Section 5
TASK EXECUTION

In this section: See page

In this section (cont):

Summary

This section describes the processing of priority levels, including context saving of interrupted tasks and the assignment of priority levels and logical resource numbers to tasks. This section also describes task communication and coordination as well as deferred processing.

INTRODUCTION

A task can be characterized as the execution of a sequence of instructions that has a starting point and an ending point, and performs some identifiable function. A task can initiate another task for execution or terminate itself by calling the task management commands or macrocalls. Multiple tasks can operate independently of and asynchronously to each other.

Each application, system, or device driver task operates at an interrupt priority level, one of the 64 priority levels provided for each central processor by the hardware and firmware.

CENTRAL PROCESSOR INTERRUPT PRIORITY LEVELS

All system tasks, device drivers, and application tasks are assigned interrupt priority levels that indicate the order of their execution. This order of execution may be changed due to timeslicing (see below) or because this is a multiprocessor system.

Control of the central processor is given to the highest active interrupt level. However, in multiprocessor systems, a task at a lower priority may execute at the same time as a task of higher priority since each task is executing on a different central processor.

Number of Levels

Each central processor provides 64 potential interrupt priority levels that are used by the hardware to order the processing of events. These levels are numbered from the highest priority (level 0) to the lowest priority (level 63). Levels 0 through 5, 62, and 63 are reserved. The intervening levels (6 through 61) are assigned to logical resources (that is, devices and tasks).

Activity Indicators

The determination of which priority level is to receive central processor time is based on a linear scan of the level activity indicators. The level activity indicators are maintained by the hardware in four contiguous dedicated memory locations in each central processor (see Figure 5-1). Each bit that is "on" denotes an active priority level; each bit that is "off" denotes an inactive level.

Interrupts

Bits can be set "on" by software or by hardware events (interrupts). Most interrupting hardware devices are associated with priority levels during system configuration (by directives in the CLM_USER file).

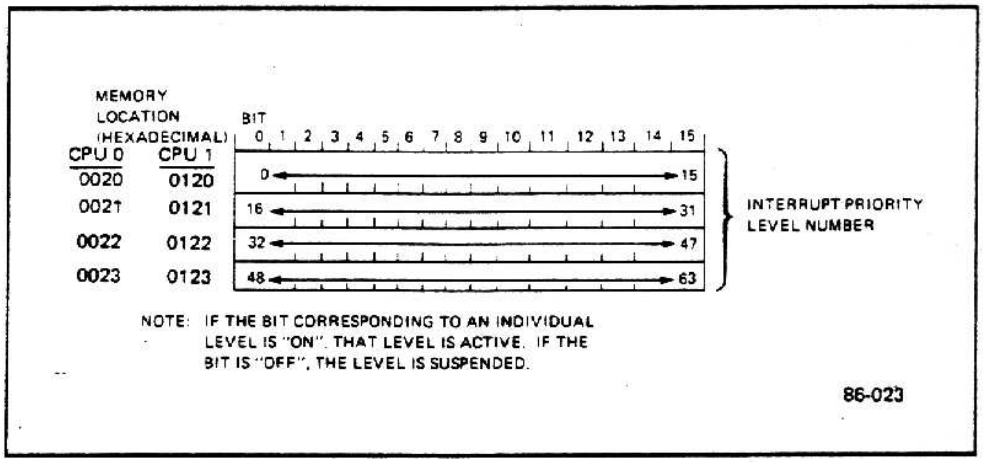

Figure 5-1. Format of Level Activity Indicators for Each Central Processor

Available Levels

The three highest priority levels have dedicated assignments of special hardware/firmware functions such as incipient power failure, watchdog timer runout, and trap save area overflow. Priority level 3 is reserved as an inhibit level, level 4 is reserved for the timeslice clock, and level 5 is dedicated to the real-time clock. Succeeding levels are user-configurable as device levels. Following these are five levels reserved for system use. Except for levels 62 and 63, the remaining levels can be used for application tasks. Level 62 is reserved for system use. Level 63 is reserved for an always active software idle loop or, in multiprocessor systems, for the task dispatcher.

Interrupt Processing

When a given priority level is the highest active level, it receives all available central processor time until it is interrupted by a higher priority level or until it relinquishes control by suspending itself (setting its level activity indicator off). If a priority level is interrupted by a higher priority level, its level activity indicator remains on and it will resume execution of the interrupted logical resource when it again becomes the highest priority level.

Each time a priority level change occurs, the hardware/firmware saves the central processor context of the task running at the previously highest active level and restores the central processor context of the task running at the new highest active level. Interrupting a task, saving the context of a task, selecting and starting the highest priority level task, and restoring the context of a task are done without software involvement.

INTERRUPT SAVE AREA

Task Context

The context of a level (task) can include the contents of the program counter, S-register, B-registers, I-register, K-registers, R-registers, M-registers, SIP registers, and CIP registers. The context is stored for each central processor in a block of memory known as an Interrupt Save Area (ISA).

