iRMX®
System Concepts

RadiSys Corporation
5445 NE Dawson Creek Drive
Hillsboro, OR 97124
(503) 615-1100
FAX: (503) 615-1150
www.radisys.com
07-0635-01
December 1999

December 1999

Quick Contents

Notational Conventions

Most of the references to system calls in the text and graphics use C syntax instead of PL/M (for example, the system call **send_message** instead of **send$message**). If you are working in C, you must use the C header files, *rmx_c.h*, *udi_c.h* and *rmx_err.h*. If you are working in PL/M, you must use dollar signs ($) and use the *rmxplm.ext* and *error.lit* header files.

This manual uses the following conventions:

- Syntax strings, data types, and data structures are provided for PL/M and C respectively.
- All numbers are decimal unless otherwise stated. Hexadecimal numbers include the `H` radix character (for example, `0FFH`). Binary numbers include the `B` radix character (for example, `11011000B`).
- Bit 0 is the low-order bit. If a bit is set to 1, the associated description is true unless otherwise stated.
- `Data structures and syntax strings appear in this font.`
- **System call names and command names appear in this font.**
- PL/M data types such as BYTE and SELECTOR, and iRMX data types such as STRING and SOCKET are capitalized. All C data types are lower case except those that represent data structures.
- Whenever this manual describes I/O operations, it assumes that tasks use BIOS calls (such as **rq_a_read**, **rq_a_write**, and **rq_a_special**). Although not mentioned, tasks can also use the equivalent EIOS calls (such as **rq_s_read**, **rq_s_write**, and **rq_s_special**) or UDI calls (**dq_read** or **dq_write**) to do the same operations.

Note

Notes indicate important information.

CAUTION

Cautions indicate situations which may damage hardware or data.

Contents

Section I: NUCLEUS PROGRAMMING CONCEPTS

1 Jobs

2 Tasks

5 Regions

6 Ports

8 Exception Handling and System Accounting

10 Interrupts

13 OS Extensions and Type Managers

Section II: KERNEL PROGRAMMING CONCEPTS

14 iRMX Kernel Programming Concepts

Section II: I/O SYSTEMS PROGRAMMING CONCEPTS

15 I/O System Basic Concepts

16 I/O Jobs and Connections

17 Named Files

Section V: HUMAN INTERFACE PROGRAMMING CONCEPTS

25 Human Interface Basic Concepts

26 The Command Line Interpreter

27 Writing and Parsing Commands

Tables

Figures

NUCLEUS PROGRAMMING CONCEPTS

This section documents the iRMX® Nucleus subsystem. Its functions include:

- Providing objects for communication and resource access control
- Scheduling tasks based on priority
- Handling interrupts based on interrupt level

The Nucleus consists of:

Kernel	Provides low level interfaces and primitives.
Resident Nucleus	Provides high level interfaces, memory protection and validation.
Nucleus Messaging Service (NMS)	Provides high level message passing
Interface libraries	Provide communication between OS layers.

These are the chapters in this section:

Jobs 1

What is a Job?

A job consists of a set of tasks and the resources they use, in a shared address space: the job's memory pool. Tasks within a job use and share the job's resources to do their work. A job isolates its tasks and resources from other jobs because jobs cannot share memory pools.

When you have tasks and resources that need to be isolated, create a separate job for them.

Job Hierarchy

Jobs are arranged in a hierarchy; the root job is always topmost; other jobs descend from the root job. A *parent* job is a job that contains tasks that create other jobs. The created jobs are *child* jobs.

The Nucleus maintains the job hierarchy, keeping track of the relationships of parent and child jobs.

See also: Jobs, *Introducing the iRMX Operating Systems*, for basic information on job hierarchy

Job Types

The iRMX OS supports several kinds of jobs.

Job Type	Description
Root job	Created by the Nucleus at system initialization. All jobs in the system descend from the root job.
First level job	Created by the Nucleus at system initialization. First level jobs are child jobs of the root job. The BIOS and EIOS, for example, are first-level jobs. In ICU-configurable (Interactive Configuration Utility) systems, you can specify your application as a first-level job.
Loadable job	Loadable jobs are child jobs of the HI. You can create your applications as one or more loadable jobs. Other loadable jobs are the shared C library, network jobs, I/O jobs, and device and file drivers.
System job	System jobs include servers and networking. In ICU-configurable systems, system jobs are first-level jobs created by the Nucleus at system initialization. They are child jobs of the root job. You can use the ICU to specify which of the system jobs supplied by Intel to include in your system. You can load some system jobs using the *loadinfo* file rather than making them first-level jobs with the ICU.
Dependent/child job	Descend from other jobs. They are created dynamically as the system runs. Parent jobs create dependent child jobs. Most of the jobs you create are dependent jobs.
I/O jobs	Dependent jobs that provide the environment for EIOS system calls. You create I/O jobs for tasks that use these calls. I/O jobs are child jobs of the EIOS and HI.

Your application will probably contain more than one first-level or dependent job because you will have tasks and resources that need isolation. The number of jobs depends on the complexity of the application and the modularity of your design.

See also: I/O jobs, in this manual;
system jobs and loadable jobs, *System Configuration and Administration*

What Does a Job Contain?

A job can contain these resources:

Resource/Object	Description
Task	A thread of execution. Tasks do the work of the system. A job may contain several tasks. One is the initial task created by the Nucleus. The remainder are created by the initial task. You group related tasks in the same job.
Object directory	A list of object names and tokens which tasks in jobs share with each other. You catalog objects in the object directory.
Memory pool	A pool of up to 4 Gbytes that provides the memory that tasks share and use to do their work within the job. You specify the size of the memory pool.
Memory segment	A contiguous sequence of bytes that tasks use for any purpose. You have to create and delete segments.
Buffer pool	Dynamically allocable memory. First you create the buffer pool and load it with segments. Then to allocate memory you only need to specify the buffer pool's token and how much memory you need. The Nucleus allocates and returns memory to the buffer pool.
Mailbox	Passes messages or data between tasks. You can pass messages and data between tasks in different jobs.
Semaphore	Synchronizes tasks. A semaphore is a counter.
Region	A one-unit semaphore with special suspension, deletion and priority-adjustment features. Regions provide mutual exclusion. Only one task can access a region at a time.
Port	Passes messages between tasks in the same or different jobs. Synchronizes operations between boards in a Multibus II system.
Exception handler	Specifies what to do when a hardware, programmer or environmental error occurs.

Each object you create uses an entry in the Global Descriptor Table (GDT).

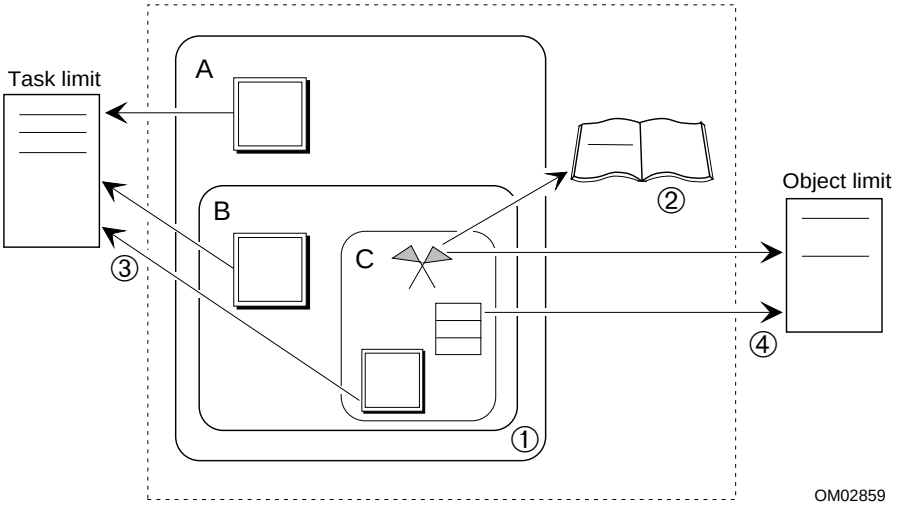
See also: Individual objects and exception handling in *Introducing the iRMX Operating Systems*;
individual object chapters in this manual

Creating a Job

When you create a job using **create_job** or **rqe_create_job**, you specify its resources, which is a parameter object the parent job can pass to the child, and an initial task. These resources are taken from the parent job's memory pool.

Resource Sharing

When you create a job that will have an extensive hierarchy beneath it, be sure you specify enough resources (memory, object directory entries, tasks and objects) in the new job because all of the tasks in the new job and any subsequent child jobs will share the resources of the new job. Since a child job gets its resources from its parent job, resources in child jobs cannot exceed those of the parent, as shown in Figure 1-1.



1. The memory pools for child tasks B and C are allocated from the memory pools of their parent jobs.
2. Any objects cataloged by child jobs in the parent job's object directory reduce the number of entries remaining to be made in the parent job.
3. Any tasks created by the child jobs reduce the number of tasks remaining to be created in the parent job.
4. Any objects created by the child jobs reduce the number of objects remaining to be created in the parent job.

Figure 1-1. Resource Sharing in Jobs

Specifying Resources

These are the resources you specify when you create a job:

- Maximum number of entries allowed in the job's object directory. Alternatively, you can specify no object directory if tasks do not share objects.
- Maximum and minimum sizes of the job's memory pool, to be shared by all tasks in the job and any child jobs they create.

  See also: Borrowing memory, in this manual
- Maximum number of objects that tasks in the job can create. You can specify that an unlimited number of objects can be created by tasks in the job.
- Maximum number of tasks allowed to exist within the job at a given time. Since the Nucleus always creates an initial task, you cannot specify 0. You can specify that an unlimited number of tasks can be created.
- Maximum (numerically lowest) task priority at which any task contained in the job can execute. You can specify that the child job inherits the maximum task priority of its parent. You cannot specify a maximum task priority that exceeds the maximum task priority in the parent job.
- Exception handler to use for tasks in the job and when to pass control to it. Alternatively, you can use the default exception handler, which deletes the job.
- Whether the Nucleus should validate system call parameters for calls made within the job's tasks and in child jobs. You can enable parameter validation in a child job even if you disabled it in the parent job.

The Parameter Object

When you create a child job, the parent job can pass a *parameter object* to the child job, if needed. The parameter object can be of any object type and can be used for any purpose. For example it can be a segment containing data, arranged in a predefined format, which the child job needs. The child job accesses the parameter object by getting its token with the **get_task_tokens** system call.

If there is no need to pass a parameter object, don't specify one.

The Initial Task

The Nucleus creates the initial task for the new job. This task reduces by one the maximum number of tasks in the parent job.

You program the initial task to do initializing or housekeeping needed when the job starts running. You can program the initial task to either delete itself or continue to exist as a regular task, perhaps doing other housekeeping operations.

You supply the same information to the Nucleus about the initial task as you do when you create a task yourself:

- The priority of the initial task, which must not exceed the new job's maximum task priority. Alternatively, you can specify that the task use the job's maximum task priority.
- A pointer to the initial task's start address.
- A token for the initial task's data segment. Or you can let the Nucleus assign the segment.
- A pointer to the initial task's stack. Unless you have a specific reason to do so, let the Nucleus create the stack and assign the stack pointer. Otherwise, particularly in first-level jobs, results may be unpredictable.
- The size of the stack.

  See also: Stack size, *Programming Techniques*
- Whether the initial task contains floating-point instructions.

Deleting a Job

Before you delete a job using **delete_job**, you have to delete all its child jobs and its extension objects, if any exist.

Use the **rqe_offspring** system call to find the child jobs. Delete jobs starting from the bottom of the job's hierarchy, beginning with childless jobs. After you have deleted all child jobs, delete the job itself; all the job's objects are deleted too, even if tasks in other jobs have access to them. The deleted job's memory is returned to the parent job.

Use the **delete_extension** system call to delete extension objects and their composite objects.

Job System Calls

These are the system calls that relate directly to jobs:

rqe_create_job
create_job
delete_job
end_init_task (ICU-configurable systems only)
rqe_offspring
get_task_tokens
set_pool_min
rqe_set_max_priority

Table 1-1 lists common operations related to jobs and the Nucleus system calls that do the operations.

Table 1-1. Job System Calls

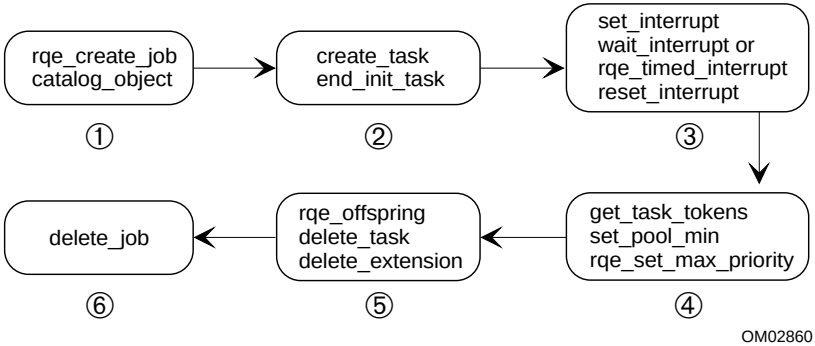
Operation	Description
create job	**Rqe_create_job** and **create_job** create a job with an initial task and returns a token for the job.
delete job	**Delete_job** deletes a job that has no child jobs or extension objects.
signal Nucleus	Use **end_init_task** in the initial task to signal the Nucleus that initialization is complete.
get token for object	**Get_task_tokens** gets a token for a parameter object or for the task's job, parent job or root job so you can catalog objects.
find child jobs	**Rqe_offspring** gets tokens for all child jobs so you can delete them. It returns the list in a structure you supply.
set minimum size of job's memory pool	**Set_pool_min** changes the minimum size of the job's memory pool from its creation size. If the new minimum is greater, memory will be obtained from the job's memory pool if possible.*
set maximum priority of tasks in job	**Rqe_set_max_priority** dynamically changes the maximum priority of a task in a job. The new maximum task priority must not be greater than (numerically less than) the job's maximum task priority.

***** The amount actually allocated depends on the current allocation, the requested minimum and maximum, granularity of units allocated, and how memory is already allocated from the memory pool. The minimum pool size must not exceed the maximum pool size.

See also: Nucleus system calls, *System Call Reference*

How to Use Job System Calls

Figure 1-2 shows the order in which you make job system calls and mentions calls that tasks in jobs frequently use.



1. Make these calls from the task that needs to create the new job.
2. Make these calls from the initial task created by the Nucleus.
3. Make these calls from the job's interrupt tasks.
4. Make these calls from any tasks in the job. You will also use calls that:
   - Create, catalog, manipulate, and delete objects
   - Change a task's priority or execution state
5. Make these calls from the initial task or another housekeeping task.
6. Make this call from the task that created the job.

Figure 1-2. Job System Call Order

❑❑❑

Tasks 2

What is a Task?

A task is a thread of execution that does the work of the system. It is only active object in the system. It runs a sequence of instructions to manipulate data and objects. It is the active object within a job.

You will probably have several tasks in one job. One will be the initial task which you specified in the **create_job** call and which the Nucleus created to initialize the job environment. You may create other tasks and group them together in one job environment because:

- They have similar or related purposes.
- They share resources.
- They exist for similar lengths of time.

There is no hierarchy among iRMX tasks: all tasks in a job belong to the job, even if one task has created the others. All objects in a job belong to the job, not to the tasks that created them.

The executable part of a task is a procedure without parameters that never returns, similar to `main()` in C programs. A task makes system calls and may call other procedures. You can write a procedure specifically for one task, or share it among several tasks.

