

The AP-FORTRAN user places the new overlay in PS memory by:

```
CALL APOVLD (n)
```

A subroutine in the new overlay can then be called as usual:

```
CALL NEWSUB (args)
```

Refer to the LOD100 Reference Manual for more details on overlays.

6.2.2 Making a Load Module with LOD100

This section discusses the techniques, restrictions, and potential pitfalls involved in making load modules using LOD100.

6.2.2.1 Space Allocation

When the user creates a load module for the FPS-100, all tasks and ISRs are allocated space in MD, where they will permanently reside as back-up storage, and in PS, into which they are overlaid from MD. This implies two things:

- The user's entire job must fit into MD.
- The system is static in the sense that task X always runs in the same area of PS.

The job which must fit into MD includes all tasks, host-callable subroutines, and ISRs (at two MD words per PS word), plus system data space (TCBs, maps, system commons) and user data space (exchanges, messages, data). It does not include the permanently PS-resident supervisor.

Since the PS space is pre-allocated, the user must decide at LOD100 time which routines should share PS space, if necessary. To obtain the fastest execution time, the amount of overlaying from MD to PS should be kept to a minimum. The best way to achieve this is by having each area of PS dedicated to only one task, with each overlay segment within each task in a dedicated sub-area. Thus, once made resident in PS, it remains permanently resident. Unfortunately, there may not be enough PS space for all tasks to reside permanently, in which case the user must decide where two or more overlay segments or even whole tasks overlaying each other within the same area of PS would cause the least overhead.

For example, given 4K of PS minus 1K for the supervisor (leaving 3K of available space), the problem is to allocate space for three tasks: task 1 = 2K, task 2 = 1/2K, and task 3 = 1K. Figure 6-1 presents three possible allocations.

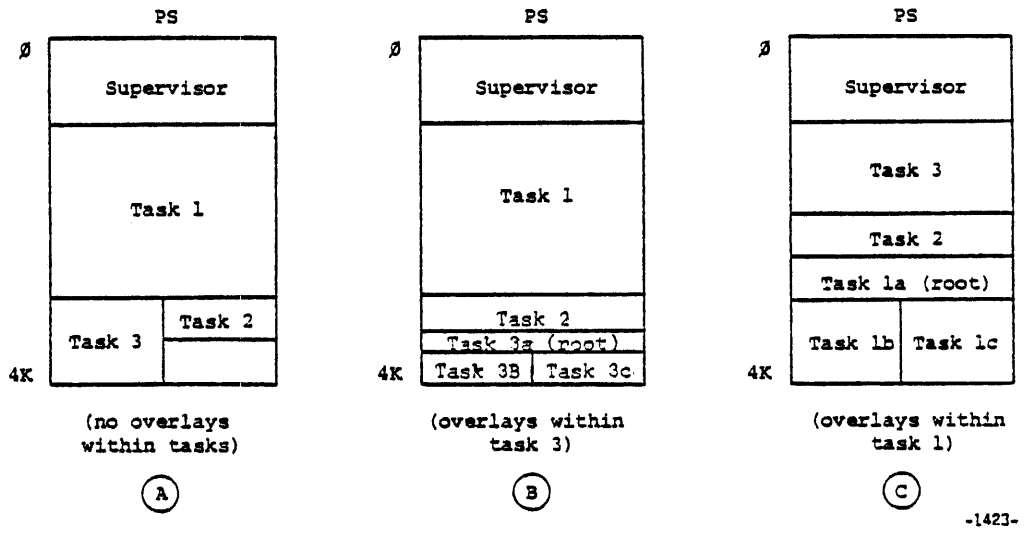


Figure 6-1 Possible PS Allocations

The appropriate choice for the application is based upon relative desired response times for each task.

Unless directed otherwise, the loader allocates PS space for tasks on a non-sharing basis. That is, if one task is allocated locations n through m, then the next task loaded is allocated space starting at location m+1. If PS sharing is necessary between tasks (as in Figure 6-1A), then the user must reset the base PS address used by the loader (via the PS offset command) before loading the next task. However, the sharing of space within a task is automatically taken care of by the form of the TREE command (refer to the LOD100 Reference Manual).

Thus, the loading sequence for Figure 6-1B (assuming task 3 is split into overlays 30, 31, and 32) includes the following commands:

```
TREE ((1))
OV 1
LOAD task 1 object
LINK
MAP 0 task 1 mapfile
TREE ((2))
OV 2
LOAD task 2 object
LINK
MAP 0 task 2 mapfile
TREE ((30 (31) (32) ))
OV 30
LOAD task 3 root object
OV 31
LOAD overlay segment 31 object
OV 32
LOAD overlay segment 32 object
LINK
MAP 0 task 3 mapfile
```

This produces a PS space allocation such as that depicted in Figure 6-1B. Figure 6-1C has the same load sequence, with the exception that task 3 takes the place of task 1 and vice versa. For Figure 6-1A, the sequence is:

```
TREE ((1))
OV 1
LOAD task 1 object
LINK
MAP 0 task 1 mapfile
TREE ((2))
OV 2
LOAD task 2 object
LINK
MAP 0 task 2 mapfile
MAP
```

(Here the user must check the map (entering "MAP" displays the map on the terminal) for task 2 to find out its PS starting location, which should be the same for task 3. Assuming location n, the command sequence proceeds as follows:)

```
PS n
TREE ((3))
OV 3
LOAD task 3 object
LINK
MAP 0 task 3 mapfile
```

Note that the use of "TREE ((k))" and "OV k" for task k is not necessary; it merely makes reading the maps easier. "TREE ((1))" and "OV 1" are acceptable for any one-segment task. These commands must be present for even the simplest task, so that the supervisor tables can be properly initialized by the loader.