Interrupt Vectors

The hardware/firmware context save/restore function finds the appropriate ISA through a pointer supplied in the interrupt vector for that level. The interrupt vectors are a set of contiguous memory locations containing an entry for each potentially active priority level and ordered by ascending priority level number. Figure 5-2 illustrates the order of the priority levels, their corresponding interrupt vectors, and the format of an ISA.

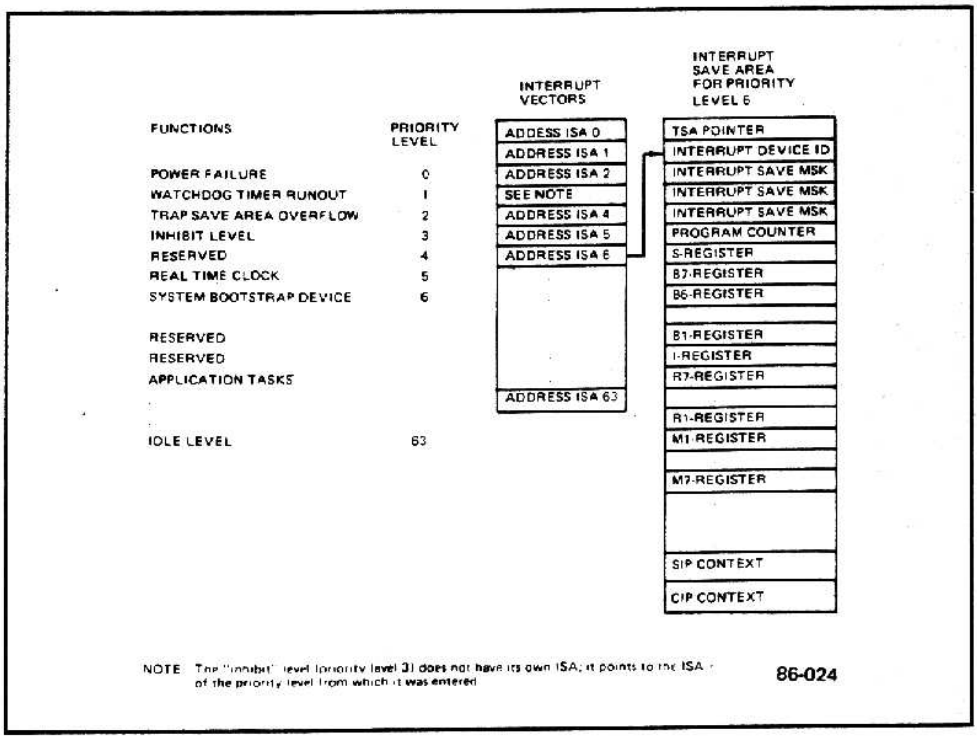

Figure 5-2. Order of Interrupt Vectors and Format of Interrupt Save Areas for Each Central Processor

TASK DISPATCHING

The way in which a task receives central processor time depends on whether the system has one or more than one central processor.

In a monoprocessor system, tasks are dispatched according to their priority level. The task at the highest priority level receives all available central processor time until it is interrupted by a task with a higher priority level or until it suspends itself. In a multiprocessor system, all tasks are dispatched from a general ready queue. The tasks are placed in the queue according to their priority level, with higher priority tasks at the top of the queue. The level at which a task executes stays the same, but the central processor in which it executes may vary.

Monoprocessor Task Dispatching

When a task in a monoprocessor system is at the highest active priority level, it receives all available central processor time until it is interrupted by a task at a higher priority level, until it relinquishes control by suspending itself, or until it has control taken away from it due to timeslicing (see below). If a task is interrupted by a higher priority task, it will resume execution when it again becomes the task at the highest priority level.

When more than one task is assigned the same priority level, a system software task at a higher level regulates in round-robin fashion the sharing of the level between tasks. Thus a task does not block a level when the task is timesliced (refer to "Timeslicing" below) or when it is put in a waiting state after a request to wait, wait on list. request semaphore, or terminate, or after a system service macrocall that waits for a data transfer. Instead, the context of another task on the same level will be linked to the level interrupt vector.

Multiprocessor Task Dispatching

In a multiprocessor system, the Executive maintains a queue of ready tasks ordered by priority level. This queue is called the general ready queue. The Executive dispatches the task at the top of the queue whenever a central processor becomes free to provide service. A dispatcher task runs at level 63 in each central processor and dispatches a task whenever it receives central processor time. The dispatcher tasks attempt to balance the load so that high priority tasks are serviced before low priority tasks, and all processors are used as fully as possible.

TIMESLICING

The technique of timeslicing minimizes the ability of user tasks, that use large amounts of central processor time, to interfere with interactive users of the system. Timeslicing uses a sampling technique to monitor all tasks at user levels. Configuration of timeslicing is automatic. All user levels execute in a timesliced manner. Timeslicing options are discussed in the *System Building and Administration* manual.

Each running user task is monitored by the software timeslicer logic. Each user task is given a CPU time allocation (a time slice), the amount of which has been determined by system software and is governed by the arguments (default or explicit) associated with the DSLICE bound unit.

While a user task is running, timeslicing software is counting down the task's time slice allocation.