The Nucleus schedules tasks so that each task sees itself as having its code executed continuously. Depending on the needs of the application, a task may execute in these, or other, ways:

- Execute once, then delete itself
- Execute in an infinite loop, spending most of its time waiting for an event to occur, such as a message arrival, an interrupt, or an elapsed time interval
- Execute in an infinite loop, spending most of its time performing its function

Task Types

These are the basic types of tasks.

Type	Task Function
initial	Initializes the job environment and creates one or more tasks for the job. It is created by the Nucleus and is the first task to run in a new job. It may exist for the life of the job, performing housekeeping and other functions or execute once.
ordinary	Typically responds to internal events. Does work required of the application.
interrupt	Services incoming interrupts.

Task Attributes

A task inherits some attributes from its parent job, such as its exception handler and exception mode. It also has these attributes of its own:

- An instruction pointer that points to the currently executing instruction in the task
- The task state at initialization and the current execution state

  See also: Task execution states in this section
- The current suspension depth of the task

  See also: Task execution states in this section
- Whether the task is an interrupt task
- The parent job
- The code, data and stack segment register context

Once you create a task, the Nucleus keeps track of these attributes.

Creating a Task

When you create a task using **create_task**, the Nucleus takes resources that it needs (such as memory for a stack) from the parent job. These are the resources you specify when you create a task:

- The priority of the task, which must not exceed the job's maximum task priority. Alternatively, you can specify that the task use the job's maximum task priority.
- A pointer to the task's start address.
- A token for the task's data segment. Or you can let the Nucleus assign the segment.
- A pointer to the task's stack. Unless you have a specific reason to do so, let the Nucleus create the stack and assign the stack pointer. Otherwise, particularly in first-level jobs, results may be unpredictable.
- The size of the stack.

  See also: Stack size, *Programming Techniques*
- Whether the task contains floating-point instructions.

These are the system calls for creating the three types of tasks:

Task Type	System Call
initial	**create_job**
ordinary	**create_task**
interrupt	**create_task** followed by **set_interrupt** called from within the new task, which will become the interrupt task

Deleting a Task

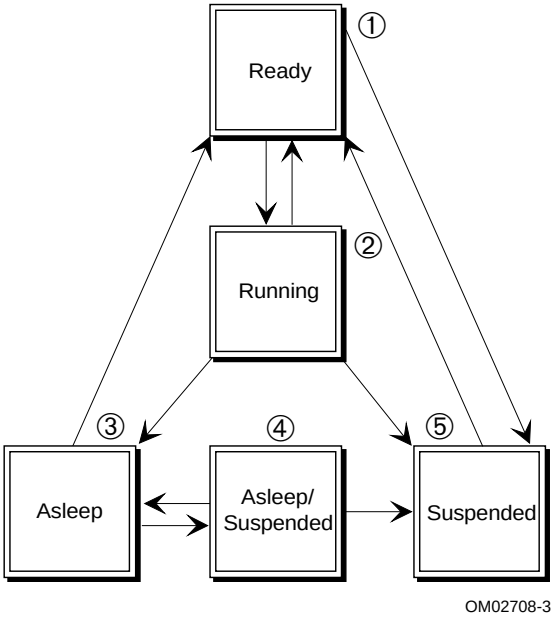
When you delete a task using **delete_task**, the task is disassociated from its parent job, and any stack segments created for it are reclaimed for allocation to new tasks. The task's resources are returned to the parent job. These are the system calls to delete the three kinds of tasks:

Task Type	System Call
initial	**delete_task**
ordinary	**delete_task**
interrupt	**reset_interrupt**; when an interrupt task is reset, **delete_task** is automatically called

If a task makes C library calls, call **c_stop** before calling **delete_task**.

Task Execution States

A task exists in one of the execution states shown in Figure 2-1.



1. A task is usually created in the ready state. It is not running, asleep, and/or suspended. Tasks created in I/O jobs can start in suspended state.
2. The task's instructions are being executed; only one task can execute at a time.
3. The task is voluntarily waiting for something to wake it up; it controls the length of time it stays asleep. A task goes to sleep because:
   - It makes a request that cannot be done at once and it will wait (forever if necessary).
   - It puts itself to sleep for a specified time. The task will not specify *sleep forever*.

   The asleep state is the most common state for tasks waiting for an event. A task may not put another task to sleep.
4. The sleeping task is suspended. The suspension depth increases by one each time the task is suspended. If the task's sleep time expires first, it enters the suspended state. If the task is resumed first, it enters the asleep state.
5. The task had its execution postponed because it has suspended itself by waiting for an event or interrupt or has been suspended by another task. The suspension depth increases by one each time the task is suspended.

Figure 2-1. Task Execution States

Task Execution State Transitions

As an application runs, a task often transitions from one execution state to another. A task in any state except ready cannot run, even if it has the highest priority. You can delete a task from any state. Creating a task instantly makes it ready.

Transition	Reason
Ready to running	The task becomes ready, has the highest priority of all ready tasks and one of these: • It has a higher priority than the running task. • The running task is suspended, put in the sleep state, or deleted. • The running task's time quota has expired, and the ready task is next in the queue.
Running to ready	One of these: • A higher priority task becomes ready. • The task uses all of its time quota in round-robin scheduling.
Running to asleep	One of these: • The task puts itself to sleep for a specified time. • The task requests something that cannot be done immediately and it can wait.
Asleep to ready, or asleep-suspended to suspended	One of these: • The sleep time expires. • The sleep time expires before a request is granted. • The request is granted because another task sends a message and the message is received. • The object the task is waiting at is deleted.
Running to suspended	The task suspends itself.
Ready to suspended or asleep to asleep-suspended	The task is suspended by another task. The suspension depth increases by one each time the task is suspended.
Suspended to ready or asleep-suspended to asleep	The suspension depth is one and the task is resumed by another task.

These are the system calls that cause execution state transitions:

catalog_object	**receive_units**
create_job	**receive_task**
create_task	**send**
enable	**send_control**
enable_deletion	**send_data**
end_init_task	**send_reply**
force_delete	**send_message**
lookup_object	**send_rsvp**
receive	**send_signal**
receive_control	**send_units**
receive_data	**sleep**
receive_fragment	**suspend_task**
receive_message	**timed_interrupt**
receive_reply	**wait_interrupt**
receive_signal	

Suspending and Resuming Tasks

You will not encounter problems when a task uses **suspend_task** to suspend itself. You may get unpredictable results when using **suspend_task** to suspend another task for synchronization. Whenever possible, use a semaphore or mailbox to synchronize tasks instead.

Each time you call **suspend_task**, the suspension depth increases by one. The Nucleus keeps track of the task's suspension depth, up to 255. The larger the number of calls made, the greater the depth. When the suspension depth is >0, you must make a corresponding number of **resume_task** calls to bring the task out of suspension. You cannot obtain the suspension depth of a task from the Nucleus.

You need to make multiple calls to **resume_task** from another task to make a task ready when the suspension depth is >1. Each time you call **resume_task**, the suspension depth decreases by one. You do not have to make the **resume_task** calls from the task that suspended the task.

See also: **suspend_task** example, *Programming Techniques*

Prioritizing Tasks

The Nucleus handles task scheduling, based on priority or interrupt level; what job a task belongs to has no effect on scheduling. The Nucleus always executes the highest priority running task until it is interrupted, is preempted by a higher priority ready task, or it puts itself to sleep, suspends itself, or completes and relinquishes control.

Task Priority Level

The priority level of a task determines its importance in relation to other tasks and interrupts. You specify a task's static priority when you create it or later using **set_priority** if you need to. The task's priority may be adjusted by the OS when using a region (described later); this is called *dynamic priority*. The priority is an integer value from 0 through 255, with 0 being the highest priority.

Range	Used For
0 - 16	Used by the OS for servicing hardware exceptions.
17 - 127	Used by the OS for servicing external interrupts. Let the Nucleus assign these levels to handlers and interrupt tasks, based on the order in which you attached your external interrupt sources to the PICs. In general, don't create tasks in this range. A task running in this range masks everything numerically lower, meaning response time to external interrupts is slower and interrupts may be lost.
128 - 130	Use for tasks that communicate with interrupt tasks. These tasks may, for example, do some asynchronous processing that is related to, but not required for servicing the interrupt.
131 - 255	Use for tasks that handle internal events, like message passing and computation. Typically, you don't assign a task to every level in this range. You might put important tasks, such as mailbox managers, in the range 140 - 160. Leave some gaps if you plan to add features and tasks later on. You can usually start using round-robin scheduling at about 200. See also: Round-robin Scheduling in this section

Interrupt Task Priority Level

Interrupt tasks are tasks that you create, using the **create_task** call; assign the default priority for the task's job. Then associate the task to an interrupt handler using the **set_interrupt** call. You may use the **rqe_set_max_priority** call to adjust the job's maximum task priority, if needed.

Typically, you create interrupt tasks in related jobs. If you are using the I/O System, however, interrupt tasks are created within the BIOS job.

See also: Managing Interrupts in this manual for more information on interrupt handling

Round-robin Scheduling

You can assign the same priority level to more than one task and let the tasks take turns running. Typically you do not do this with important tasks.

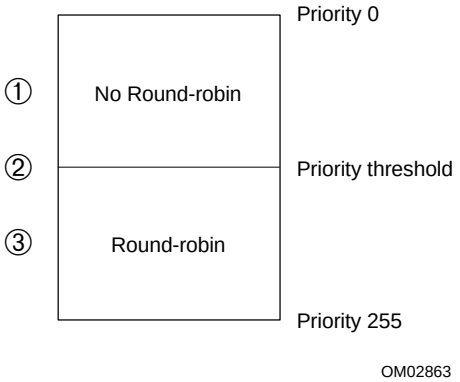
Unless you use round-robin scheduling, the first task, Task A, at any given priority level can run until interrupted or put in the ready state by a higher-priority task. Task A will regain control after the interrupt has been serviced or the higher-priority task completes. Other tasks assigned the same priority level can be left waiting indefinitely unless Task A voluntarily gives up control of the CPU. This could be disastrous in a multiuser environment.

See also: *Introducing the iRMX Operating Systems* for basics on round-robin scheduling

The default round-robin level is 140. You set two parameters that affect round-robin scheduling: the threshold priority level and the time quota each task can run before it is preempted. In an ICU-configurable system, you can use the Nucleus screen to set the RRP and RRT parameters; otherwise you use the *rmx.ini* file to set them.

See also: RRP and RRT, *ICU User's Guide and Quick Reference*;
Loadtime parameters, RRP and RRT, *System Configuration and Administration*

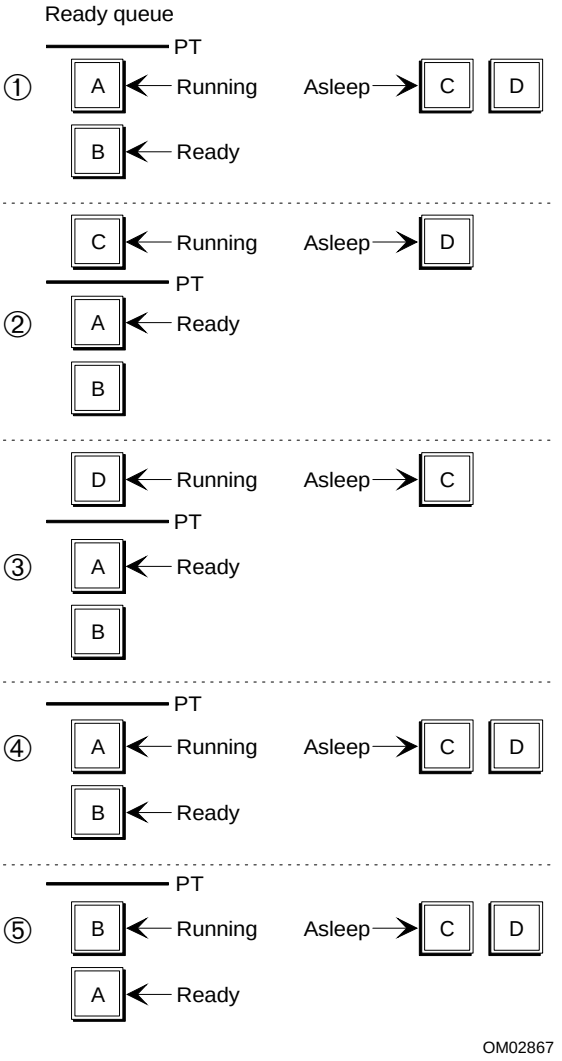
Figure 2-2 illustrates round-robin scheduling and the priority threshold.



1. At or above the priority threshold, no round-robin scheduling occurs.
2. You set the priority threshold at configuration.
3. Below the priority threshold, round-robin scheduling automatically occurs between tasks of equal priority.

Figure 2-2. The Round-robin Priority Threshold

Figure 2-3 shows how round-robin scheduling works with priority-based scheduling.



PT = Priority Threshold

Figure 2-3. Round-robin and Priority-based Scheduling within the Ready Queue

The priority threshold in the figure is 200. There are tasks A, B, C, and D with these priorities in the figure:

Priority **Tasks**

130 Task C
140 Task D
200 Tasks A and B

1. Task A runs for 2 clock ticks when Task C becomes ready.
2. Task C has higher priority than A and B, so it gains control and runs until done. In the meantime, Task D becomes ready.
3. Task D has higher priority than A and B, so it gains control and runs until done.
4. Task A runs again for its remaining 3 clock ticks, then relinquishes control to Task B.
5. Task B runs until it has used all of its 5 clock ticks or completed. It relinquishes control. Task A begins running for another 5 clock ticks.

Communicating Between Tasks

Tasks communicate with each other to exchange data and synchronize execution. The OS provides four exchange objects used in exchanging data and synchronizing tasks. They are:

- Mailbox
- Port
- Semaphore
- Region

See also: Chapters about each object, in this manual;
Designing an Application, *Programming Techniques*;
examples in */rmx386/demo/c/intro* directory

For tasks to share the exchange objects, you must create them, then catalog them from the creating task using **catalog_object**. Then other tasks can use **lookup_object** to get the object's token so they can access the object.

Using Mailboxes and Ports

Tasks commonly use mailboxes or ports to request a service from another task. The client task sends a message that specifies parameters for the service call; the service task receives the message and provides the specified service. The service task can return results of the service, if any, to the client using a mailbox or port.

See also: Mailboxes and ports, *Introducing the iRMX Operating Systems*;
Chapters about mailboxes and ports, in this manual

Advantages and Disadvantages of Mailboxes

You use a mailbox to send variably-sized messages between tasks in the same or different jobs on the same host processor. A mailbox is easy to use for single-message exchanges. A data mailbox, on the other hand, requires iRMX support at both the sending and receiving task. Additionally, a message arriving at a mailbox where no task is waiting is copied by the OS into buffer space in the mailbox message queue; this is the first copy. When a task arrives to receive the message, the message is copied by the OS from the mailbox queue into the task's message buffer; this is the second copy. There is no copying with message-type mailboxes, which pass tokens only.

Advantages and Disadvantages of Ports

A port transmits large messages between tasks on the same processor (short-circuit message passing) and communicates between tasks on different processors in a Multibus II system. A port provides access to non-iRMX applications using the Multibus II transport protocol.

A port provides a transaction-based protocol; request messages are tied to specific response messages. This enables a task to send messages to many tasks and to distinguish the replies from one another. Tying the request to the response is handled automatically by the iRMX OS.