Note that, within a task, overlay segments which are conceptually on the same level of the tree structure may be put below one another on the tree without unnecessary overlaying resulting. That is, the higher nodes do not always have to be made resident in PS when lower nodes are needed.

Figure 6-2A illustrates an overlay tree structure which can, if desired, be expressed in the form of Figure 6-2B. The advantage of B, if space is available, is that segment 4 does not have to share PS space with segments 2 and 3. At the same time, it is not necessary for segment 3 to become resident for segment 1 to call segment 4.

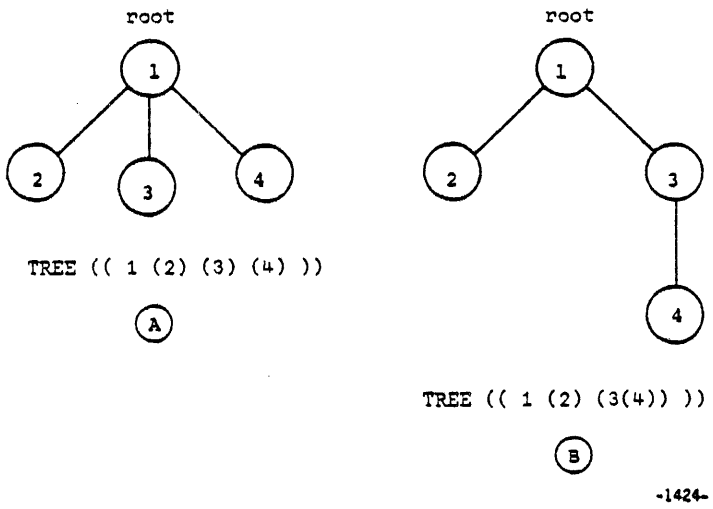


Figure 6-2 Overlay Tree Structures

6.2.2.2 Host-callable Subroutines

Host-callable subroutines are identified at LOD100 time by the CALL command. This causes the loader to create FORTRAN subroutines (referred to as HASIs) for the host FORTRAN program to call. When the host calls such a routine, control goes to the HASI on the host, then to the APX100 task on the FPS-100, and finally to the subroutine itself. Thus all host-callable subroutines are considered part of the APX100 task under MTS100. They can be loaded as separate overlay segments of the task or in any combination, with the FPS-supplied APX100 task as the root. If only one host-callable subroutine exists, it should be loaded in the same overlay as the APX100 task.

For example, consider two host-callable subroutines that are called repeatedly. If there is enough PS space for them to remain permanently resident, it is most efficient for them to be loaded into PS as only one overlay, with no sharing of PS space either within the APX100 task itself or between it and other tasks. The command sequence for this is as follows:

```
TREE ((1))
OV 1
LOAD UPEX.B       (load APX100 task root)
CALL SUBR1  /     (create HASI for following subroutine)
LOAD SUBR1.B      (load first subroutines object)
CALL SUBR2  /     (create HASI for second subroutine)
LOAD SUBR2.B      (load second subroutines object)
LINK
MAP 0 mapfile
```

Note the slash in the CALL command. This is necessary to ensure that the subroutine is resident in PS before jumping to it.

Now consider another example, in which three host-callable subroutines must be put into such limited PS space that they must all use the same area. This case is equivalent to the tree structure shown in Figure 6-2A. The command sequence is:

```
TREE ((1 (2) (3) (4) ))
OV 1
LOAD UPEX.B               (APX100 task root)
OV 2                      (define next overlay segment)
CALL SUBRA  /             (make HASI for SUBRA)
LOAD SUBRA.B              (load its object)
OV 3
CALL SUBRB  /
LOAD SUBRB.B
OV 4
CALL SUBRC  /
LOAD SUBRC.B
LINK
MAP 0 mapfile
```

The resulting PS space allocation is shown in Figure 6-3.

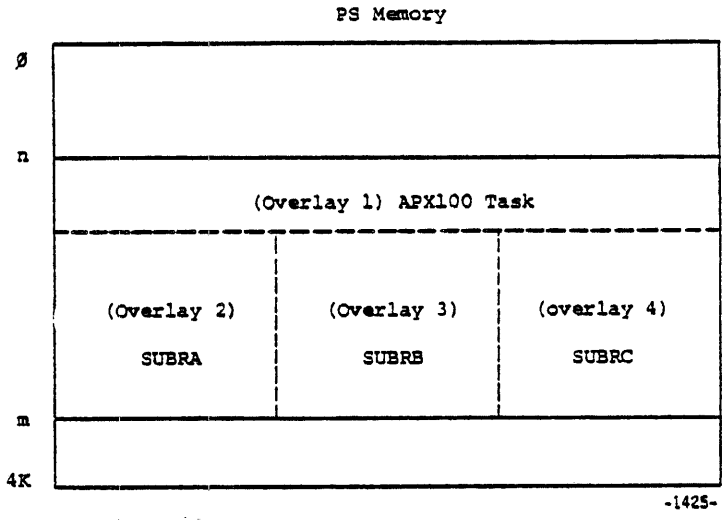


Figure 6-3 PS Allocation

On the other hand, the tree structure shown in Figure 6-2B has the same load sequence as that for Figure 6-2A (except for the TREE command), yet produces the PS space allocation shown in Figure 6-4.