When a task's time slice allocation becomes exhausted, the timeslice software, in general, will dequeue the sliced task and then decide whether to demote the sliced task one level or to place the sliced task behind all other tasks ready to run on its current level. In either event, the system software will establish a new time slice allowance for the task. This allowance will be in effect when the task next runs.

If a task voluntarily waits (a synchronous I/O order, for example) while it still has time remaining on its current time slice, the reason for the wait is evaluated by system software. If the wait is for a "long" interval, thereby making CPU time available to competing tasks, the task will be promoted to (or stay assigned to) its native level when it next runs. If the wait is for a "short" interval, thereby limiting the CPU time available to other tasks, the task will be given priority for completing its remaining slice when it next runs, but will then become a candidate for possible demotion.

The result is that all tasks are facilitated in using a given time slice, but those that voluntarily wait for "long" intervals become candidates for promotion, and those that wait for "short" intervals (or do not voluntarily wait at all) become candidates for demotion.

TRAP HANDLING

The hardware provides a means by which certain events that occur during the execution of a task can be "trapped", with control being passed to software routines designed specifically to cover the condition causing the trap. Events such as the execution of a system monitor call, or the detection of a program error, hardware error, arithmetic overflow, or uninstalled optional instruction cause traps (control transfers to designated software routines) to occur.

Trap Classes

Traps are divided into two classes: (1) standard system traps, for which routines are supplied with the system, and (2) user-specific traps, for which users supply their own routines (handlers).

User Traps

An application program can designate which user-specific traps are to be handled by using the enable/disable user trap macrocalls (refer to the *System Programmer's Guide - Volume II* for details). If an enabled trap occurs in the user program, the Trap Manager transfers control to the connected trap handler for the condition causing the trap. A trap that is enabled is local to a task. Such a trap neither affects nor is affected by the handling of the same trap in another task, even within the same task group.

Any trap that occurs when its handler is not enabled, or that does not have a handler to process it, causes the executing task to be aborted.

SYSTEM FEATURES AFFECTING TASK EXECUTION

While the system does monitor resource use within a task group and among task groups, tasks and task groups must cooperate in their use of system resources to ensure smooth operation of the application.

Priority Level Assignments

Priority levels 6 through 61 are available for assignment to system, device driver, and application tasks. The system builder establishes the priorities of system and driver tasks during configuration. (On the DPS 6/22, the Autoconfigurator establishes these priorities.) You assign the priorities of application tasks when you create task groups. Priority levels with low numeric values have higher priority than those with high numeric values. The procedures for establishing priorities are described below.

Assigning Priority Levels to Devices and System Tasks

The system builder specifies hardware interrupt priority levels through an argument of the DEVICE directive in the CLM_USER file. (The Autoconfigurator is used on the DPS 6/22.) When the system builder specifies a particular type of device, the appropriate Honeywell Bull written device driver is loaded as part of the system. The three priority levels following the last one assigned to a configured device are used by system tasks and cannot be assigned to application tasks.

Reserved Levels

One example of priority level assignment is shown in Table 5-1. Levels 0 through 5 are assigned by the system and are not available to any user. The operator terminal is assigned to level 8; however, the system builder can assign any appropriate level to the operator terminal through a DEVICE directive. (The operator terminal must be at a lower (numerically higher) level than the Communications Supervisor.) At initialization, the system bootstrap device is assigned to level 6. This assignment remains in effect unless changed by a DEVICE directive.

Peripheral Devices

Peripheral devices may be assigned to levels on both central processors in a multiprocessor system. This assignment is done automatically by the system.

Table 5-1 indicates Input/Output (I/O) devices, and not device drivers, to stress that each peripheral device must have at least one level assigned to it. Except for communications devices, peripheral devices cannot share a level. If there are two printers, each must be assigned a unique level even though there is only one copy of the associated I/O driver. Communications configurations require at least one nonshareable level dedicated to processing communications interrupts. This level must be higher than any level assigned to a communications device.

Communications devices can share a level. For example, four teleprinters (TTYs) and one Visual Information Projection (VIP) terminal can be configured to share one level or to use up to five levels. The priorities in Table 5-1 provide maximum throughput because devices with high transfer rates are assigned higher priorities than devices with low transfer rates.

Assignable Levels

Theoretically, the system builder could assign a level number as high as 58 to a device. In this case, levels 59 and 60 would be used by the system and only level 61 would be available for user task groups. In practice, however, the system builder would want to reserve more than one level for user task groups, especially for a system with a large number of devices. If priority levels 6 and 7 are assigned as shown in Table 5-1, the theoretical range of levels assignable through CLM COMM directives is 8 through 58. For a device associated with a COMM directive, the range is 9 through 58.