A port copies a message only once. If a message arrives at a port where no task is waiting, the message is copied into buffer space (which you must allocate) in the port's buffer pool; this is the only copy. When a task arrives to receive the message, the OS gives the task a pointer to the message buffer. A second copy of the message is not made.

Using Semaphores and Regions

Both semaphores and regions provide mutual exclusion to shared resources.

See also: Semaphores and regions, *Introducing the iRMX Operating Systems*; Chapters about semaphores and regions, in this manual

You can use a semaphore with more than one unit as a general purpose counter to synchronize the actions of multiple tasks. A semaphore with one unit can also provide mutual exclusion of tasks, but without the dynamic priority adjustment and deletion protection provided by regions. Semaphores do not enforce synchronization or mutual exclusion. Semaphores provide more flexibility in waiting for access to resources than regions; you can specify a time limit for waiting in the task queue. A task using a region cannot set a time limit.

Tasks use regions to enforce mutual exclusion to a specific resource or data. Only one task at a time can control a region. The task that is controlling the region cannot be deleted or suspended. If the region has a priority-based task queue, the task in the region will have its priority dynamically adjusted so that it is always at least as high as the highest priority task waiting in the queue. When a task gains control of several regions, then gives up control of the regions one at a time, the task's dynamic priority is not readjusted to its static priority until the task gives up control of the last region; this improves performance.

Task and Message Queues

If a task makes a request that cannot be filled immediately and the task is willing to wait, the task stops executing, goes into a task queue and goes to sleep. More than one task can wait in a queue. You specify whether the OS places tasks in the queue in either a first-in-first-out (FIFO) or a priority-based manner when you create the exchange object.

- In a FIFO queue, tasks are queued in the order they arrive at the exchange object.
- In a priority queue, the highest priority tasks move to the head of the queue. Tasks of equal priority are arranged in order of arrival.

You specify the maximum length of time the task can wait in the queue when you tell the task to receive data or a message, a signal, or semaphore units.

Besides the task queues maintained by all exchange objects, mailboxes and ports also have message queues to hold incoming messages for tasks. Message queues are always FIFO-based.

Task System Calls

These are the system calls that relate directly to tasks:

create_task
delete_task
reset_interrupt
sleep
suspend_task
resume_task
get_priority
set_priority
get_task_tokens

Table 2-1 describes common operations on tasks and the system calls that perform the operations.

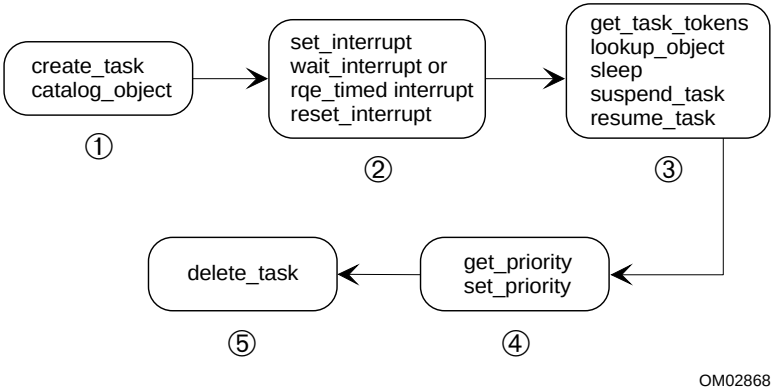
Table 2-1. Task System Calls

Operation	Description
create task	**Create_task** creates a new task and returns a token for it.
delete task	**Delete_task** deletes the specified task. It calls reset_interrupt for interrupt tasks.
put task to sleep	**Sleep** puts the calling task to sleep for a specified time. One task may not put another to sleep.
suspend task	**Suspend_task** lets tasks suspend themselves and other tasks. **Suspend_task** increases the suspension depth by one.
resume task	**Resume_task** decreases the suspension depth by one.
modify task priority	**Get_priority** checks the priority of the specified task. **Set_priority** sets task priority to the specified level, which must be • Equal to or greater than the parent job's priority level • Within the allowable range of priorities (0 to 255) You cannot change the priority level of interrupt tasks.
obtain specific token	**Get_task_tokens** finds out the token for any one of these objects: • Task's own token • Task's job • Parameter object of the task's job • Parent job of the task's job • Root job of the system

See also: Nucleus system calls, *System Call Reference*;
examples in the */rmx386/demo/c/intro* directory

How to Use Task System Calls

Figure 2-4 shows the order in which you make task system calls and mentions calls that tasks frequently use.



1. Make these calls from the task that needs to create the new task.
2. Make these calls if the new task is to be an interrupt task.
3. Make the **get_task_tokens** and **lookup_object** calls from the new task to obtain tokens for other jobs, tasks and objects in the system. Make the **sleep** call if the task needs to wait. Make the **suspend_task** call from a task that has completed and no longer needs to run. Make the **resume_task** call from another task.

   You will also use calls that create, catalog, manipulate and delete objects.
4. Make these calls from the new task to change its own or another task's priority.
5. Make this call from the task that created the task.

Figure 2-4. Task System Call Order

□□□

Mailboxes 3

What is a Mailbox?

Tasks exchange information by sending messages to and receiving messages from mailboxes. A message may be either an object token or a stream of data.

You can create two kinds of mailboxes: object (usually for object tokens) or data. The choice depends on the information your tasks need to exchange. An object mailbox cannot pass data (except in segments).

Sending and receiving data uses different system calls than sending and receiving object tokens.

See also: Nucleus system calls, *System Call Reference*; examples in the */rmx386/demo/c/intro* directory

Object Mailboxes

You use an object mailbox to pass an object token, usually a segment token, to another task. To use an object mailbox to send a segment, you must create the segment, then send the segment's token to an object mailbox. An object mailbox is also called a message mailbox.

Data Mailboxes

Use data mailboxes for passing small amounts of information. You won't have to create and delete segments or dereference a segment token after a task receives it.

Although the amount of data per message is limited to 128 bytes, the data can be a pointer to a larger area. Passing data instead of objects can be important in systems where the GDT is almost full, because each object uses an entry in the GDT.

Creating a Mailbox

When you create a mailbox using **create_mailbox**, the Nucleus takes resources that it needs from the task's parent job. These are the parameters you specify when you create a mailbox:

- Whether the mailbox passes data or objects.
- For object mailboxes, the number of objects that can be in the high-performance message queue. By default, the OS creates a high-performance queue of eight objects. For data mailboxes, you do not specify the size of the queue.
- Whether the task queue is FIFO or priority based.

Mailbox Queues

Each mailbox has two queues: a task queue and a message queue. At any given time, at least one of the queues is empty, because the Nucleus sees that waiting tasks receive messages as soon as they are available.

See also: Task and message queues in Chapter 2

Queues For Object Mailboxes

By specifying a high-performance queue that is large enough to contain all the objects queued during normal operations, you improve the performance of **send_message** and **receive_message** when these calls get or place objects in the queue. The Nucleus permanently allocates memory for a high-performance queue even if no objects are stored in it, so memory does not have to be allocated dynamically.

The Nucleus automatically handles overflow. When more objects arrive than the high-performance queue can hold, the Nucleus creates a temporary overflow queue that holds up to four messages. The overflow queue is not deleted until it empties. Because the overflow queue is created once for every additional four messages, performance is only affected when a **send_message** system call causes the allocation of an overflow queue. Then, extra time is required for the allocation.

Queues For Data Mailboxes

The default queue for data mailboxes is three messages, 128 bytes each. When more messages arrive than the queue can hold, the Nucleus creates a temporary overflow queue that holds up to 400 bytes. The overflow queue is not deleted until it empties.

Reconfiguration Mailboxes

Multibus II systems that are configured with the watchdog timer to support live insertion use reconfiguration mailboxes in the iRMX OS. The watchdog timer sends messages to these mailboxes to indicate board failures or resets. You create a reconfiguration mailbox by first creating a mailbox with **create_mailbox**. Then, you use the **add_reconfig_mailbox** system call to specify the mailbox as a reconfiguration mailbox. A reconfiguration mailbox must be a data mailbox.

See also: **add_reconfig_mailbox** system call, *System Call Reference*; Live Insertion, Chapter 12

Deleting a Mailbox

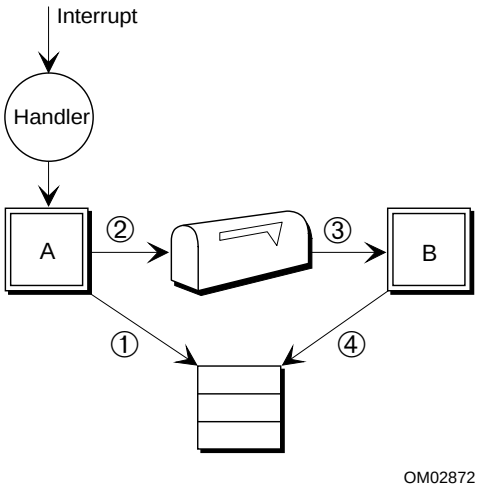
When you delete a mailbox using **delete_mailbox**, the Nucleus:

- Awakens any tasks waiting at the mailbox with an E_EXIST condition code
- Discards any messages in the queue

Exchanges Between Tasks in the Same Job

Figure 3-1 on page 52 illustrates an exchange between two tasks in a single job. When tasks in the same job use mailboxes, they can use object mailboxes.

In this figure, Task A is an interrupt task associated with an interrupt handler; Task B is an ordinary task. If Task A sends messages to the mailbox faster than Task B can receive them, the messages will be queued at the mailbox until Task B can get to them. This is a good situation to use a high-performance queue.



1. Interrupt Task A creates a segment to store the data it expects to receive from the interrupt handler using **create_segment**.

2. Interrupt Task A creates an object mailbox using **create_mailbox** and catalogs it using **catalog_object**. Task A goes to sleep by waiting for a signal from an interrupt handler, using **wait_interrupt**.

   Task B looks up Task A's mailbox using **lookup_object**. Task B goes to sleep by waiting for a message at the mailbox, using **receive_message.** (If the tasks share a common data segment, you could store the mailbox token there and avoid using **lookup_object**.)

   When Task A receives the signal from the interrupt handler, it wakes up, places data into the segment and sends the token to Task B using **send_message**.

   Task A then creates a new segment for the next interrupt and waits.

3. Task B wakes up and receives the token in the mailbox.

4. Task B processes the segment, then deletes the segment.

Figure 3-1. Exchanging Objects Between Tasks in the Same Job

Using send_message

Send_message sends a single object to a mailbox and enables a task to request acknowledgment from the receiving task.

When you send a message:

- If a task is waiting, it receives the message immediately. If the receiving task has been asleep, it moves either from asleep to ready or from asleep-suspended to suspended.
- If no task is waiting, the message is placed at the tail of the message queue. Message queues are processed as FIFO, so the message remains in the queue until it moves to the head of the queue and is given to a task.

Using receive_message

When a task is waiting to receive a message:

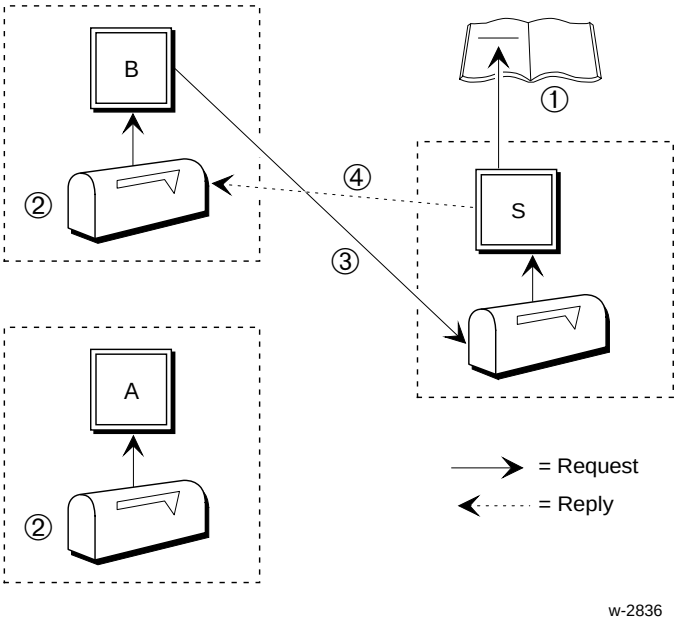
- If there is a message in the queue when a task arrives at a mailbox, the task receives the message immediately.
- If there is no message in the queue, the task may or may not wait in a task queue.
  - If the **receive_message** call indicates that the task can wait, it is placed in the task queue and goes to sleep. This is how you use mailboxes to synchronize tasks as well as pass messages. A sleeping task wakes when a message arrives or when a specified time limit expires.

    The task receives an E_TIME condition code if the designated waiting period elapses before the task gets a message.
  - If the **receive_message** call specifies that the task cannot wait, the task remains ready and immediately receives an E_TIME condition code.

If you use **receive_message**, check to see if an acknowledgment has been requested.

Exchanging Data Between Tasks in Different Jobs

Figure 3-2 shows a server task that does similar services for several client tasks in different jobs. The server and clients have their own mailboxes. The server should catalog its mailbox in the root job's object directory. Each client sends the token for its mailbox to the server so the server will know where to reply.



1. The Server Task S creates a data mailbox using **create_mailbox** and catalogs it in the root job's object directory using **catalog_object**. The Server Task puts itself to sleep using **receive_data**.

2. Each Client Task creates its own mailbox. Each Client Task looks up the token for the server task's mailbox using **lookup_object**.

3. When either client (Task B in this example) sends data using **send_data**, the client includes the token for its mailbox in the call.

4. When a message from either client arrives, the Server Task wakes up and processes the data. It sends a reply to the appropriate client task's mailbox (Task B in this example) using the mailbox token included in the **send_data** call.

Figure 3-2. Exchanges Between Tasks in Different Jobs

Using send_data

The maximum amount of data transferred by the **send_data** system call is 128 bytes. You must create a send buffer for the data and pass a token or a pointer to it. Pass a token if you have created a segment using the **create_segment** call or a pointer if you have declared a data structure in a portion of the DS.

The original data area becomes available for re-use after **send_data** returns.

You cannot request acknowledgment from the receiving task when you use **send_data**.

If there is a task waiting at the mailbox when the message arrives, the message is copied directly to the task's receive buffer. Otherwise the message is copied into the Nucleus-provided message queue.

Using receive_data

The **receive_data** call requests a message from a mailbox. Always specify a buffer of at least 128 bytes in the **receive_data** system call. You must create a receive buffer; you can create a segment or declare a data structure in a portion of the DS.

If the task calling **receive_data** is waiting at the mailbox when the message arrives, the message is copied directly to the waiting task's receive buffer. Otherwise the message is copied into the Nucleus-provided message queue.

When a task is waiting to receive data:

- When a message arrives at the mailbox, the data is copied from the send buffer into the task's receive buffer. It does not go into the message queue.
- If there is no message, the receiving task goes into the task queue. The task goes to sleep for the specified time limit or until a message arrives, whichever comes first. If the message arrives, it is copied from the send buffer to the receive buffer. If no message arrives during the time limit, the task will awaken with an E_TIME condition code.

When a message is waiting to be received:

- The receiving task receives the message without going into the task queue.
- If no task is waiting, the message goes into the message queue.

The amount of time necessary to receive a message can potentially be longer than the specified time limit. A time-out error will not occur after the message transmission into the receiver's segment begins. The transmission time is significant only for very long messages.