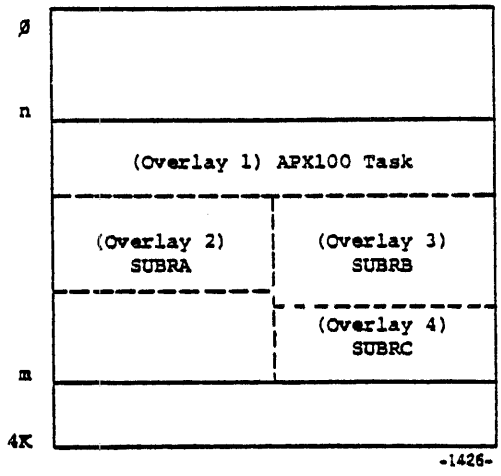


Figure 6-4 PS Allocation

Still another way of defining the structure is useful when, for example, SUBRB and SUBRC are often called in pairs. They then can be put into the same overlay, as Figure 6-5 shows:

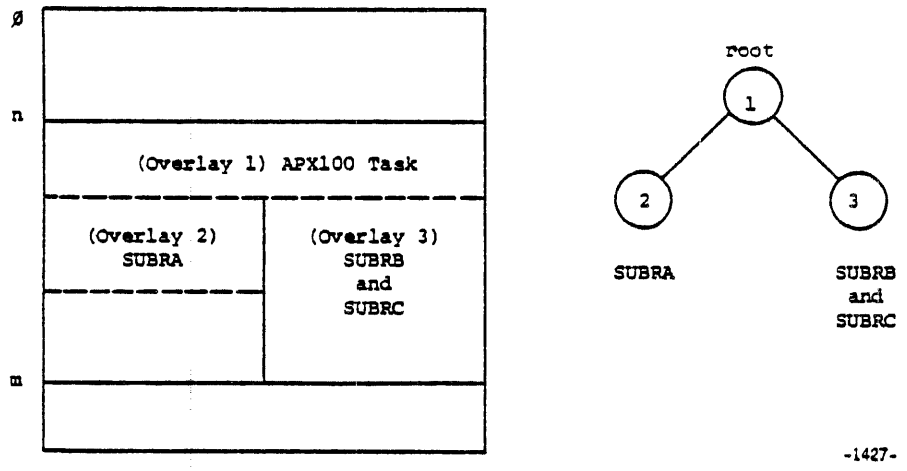


Figure 6-5 PS Allocation and Tree Structure

The above allocation can be achieved by using this command sequence:

```
TREE ((1 (2) (3) ))
OV 1
LOAD UPEX.B
OV 2
CALL SUBRA  /
LOAD SUBRA.B
OV 3
CALL SUBRB  /
LOAD SUBRB.B
CALL SUBRC  /
LOAD SUBRC.B
LINK
MAP 0 mapfile
```

NOTE

The CALL command, which tells the loader to create a HASI for the next routine loaded, must come <u>after</u> the OV command.

For information on the various types of HASIs (UDC and ADC), refer to the APX100 and LOD100 manuals (Table 1-1).

6.2.2.3 User Exchanges and Messages

If the tasks being loaded reference exchanges and messages, these data structures should be loaded beforehand, followed by the MARK command to ensure that the common block names will remain global symbols (i.e., accessible by all following tasks). They can also be loaded as part of a task, but they still must be MARKed.

6.2.2.4 Loading Device Handlers

For each I/O device to be supported, the user must load an interrupt service routine (ISR) and usually an I/O controller task. The controller task is loaded in the same way as any other task. An ISR, however, does not require the TREE or OV commands, since it always consists of one segment only. Normally, ISRs should not share PS space unless size restrictions absolutely require it; otherwise, interrupt response time suffers.

The most familiar ISR is the one for the host, call HIRP. This is loaded as follows:

```
LOAD HIRP.B
LINK
MAP 0 mapfile
```

Other ISRs are loaded in the same manner.

6.2.2.5 Loading the Supervisor

The supervisor kernel itself, the system exchanges and tables, the host communication routines, and the real-time clock queue support are all loaded via an FPS-supplied command file called LODSYS. When the command file terminates, it leaves the loader ready to load ISRs, tasks, etc., at the user's direction.

6.2.2.6 A Sample LOD100 Session

In the following example, two tasks and two host-callable subroutines are each one segment long, with no I/O devices other than the host. PS space need not be shared.

```
(run LOD100)
OUT hasifile lmfile  /D   (open files for HASIs and load module)
INPUT LODSYS              (load supervisor)
L HIRP.B                  (load host ISR)
LINK                      (link)
MAP 0 mapfile1            (output map)
TREE ((1))                (form structure for 1-segment task)
OV 1
LOAD TASK1.B              (load it)
LINK
MAP 0 mapfile2
TREE ((2))          )
OV 2                |
LOAD TASK2.B        }     (second task)
LINK                |
MAP 0 mapfile3      )
TREE ((3))                (APX100 task)
OV 3                      (everything in 1 overlay segment)
LOAD UPEX.B               (load APX100 root)
CALL SUBR1  /             (make HASI for SUBR1)
LOAD SUBR1.B              (load its object)
CALL SUBR2  /             (make HASI for SUBR2)
LOAD SUBR2.B              (load its object)
LINK                      (link)
MAP 0 mapfile4
EXIT                      (terminate loader)
```

On the other hand, if it is essential to conserve PS space, everything can be loaded starting at the same location, although in the following load sequence the ISR is still loaded in its own space.

```
(run LOD100)                    )
OUT hasifile lmfile  /D         |
INPUT LODSYS                    |
L HIRP.B                        |
LINK                            |
MAP 0 mapfile1                  } (same sequence as previous example)
TREE ((1))                      |
OV 1                            |
LOAD TASK1.B                    |
LINK                            |
MAP 0 mapfile2                  )
MAP                             (display map on terminal)
(read map to find TASK1's PS starting location n)
PS n                            (reset base PS address)
TREE ((2))                      )
OV 2                            |
LOAD TASK2.B                    } (make TASK2 as before)
LINK                            |
MAP 0 mapfile3                  )
PS n                            (reset base PS address again)
TREE ((3 (4) (5) ))             (structure subroutines to share space)
OV 3
LOAD UPEX.B                     (load APX100 root)
OV 4                            (next overlay segment)
CALL SUBR1  /                   (make HASI for SUBR1)
LOAD SUBR1.B                    (load its object)
OV 5                            (next overlay segment)
CALL SUBR2  /                   (make HASI for SUBR2)
LOAD SUBR2.B                    (load its object)
LINK                            (link)
MAP 0 mapfile4                  (generate map)
EXIT                            (terminate)
```

Note that the LINK command is always used after loading an ISR and after loading all the overlays in a task. It is wise to generate a map at each of those points, since some information disappears when the next item is loaded. The mapfile must be different in each case, or maps can be sent directly to the line printer. Alternatively, if the host can record terminal I/O in a file, the user should record the entire load session with all "MAP 0 mapfile" commands changed to "MAP". The maps will thus appear both on the terminal and, therefore, also in the record, eliminating the need for numerous separate map files.