Table 5-1. Sample Priority Level Assignments for Tasks and Devices (Sheet 1 of 2)

Physical Priority Level	Base Priority Level	Use	Comments
0	N/A	Power failure handler	Levels 0 through 5 are automatically assigned by the system.
1	N/A	Watchdog timer runout	
2	N/A	TSA overflow	
3	N/A	Inhibit interrupts	
4	N/A	Reserved	
5	N/A	Real-time clock	
6	N/A	System bootstrap device	Set to level 6 at system initialization but can be changed.
7	N/A	Communications Supervisor	Must be higher level than any communications device.
8	N/A	Operator terminal	Can be assigned any available level.
9	N/A	TTY device	Communications devices can share priority levels.
9	N/A	TTY device	
9	N/A	TTY device	
10	N/A	Removable cartridge disk	The priority level for a pair of fixed/removable disks must be the same.
10	N/A	Fixed cartridge disk	
11	N/A	Diskette	
12	N/A	Diskette	
13	N/A	Diskette	
14	N/A	Line printer	
15	N/A	Card reader	

Table 5-1. Sample Priority Level Assignments for Tasks and Devices (Sheet 2 of 2)

Physical Priority Level	Base Priority Level	Use	Comments
16 17 18	N/A N/A N/A	Reserved by system Reserved by system Reserved by system	The three levels following the last device-assigned level are used by the system.
19 20 . . .	0 1 . . . 10	Task group A Task group B . . . Task group n	
. . .			
62 63	N/A N/A	Reserved by system System idle loop or task dispatcher	Always active.

Assigning Priority Levels to Application Tasks

Base Priority Level

You assign priority levels to user task groups and tasks when you create or spawn them. The command to generate a task group contains an argument that specifies the base priority level for the task group. The base priority level is relative to the highest number priority level assigned to a configured device.

When a task group is assigned a base priority level of zero, the lead task of the group executes at the physical interrupt priority level that is three level numbers above the highest level number assigned to a configured device. When other tasks in the same task group are created or spawned, they are given level numbers relative to the base priority level assigned to the task group.

Physical Level

The physical interrupt level at which a task executes is the sum of the following:

1. The highest level number assigned to a configured device plus 4.
2. The base priority level number of the task group
3. The relative priority level of the task within that group.

This sum must not exceed 61.

Recommended Priorities

Interactive user tasks are usually given higher priorities (lower level numbers) than absentee user tasks. Tasks that are I/O-bound should be run at a higher priority than tasks that are Central Processor (CP) bound. This permits I/O-bound tasks, which run in short bursts, to issue I/O data transfer orders as needed, wait for I/O completion and, while in the wait state, relinquish control of the central processor to CP-bound tasks. Otherwise, if the CP-bound tasks have a higher priority, the I/O devices would be idle while I/O-bound tasks waited to receive central processor time. (Timeslicing minimizes the ability of CP-bound tasks to interfere with interactive and I/O bound tasks.)

Logical Resource Numbers

A Logical Resource Number (LRN) is an internal identifier used to refer to task code and devices independently of their physical priority levels. Use of LRNs makes Assembly language application task code independent of priority levels so that, if circumstances require a change in priority levels, the task code does not have to be reassembled.

Device Logical Resource Numbers

The system uses DEVICE directives to assign LRN values. Device LRNs may have values from 2 through 252, and from 256 through 4002. LRN 0 is used for the operator terminal; LRN 1 is used for the bootstrap device; and LRNs 4003 through 4095 are reserved for other system uses. Figure 5-3 is an example of LRN assignments for devices and system tasks.

LRN	Use
0	Operator terminal
1	System disk
2	Reserved
3	System disk companion (if any)
4	User aspect of dual-purpose operator terminal
5	Other devices
.	
.	
.	

Figure 5-3. Example of LRN Assignments for System Tasks and Devices

Application Task Logical Resource Numbers

LRN assignments to application program tasks within each task group are not dependent on the system configuration on which the application task group is running. You can assign LRNs or have the system select the highest numbered LRN available at task creation.

Assigning LRNs

LRNs are assigned to task code within an Assembly language application program through specification of the Create Group and Create Task macrocalls as well as the macrocalls that build data structures ($IORB, $TRB, and so forth). LRNs can be assigned at the control language level through the commands for the creation of task groups and tasks.

LRN Values — An LRN for an application task can have any value from 0 through 4095. Within a task group, the LRN for each task must be unique. More than one LRN can be associated with the same priority level (for example, two tasks at level 23 can have LRNs of 28 and 29, respectively).

Non-LRN Tasks — Two kinds of tasks do not have LRNs:

- The lead task of any task group
- Any spawned task.

Logical File Numbers

Logical file numbers (LFNs) are internal file identifiers associated with file pathnames at the source language program level or at command level. LFNs can be used to reduce program dependence on actual file pathnames (which are likely to vary).

Assigning LFNs — LFNs can be associated with file pathnames in Assembly language or COBOL programs, or through Create File, Get File, and Associate commands.

LFN Values — An LFN can have any value from 0 through 4095.

Task and Resource Coordination

Tasks can be coordinated in either of two ways:

- Through the use of tasking requests
- Through the use of semaphores.

Task Requests

One task can request another to execute asynchronously with it, or the requesting task can later wait for the completion of the requested task. Both tasks have access to the request block provided by the requesting task, and can use it to pass arguments between them.

Semaphores

Semaphores support an application-designed agreement among tasks to coordinate the use of a resource such as task code or a file. A semaphore is defined by a task within a task group and is available only to the tasks within that group. Use of semaphores in an application is essential if the application has multiple tasks and is sharing data in memory.