Mailbox System Calls

These are the system calls that relate directly to mailboxes:

add_reconfig_mailbox
create_mailbox
delete_mailbox
send_data
receive_data
send_message
receive_message

Table 3-1 lists common operations on mailboxes and the mailbox system calls that do the operations.

See also: Nucleus system calls, *System Call Reference*

Table 3-1. Mailbox System Calls

Operation	Description
create mailbox	**Create_mailbox** creates a new mailbox and returns a token for the mailbox.
specify reconfiguration mailbox	**Add_reconfig_mailbox** specifies an existing data mailbox as a reconfiguration mailbox (used with the watchdog timer).
delete mailbox	**Delete_mailbox** takes a token for a mailbox and deletes the mailbox.
send data	**Send_data** sends up to 128 bytes of data to a data mailbox.
receive data	**Receive_data** receives up to 128 bytes of data from a data mailbox.
send message	**Send_message** sends an object token to a mailbox.
receive message	**Receive_message** receives an object token from a mailbox.

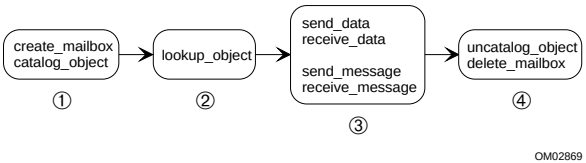
See also: Nucleus system calls, *System Call Reference*;
examples in the */rmx386/demo/c/intro* directory

How to Use Mailbox System Calls

These are the rules for mailboxes:

- A task can send a message to any mailbox for which it has a token.
- A mailbox can receive a message from any task that has its token.
- The size of a data message is limited to 128 bytes.

Figure 3-3 shows the order in which you make mailbox system calls.



1. Make these calls from a task in the job that needs to receive messages from the new mailbox.
2. Make this call from the task that needs to send information to the mailbox.
3. Make the **receive**_ calls from a task in the job that created the mailbox. Make the **send**_ calls from any task that has the mailbox token.
4. Make these calls from the task that created the mailbox.

Figure 3-3. Mailbox System Call Order

Use the **send_data** and **receive_data** system calls with data mailboxes. Use the **send_message** and **receive_message** with object mailboxes. If you try to pass information with the wrong system call, for example sending a object with **send_data**, the Nucleus issues an E_TYPE condition code.

❑❑❑

4 Semaphores

What is a Semaphore?

A semaphore is a counter that takes positive integer values called *units*. Tasks send units to and receive units from the semaphore. A semaphore can:

- Synchronize a task's actions with other tasks
- Provide mutual exclusion from data or a resource

See also: Semaphores, *Introducing the iRMX Operating Systems*;
Examples in the */rmx386/demo/c/intro* directory

Creating a Semaphore

These are the parameters you specify when you create a semaphore using **create_semaphore**:

- The initial number of units in the custody of the new semaphore.
- The maximum number of units the semaphore can have in custody at any given time. The lower limit is automatically 0.
- Whether the task queue is FIFO or priority based.

Task Queue

Use a priority-based queue so high-priority tasks do not wait behind lower-priority tasks in the queue. Within a priority-based queue, tasks of equal priority are FIFO queued.

See also: Priority Bottlenecks and Blocking, in this chapter

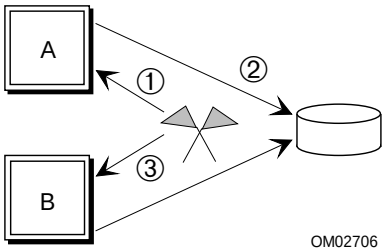
Deleting a Semaphore

When you use **delete_semaphore**, the Nucleus awakens any tasks waiting to receive units at the semaphore with an E_EXIST condition code.

Binary Semaphores and Mutual Exclusion

If a task asks a binary (single-unit) semaphore for a unit to gain access to a resource and a unit is not available, it means some other task is using the resource. The requesting task can't access the resource until the unit is returned.

Figure 4-1 illustrates a binary semaphore guarding a resource. Tasks queue up for access to the resource; in general, use a priority-based queue in your applications.



Create the semaphore with one initial unit and a maximum of one unit, using **create_semaphore**.

1. Task A requests a unit from the semaphore using **receive_units**. The semaphore sends the unit. Task B also requests a unit from the semaphore using **receive_units** and specifies it is willing to wait. Task B goes to sleep by waiting in the queue until Task A has returned the unit.
2. Task A accesses the resource. No other task can access the resource at the same time. When Task A is done using the resource, it returns the unit to the semaphore using **send_units**.
3. Now Task B will wake up and receive the unit. If Task B has a higher priority than A, it will begin running. Otherwise it will be ready.

Figure 4-1. Mutual Exclusion Using a Binary Semaphore

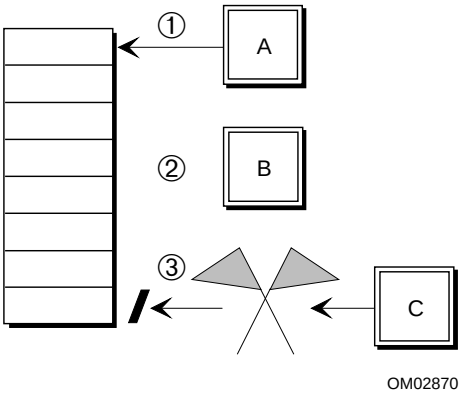
Priority Bottlenecks and Blocking

You may encounter several problems when you use semaphores for mutual exclusion of shared data. To eliminate the problems, use regions rather than semaphores to control shared resources.

See also: Regions, in this manual

The first bottleneck is a high priority ready task blocked by a lower priority running task. This occurs if the lower-priority task obtained the required units before the higher-priority task became ready. The running task, regardless of priority, controls the resource until it returns the units to the semaphore.

The second bottleneck, *priority inversion*, occurs when a low priority task obtains the required units to access a resource, then is preempted by a medium-priority task, which is then preempted by a high-priority task that needs to access the resource. Figure 4-2 shows what could happen:



1. Low priority Task A is running and obtains a unit from a binary semaphore to access some data. It starts accessing the data.
2. Task B, a medium priority task, preempts A.
3. Higher priority Task C preempts B, but cannot access the data while the low priority task holds the unit. The low priority Task A cannot complete its operation and return the unit because it is preempted by B.

Figure 4-2. Priority Inversion Bottleneck with Semaphores

The third bottleneck occurs when a task holding a semaphore unit and using shared data is suspended or deleted; no other task can gain access to the shared data. Only after the suspended task is resumed and returns the semaphore can the data be used by the other tasks. In the case of a deleted task, the semaphore prevents any other tasks from ever using the shared data.

Multi-unit Semaphores

You typically use a multi-unit semaphore as a counter, for example managing the available space in a circular buffer. A task can request more than one unit from a multi-unit semaphore and the semaphore tries to satisfy the request.

The semaphore either sends all the units requested or none at all. So, a multi-unit semaphore might have tasks waiting for units and also have units that have not been granted available, but not enough to satisfy the task at the head of the task queue. This is a possible scenario:

> Two tasks are queued waiting for units.
> Task A is first in the queue and wants three units.
> Task B is second in the queue and wants one unit.
> The semaphore has zero units available when the requests are made.

These are possible outcomes for a FIFO queue:

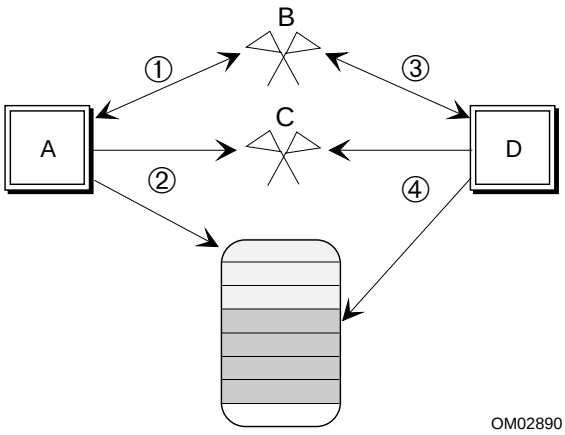
- The semaphore receives three units. Task A receives the units, awakens and runs while B remains asleep in the queue.
- The semaphore receives two units. Both tasks remain asleep. There aren't enough units for Task A and Task B's request cannot be satisfied, because Task A is still ahead of it in the queue.
- The semaphore receives four units. Both A and B receive their requested units and are awakened. Task A runs first because it is first in the queue.

These are possible outcomes for a priority queue, with Task B having a higher priority than A:

- The semaphore receives two or three units. Task B receives a unit, awakens and runs while A remains asleep in the queue.
- The semaphore receives four units. Both A and B receive their requested units and are awakened. Task B runs first because it is higher priority.

Figure 4-3 shows how tasks can share a fixed-length list of buffers using two semaphores: one binary and one multi-unit counting semaphore.

- The binary semaphore prevents two different tasks taking buffers from the list at the same time.
- The counting semaphore prevents a task spending time searching the list for an available buffer when there is none.



Create a binary semaphore B that provides mutually-exclusive access to the buffer list using **create_semaphore**.

Create a counting semaphore C that tracks the number of available buffers, eight in this example, using **create_semaphore**. Set the initial units and maximum units equal to the number of buffers: eight.

1. Task A requests the only unit from semaphore B using **receive_units**. The semaphore sends the unit. Now, only task A can request units from semaphore C.
2. Task A requests three units from semaphore C using **receive_units**. The semaphore sends the units. Now task A has access to three buffers in the shared list. Task A then returns the unit to semaphore B using **send_units**.
3. Task D requests the unit from semaphore B using **receive_units** and receives it.
4. Task D can now request four units from semaphore C. Since the semaphore has enough remaining units to satisfy the request, Task D will receive them. If it had not, D would have waited.

All tasks should return their units to C as soon as possible to free resources for other tasks.

Figure 4-3. Multi-unit and Binary Semaphores Allocating Buffers

Using send_units

A task does not have to receive a unit from a semaphore in order to send a unit to it.

When a task sends units to a semaphore, and no task of equal or higher priority is waiting, the task remains running. If a higher priority task is waiting for the unit, it preempts the lower priority task.

The semaphore returns an E_LIMIT condition code when:

- You try to send zero units.
- You try to send more units than the maximum number of units the semaphore is allowed to have. In this case, the number of units in the custody of the semaphore remains unchanged.

Using receive_units

Use **receive_units** to find out how many units are available by specifying 0 in the `units` parameter.

You can specify how long a task using **receive_units** will wait for a semaphore unit. Two factors determine whether the task receives the units and how soon: how many units the task asks for, and where the task is in the queue.

- If the number of units requested is within the semaphore's current supply of units and the specified maximum for that semaphore, the request is valid.
  - If the task is at the front of the queue, the request is granted immediately, and the task stays running.
  - If a request is valid but cannot be granted immediately, the task can either wait or not.

    If the **receive_units** call specifies that it can wait, the task goes into the task queue and goes to sleep by waiting. If the time elapses before the task gets the units it asked for, the task awakens and receives an E_TIME condition code.

    If the **receive_units** call specifies that the task cannot wait, the task receives an E_TIME condition code.
- If the task asks for more units than the maximum number allowed for a particular semaphore, the request is invalid and the semaphore returns an E_LIMIT condition code.

Semaphore System Calls

These are the system calls that relate directly to semaphores:

create_semaphore
delete_semaphore
send_units
receive_units

Table 4-1 lists common operations on semaphores and the semaphore system calls that do the operations.

Table 4-1. Semaphore System Calls

Operation	Description
create	**Create_semaphore** creates a new semaphore and returns a token for it.
delete	**Delete_semaphore** deletes the semaphore.
send units to semaphore	**Send_units** gives a specified number of units to a semaphore.
receive units from semaphore	Request units from a semaphore with the **receive_units** system call.

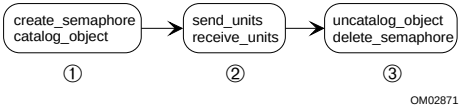
See also: Nucleus system calls, *System Call Reference*

How to Use Semaphore System Calls

These are the rules for semaphores:

- A task does not have to receive a unit from a semaphore in order to send a unit to it.
- A semaphore cannot receive more units than the maximum specified when it was created.

Figure 4-4 shows the order in which you make semaphore system calls.



1. Make these calls from the task that has the resource that needs to be shared.
2. Make these calls from the tasks that need to use the resource.
3. Make these calls from the task that created the semaphore.

Figure 4-4. Semaphore System Call Order

❑❑❑

Regions 5

What is a Region?

A region is a binary semaphore with special suspension, deletion, and priority-adjustment features. Regions provide mutual exclusion from resources; only one task may control a region at a time; only the task in control of the region can access the resource.

Deletion and Suspension Protection

Tasks that have control of a region, or are queued at a region, cannot be deleted or suspended by other tasks until they give up control of the region.

Tasks in control of a region cannot be preempted by other tasks wanting control of the region. A task can, however, be preempted by a higher-priority task that does not want control of the region.

Priority Adjustment

If you use a priority-based queue, the priority of the task controlling the region will be dynamically raised whenever the task at the head of the region's task queue has a priority higher than that of the controlling task. The priority of the controlling task is raised to match that of the queued task. This priority adjustment prevents the priority inversion bottleneck that can occur when tasks use semaphores to obtain mutual exclusion.

Once a task's priority is raised in this way, the priority is not lowered until the task gives up control of all regions. It is not sufficient to give up control of the region that raised the priority, if the task still controls another region.

Creating a Region

The only parameter you specify when you create a region using **create_region** is whether the task queue is FIFO or priority based.

Task Queue

Tasks of equal priority in a priority-based queue are queued in a FIFO manner.

A task in the region's task queue sleeps until the region becomes available; it can wait indefinitely.

Deleting a Region

When you delete a region using **delete_region**, the Nucleus awakens any tasks waiting for control of a region with an E_EXIST condition code.

A task cannot delete a region it controls. It must give up control of the region first. Otherwise, an E_CONTEXT condition code returns.

Misusing Regions

Misuse of regions can corrupt the interaction between tasks in an application system. Before writing a program using regions, you must have a complete understanding of regions, the OS, and the entire application system. Avoid these problems:

Deadlock
: This occurs if two tasks need control of two regions for access to the same two resources at the same time and each task has control of one region.

  Since there is no time limit on waiting for control, deadlocked tasks can remain so indefinitely. Any other tasks entering the region's task queue will also become deadlocked.

Deletion immunity
: If you create and a task obtains control of the region, the task will be immune to deletion until it gives up control of the region. If the task never gives up control, it can never be deleted.

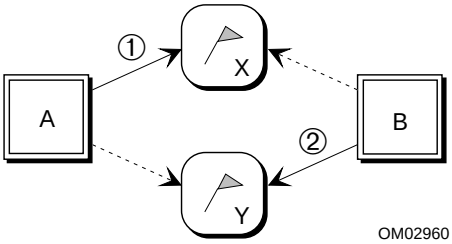
No time limit
: If control is not immediately available, there are two options. If the task cannot wait, it receives a condition code. If the task waits, it may never run again. If these are not acceptable, use a semaphore instead.

See also: Semaphores, in this manual

Nesting Regions

A task can take control of more than one region at a time, which is called *nesting regions*. Regions are released in a last-obtained, first-released order. When a task releases control of a region and has control of multiple regions, the most recently obtained region is released first.

Deadlock occurs with multiple nested regions as shown in Figure 5-1. The example uses the **receive_control** system call to gain control of the regions.