6.2.2.7 Potential Problems

The user should experiment with the loader, storing or displaying maps after each command to get a feel for the kinds of symbols which appear, disappear, change, and remain permanently. One item which never appears is a table which the loader generates and initializes after the EXIT command is given. This table currently occupies 50 words of MD, starting at the next available location (DBBRK on the last map before EXITing). Following this table is the parameter passing area, used by ADC HASIs to pass host-callable subroutine parameters (possibly arrays) to the FPS-100. The symbol .PPA. which appears on the maps refers to this area inaccurately and should be ignored. The length of this area depends on the parameters sent by the host at run time and so is not known at load time, except that ostensibly it is allocated the remainder of MD. This may also be an area in which users with UDC HASIs can explicitly pass parameters via APPUT and APGET calls. Users who use both types of HASIs must take care not to transfer UDC parameters in the true parameter passing area used by ADC HASIs. Also, sufficient room for data must be allowed. Use of the BUFFER command is usually helpful for this problem.

Other symbols appearing on the maps include items of the form ".MPnnn". These are tables for the supervisor and can be ignored by the user. Other map-reading information may be found in the LOD100 Reference Manual (Table 1-1).

TCBs appear on the map because users calling the SETPRI SVC need to reference TCBs by name. If a user loads a task which references another task not yet loaded, the user should do a MARK after loading the first task; otherwise, the TCB for the second task is defined twice, and the reference to it will be incorrect.

6.2.3 The Host FORTRAN Mainline

The FORTRAN mainline, making calls to affect the FPS-100, is on the host's side of the job. While the user should read the APX100 Reference Manual (Table 1-1) to understand the format and purpose of the appropriate routines, they are mentioned here for the sake of completeness.

First, the user should call APINIT to assign and initialize the FPS-100. Then, APLLI should be called to establish which file contains the desired LOD100 load module to be downloaded into the FPS-100. Next, MTS1 should be called to download and start the supervisor running (assuming that the load module ID is 1, the default; for load module n, the call should be made to MTSn).

Next, the user can call any host-callable subroutines on the FPS-100 (user-written or FPS Math Library routines) or can transfer data in either direction. (See descriptions of APPUT, APGET, APWD, and APWR in the APX100 Reference Manual.) The user can also communicate with the FPS-100 via HPUT, HGET, and HTST.

When the job is done, the user releases the FPS-100 by calling APRLSE.

6.3 ADVANCED TECHNIQUES

This section explains techniques for faster SVC execution and for adding SVCs and ISRs to the system.

6.3.1 Faster Execution of SVCs

Normally, the user executes a TRAP instruction to call a system service. The overhead involved in this method includes at least a minimum state save and restore, which usually takes longer than the SVC itself. One way of bypassing this overhead is to execute a JSR directly to the SVC routine (e.g., JSR SEND). The advantage is speed. The disadvantage is that many of the user's registers may be lost. In general, the user loses s-pads 0-7, DPX(0)-(3), DA, TMA, and any pending memory fetches. If the user saves and restores any of those registers which may be in use, the tradeoff may be worthwhile.

Assuming that the user is running in user mode, with user MA, and interrupts on, the sequence to JSR to an SVC should include the following instructions:

```
IOFF
SETMOD
JSR svcname
CLRMOD
ION
```

The user must not switch to SMA to do a JSR to an SVC.

NOTE

The only SVC types to which the user cannot JSR are the WAITs (WAIT, TWAIT, WAITA, TWAITA). The exception is that one can JSR to TWAIT or TWAITA with a time limit of zero.

6.3.2 How to Write and Add SVC Routines

The user's own supervisor services can be added to the system easily. There are no jump tables to change and no limit on the number of SVCs allowed. The user has only to load them at LOD100 time with the rest of the SVCs. This can be accomplished automatically by changing the system loading command file, LODSYS, as follows: after the line "LOAD SYSSVC.B", insert the line "LOAD mysvc.B", where "mysvc.B" is the object of the user's SVC routine(s), assembled by ASM100.

The rules for writing an SVC routine are:

- An SVC routine can have up to four parameters, which are passed in DPX(0)-DPX(3).
- An SVC routine is entered in supervisor mode, with interrupts off; it must exit in the same state. In supervisor mode, memory fetches must be "pushed" via the STATMA instruction or any other memory-referencing instruction.
- An SVC routine can use only minimum-state registers except s-pad 7 (i.e., it can use s-pad 0-6, DPX(0)-DPX(3), DA, TMA, and MD references).

User SVC routines can be accessed either by jumping directly to them via a JSR instruction, or by executing the following instruction:

```
TRAP; DB=@mysvc; LDTMA
```

6.3.3 How to Write and Add ISRs to the Supervisor

Interrupt service routines (ISRs) are identified by the assembly code $ISR pseudo-op. Although they run in user mode, they must explicitly enable interrupts with the ION instruction upon entry and disable them (with IOFF) upon exit. The available register set is minimum-state registers minus s-pad 7, the same as for SVCs.