For each resource to be controlled, you define a semaphore and give it a 2-character (ASCII) name. The semaphore name is a system symbol recognized by the system control software; it is not a program symbol that needs Linker resolution.

Resource Coordination

In controlling resources, the agreement is that each requestor of a resource whose use must be coordinated issues appropriate system service macrocalls to the named semaphore to request or release the resource. The task that defines the semaphore assigns the semaphore's initial value. The system control software maintains the current value of the semaphore so as to coordinate requesters of the resource being controlled. A requester obtains use of a resource if the semaphore value is greater than zero at the time of the request. If the value is zero or negative, the requester is either suspended (waiting for the resource) or notified that no resource is available, depending on how the request was made.

Semaphore Macrocalls

System service macrocalls are used to:

- Define a semaphore and give an initial value ($DFSM).
- Reserve a semaphore-controlled resource ($RSVSM). This macrocall subtracts a resource or queues a waiter for the resource (that is, it decrements the current-value counter). $RSVSM suspends the requesting task until the resource is ready.
- Release a semaphore-controlled resource ($RLSM). This macrocall adds a resource or activates the first waiter on the semaphore queue (that is, it increments the current-value counter).
- Request the reservation of a semaphore-controlled resource ($RQSM). This macrocall queues a request block (SRB) if the resource is not available (that is, it decrements the current-value counter). The requesting task must test the queued SRB subsequent to the request in order to determine when the resource is granted. The requesting task continues executing until it executes a $RSVSM macrocall; then it waits.
- Delete a semaphore ($DLSM).

Example

A semaphore is a gating mechanism. The initial value you give to it depends on the type of control you want to exercise. For example, assume that you want to restrict access to a particular resource to one user at a time. The mechanism would work in the following way:

1. Task A defines a semaphore by issuing the macrocall:

   ```
   $DFSM ZZ
   ```

   Omission of the value argument causes the initial value to be set at 1.

2. Task B now issues a $RSVSM macrocall. The counter is decremented to 0. Task B gets the resource for itself, knowing that no other task using the semaphore mechanism is now using or can obtain the resource.

3. Task C issues a $RSVSM macrocall. The counter is decremented to -1. Task C is suspended and put on the semaphore queue in first-in/ first-out order (because Task B is still using the resource).

4. Task B issues a $RLSM macrocall when it finishes with the resource. The counter is incremented to 0. Task C now gets the resource. After Task C issues the $RLSM macrocall, the value again becomes 1.

Use of resources by more than one user at a time can be arranged by adjusting the initial value of the semaphore. For example, an initial value of 2 allows two users, a value of 4 allows four users, and so on. The value chosen as the initial value of the semaphore depends on the nature of the resource and its intended use.

If it is undesirable for a task to be suspended while a resource is in use, the $RQSM macrocall can be used instead of $RSVSM to reserve a resource. $RQSM is an asynchronous reservation request ($RSVSM is a synchronous request) that causes a request block to be queued for the resource so that the issuing task can do other processing before the needed resource is available.

TASK HANDLING

More than one task can be concurrently active. In a multiprogramming environment, a task in each of several task groups can be active and compete for system resources. Another possibility is a multitasking application where several tasks executing under one task group can be active to compete for system resources among themselves and with tasks from other task groups.

A FORTRAN or Assembly language program can include requests to activate several tasks and synchronize their execution; these requested tasks can execute concurrently. A COBOL, BASIC, C, Pascal, or Ada program executes as a single task, but can include commands to activate other tasks.

Task Priority Levels

For the system to sequence the execution of tasks, each task must be assigned a priority level. In monoprocessor systems, task competition for the central processor resource is governed by the hardware/firmware linear priority scan of level activity indicators. Tasks on the same priority level execute serially in the order in which they are requested. In multiprocessor systems, tasks are ordered in a software queue according to their priority levels. The task at the top of the queue is dispatched when a processor becomes free. When it is assigned to a processor, the task executes at the same priority level as it would on a monoprocessor system.

The highest priority active task receives all available central processor time until it waits, exceeds the timeslice value, terminates, or is suspended. In both monoprocessor and multiprocessor systems, the task can be interrupted by a higher priority task.

Device Drivers

It should be noted that all device drivers are considered to be tasks in the above sense. Using the File System, buffered device drivers can execute concurrently with tasks. Drivers execute on the central processor priority levels assigned to individual devices and thus have their own contexts. The device drivers provided in the system are written in reentrant code, are capable of servicing multiple devices, and execute on any central processor in a multiprocessor system.

Task Activation

A user task becomes active when a Spawn Task or Enter Task Request command is issued for it. The Spawn Task command can request that the invocation of the task be delayed until a specified time interval has elapsed. FORTRAN programs can cause a task to become active through the START and TRNON statements. Assembly language programs can issue a $RQTSK OR $SPTSK macrocall to activate a user task. Any application program can issue a command to spawn or request a task by calling the ZXEXCL run-time routine.

When you want more than one task to execute concurrently, you must specify each task in a Create Task or Spawn Task command (or system service macrocall).