1. Task A requests and obtains control of Region X. It also needs control of Region Y.
2. Task B preempts Task A. It requests and obtains control of Region Y. It also needs control of Region X.

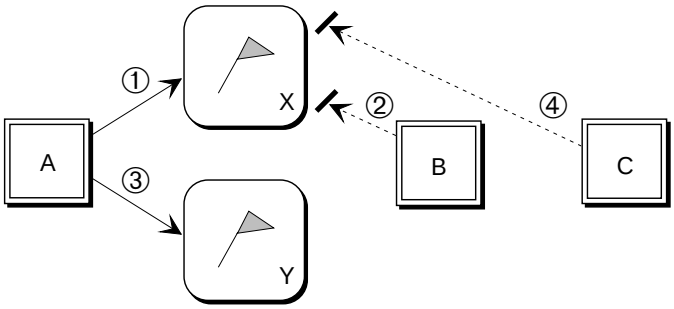
Neither task can run. Neither task can be deleted. If any other tasks try to obtain control of either region, they will also become deadlocked.

Figure 5-1. Deadlock and Nested Regions

Prevention

There are two ways to prevent deadlock in nested regions:

- Use the **accept_control** system call. Tasks using **accept_control** cannot deadlock at a region unless they keep trying endlessly to accept control.
- If you use **receive_control**, have all tasks request control in a consistent order; it doesn't matter what order as long as all tasks obey it. List the names of all regions in any order and label them in sequential order. As you program a task that nests any of the regions, be sure the task requests control in ascending order and releases the regions in descending order. If you follow this rule consistently, you can safely nest regions to any depth. Figure 5-2 on page 70 shows how sequential ordering works.



1. Task A, priority 140, requests and obtains control of region X. It also needs control of Region Y.
2. Task B, priority 135, preempts Task A. It requests control of region X. Task A's priority is raised to equal B's. Task B can't obtain control so it enters the task queue.
3 Task A requests and obtains control of region Y.
4. Task C, priority 130, preempts Task A. It requests control of region X. Task A's priority is raised to equal C's. Task C can't obtain control so it enters the task queue.

Task A runs and then releases region Y, followed by region X. Then, its priority is adjusted to its static level, 140. Task C will then wake up, preempt A, and obtain control of both regions.

Figure 5-2. Preventing Deadlock in Nested Regions

If a task has control of several regions, and multiple tasks with different priorities are waiting for the regions, the priority of the controlling task may be raised more than once. But the controlling task must surrender control of all the regions it controls before its priority reverts to its original static value.

Using receive_control

The **receive_control** system call enables a task to wait for a region to become available. But if access never becomes available, the task never runs again. An error occurs if a task requests control of a region it already controls.

Using accept_control

If control is not immediately available, the task does not wait at the region. Instead, it receives a condition code and remains ready. To gain control, the task must make repeated calls to **accept_control**.

Region System Calls

These are the system calls related to regions.

create_region
delete_region
receive_control
send_control
accept_control

Table 5-1 lists common operations on regions and the region system calls that do the operations.

Table 5-1. Region System Calls

Operation	Description
create region	**Create_region** creates a new region and returns a token for it.
delete region	**Delete_region** takes a token for a region and deletes the region.
get control of region	**Receive_control** gives a task control of a region when it becomes available. The task sleeps in the task queue until control is granted.
give up control	**Send_control** informs the Nucleus that the calling task is giving up control of the last region it controlled. A different task can then be given access to the shared data.
get control immediately	**Accept_control** allows a task to gain access to shared data when access is immediately available.

See also: Nucleus system calls, *System Call Reference*;
examples in the */rmx386/demo/c/intro* directory

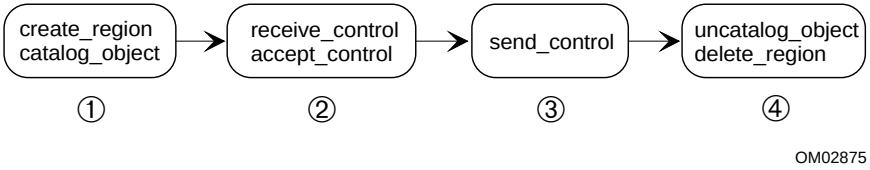
How to Use Region System Calls

These are the rules for regions:

- Do not let a task suspend itself when it controls a region. Unless the task is resumed by another task, the region may permanently exclude other tasks from a shared resource.

  In addition, the task will never run again and its memory will not be returned to the memory pool. Tasks in the region's task queue are also immune to deletion and will encounter the same memory pool problems.
- Do not use regions in Human Interface applications. If a task in an HI application uses regions, the application cannot be stopped asynchronously (using <Ctrl-C> entered at a terminal) while the task is accessing data guarded by the region.
- When the running task no longer needs control, it should release control of the region, which enables a waiting task to access the resource.
- A task cannot delete a region it controls without first releasing the region.
- Use an arbitrary order for all tasks accessing regions when you use nested regions and the **receive_control** system call.

Figure 5-3 shows the order in which you make region system calls.



1. Make these calls from the task that has the resource that needs to be shared.
2. Make these calls from the tasks that need to obtain control of the region to access the resource.
3. Make this call to give up control of the region.
4. Make these calls from the task that created the region.

Figure 5-3. Region System Call Order

□□□

Ports 6

What is a Port?

Ports were initially implemented for Multibus II systems as an access point to the Nucleus Communication Service, but they have been extended to be an access point to any service. A task can pass messages to a service through its port. If the service connects different tasks then another task can receive that message.

When you use ports, the sending task sends the message through its port and the receiving task receives the message through its port. You can create a buffer pool or heap and attach it to the port to provide fast storage allocation for messages received at a port.

A message consists of a control part and optionally a data part. The control part contains data which usually definies what to do with the data part (if any). Control messages do not require special buffer arrangements (such as pools or heaps) but data messages do.

What is a Service?

A service is a module which processes messages from ports. Services can range from complex subsystems, such as the Nucleus Communication Service, or the IP module of the TCP/IP stack, to simple hardware interfaces, such as a low-level serial device driver. All services process messages received via ports from tasks, and they also process messages from an *interface*, which may be a hardware interface or another object, such as a mailbox or evan another port.

For example, the serial driver service manages the hardware interface using interrupts, and processes messages from tasks using the service. The Nucleus Commumication Service processes messages from tasks using the system, and also messages to and from the Multibus II interface.

Ports in Multibus II Systems

For two tasks on different boards to exchange messages, each must have access to a Nucleus Communication Service port on its own board. Each port is an access point to the message-passing protocol of the Multibus II Transport Protocol Specification.

If you use a port to communicate with a board that is running another OS, the other OS must also support the Multibus II Transport Protocol Specification.

Why Use a Port?

These are the advantages of using ports:

Variable message size
: Each message passed to a port may be a different size, from a few bytes in a control message, to include data messages of many Megabytes. Storage is allocated per message, and the message size does not have to be pre-determined.

Linking request to response
: A client task sending a message can specify a response buffer so it can receive a response tied specifically to the request. This corresponds to the client-server model of task interactions used by the Nucleus.

Providing current status
: Each message that a task receives includes status information about whether an exceptional condition prevented successful transmission (for example, a Multibus II transmission error, or insufficient buffer space at the server). *Transaction IDs* bind status messages to data message transmissions.

Short-circuit message passing
: This feature can be used where messages are passed from task to task on the same host. The message is copied directly from the sending task to the receive buffer of the receiving task. Because the interface is gerneally the same for communicating on the same processor or between processors, applications that use ports can migrate from a single host to a system with multiple hosts.

Using Heaps and Buffer Pools at Ports

Most incoming messages to a server require that you create a heap or buffer pool and attach it to the port. When a message arrives at a port, a buffer is automatically allocated from the attached object to receive the data. The receiving task can access this buffer directly, and return it to the pool or heap when it has finished with the message.

In the case of a buffer pool, depending on the message size and the buffer pool, an incoming message may be copied into a single buffer or into a series of buffers called a *data chain*. The Nucleus gives the receiving task a pointer either to a single buffer or to a data chain block that holds pointers to all buffers in the chain.

A heap or buffer pool can be attached and detached during the existence of the port.

See also: Buffer pools and heaps, in this manual

Creating a Port

These are the parameters you specify when you use the **create_port** system call.

- The name of the service to be used.
- The port's ID. In general some of the port IDs are reserved by the service. The Nucleus can assign the port ID for you if you specify a special null value (usually zero). You can create a port with no ID, called an *unbound* port.
- The number of simultaneous transactions allowed at this port.
- Whether the task queue is FIFO or priority based.
- Whether fragmentation is enabled or disabled.
- Whether the port is to have an ID assigned to it (in other words, whether the port is to be *bound*).

Fragments in Large Data Messages

When a message is delivered, the receiving port must supply storage. Whether the data is transmitted in one piece depends on the buffering capacity of the port.

The Nucleus can break up the data portion of messages that are too large to be delivered in one piece into smaller pieces, if the service supports fragmentation, and the port has enabled this feature. The Nucleus Communication Service (NCS) will send the message in fragments when the receiving buffers are too small to receive the entire message. Each of the fragments specifies the same transaction ID.

The NCS on the server delivers a single reply to the client task. The client task receives a condition code only if fragmentation fails.

The NCS expects that the server in a client-server transaction will control fragmentation whether it occurred in a request from a client or a response to a client.

If fragmentation is disabled, the sending task will receive a condition code when there is not enough buffer space at the receiving end.

Deleting a Port

When you delete a port using **delete_port**, the Nucleus deletes all messages queued at the port and cancels all outstanding transactions for the port. The Nucleus deletes message buffers allocated from the port's buffer pool and awakens any task waiting for a message at the port with an E_EXIST condition code. If you delete a sink port (described later) that is attached to one or more ordinary ports, the Nucleus detaches them.

Identifying a Port

In a multiprocessor system, each port must be uniquely identified. The paragraphs below describe the ways a port is identified.

Identifier	How It Is Used
genaddr	A genaddr (general address) is a structure which combines the port ID with the service interface address (defined by the service). This uniquely defines a port across all the hosts accessed by the service. For example a port at an Ethernet service will have a port ID and the 48-bit MAC address defined by the Ethernet specification.

An address is defined as follows:

```
DECLARE GENADDR STRUCTURE (
     port_id       WORD_16,
     address_len     BYTE,
     unused          BYTE,
     address         BYTE(*));
```

or in C,

```
typedef struct genaddr {
     unsigned short  port_id;
     unsigned char          address_len;
     unsigned char          unused;
     unsigned char          address[];
} GENADDR;
```

Socket — (NCS only) A socket is a 32-bit number that combines the host ID and port ID. It is special to the Nucleus Communication Service and is retained for legacy reasons.

These lines of code define a socket:

```
DECLARE SOCKET  STRUCTURE(
    host_id       WORD_16,
    port_id       WORD_16);
```

or in C,

```
typedef struct socket_struct {
         unsigned short       host_id;
         unsigned short       port_id;
} SOCKET_STRUCT;
```

Port ID	The port ID identifies the port among all those on a given processor board in a given service. In a multiprocessor system, more than one port can have the same port ID as long as the ports reside on separate processor boards. Any board that supports the Multibus II Transport Protocol will specify port IDs for its ports, whether running the iRMX OS or another OS. This lets boards communicate with each other regardless of the OS being used.
Host ID	The host ID is a logical address for the host, a number in the range 1 to 254 that uniquely identifies the host. It is a 16-bit value, usually equal to the slot number in a Multibus II backplane.
Address len	Defines the number of bytes in the address which are valid. Usually fixed for a given service. Adresses of up to 28 bytes are supported by iRMX.
Address	An array of bytes defining the interface address for a given service.
Token	The token is specific to the iRMX OS. iRMX tasks use the token to catalog the port in the object directory, attach the port to a sink port, or delete the port.

See also: *Multibus II Transport Protocol Specification and Designer's Guide*

Sending Data Messages

These messages follow the mailbox model. The client uses the **send** call to send the message. The server uses the **receive** call to receive the data at its port. The client does not expect a response; the server doesn't send one.

Using send

Use **send** to send data from a client to a server without expecting a reply. You must specify a valid pointer to some control information, even if your application doesn't use the information. You can optionally provide a pointer and a length for a data component. If you do, specify if the data component can be in a single segment or in a data chain.

You can specify that the message transmission be synchronous or asynchronous: if synchronous, **send** does not return until the message has been sent; if asynchronous, the system call returns immediately, letting the task continue processing while the message is being sent. A status message will be delivered to the port later if the transmission fails. Optionally a service will send a status message on the completion of every asynchromous transmission.

Using receive

Use **receive** for servers to receive messages from clients. **Receive** requires that you have created a buffer pool using **create_buffer_pool**, released buffers (segments you have created) to it using **release_buffer**, and have attached it using **attach_buffer_pool. Receive** returns a pointer to the data component of the message if there is a data component.

You must identify the receiving port. You must specify how long the task will wait for the message at the port. The calling task goes to sleep by waiting for a message at a port. If no message arrives before the specified time limit expires, the task will awaken with an E_TIME condition code.

You must supply a pointer to a structure that **receive** fills with information about the transmission.

Sending Request / Response Messages

Transaction pairs provide a client-server communication model. Clients send request messages to servers and servers send responses back. This model includes the ability to:

- Use either a control or control/data format for messages
- Identify message pairs as transactions

Control and Control / Data Format

The iRMX implementation of the Multibus II Transport Protocol Specification defines two kinds of message format: control and control/data.

Control message	A short, unsolicited message conveying control information; you do not have to create a buffer pool in the receiving task prior to sending a control message. The control message can contain application-specific control information. Control messages are delivered faster than control/data messages.
Control/data message	A message with a control portion and a data portion. Usually, you have to create a buffer pool in the receiving task prior to sending a control/data message (the exception is a client receiving a response from a server).

Although control and data portions are combined when sent in a control/data message transaction, they are stored differently at the receiving port.

- The receiving port's message queue provides storage for the control portions of incoming messages.
- The receiving port stores data portions of a message differently depending on whether they are part of a client request or of a server response.
  - For request messages, the Nucleus allocates storage from a buffer pool you attach to the server's receiving port. The Nucleus rejects data portions sent to a port without a pool or with insufficient pool resources.
  - For response messages from servers to clients, the data portions are delivered to a specific response buffer supplied by the client task; the response buffer cannot be a data chain.

Transaction Pairs

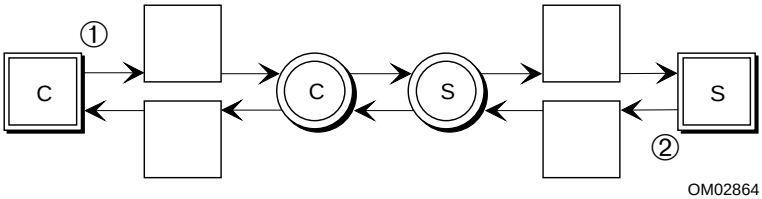
The Nucleus uses *transaction ID* numbers to match responses to requests.

The request is delivered to the server's port, where the control portion of the message is copied into a *control message queue*, and the data is transferred into space allocated from the buffer pool you created. The server acts on the request and prepares a response that may include control and data information. The NCS uses the response buffer supplied by the client as the destination for the data portion of the response message.

The client supplies a pointer to a response buffer if it expects a data message in response. The client task allocates response buffer space based on memory in the client task's job; the response buffer is not allocated from the client port's buffer pool.