These are the three parameters passed to an ISR:

```
S-PAD 0  =  bit mask
S-PAD 1  =  device order number
S-PAD 2  =  physical device address
```

When the ISR finishes execution, it must determine whether the interrupt for the device it serviced should be re-enabled or left disabled. It passes this decision to the supervisor by the contents of s-pad 0 upon exit: 0 for enable or the bit mask originally passed to it for disable.

I/O devices are identified by their bit number(s) in the IMASK register. The logical word formed by bit N on with all other bits off is referred to as the bit mask for device N. Device N also has a priority mask associated with it, which is a logical word in which bits that are on represent devices whose interrupts will be disabled during device N's interrupt processing. For example, if the priority mask for device N contains bits J and K on, then devices J and K will not be able to interrupt while ISR N is running. It is the user's prerogative to establish device priorities by setting the priority mask for each device. If left untouched, the priority mask for device N has only bit N on. The bit mask and priority mask for each device is found in the configuration table (a labeled common called CONFIG in the file TABLES). See Appendix B (I/O Device Configuration Table) for a complete listing.

Each I/O interrupt corresponds to a five-word entry in this table. The user sets the relative priority of each device by setting the priority mask (first word) of each entry. Table 6-1 shows how this is done.

Table 6-1 Setting I/O Interrupt Priority Masks

PRIORITY	BIT SETTING
Highest	Set all bits (i.e., priority mask = 177777 (octal)). This masks out interrupts from all other devices while this device is being serviced.
Lowest	Set only the device's own bit (e.g., the device corresponding to IMASK bit 1 should have a priority mask of 040000 (octal)). (Bit 0 is the left-most bit.)
Relative	Using the devices corresponding to IMASK bits 1 and 2 as examples, if the bit 1 device is supposed to have higher priority than only the device corresponding to bit 2, then its priority mask = 060000 (octal), and the bit 2 device's mask = 020000 (octal).

The system is delivered with the minimum mask for each device (i.e., only its own bit set). If the user wishes to change it, the CONFIG portion of the TABLES file must be reassembled.

The table also contains a physical device address and device order number (where appropriate) for each device. The device order number (1,2,3, etc.) is for those cases where several devices of the same type (e.g., IOPs) are serviced by the same ISR. The configuration table also contains a pointer to the appropriate ISR for each device, via the ISR's overlay map (ISRMAP). Normally, device N points to ISR N; the third word of device N's entry in the table usually contains:

```
ISRMAP+W*(N-1)
```

If devices I, J, and K are all serviced by one ISR (say ISR K), then all three devices should point to ISR K. That is, the third word of device I's entry, device J's entry, and device K's entry should contain:

```
ISRMAP+W*(K-1)
```

NOTE

When adding new ISRs to the system, remember to load them with the other ISRs at LOD100 time.

6.3.4 Modifying the Supervisor

This section explains how to make such supervisor modifications as changing the stacked save area size, changing I/O device priorities, and changing LODSYS.

6.3.4.1 Changing the Size of the System's Stacked Save Area

The file COMSYS contains a system common called SYSCOM, which contains supervisor pointers, exchanges, counters, and save areas. The save area is equivalent to a stack of TCBs for the system. The number of TCB-equivalents in the stack should be equal to or greater than the number of possible interrupt sources. These include:

```
3  internal synchronous
1  spurious
2  host
1  real-time clock
1  spurious I/O
---
8  plus 1-2 for each I/O device (e.g., 1 for GPIOP, 2 for IOP)
```

The system is delivered with a default of 10. If this is not large enough, then the file COMSYS should be modified as follows:

(for 'n' possible interrupt sources)

change the line
```
SYSIZE = MNSIZE * 10.
```
to
```
SYSIZE = MNSIZE * n.
```

Then reassemble all files using COMSYS as a $INSERT file. While these files may include user files, they more particularly include the files whose objects are loaded in the LODSYS command file, which is described in section 6.3.4.3.

6.3.4.2 Changing Device Priorities

I/O devices are assigned relative priorities via their priority masks in the configuration table. The supervisor is delivered with all devices set to lowest priority. The user can easily change this by editing and re-assembling the CONFIG portion of the TABLES file. For an explanation of the configuration table and examples of setting priority masks, refer to section 6.3.3.

6.3.4.3 Changing LODSYS

LODSYS is the LOD100 command file to load the supervisor and related system routines and data structures. The system is "generated" with each new load module. The command file ends with the loader ready to receive ISRs and tasks.

The contents and explanations of LODSYS are:

```
PS 7740          (set initial PS address near top of 4K PS memory)
LOAD BOOTSP.B    (load system initializer)
PS 0             (set base PS address to 0)
MD 0             (set base MD address to 0)
LOAD TABLES.B    (load SYSCOM and CONFIG)
LOAD SYSSVC.B    (load system services)
LOAD HSVC.B      (load host communication routines)
LOAD ENABLE.B    (load I/O device interrupt enable routine)
MARK             (make all of the above stay in loader's tables)
LOAD KERNEL.B    (load supervisor kernel)
LOAD RTC.B       (load real-time clock routines)
LINK             (link all of the above)
MAP 0 SYSMAP     (output load map to file "SYSMAP")
MODE TASK        (change loader mode to accept ISRs and tasks)
```

The user should INPUT this file immediately following the OUT command (which should be the first command given to the loader in a LOD100 session). After LODSYS terminates, the appropriate ISRs and tasks may be loaded. The user may want to make another command file for that purpose.

For users who need maximum PS space and do not require software support of the real-time clock as a system service (i.e., they do not use the timed-wait SVCs), the LODSYS command file may be altered as follows:

```
change   LOAD RTC.B
to       LOAD NORTC.B
```

This loads null routines to satisfy entry points and reduce the amount of PS space the system uses by about 190 words.

Another way to decrease the amount of space the supervisor uses is to delete those SVCs which the user never uses from the SYSSVC source file and reassemble it.