The procedural code for a requested task is either in a unique bound unit or in a bound unit shared with a task that was previously created. When a task is requested, the system searches for its identifying LRN in the table of LRNs associated with the task group under which the task is executing. The system activates the task, if it is not already active.

Task Termination

To terminate, tasks of Assembly language programs must contain a Request to Terminate ($TRMRQ) macrocall. Compilers provide this call in the object text. $TRMRQ is executed after the task completes execution.

TASK STATES

Tasks can exist in any of the logical states described below.

Dormant — A task is in the dormant state when there is no current request for it. A task enters the dormant state if it is created but never requested, or when a terminate request is issued against it. A task remains dormant until a request is placed against it or it is deleted. If deleted, it is erased, memory is reused, and the task cannot be reactivated.

Active — A task is in the active state when it is executing or when it is ready to execute if its priority level becomes the highest active level in a central processor. A task remains active until it waits, terminates, or is suspended. Tasks in the general ready queue are active.

Wait — A task is in the wait state when it is not executing. It may have caused its own execution to be interrupted until (1) the completion of an event such as the completion of a requested task, or (2) a timer request is satisfied, or (3) a task releases a semaphore. A waiting task loses its position in the priority level round-robin queue.

An I/O order to disk, magnetic tape, the operator terminal, or an unbuffered card reader usually results in a wait condition. Task code written in FORTRAN or Assembly language will also wait in the following circumstances: (1) a write order is issued to an interactive terminal or to a printer when a previous write has not completed, (2) a read order is issued before the transfer of the current message from an interactive terminal is complete (the RETURN key is not pressed). In COBOL, these circumstances result in a wait if the program is executing its I/O statements in synchronous mode; otherwise, if in asynchronous mode, a status code value of 9I is returned with no waiting.

Suspend — A task is in the suspend state when it is removed from execution by an external human action (for example, the operator entering a Suspend Group command or a user interrupting a program with a Break action). The task is activated through another human action (for example, the operator enters an Activate Group command or a user enters a command after the Break action).

INTERTASK AND INTRATASK GROUP COMMUNICATION

Information can be passed among task groups and tasks by means of request blocks, common files, and the message facility.

Request Blocks

Task code written in Assembly language can pass information to other Assembly language tasks in the same task group by using variable-length request blocks. The request blocks can contain data or pointers to information structures. All request blocks must be in common address space so that they can be shared by the tasks. (Refer to the *System Programmer's Guide* for details on building request blocks.) Higher level languages cannot use request blocks directly; they require called subroutines written in Assembly language.

Common Files

Tasks within the same task group and tasks within different task groups can communicate through disk files. The concurrency status must be the same for all tasks using the files. The requesting tasks must have access rights to the files.

Pipes

A pipe is a special type of sequential file that also provides synchronization and queuing facilities to cooperating tasks. Pipes are used by tasks in different task groups (applications) or in the same task group, to communicate with each other.

Message Facility

The message facility allows two or more task groups (users) or two or more tasks within a task group to communicate with one another. This communication is accomplished through containers called mailboxes. Messages (requests) sent to a task or task group are queued in a mailbox and are dequeued when received.

To control the sending and receiving of messages, the message facility provides a number of macrocalls and commands. One set of macrocalls (Initiate, Send, and Terminate Message Group) allows a message (a request) to be sent to a mailbox; another set of macrocalls (Accept, Receive, and Terminate Message Group) allows a message to be received from a mailbox. Commands are provided to allow you to send, receive, list, and cancel messages (requests). The Mail command is provided to allow you to send messages (mail) to another user's mailbox and to display mail in your own mailbox.

Deferred processing of print and task group requests is carried out through the use of the message facility. Deferred processing is described later in this section.

Before the message facility commands or macrocalls can be used, and before the deferred processing of print and task group requests can be initiated, you (or the operator) must create the mailboxes and activate the message facility task.

The paragraphs below describe mailbox creation, the activation of the message facility task, and the command and macrocall interfaces.

Creating Mailboxes

Three steps are involved in the construction of mailboxes. You must (1) create the mailbox root directory, (2) create the mailboxes, and (3) set access controls on the created mailboxes. (Refer to the *Commands* manual for details.)

The mailbox root directory is the directory that is to contain the simple names of the mailboxes.

The system assumes that the mailbox root directory is in the >MDD directory. (An MDD directory is supplied on the system root volume.) You, however, are free to create your own mailbox root directory through the Create Directory command.

Each mailbox is created through the Create Mailbox command. This command creates a directory corresponding to the mailbox name and a file ($MBX) within that directory defining the mailbox attributes.

Setting Access

To prevent unauthorized use of the message queues, you should set access controls as follows:

- Senders must be given list access on the directory defining the mailbox.
- Receivers must be given read access on the $MBX file for a given mailbox.

Individual mailboxes can be deleted using Delete Mailbox commands.

Activating Message Facility Task

The Start Mail operator command activates the message facility. This command contains an optional argument used to set the name of the mailbox root directory to other than the default directory pathname (>MDD).