Basic Request / Response Transactions

The client sends a control/data message that tells the server to perform a service on the data (for example, write it out to disk). The data component in this example is 1 Kbyte long. All buffers in the example are 1 Kbyte. Figure 6-1 shows this transaction.



1. The client Task C calls the **send_rsvp** system call. Then the client calls **receive_reply** and goes to sleep by waiting for the response.

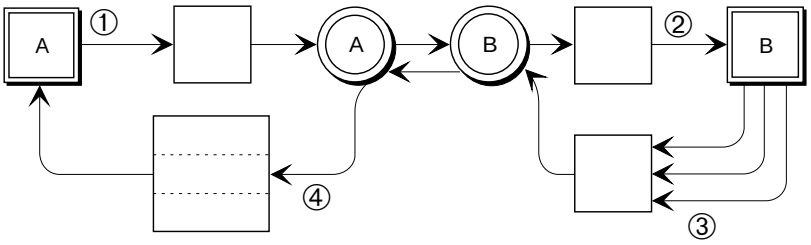
   The server Task S has previously called **receive** and is waiting for the message. The message goes through Port C and arrives at the server's port, Port S. Because the receive buffer in the port's buffer pool is large enough, the message is delivered. The server Task S receives the message and begins processing the data. The server also receives a transaction ID that the Nucleus uses to match the server's response to this request.

2. When it is done processing, Task S calls **send_reply** to send a reply to the client; the server includes the transaction ID supplied in the original request. Task C receives the reply in its response buffer and, having previously called **receive_reply**, wakes up; the transaction is complete.

Figure 6-1. Basic Request / Response Using Ports

Fragmented Response Transactions

Figure 6-2 shows a fragmented response. Three buffers in the example are 1 Kbyte; the request is for 3 Kbytes and the response buffer is 3 Kbytes.



1. The client Task A uses **send_rsvp** to send a request to read 3 Kbytes from a disk to the server, Task B. Task A passes a pointer to a 3 Kbyte response buffer so it can receive the entire response in one block. The client calls **receive_reply** and goes to sleep by waiting for the response.

2. The server, Task B, receives the request message, having previously called **receive**, and initiates the service.

3. This server has a 1 Kbyte buffer limit, so the server cannot send the entire 3 Kbyte response message in one operation. The server fragments the response message, repeatedly calling the **send_reply** system call with 1 Kbyte fragments until the entire message is sent.

   Send_reply parameters include an EOT (end-of-transaction) indicator. As long as the server is sending fragments, it sets the EOT field to FALSE so the transaction remains open. When it sends the last fragment, the server sets EOT to TRUE.

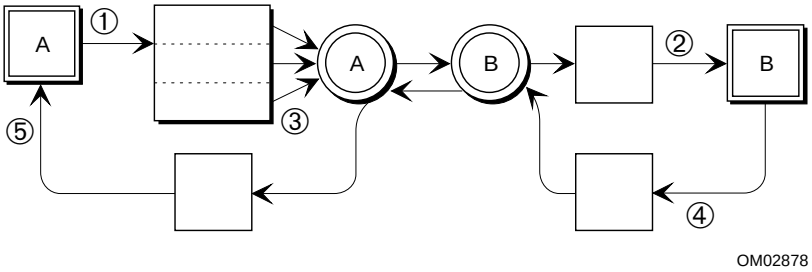
4. Only the control part of the last fragment, EOT=TRUE, is sent to the receiver.

   At that point, the client, Task A, will awaken from the **receive_reply** system call with 3 Kbytes of data in the response buffer, and the transaction ends.

Figure 6-2. Fragmented Response Using Ports

Fragmented Request Transactions

In Figure 6-3, the client sends a control/data message to the server that includes 3 Kbytes of data. This server can receive data only in 1 Kbyte blocks, so the message must be fragmented before it can be received. The client will not be aware that this transmission was fragmented.



1. The client, Task A, calls the **send_rsvp** system call. The client calls **receive_reply** and goes to sleep by waiting for the response.
2. When the server, Task B, tries to receive the incoming data message using **receive**, it will receive an E_NO_LOCAL_BUFFER status message. The message includes the length of the data message.
3. A loop in the server task calls the **receive_fragment** system call three times to receive the 3 Kbyte data message in 1 Kbyte fragments.
4. When the server has called **receive_fragment** often enough to receive the entire message, it calls **send_reply** to send a response to the client. The transaction ID matches the response to the request.
5. The client, Task A, awakens from the **receive_reply** system call and the transaction ends.

Figure 6-3. Fragmented Request, Example

Using send_rsvp

Use **send_rsvp** for a client to send a request to a server, expecting a response. You must identify the receiving port. You must specify a valid pointer to some control information; you can optionally provide a pointer and a length for a data component. If you do, specify if the data component can be in a single segment or in a data chain.

You must specify whether to use the **receive** or **receive_reply** system call for receiving the response from the server. Use **receive** when a client task initiates multiple transactions from a task; you might also have a separate task use **receive** to pick up the responses.

Specify the size of the client's response buffer and a pointer to it. The response buffer cannot be a data chain; it must be a contiguous block.

Specify whether to use synchronous or asynchronous transmission. If you specify synchronous when you call **send_rsvp**, the client task will wait until all the fragments have been received. Otherwise, the client will go on with other processing.

Using receive_fragment

Use **receive_fragment** for servers to receive messages from clients when insufficient buffer space is available to receive a message in one piece. If the **receive** status code is E_NO_LOCAL_BUFFER, you need to code a loop that makes calls to **receive_fragment** to receive fragments.

The `info_ptr` structure of the **receive** call will contain the length of the data message received, the transaction ID, and the sending socket. When you use **receive_fragment**, you specify the size of fragments according to the how the server's buffer pool is set up. You are responsible for determining how many times to call **receive_fragment**, based on the size of the message and the size of the buffers available for receiving the fragments.

Using send_reply

Use **send_reply** for a server to send a response to a client. The message goes to the response buffer supplied by the client, not its heap or buffer pool.

You must supply the transaction ID; this is the `trans_id` parameter in the **send_rsvp** system being answered.

You must identify the receiving port. You must specify a valid pointer to some control information; you can optionally provide a pointer and a length for a data component. If you do, specify whether the data component can be in a single segment or in a data chain.

You must specify whether the transmission is synchronous or asynchronous. If the reply is fragmented, you must specify whether this is the last fragment.

Using receive_reply

Receive_reply enables a client to wait for a response from a server. The data component is received in the response buffer originally specified in the **send_rsvp** call.

You must supply the same `port_token` and `rsvp_trans_id` parameters you used in the **send_rsvp** call. You must also specify how long the client task will wait for the message at the port. The client task goes to sleep by waiting for a message at a port. If no message becomes available before the specified time limit expires, the task will awaken with an E_TIME condition code.

Using broadcast

This call is commonly used by Multibus II hosts to broadcast status information system-wide to dedicated ports with the same port ID on each host in a system. You can dedicate a task on each host to wait for messages at the agreed-upon port.

You can also use this call to locate servers in the system. The clients can send a broadcast message to the server port ID. The server sends a message back to the sender, and the client obtains the server's host ID from the message.

In general, indicate to a service that you want to broadcast by setting the BROADCAST flag in the rqe_send call. The broadcast call is specific to the Nucleus Communication Service.

Using cancel

You can cancel **send** or **send_reply** control/data messages, but not control-only messages. You can also cancel a **send_rsvp** message, which disassociates the response buffer from the source port. You specify the transaction ID and the port ID for the operation to be canceled. **Cancel** is a local operation only, affecting only the specified port. It does not notify the remote socket involved in the transaction.

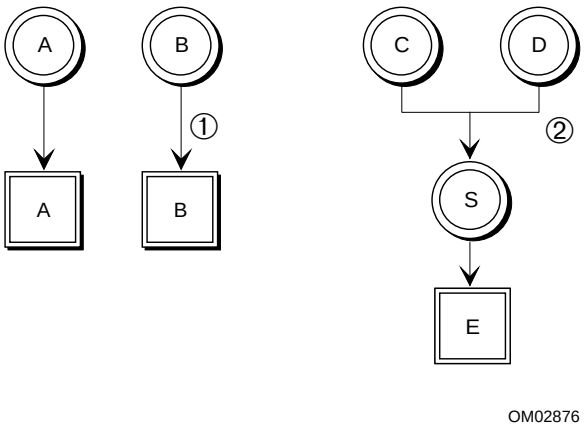
Setting Up Special Ports

This section describes attaching and detaching a sink port to ordinary ports and connecting a port to a default remote socket.

Forwarding Messages from Sink Ports

Message forwarding allows messages from several connected ports to be received by a single task waiting at a sink port. Sink ports help avoid duplication of code in several tasks. All messages come to the *sink port*, which forwards them to the appropriate task. The sink port must be on the same host as the receiving task. The sink port must be of the same service as all of its forwarding ports, or it must be an *anonymous* sync port, created by specifying a null service name in rqe_create_port.

The example in Figure 6-4 is an I/O Server that receives messages at two different sockets and accesses a single hard disk to fill the requests. The example illustrates using only ordinary ports (on the left) and using a sink port (on the right).



1. Ports A and B receive a request and passes data to the receiving Tasks A and B. A and B both write data to disk, so there is duplication of code.

2. Ports C and D are forwarded to a single sink port. The sink port determines which of the ports forwarded the message to it (so it can send a reply) and sends the data to Task E. Task E handles requests from both ports C and D.

Figure 6-4. Forwarding Messages Using Ports

Using attach_port and detach_port

The **attach_port** system call enables you to attach ordinary ports to a sink port. An ordinary port can be attached only to one sink port.

After you attach a port to a sink port, all subsequent messages to the ordinary port are forwarded to the sink port. Messages that were queued at the ordinary port at the time of the attachment remain queued at the ordinary port and are not forwarded, so you must ensure that the queue is empty before attaching the sink port. A task that was queued to receive a message at an ordinary port with an empty message queue will remain in the task queue until it times out or until the sink port is detached and a message arrives at the ordinary port.

Only a single level of forwarding is supported; a sink port may not be attached to another sink port.

When you detach a sink port using **detach_port,** subsequent messages to the ordinary port will not be forwarded to the sink port. Messages previously forwarded to the sink port remain queued at the sink port until they are removed with a receive operation or the port is deleted.

Using connect

You can use **connect** to connect a port on the host to a *default remote address*, which you specify, so that messages sent from the host port are automatically routed to that particular address on the remote host. While the connection exists, the port on the client can only receive messages from the specified socket. The connection is active when you specify a default remote socket with the **connect** system call

To disconnect the default remote socket, specify a 0 for the default remote socket with the **connect** system call. Once disconnected, the port remains disconnected until specifically connected again. A port can be connected to a remote socket more than once, with the most recent connection overriding all previous connections.

Port System Calls

Operations on ports fall into two broad categories: setup and message passing.

These are the system calls that relate directly to ports.

Setup calls	**create_port** **delete_port** **connect** **attach_port** **detach_port** **get_port_attributes**
Message-passing calls	**send** **send_rsvp** **send_reply** **receive** **receive_reply** **receive_fragment** **cancel**

Table 6-1 describes operations on a port and what the related system calls are.

Table 6-1. Port System Calls

Operation	Description
create port	**Create_port** creates a new port and returns a token for the port.
delete port	**Delete_port** deletes the port.
connect port	**Connect** connects a specified port with a specified socket.
attach sink	**Attach_port** attaches a specified port to a specified sink port.
detach sink	**Detach_port** detaches the specified port from its sink port.
get port attributes	**Get_port_attributes** fills in a data structure containing the specified port's attributes. Supply a pointer to a port_attrib structure.
send, no reply	**Send** sends a message and returns a transaction ID.
receive	**Receive** receives a message at a specified port.
receive message in fragments	**Receive_fragment** receives a fragment of a request message. It is typically used by a server when insufficient buffer space is available to receive a message in one piece.

continued

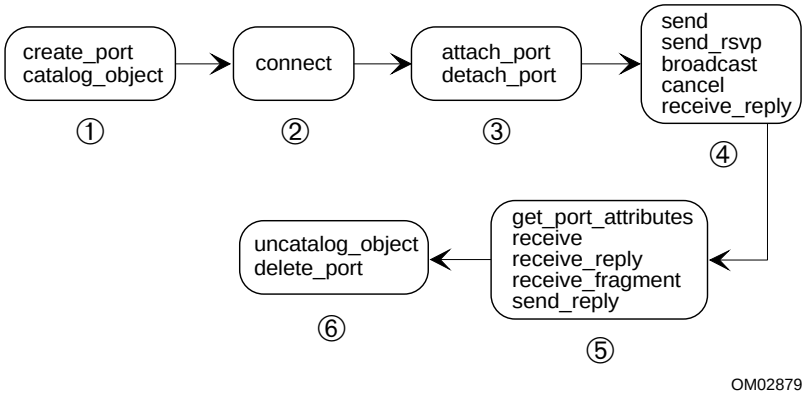
Table 6-1. Port System Calls (continued)

Operation	Description
send, expect reply	**Send_rsvp** sends a message from a client to a server with an implied request for a response from the server.
send response	**Send_reply** sends a reply from a server to a client in response to an earlier **send_rsvp** message.
receive response	**Receive_reply** call receives a reply to an earlier **send_rsvp** message.
broadcast a message	**Broadcast** sends a message from a specified port to a specified socket on every host processor in the system. **Broadcast** ignores the host ID, so the call effectively sends a message to the specified port ID on every host.
cancel message	**Cancel** cancels synchronous or asynchronous **send_rsvp** messages.

See also: Nucleus system calls, *System Call Reference*;
examples in the */rmx386/demo/c/intro* directory

How to Use Port System Calls

Figure 6-5 shows the order in which you make port system calls.



1. Make these calls from the client or sending task.
2. Make this call from the client or sending task to connect to a default remote socket.
3. Make these calls to attach an ordinary port to a sink port. The sink port and ordinary ports must reside on the same host.
4. Make these calls from the client or sending task.
5. Make these calls from the receiving or server task.
6. Make these calls from the task that created the port.

Figure 6-5. Port System Call Order

❑❑❑

Memory Pools, Memory Segments, Heaps, and Buffer Pools 7

Tasks satisfy their memory needs by using Nucleus system calls to allocate and deallocate memory. Memory includes:

- Memory pools, which control memory allocation and management in the iRMX OS. Memory pools are maintained by the Nucleus.
- Memory segments, which are the fundamental building blocks of the OS; they are maintained by your application.
- Buffer pools, which provide a way to allocate a set of segments so they will be available quickly and dependably during time-sensitive operations; after you create them, buffer pools are maintained by the Nucleus.

Flat Memory Models

The flat memory model is a 32-bit memory model where an application runs entirely in a single segment. Memory management differs between flat memory model applications and 32-bit segmented memory models. This chapter focuses on the 32-bit segmented memory model.

See also: Using the Flat Memory Model, *Programming Techniques* for information on the flat memory model, the paging subsystem that supports this model, and the system calls used in managing memory.

What is a Memory Pool?

The iRMX OS allocates a contiguous block of memory to a job from free space memory; you specify the minimum and maximum size. Each job has one memory pool, which is the source of memory for objects created within the job. When you create the job, the Nucleus creates a minimum size memory pool by allocating memory from the parent memory pool. There is a tree-structured hierarchy of memory pools, identical in structure to the hierarchy of jobs.

Memory that a job subsequently borrows from its parent remains in the pool of the parent but is temporarily allocated to the child. Until the child job releases the borrowed memory, it is only available to tasks in the child job, not to tasks in the parent job.