If the user always loads certain ISRs (such as HIRP.B, for the host), the command sequences to do so can be appended to the LODSYS file.

6.4 AP-FORTRAN TASKS

Tasks may be written in AP-FORTRAN as well as ASM100. However, AP-FORTRAN offers no way to define a routine as a task. Therefore, this designation is made at load time via the TASK command in LOD100. This command is basically the same in format as the $TASK pseudo-op of ASM100. Unfortunately, it defines the task ID at load time instead of at assembly time, and the user has to remember which object modules to use it on.

Alternatively, the user can assemble a file consisting of nothing but the $TASK pseudo-op and concatenate its object to the beginning of the AP-FORTRAN object. This prevents the user from having to treat it differently at load time and ties the ID to the task.

In all other respects, AP-FORTRAN tasks are the same as other tasks. They can call SVCs via the regular FORTRAN CALL statement, and thus can communicate with other tasks. However, they tend to be larger in size and, therefore, are more likely to be split into several overlays.

6.5 VFC AND HSR MODES

This section discusses the use of the VFC in conjunction with the supervisor, and alternatives to the HSR mode of operation.

6.5.1 Vector Function Chainer

The Vector Function Chainer (VFC) output can be considered equivalent to any other host-callable subroutine; therefore, its object should be loaded as an overlay within the APX100 task. However, since VFC programs modify their own PS space, they must not share that space with any other overlays within the APX100 task, nor with any other tasks. This is because tasks are not swapped out when overlaid by other tasks; that is, code is copied from MD to PS, but not from PS to MD. Therefore, self-modifying code which has been interrupted, overlaid by something else, and then loaded into PS again, is incorrect.

6.5.2 HSR Mode

Running the FPS-100 in HSR mode is incompatible with running the supervisor. The FPS-100 can be run in HSR mode without the supervisor. Supervisor users wishing to access the FPS-supplied Math Library routines must reference them at load time so that LOD100 can create UDC or ADC HASIs for them.

For example, the following LOD100 command sequence loads the Vector Add routine as part of the APX100 task and creates a UDC HASI (SUBROUTINE VADD) for the user to call:

```
TREE ((1))
OV 1
LOAD UPEX.B        (load APX100 root)
CALL VADD /        (create HASI)
FORCE VADD         (get VADD from library and load it)
LIB APLIB
MAP 0 mapfile1     (to see symbols which will disappear)
LINK
MAP 0 mapfile2
```

6.6 PROCEDURE FOR A COMPLETE JOB

The procedure for getting a complete job through MTS100 is as follows:

1. Write the FORTRAN routines to run on the host. Write any tasks or host-callable subroutines in ASM100 or AP-FORTRAN.

2. Assemble FPS-100 assembly language with ASM100 and compile AP-FORTRAN routines with the AP-FORTRAN compiler to create object modules.

3. Load the object modules of the supervisor, tasks, host-callable subroutines, ISRs, exchanges, messages, etc., with LOD100.