Message Facility Command Interface

The commands that can be used to send/receive messages (mail) are Mail (MAIL), Send Message Mailbox (SMM), and Accept Message Mailbox (AMM). Commands are also provided to list and delete messages.

Mail Command

The Mail command (also referred to as the local mail facility) is used to send and receive multiline messages to/from the mailbox whose name (id) is the same as the person id of the receiving user. A message sent by a Mail command is queued in the mailbox and displayed only if the receiving user issues a Mail command.

To send a mail message, you issue the Mail command, specifying the mailbox id (person id) of the user to receive the message. The message to be sent can be located in a file (named by an argument of the command) or it can be entered after the Mail command has been invoked.

To receive mail messages, you issue the Mail command without arguments. The contents of your mailbox are displayed when the command is executed. If you request deletion of the messages, they are deleted from the mailbox after being displayed. Otherwise, the messages remain in the mailbox.

SMM and AMM Commands

The Send Message Mailbox (SMM) and Accept Message Mailbox (AMM) commands (also referred to as the local message facility) are used for single-line messages that must be viewed immediately or at a specified time. The AMM command is used to specify that messages sent by the SMM command be displayed when received or at the time specified in the SMM command.

To send a message, you issue the SMM command, specifying the person id (mailbox) to which the message is to be sent. You include the message in the command line. You can include the -TIME argument to specify a delivery time for the message. You can send a broadcast message by specifying * in place of the mailbox in the SMM command. The message will be sent to all logged-on users who have issued an AMM command indicating their willingness to accept messages.

To receive messages, you issue the AMM command. Messages already in your mailbox are displayed. Subsequent messages are displayed when placed in your mailbox. Messages whose date and time for display have not been reached are not displayed. Messages are deleted from your mailbox as soon as they are displayed.

Senders can use both the Mail and SMM commands. A receiving user who issues a Mail command receives both types of messages. If you issue an AMM command, you receive only messages sent by the SMM command, unless you specified the -IMBX * argument. Only when this argument is specified will you receive both types of messages via the AMM command.

The -IMBX argument also allows you to specify by name (person id) the sending user from whom you will accept messages for immediate display. Messages sent by other senders are stored in your mailbox. The -AMBX argument allows you to obtain messages from mailboxes other than the one associated with your person id.

Message Facility Macrocall Interface

You can use the message facility on the Assembly language level by using the macrocall interface. To permit the sending and receiving of messages, the message facility provides the following macrocalls:

- Initiate Message Group ($MINIT)
- Send ($MSEND)
- Accept Message Group ($MACPT)
- Receive ($MRECV)
- Terminate Message Group ($MTMG)
- Count Message Group ($MCMG)
- Cancel Enclosure Level ($MCME).

The information associated with these macrocalls can be passed by means of request blocks.

Send Message

A task group that wishes to send a message to a mailbox must issue a $MINIT macrocall to open the send-message session. The mailbox is identified by a name entered in the request block. As a result of this macrocall, the message facility returns a message id, unique to the task group, to identify the message to the other macrocalls (that is, the send).

The task group then issues one or more $MSEND macrocalls to send message data. The send-message session is closed by the $MTMG macrocall or, alternatively, by the $MSEND macrocall. The sending task group can issue the $MCME macrocall to delete the last record of an incomplete quarantine unit or the entire incomplete quarantine unit. (A quarantine unit is the smallest amount of transmitted data available to a receiving task group.) Receipt of the message can be deferred by the sender.

Receive Message

A task group wishing to receive a message from a mailbox issues a $MACPT macrocall to open the receive-message session. The mailbox is identified as described above for the $MINIT macrocall, and the message facility returns a message id to be used by the $MRECV and $MTMG macrocalls.

The task group then issues one or more $MRECV macrocalls to receive message data. The receive-message session must be closed with a $MTMG macrocall.

The message may be accepted on the following selection criteria:

- First available message
- Sequence number
- Submitter name
- Submitter name and sequence number.

The receiving task group can request the message in record sizes other than those in which the message was sent. The receiving task group delimits the amount of received data by range or enclosure level.

Effective Use

The message facility can be used most effectively by two task groups wishing to communicate if they both simultaneously send and receive a message. To accomplish this, each of the task groups should issue the $MINIT macrocall to open the send-message session and the $MACPT macrocall to open the receive-message session. In this case, the quarantine unit is the vehicle used to exchange data between the two task groups.

DEFERRED PROCESSING FACILITIES

The system's deferred processing facilities are supported by the message facility (refer to "Message Facility" above). In deferred processing, the messages are requests.

Deferring the execution of interactive and absentee task group requests makes it possible for you to gain greater control over the processing sequence. Deferring print requests allows you to obtain program independence from the availability of print devices. Queuing and later transcribing reports provides a spooling capability that places printing and punching outside of program context.

Deferring Task Group Requests

When placing an interactive or absentee task group request, you can have the request entered in a disk queue and can postpone any action being taken on the request until a specified time. When the request queue structures are on disk, memory space is conserved and the data in the queues can be recovered in the event of a system failure (refer to Section 6).