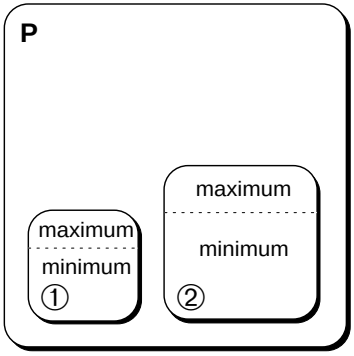
A memory pool for a job does not have a token. You cannot refer to a memory pool explicitly or manipulate it like an object.

Creating a Memory Pool

You create a memory pool when you use the **rqe_create_job** call to create a job. Two parameters of the **rqe_create_job** system call, `pool_min` and `pool_max`, set the size range. The upper limit of both `pool_min` and `pool_max` is 4 Gbytes. The job begins with the specified minimum amount of memory, and it can borrow memory from the parent memory pool up to the specified maximum size. You delete a memory pool by deleting the job.

Initially, a job's memory pool is a physically contiguous block equal to the specified pool minimum. If the job borrows memory from its parent job, the borrowed memory is also a contiguous memory block, but not contiguous to the initial memory pool. The maximum amount of memory that a job may borrow is equal to pool_max - pool_min. It is possible that a memory request in a pool can fail even if the pool has not reached its specified maximum limit.

Figure 7-1 shows two jobs that have been allocated from the same parent memory pool.



P is the parent pool. Its size is 512 K.

1. Job 1 has a minimum of 200 K and a maximum of 300 K.
2. Job 2 has a minimum of 200 K and a maximum of 350 K.

Since the total minimum size for the jobs is 400 K, both jobs can be created. Since the total maximum size is 650 K, the pools will not be able to reach their maximum sizes simultaneously.

Figure 7-1. Consequences of Minimum-Maximum Memory Pool Values

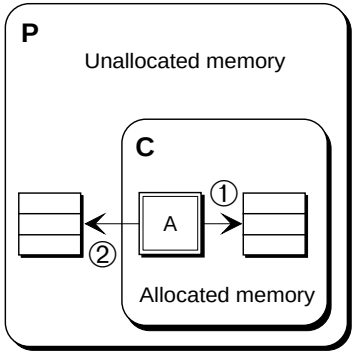
Allocating Memory

Memory in a job is unallocated unless it has been requested by tasks in the job or is on loan to a child job. A request for memory is explicit when you call the **create_segment** system call and implicit when you create any other object.

The amount of memory actually allocated to objects is between 18 and 33 bytes longer than the specified size. These extra bytes are for internal use by the Nucleus. However, each returned selector points to the first address available to the task.

Borrowing Memory

When you try to create a segment or other object, and the unallocated part of the job's pool is too small to satisfy the request, the Nucleus tries to borrow more memory, up to the pool's specified maximum, from the job's parent and on up the job hierarchy if necessary. Figure 7-2 illustrates borrowing memory.



1. Task A creates a segment object using **create_segment**. The memory is available from Job C. When Task A no longer needs this segment, it should delete it using **delete_segment**. The memory returns to Job C's pool.

2. Task A creates another segment. This time, the memory is not available in Job C's pool, so it is borrowed from the parent job's pool, P. When Task A no longer needs this segment, it should delete it using **delete_segment**. The memory returns to pool P.

When Job C is deleted, the memory in its pool becomes unallocated, and it is available to the parent job.

Figure 7-2. Borrowing Memory From the Parent Job

Borrowing increases the pool size of the job that is doing the borrowing and is restricted to the job's maximum. If a job has equal pool minimum and maximum attributes, its pool is fixed at that common value, and the job cannot borrow memory from its parent.

Using rqe_get_pool_attrib

You can determine the source pool for a job by getting the attributes of the job's memory pool using **rqe_get_pool_attrib**. Supply a pointer to the `attrib_ptr` data structure when calling **rqe_get_pool_attrib**, and the system call fills in the fields of the structure with the pool's attributes.

Pool attributes include: minimum and maximum allowable pool size, initial pool minimum size, number of allocated paragraphs of memory, number of available paragraphs (not including memory that could be borrowed from the parent job), the parent job token, and the amount of memory borrowed.

What is a Memory Segment?

A memory segment is a contiguous sequence of bytes, ranging in size from 1 byte to 4 Gbytes.

There are no restrictions on what you can use memory segments for; you can create segments to hold data of whatever size and internal structure you need. The Nucleus itself creates segments in response to a wide range of system calls; all the iRMX objects are constructed from segments.

Creating a Segment

The only parameter you specify in the **create_segment** system call is the size of the new segment. If enough memory is available, the Nucleus returns a token for the segment.

A segment's physical starting address is on a 16-byte (paragraph) boundary. The Nucleus assigns each segment a slot in the Global Descriptor Table (GDT). That GDT slot multiplied by 8 serves as the segment token.

You can use the segment token when you use **send_message** to send a message, for example. You can also use the token for a segment as the selector portion of a pointer to the segment when placing data into the segment. The SELECTOR data type is especially useful in referring to the segment.

You can use the **rqe_change_object_access** call to change data segments to read-only and read/write for data segments or execute-only or execute/read. You can use **get_size** to get a segment's size in bytes.

See also: Data types, Nucleus examples, *System Call Reference*

Boundary Alignment

In a Multibus II system, solicited messages pass across the system bus more efficiently if buffers are aligned on a 4 byte boundary. Both the base address of the segment and its length are multiples of four. The **create_segment** system call automatically creates buffers that adhere to this convention. Because of the 4 byte boundary, Direct Memory Access (DMA) can be done in one cycle (fly-by mode), in which the DMA controller directly transfers data between the Message Passing Coprocessor (MPC) and memory. The Nucleus Communication Subsystem (NCS) supports one-cycle transfers for aligned buffers.

See also: MCO, MCT, and MDC parameters on the MBII screen, *ICU User's Guide and Quick Reference*

If the buffer is not aligned on a 4 byte boundary, each DMA operation requires two cycles: one to place the information in the DMA controller's buffer and another to move it to the desired destination.

For example, on an SBC 486/125 or 486/150 board, solicited data can be transferred at 13.3 Mbytes per second using one-cycle transfers; using two-cycle transfers, the rate is 4 Mbytes per second. On SBC 386/133, 486/125, or 486/133SE or MIX n86/020A, 486SX33, 486DX33, or 486DX66 boards, alignment on a 16-byte boundary and length allows even faster DMA burst mode. For example, on an SBC 486/125 or 486/150 board, solicited data can be transferred at 20 Mbytes per second using burst mode. The NCS picks the fastest mode possible.

Deleting a Segment

You delete a segment using **delete_segment**; any task that knows the segment's token can make the call.

Access Rights and Hardware Types

When the microprocessor is operating in protected mode, a segment's access bytes define the way the segment can be used by instructions in other segments.

When you create an object, its corresponding segment is assigned a read/write access type. Before the OS performs any operation, the processor checks the access type. If you have entered the wrong access type, the processor causes a hardware exception.

You can check an object's type using **rqe_get_object_access**. Provide a pointer to the `access_ptr` data structure and the system call fills in the results.

You can change an object's access type for segment objects, descriptor objects, or composite objects using **rqe_change_object_access**. Access rights for all other objects cannot be changed. This system call uses the access byte format provided by the microprocessor for both code and data segment descriptors.

See also: **rqe_get_object_access** and **rqe_change_object_access**, *System Call Reference*;
the user's manual for your microprocessor

CAUTION

Do not change bits in a token. This can cause a hardware exception.

Heap Management

The iRMX III OS includes a memory object called a heap. The interfaces described in the following subsections provide the object management interface for heap objects.

A heap is created, then buffers may be requested from and subsequently released to the heap. Finally, the heap is deleted, which also deletes the underlying segment. Two additional interfaces are provided: one finds the size of a buffer allocated from a heap given the heap token and the buffer pointer, and one interrogates the heap object to determine how much of its resources have been used.

When buffers are requested (allocated) from the heap, a full pointer is returned and the base (segment) of the pointer is used to validate the buffer when it is returned to the heap. If a flat-model application creates a heap, that heap's segment is mapped into the job's virtual segment.

No reference is returned to the user, because no direct access of the segment is intended. When buffers are allocated from the segment, the interface library must adjust the buffer offset according to the task's mapping of the heap segment so that a valid pointer is returned to the caller. If a heap token is used by a flat-model job which did not create it, the heap object must be mapped into the new job's virtual segment and an internal reference of this mapping be kept by the object. This has two purposes:

- To allow a buffer allocated to a task in the new job to be mapped correctly, and
- If the heap object is deleted, the mappings may be deleted from each job's virtual segment.

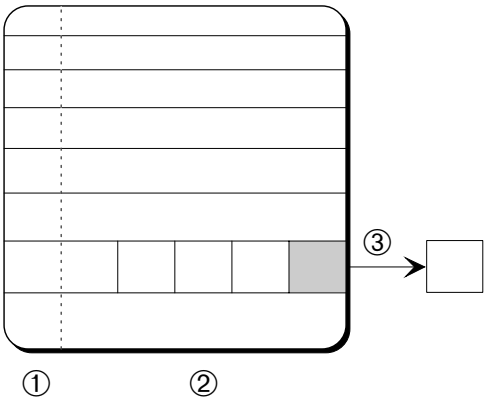
What is a Buffer Pool?

A buffer pool manages a preallocated set of segments that you can allocate dynamically so they will be available quickly and dependably during time-sensitive operations. The Nucleus maintains the buffer pool, and your application maintains the segments that it holds. You reference a buffer pool with its token.

The pool can maintain eight different sizes of segments. The buffer pool maintains a linked list of buffers for each size of segment; all segments in a given list have the same size. Each linked list can contain as many segments as you need.

After you create and fill a buffer pool, you just specify a token to identify the pool and how much memory you need. The Nucleus automatically takes care of the rest.

Figure 7-3 shows a buffer pool and its associated buffers.



1. This area is the linked list for each of eight segment sizes. Each linked list can have as many buffers as you need.
2. This area holds the buffers. The individual buffers are segments with read and write access enabled.
3. During its existence, a pool gives buffers to tasks when they call **request_buffer**. When the pool delivers a buffer to a requesting task, the buffer is removed from the list of available buffers. The task releases the buffer back to the pool using **release_buffer**.

Figure 7-3. Buffer Pool with Associated Buffers

Creating and Initializing a Buffer Pool

These are the parameters you specify when you use the **create_buffer_pool** system call to create the buffer pool.

- The maximum number of buffers that can exist at one time in the buffer pool
- Whether data chains are supported, or only contiguous buffers

A newly created pool is an object with a set of attributes that defines its capabilities. Buffer pools incur a certain amount of system overhead in their creation. This formula defines the amount of resources required.

(Max Buffers * 4) + 108 bytes = memory used by a buffer pool

Buffers are not allocated for a buffer pool at creation. You must allocate a set of segments for the buffer pool. Create the segments using the **create_segment** system call. You can create as many segments as you need in up to eight different sizes.

Then, you place the segments in the buffer pool using **release_buffer**. This process is called *initializing the buffer pool*. Do it early in your program rather than in the middle of real-time operation. Do not release segments created in different jobs to the buffer pool.

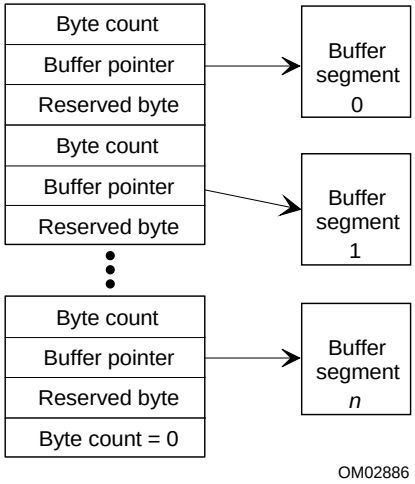
Using Data Chains

> **Note**
> You can use data chains only in the iRMX III OS. The configuration of DOSRMX and iRMX for PCs does not allow the use of data chains.

If you enable data chaining, a message may be copied into a data chain instead of a single buffer, depending on its size. The NCS strings small buffers together to fill a large request made in **request_buffer**. If the pool constructs a data chain, it returns a selector to a data chain block that holds pointers to the segments that make up the chain. The buffer pool also returns the E_DATA_CHAIN condition code to **request_buffer** so the requesting task will know it has received a chain block.

The amount of data in the message determines the number of buffers used. When a data chain is created, the required number of buffers are removed from the buffer pool and made into the chain. One additional buffer is taken from the buffer pool and used as the chain block, which contains a list of all the buffers in the chain.

Figure 7-4 on page 102 shows the structure of a chain block.



The buffer pointer is a selector:offset pair with 16-bit offset (not 32-bit). To use as a pointer in PL/M-386, use this: buf$p = build$ptr(chain(n).base, chain(n).off)

The byte count data type is WORD_16; each component buffer cannot exceed 64 Kbytes in length.

Byte count = 0 is the chain terminator.

Figure 7-4. Structure of a Chain Block

The minimum buffer size for data chains is 1Kbyte in length, and you must request at least one buffer to enable the system to build the data chain block.

- The minimum data chain block size is (max_elements*8) + 2 BYTES.
- The maximum number of elements is a configuration option. At least one 1026 byte buffer will be available in the buffer pool for chain allocation.

See also: MCE parameter on the NUC screen, *ICU User's Guide and Quick Reference*

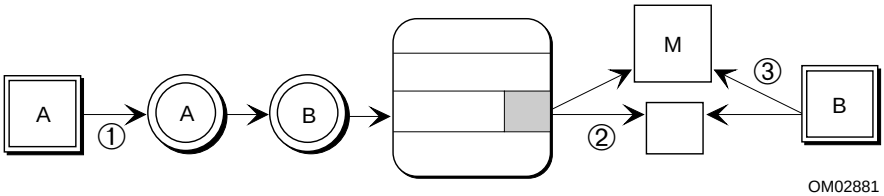
Note

Data chains are not supported as message passing buffers on the SBC 486DX33 and SBC 486SX25 boards.

Using attach_buffer_pool

If you have created the buffer pool to use with a port, you have to attach it to the port with the **attach_buffer_pool** system call. When the NCS delivers a message to the port, it will automatically store data messages coming to the port in buffers from the pool. The NCS requests buffers at the same time it receives a buffer request. The NCS grants the request, using buffers from the attached pool. The buffer or buffers are then passed to the task that receives the message. Even if the buffer pool is attached to a port, you can still use the buffer pool token to perform operations on the buffer pool.

Figure 7-5 shows the relationship of a buffer pool and an attached port. It also shows how a pointer to a buffer is passed to a task receiving a message at the port.



1. Task A sends a message through port A to port B.
2. The NCS requests a buffer from the pool attached to port B and places the message directly into the buffer. A pointer M to the message buffer is placed into the message queue of the port.
3. Task B receives the pointer to the message buffer and accesses it. Task B should release the buffer back to the pool when finished with the message.

Figure 7-5. Relationship of Buffer Pool and Port

Using detach_buffer_pool

This call does not delete the buffer pool, it only removes the association to the port. If no buffer pool is attached, the E_STATE condition code returns.

Using request_buffer

You can use **request_buffer** when you need a buffer from a pool for any purpose. The NCS calls **request_buffer** when it needs space to store an incoming data message at a port.

The pool returns a pointer to the smallest buffer that fills the request; the buffer may be equal to or larger than the requested size.