4. Compile the host FORTRAN mainline, its subroutines, and the HASIs created by LOD100 in step 3.

5. Use the host loader to load the items in step 4.

6. Run the result. This causes the load module created by step 3 to run on the FPS-100.

APPENDIX A

SYSTEM STANDARD DEFINITIONS

The following listing is the contents of the SYSDEF file, used by the supervisor in accessing various data structures such as TCBs, messages, etc. The user may find this file helpful, if any of the symbols (such as R0-R7) are used in a user program. This file may be put into an ASM100 subroutine by means of the following pseudo-ops:

```
$NOLIST
$INSERT SYSDEF
$LIST
```

```
"******* SYSDEF = SYSTEM STANDARD DEFINITIONS = **********************
"
"
"  S Y S T E M   S T A N D A R D   D E F I N I T I O N S
"
"  THIS $INSERT FILE INCLUDES ALL REGISTER DEFINITIONS AND DATA AREA
"  OFFSETS USED WITHIN THE FPS-100 SUPERVISOR.  IT IS INCLUDED WITH
"  THE ASM100 PSEUDO-OP:  $INSERT SYSDEF.
"
"  PHYSICAL DEVICE ADDRESSES
"
IMASK   = 373
APST2   = 377
APST3   = 376
RTCCTL  = 372
RTCCST  = 371
RTCCTR  = 370
"
"
"  SPAD REGISTER DEFINITIONS:
"
R0      =       0
R1      =       1
R2      =       2
R3      =       3
R4      =       4
R5      =       5
R6      =       6
R7      =       7
"
"  DATA PAD (ARGUMENT) REGISTER DEFINITIONS:
"
X0      =       0
X1      =       1
```

```
X2      =       2
X3      =       3
"
"
"  STANDARD DATA OBJECT HEADER OFFSETS:
"
HEADER  =  0            " DATA OBJECT HEADER
RLINK   =  HEADER       " RIGHT (SUCCESSOR) LINK
LLINK   =  RLINK+1      " LEFT (PREDECESSOR) LINK
RPRI    =  LLINK+1      " PRIORITY OF DATA OBJECT
TYPE    =  RPRI+1       " TYPE OF DATA OBJECT
   INTBIT  =  040000    "    INTERRUPT MESSAGE DATA OBJECT BIT
   ANSBIT  =  020000    "    ANSWER MESSAGE DATA OBJECT BIT
   TIMBIT  =  010000    "    TIMEOUT MESSAGE DATA OBJECT BIT
   MSGBIT  =  002000    "    EXCHANGE MESSAGE QUEUE IN USE BIT
   TSKBIT  =  001000    "    EXCHANGE TASK QUEUE IN USE BIT
   BSYBIT  =  000001    "    MESSAGE BUSY BIT (IN EXPONENT)
LENGTH  =  TYPE+1       " LENGTH OF DATA OBJECT (IN MD WORDS)
ANSKEY  =  LENGTH+1     " ANSWER EXCHANGE ADDRESS
"
HDREND  =  ANSKEY
"
" FIELD FOR MESSAGES ONLY
"
MISC    =  HDREND+1
"
"
"  TASK CONTROL BLOCK (TCB) OFFSETS:
"
TASKID  =  HDREND+1     " TASK IDENTIFIER
OVLPTR  =  TASKID+1     " POINTER TO FIRST ENTRY IN OVERLAY TABLE
OVLCNT  =  OVLPTR+1     " NUMBER OF ENTRIES (OVERLAY SEGMENTS)
DPRI    =  OVLCNT+1     " DEFAULT TASK PRIORITY
STATUS  =  DPRI+1       " CURRENT TASK STATUS FLAGS
   SLVBIT  =  010000    "    TASK PRIORITY IS SLAVED
   CXTBIT  =  004000    "    TASK HAS FULL CONTEXT
   RDYBIT  =  001000    "    TASK IS READY TO EXECUTE
   RDYOFF  =  176777    "    COMPLEMENT OF RDYBIT
   CLKBIT  =  000400    "    CLOCK QUEUE BIT
LSTMSG  =  STATUS+1     " ADDRESS OF LAST MESSAGE RECEIVED (FOR ANSWER)
RCLOCK  =  LSTMSG+1     " CLOCK QUEUE THREAD RIGHT LINK
LCLOCK  =  RCLOCK+1     " CLOCK QUEUE THREAD LEFT LINK
ICLOCK  =  LCLOCK+1     " CLOCK QUEUE INCREMENT
TADDR   =  ICLOCK+1     " TASK'S BEGINNING ADDRESS (IN MD)
"
"  TCB STATE SAVE AREA
"
FIFO    =  TADDR+5      " MD FIFO (3 WORDS)
DPBS    =  FIFO+3       " DATA PAD BUS
DPXW    =  DPBS+1       " DPX WRITE BUFFER
STAT1   =  DPXW+1       " CURRENT FLAGS AND EXCEPTION STATUS
DPX0    =  STAT1+1      " X0 REGISTER FOR ARGUMENTS
DPX1    =  DPX0+1       " X1 REGISTER FOR ARGUMENTS
```

```
DPX2    =  DPX1+1       " X2 REGISTER FOR ARGUMENTS
DPX3    =  DPX2+1       " X3 REGISTER FOR ARGUMENTS
DA      =  DPX3+1       " CURRENT DEVICE ADDRESS
SPD     =  DA+1         " SPAD DESTINATION
SPAD0   =  SPD+1        " R0 REGISTER
SPFUNC  =  SPAD0+1      " SPFN RETURNED (ERROR CODE ON SVC'S)
SPAD1   =  SPFUNC+1     " R1 REGISTER
SPAD2   =  SPAD1+1      " R2 REGISTER
SPAD3   =  SPAD2+1      " R3 REGISTER
SPAD4   =  SPAD3+1      " R4 REGISTER
SPAD5   =  SPAD4+1      " R5 REGISTER
SPAD6   =  SPAD5+1      " R6 REGISTER
SPAD7   =  SPAD6+1      " R7 REGISTER
STAT2   =  SPAD7+1      " APSTAT2 (SUPERVISOR/INTERRUPT STATUS)
TMAREG  =  STAT2+1      " TABLE MEMORY ADDRESS
TMREG   =  TMAREG+1     " CURRENT TM VALUE
FFT     =  TMREG+1      " FFT STATUS BITS
MAREG   =  FFT+1        " MAIN DATA ADDRESS
STAT3   =  MAREG+1      " SUBROUTINE RETURN STACK ADDRESS (APSTAT3)
SRS     =  STAT3+1      " SRS(15)-(0)
"
MINTCB  =  SRS+15.      " LAST LOCATION IN MIN SAVE AREA
"
"  TASK CONTEXT AREA
"
SPAD8   =  MINTCB+1     " SPAD 8 - 15
DPYW    =  SPAD8+8.     " DPY WRITE BUFFER
DPY0    =  DPYW+1       " DPY(0)-(3)
DPAD    =  DPY0+4       " REMAINING DPX AND DPY
DPA     =  DPAD+56.     " DPA
FLADD   =  DPA+1        " FLOATING ADDER STATE
FLMUL   =  FLADD+5      " FLOATING MULTIPLIER STATE
FLAGS   =  FLMUL+6      " AP FLAGS (4 WORDS)
MAXTCB  =  FLAGS+3      " LAST WORD IN TCB
"
"
"  SUPERVISOR ERROR CODES:
"
ERRBSY  =  -1           " MESSAGE BUSY ERROR
ERRMSG  =  ERRBSY-1     " INVALID MESSAGE ERROR
"
"
" OTHER USEFUL CONSTANTS
"
FPEON   =  000100
FPEOFF  =  177677
"
"
```