Assuming that an interactive and/or absentee task group has been created, two steps are required to defer group requests. The operator must create the request queues (mailboxes), and you must issue task group requests with optional arguments specifying the time each request is to be activated.

Creating Task Group Request Queues

The operator uses the Create Group Request Queue command to create queue structures in which requests issued to a given task group will be stored. The operator must also issue a Start Mail command if one had not been previously issued.

Queuing Task Group Requests

You queue task group requests by issuing an Enter Group Request command. You can postpone action being taken on a request by specifying the -DFR (defer for interval) or -TIME (defer until date/time) arguments.

Once the operator has issued a Create Group Request Queue command for a task group, all further requests for that group are queued whether or not the requests are being deferred.

If the operator does not issue a Create Group Request Queue command, you can still submit group requests but will not be able to defer the requests.

Deferring Print Requests

The system provides a deferred printing capability under which your requests for printing specified files are queued in memory or disk mailboxes. The actual transcription of the files is done at a later time, under the control of an operator-created system task group called a daemon.

After you submit a deferred print request, you can resume normal activities, log off, or reboot the system without losing the request.

The three steps involved in deferred print processing are (1) creating the mailboxes, (2) activating the daemon, and (3) queuing the print requests. The information in the following paragraphs is conceptual. Detailed procedures for deferred printing are given in the *System User's Guide*.

Creating Print Request Mailboxes

The operator establishes the mailboxes that are to contain the queued print requests. The mailboxes can be in memory or on disk. The mailbox names must be in the form $PR.Qn (n is an integer from 1 through 9 that identifies the relative priority of the queue, with 1 being the highest priority and 9 the lowest).

Creating the Print Daemon

The operator is responsible for defining and activating the daemon to process the print requests.

To create a daemon task group, the operator issues a Start Mail command (if one was not already issued), a Create Group command naming the daemon to be created, and an Enter Group Request command identifying the mailboxes to be used for queuing the requests and the devices to be used for printing.

Multiple daemon task groups can be run concurrently, using common or separate sets of mailboxes and printers.

Queuing Print Requests

Once the daemon task group is active, you can queue, print, or punch requests by issuing Deferred Print commands. You can employ the -TIME argument to defer the printing of a file until a specified date and time.

Queuing and Transcribing Reports

Any file in print or punch format (i.e., any report file) can be queued and subsequently transcribed to an available printer or card punch. Report queuing and transcription is a spooling capability that provides automatic and manual report transcription, time-of-day printing or punching, and an automatic setup function that includes a sample transcription file (template).

The report queuing and transcription facilities control report transcription outside the context of the program. Reporting procedures for identical software can be totally different in different situations without requiring reprogramming.

Report queuing and transcription have three major aspects: creating a report queue, queuing a transcription request, and transcribing a report.

Creating Report Queues

A report queue is a directory that allows you to place a report transcription request in a queue and subsequently transcribe the report. Report queues are created, modified, and deleted through Report Queue Maintenance (RQM) commands. The characteristics of the report queues are determined when the queue is created; the contents are determined when a report transcription request is placed in the queue.

Report Queue Profile

When the report queue is created, a report queue profile file is built. The report queue profile file designates the characteristics of reports whose transcription requests will be entered in the report queue. The report characteristics include:

- Name of form descriptor
- Format of reports to be queued (print or punch)
- Transcription mode (automatic or manual)
- Column number at which printing is to begin
- Line at which printing is to begin (head of form)
- Number of print lines per inch
- Number of copies of report
- Time at which report is to be transcribed
- Heading line
- Destination line.

The report queue profile file is complete when the report queue is created; however, various aspects of the profile can be overridden when the report transcription request is queued.

Queuing Report Requests

The name of a report to be subsequently printed or punched is placed in a report queue through the Queue Report (QRPT) command. This command also associates with the report a specialized report queue profile file that governs the details of the report transcription. Once a request has beer queued, it remains queued until the file has been transcribed or the request pathname has been deleted through a report queue maintenance renew or delete function.

Transcribing Reports

Previously queued reports are written to a printer or card punch through the Unspool (UNSP) command. A single UNSP command can unspool all current and future requests. The printing or punching characteristics are determined by the report queue profile file created through the RQM command, the specialized report queue profile file created by the QRPT command, the user's activities, and the arguments specified in the UNSP command.

The UNSP command defines the report queue and the hard copy device to be used. After the command is executed, the specialized report file (if any) is deleted from the report queue. All reports whose profile matches the specified profile are unspooled in a single invocation of UNSP.

The report queue profile file can specify that the report is to be transcribed automatically or manually.

Automatic Mode

Automatic transcription is used when constant monitoring of a report queue is desired. When there is no transcription activity in progress, the unspool routine suspends itself for 1-minute intervals. When transcription of the queue is activated, each report in the queue is printed immediately unless one of the following is true:

- Manual mode was specified in the controlling profile.
- The specified time of day for report transcription has not been reached (or exceeded).

Manual Mode

Manual mode is used to transcribe reports in a nonautomated fashion. When you require the reports, you issue the UNSP command. All reports on the queue are transcribed immediately, regardless of time or mode. When the queue is empty, UNSP terminates.