APPENDIX B

I/O DEVICE CONFIGURATION TABLE

The following listing is the CONFIG portion of the TABLES file. The configuration table contains data pertinent to each I/O device on the FPS-100.

```
"****** CONFIG = I/O DEVICE CONFIGURATION TABLE ******
"
$TITLE CONFIG
$COMMON /CONFIG/ DEV1(5) /T, DEV2(5) /T, DEV3(5) /T, DEV4(5) /T,
                 DEV5(5) /T, DEV6(5) /T, DEV7(5) /T, DEV8(5) /T,
                 DEV9(5) /T, DEV10(5) /T, DEV11(5) /T, DEV12(5) /T,
                 DEV13(5) /T, DEV14(5) /T, DEV15(5) /T
$COMMON /ISRMAP/ DUMMY(120.) /I
"
"
" THIS IS A CONFIGURATION TABLE FOR THE I/O DEVICES ON THE FPS-100.
"
" THEIR ORDER IS THE SAME AS THAT OF THEIR BITS IN THE IMASK REGISTER.
"
" THE FORMAT OF EACH ENTRY IN THIS TABLE IS:
"
"   WORD 1:  PRIORITY MASK (TO MASK OUT LOWER PRIORITY DEVICES)
"   WORD 2:  BIT MASK (THIS DEVICE'S BIT ON)
"   WORD 3:  POINTER TO SERVICE ROUTINE'S ENTRY IN OVERLAY MAP
"   WORD 4:  DEVICE ORDER NUMBER  (EXP)
"            PHYSICAL DEVICE ADDRESS (LMAN)
"   WORD 5:  SAVE AREA FOR OLD IMASK
"
W = 8.  "WIDTH OF OVERLAY TABLE ENTRY
"
$DATA DEV1(1)    0,0,    040000
$DATA DEV1(2)    0,0,    040000
$DATA DEV1(3)    0,0,    ISRMAP+0
$DATA DEV1(4)    0,0,    0
$DATA DEV1(5)    0,0,    0
"
$DATA DEV2(1)    0,0,    020000
$DATA DEV2(2)    0,0,    020000
$DATA DEV2(3)    0,0,    ISRMAP+(W*1)
$DATA DEV2(4)    0,0,    0
$DATA DEV2(5)    0,0,    0
"
$DATA DEV3(1)    0,0,    010000
$DATA DEV3(2)    0,0,    010000
$DATA DEV3(3)    0,0,    ISRMAP+(W*2)
```

```
$DATA DEV3(4)     0,0,     0
$DATA DEV3(5)     0,0,     0
"
$DATA DEV4(1)     0,0,     004000
$DATA DEV4(2)     0,0,     004000
$DATA DEV4(3)     0,0,     ISRMAP+(W*3)
$DATA DEV4(4)     0,0,     0
$DATA DEV4(5)     0,0,     0
"
" REAL TIME CLOCK
"
$DATA DEV5(1)     0,0,     002000
$DATA DEV5(2)     0,0,     002000
$DATA DEV5(3)     0,0,     ISRMAP+(W*4)
$DATA DEV5(4)     0,0,     372
$DATA DEV5(5)     0,0,     0
"
" HOST (DMA)
"
$DATA DEV6(1)     0,0,     001000
$DATA DEV6(2)     0,0,     001000
$DATA DEV6(3)     0,0,     ISRMAP+(W*5)
$DATA DEV6(4)     0,0,     0
$DATA DEV6(5)     0,0,     0
"
" HOST (NON-DMA INTERRUPT)
"
$DATA DEV7(1)     0,0,     000400
$DATA DEV7(2)     0,0,     000400
$DATA DEV7(3)     0,0,     ISRMAP+(W*6)
$DATA DEV7(4)     0,0,     0
$DATA DEV7(5)     0,0,     0
"
" IOP16 (DMA)
"
$DATA DEV8(1)     0,0,     000200
$DATA DEV8(2)     0,0,     000200
$DATA DEV8(3)     0,0,     ISRMAP+(W*8.)
$DATA DEV8(4)     1,0,     10
$DATA DEV8(5)     0,0,     0
"
" IOP16 (NON-DMA INTERRUPT)
"
$DATA DEV9(1)     0,0,     000100
$DATA DEV9(2)     0,0,     000100
$DATA DEV9(3)     0,0,     ISRMAP+(W*8.)
$DATA DEV9(4)     1,0,     10
$DATA DEV9(5)     0,0,     0
"
" IOP38 (DMA)
"
$DATA DEV10(1)    0,0,     000040
$DATA DEV10(2)    0,0,     000040
```

```
$DATA DEV10(3)     0,0,    ISRMAP+(W*8.)
$DATA DEV10(4)     1,0,    20
$DATA DEV10(5)     0,0,    0
"
" IOP38 (NON-DMA INTERRUPT)
"
$DATA DEV11(1)     0,0,    000020
$DATA DEV11(2)     0,0,    000020
$DATA DEV11(3)     0,0,    ISRMAP+(W*8.)
$DATA DEV11(4)     1,0,    20
$DATA DEV11(5)     0,0,    0
"
" IOP38 (DMA)
"
$DATA DEV12(1)     0,0,    000010
$DATA DEV12(2)     0,0,    000010
$DATA DEV12(3)     0,0,    ISRMAP+(W*8.)
$DATA DEV12(4)     2,0,    40
$DATA DEV12(5)     0,0,    0
"
" IOP38 (NON-DMA INTERRUPT)
"
$DATA DEV13(1)     0,0,    000004
$DATA DEV13(2)     0,0,    000004
$DATA DEV13(3)     0,0,    ISRMAP+(W*8.)
$DATA DEV13(4)     2,0,    40
$DATA DEV13(5)     0,0,    0
"
" GPIOP
"
$DATA DEV14(1)     0,0,    000002
$DATA DEV14(2)     0,0,    000002
$DATA DEV14(3)     0,0,    ISRMAP+(W*13.)
$DATA DEV14(4)     1,0,    100
$DATA DEV14(5)     0,0,    0
"
" GPIOP
"
$DATA DEV15(1)     0,0,    000001
$DATA DEV15(2)     0,0,    000001
$DATA DEV15(3)     0,0,    ISRMAP+(W*13.)
$DATA DEV15(4)     2,0,    200
$DATA DEV15(5)     0,0,    0
"
$END
```

APPENDIX C

ALPHABETICAL INDEX OF SUPERVISOR CALLS (SVCs) AND HOST/FPS-100 COMMUNICATION ROUTINES

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

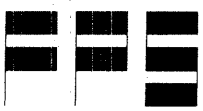
Postage will be paid by:

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