If no single buffer is large enough to fill the request, and data chaining is enabled, the NCS attempts to create a data chain and returns a pointer to the data chain block along with the E_DATA_CHAIN condition code.

Using release_buffer

Use **release_buffer** to return the buffer back to the pool when the task is done with the information in the message. Otherwise the buffer is not available to the pool.

Release_buffer adds the segment to one of the lists of buffers in the pool. If the size of the segment is different from any of the sizes currently maintained by the pool, the pool creates a new list for segments of that size. Up to eight lists are supported.

If you are releasing a chain block, a single call to **release_buffer** releases all data chain buffers to the pool, including the data chain block buffer. Use the `flags` parameter to indicate whether the segment is a single buffer or a data chain block.

Deleting a Buffer Pool

You cannot delete a buffer pool using **delete_buffer_pool** while it is attached to a port; you must first detach it using **detach_buffer_pool**. A task attempting to delete an attached pool will receive an E_STATE condition code.

Memory Management System Calls

These are the system calls that relate directly to memory management.

Memory pool call	**rqe_get_pool_attrib**
Segment calls	**create_segment** **delete_segment** **get_size** **rqe_get_object_access** **rqe_change_object_access** **rqe_get_address**
Buffer pool calls	**create_buffer_pool** **delete_buffer_pool** **request_buffer** **release_buffer** **attach_buffer_pool** **detach_buffer_pool**
Heap management calls	**create_heap** **delete_heap** **rqe_request_buffer** **rqe_release_buffer** **get_heap_info** **get_buffer_size**

These are the rules for buffer pools:

- You can attach a buffer pool to more than one port.
- You cannot attach a port to more than one buffer pool.
- A port must be in the same job as the attached buffer pool.

Table 7-1 lists the operations used to manage memory segments and buffer pools and the system calls that do the operations.

Table 7-1. Memory Management System Calls

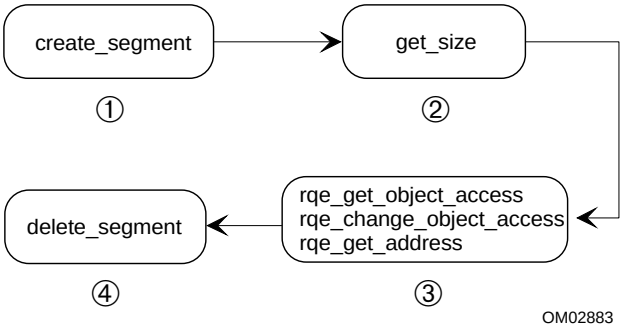
Operation	Description
get pool attributes	**Rqe_get_pool_attrib** gets information about pool status or use of its job's memory pool or another job's memory pool.
create segment	**Create_segment** creates a segment and returns a selector to a new segment.
delete segment	**Delete_segment** deletes a segment and returns the memory to the job's memory pool from which it was allocated.
get size	**Get_size** returns the size of a segment in bytes.
change access rights	**Rqe_change_object_access** changes the access rights of the segment.
check access rights	**Rqe_get_object_access** returns an object's access rights.
get physical address	**Rqe_get_address** converts an object's logical address into its physical address, which may be needed for device drivers or for creating aliases.
create buffer pool	**Create_buffer_pool** creates a buffer pool and returns a token for the pool.
delete buffer pool	**Delete_buffer_pool** accepts a token for a buffer pool and deletes the pool and any buffer segments it contains.
request buffer	**Request_buffe**r gets a buffer from a pool that has been created using the **create_buffer_pool** system call.
release buffer	**Release_buffer** adds a segment to one of the lists of buffers in the pool, either to initially fill the buffer pool or to return a segment to the buffer pool.
attach buffer pool to port	**Attach_buffer_pool** accepts a token for a buffer pool and a token for a port and associates the buffer pool with the port.
detach buffer pool	**Detach_buffer_pool** accepts a token for a port and detaches the buffer pool that is currently attached.
create heap	**Create_heap** creates a heap and returns a token for the heap.
delete heap	**Delete_heap** accepts a token for a heap and deletes the heap.
rqe request buffer	**Rqe_request_buffe**r gets a buffer from a heap that has been created using the **create_heap** system call, or from a pool that has been created using the **create_buffer_pool** system call.

rqe release buffer	**Rqe_release_buffer** returns a preciously allocated buffer space to the specified buffer pool or heap.
get heap info	**Get_heap_info** returns a structure containing information about a heap object.
Get buffer size	**Get_buffer_size** returns the size of a buffer previously allocated from a heap

See also: Nucleus system calls, *System Call Reference*

How to Use Memory Management System Calls

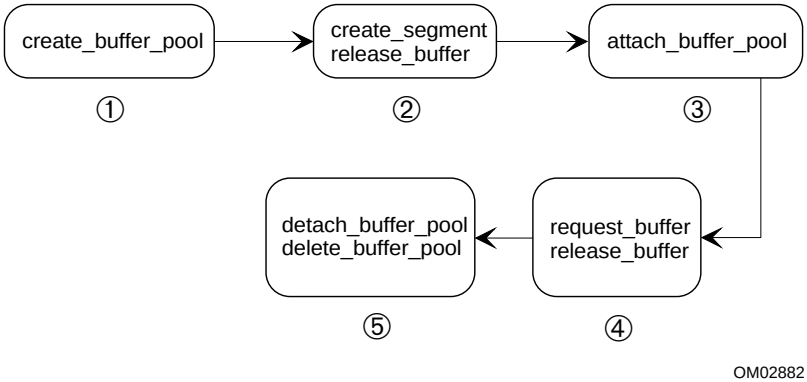
Figure 7-6 shows the order in which you make memory segment system calls.



1. Make this call from the task that needs a memory segment.
2. Make this call from any task that needs to know the size of the segment.
3. Make these calls if you need to change the segment's access rights.
4. Make this call from any task that knows the segment's token.

Figure 7-6. Segment System Calls

Figure 7-7 shows the order in which you make buffer pool system calls.



1. Make this call from the task that needs a buffer pool.
2. These calls fill the buffer pool with buffers. Make these calls from the task that created the buffer pool.
3. Make this call if the creating task has a port that needs a buffer pool.
4. Make these calls from the creating task.
5. Make this call from the task that created the buffer pool.

Figure 7-7. Buffer Pool System Calls

❑❑❑

8 Object Directories

What is an Object Directory?

An object directory contains a list of object names and corresponding tokens.

- The name contains from 1 to 12 characters; a character is a 1-byte value from 0 to 0FFH. Some tasks may know objects only by name.
- The token is a 16-bit selector or handle for objects.

The object directory enables tasks to use symbolic names to share access to objects.

Creating a Job Object Directory

The Nucleus creates an object directory for each job as you create it, using **rqe_create_job**. You specify the number of entries allowed in an object directory in the `directory_size` parameter of the **rqe_create_job** system call.

Deleting a Job Object Directory

You delete the job object directory when you call **delete_job.**

Using an Object Directory

Typically, one task creates an object and catalogs its token and name. Another task uses that name to look up the token for the object. Two or more tasks can share an object that is cataloged in an object directory.

> **Note**
> The object directory is case-sensitive: upper- and lower-case alphabetic characters are interpreted differently. The Nucleus, however, sees the name as just a string of bytes; it does not interpret these bytes as ASCII characters.

Using catalog_object

Use the **catalog_object** system call to put the object into an object directory. You specify the job in which to catalog the object, the object's token, and a name for the object. If the object directory is full, the task receives an E_LIMIT condition code.

You can catalog the object in the task's job or in any job for which you have the token. Each job has an object directory, including the root job. To make an object accessible to all tasks in the system, catalog it in the root job. Call **get_task_tokens** to obtain the token of the root job.

You can use any byte values except null in the name. You can catalog the object under several different names, all with the same token, if your application needs this. The Nucleus will return a condition code if you try to catalog an object using a name that is already cataloged in the directory.

Using lookup_object

Use **lookup_object** to get an object's token so a task can access the object. You specify the token of the job whose object directory you want to search, the name of the object and the amount of time the task can wait. If the object is not cataloged, the task goes to sleep by waiting for the specified time or until the name is cataloged, whichever comes first.

The call returns the object's token, or an E_TIME condition code if the object is not cataloged during the specified wait time. If the object directory is full and the task specified no wait time, it receives an E_LIMIT condition code instead.

Using rqe_inspect_directory

Use **rqe_inspect_directory** to view the contents of a job's object directory in a single operation. You specify the token of the job whose object directory you want to

inspect and a pointer to a structure in which the contents of the objectory will be returned. The structure also contains the maximum number of entries to be returned.

Using uncatalog_object

You remove entries from a directory using the **uncatalog_object** system call.

Object Directory System Calls

These are the system calls that relate directly to object directories.

catalog_object
get_type
inspect_directory
lookup_object
uncatalog_object
get_task_tokens

Table 8-1 lists common operations on objects in object directories and the system calls that perform the operations. Tasks can use the object directory for their job or another job.

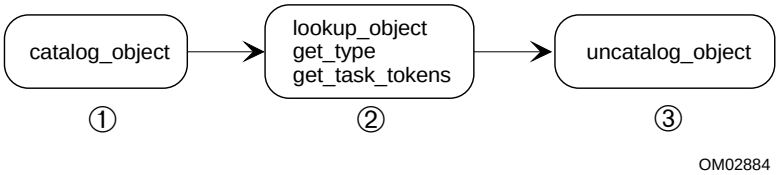
Table 8-1. Object Directory System Calls

Operation	Description
enter object in directory	**Catalog_object** catalogs the specified object in an object directory.
get type of object	**Get_type** accepts a token for an object and returns its type code. This enables you to use the appropriate calls for the object.
inspect contents of entire object directory	**Rqe_Inspect_Directory** fills a passed in structure with the entries in the specified object directory.
look up name of object	**Lookup_object** returns the token of the named object.
remove directory entry	**Uncatalog_object** removes the entry for the specified name. The name becomes available for re-use.
get token for object	**Get_task_tokens** gets a token for a parameter object or for the task's job, parent job or root job in order to catalog objects.

See also: Nucleus system calls, *System Call Reference*

How to Use Object Directory System Calls

Figure 8-1 shows the order in which you make object directory system calls.



1. Make this call from the task that created the object.
2. Make these calls from tasks that need to access the object.
3. Make this call from the task that created the object.

Figure 8-1. Object Directory System Calls

□□□

Exception Handling and System Accounting 9

This chapter describes how to handle the condition codes, or exceptions, that are returned from iRMX system calls. The primary method is to use exception handlers, either those provided with the OS or handlers you write. After handling an exception, you may want your application to investigate the state of the system by using a set of calls that return system accounting information.

Exception Handling

Whenever a task makes a system call, the system returns a *condition code* to the task to communicate the success or failure of the call. For example, a task may request memory that is not available. Conditions that represent failure or incomplete success are called *exceptional conditions*.

These are sources of exceptions:

- Hardware exceptions, such as trying to execute a read/write data segment. These occur as a result of violating a hardware protection feature.
- "Environmental errors, such as trying to write to a printer that is off-line. These conditions arise outside the control of the calling task.
- Programmer errors, such as making a mistake in a system call. These are conditions that the calling task can prevent.

See also: Nucleus interrupt and exception handling, *Introducing the iRMX Operating Systems*;
List of condition codes, *System Call Reference*

You assign an exception handler for a job when you use the **rqe_create_job** call: either the parent job's current exception handler, which normally deletes the job in which the error occurred, or a custom handler you write. You also assign the exception mode: when to transfer control to the exception handler. If you do not set a job's exception mode to transfer control to the job's exception handler, the tasks in the job must deal with exceptions either by handling them inline or by specifying their own exception handler using **set_exception_handler** or **rqe_set_exception_handler**. In either case, you must write code to handle the exception.

Exception Handler Actions

The Nucleus supports exception handlers for programmer errors in tasks. You decide how an exception handler should deal with a condition. In general, a handler does one of these:

- Corrects the cause of the exception and continues.
- Logs the error and continues.
- Deletes the job containing the task that erred.
- Deletes the task that erred.
- Suspends the task that erred.
- Ignores the error. (Note that hardware exceptions cannot be ignored.) If you choose this option, the system continues as if no error had occurred. This is generally unwise.

You can specify the System Debug Monitor (SDM) as the default exception handler when debugging an ICU-configurable system. Then SDM takes control of all the supported hardware exceptions except those from the Numeric Processor Extension (NPX). When you specify SDM as the default exception handler, hardware exceptions cause a break to SDM and send a message to the console. This is the default for systems that are not ICU-configurable.

See also: Nucleus screen, *ICU User's Guide and Quick Reference*;
sdb.job, *System Configuration and Administration*;
Writing Exception Handlers, later in this chapter

Rather than allowing SDM to take over on a hardware exception, you can write exception handlers that test for and handle these hardware traps. Since your handler is also called for programmer and environmental exceptions (unless these exceptions are handled inline), the handler must first determine the type of error and act accordingly.

Note

Because exception handlers can now process hardware traps, you must modify existing custom exception handlers to test for and process hardware traps.

See also: Writing Exception Handlers, later in this chapter

Exception Handler Modes

An exception handler normally receives control when an exceptional condition occurs, but it may not, depending on the *exception mode*. These are the exception-mode circumstances under which the handler gets control:

- Programmer errors only (all other errors handled inline)
- Environmental conditions only (all other errors handled inline)
- Always
- Never (all errors handled inline)

After detecting that a system call has encountered an exceptional condition, the Nucleus compares the condition with the calling task's exception mode. The Nucleus determines whether to pass control to the exception handler based on the mode. The exception handler then deals with the problem and returns control to the task, unless the exception handler deleted the job, deleted the task, or suspended the task. When the exception handler returns, the system call's `except_ptr` parameter points to the condition code. While the exception handler is executing, the task in which the error occurred is still the running task. The exception handler task uses the stack and environment of the task that made the system call.

Note
The only deviation from this behavior occurs for hardware traps. When a hardware trap occurs the current assigned exception handler is called regardless of the exception handler mode.

Condition Code Values and Mnemonics

Condition codes are numeric values that represent unique conditions. Each code also has a mnemonic such as E_OK, which indicates successful completion or E_MEM which indicates not enough memory.

When you write tasks, you can refer to the condition codes by their mnemonics. The OS installs include files that contain literal declarations for iRMX condition codes.

See also: Condition code numeric values and mnemonics for specific system calls, *System Call Reference*

The values of condition codes fall into ranges based on the iRMX layer that first detects the condition and the type of exception. Table 9-1 shows the ranges based on the type of error and the layer detecting the condition. Numeric values appear in hexadecimal.

Table 9-1 Condition Code Ranges

Hardware Exceptions		8100H to 8111H
Numeric Processor Extension Exceptions		8007H (NPX Error)
All Other Programming and Environmental Exceptions		
Layer	**Environmental Conditions**	**Programming Errors**
Nucleus	0H to 0FH	8000H to 800FH
I/O Systems	20H to 5FH	8020H to 805FH
Application Loader	60H to 7FH	8060H to 807FH
Human Interface	80H to AFH	8080H to 80AFH
Universal Development Interface	C0H to DFH	80C0H to 80DFH
Comm Service	E0H to EFH	80E0H to 80EFH
Reserved for Intel	F0H to 2CFH	80F0H to 82CFH
iNA Networking	2D0H to 3FFH	82D0H to 83FFH
Reserved for RadiSysl	400H to 3FFFH	8400H to BFFFH
Available for applications	4000H to 7FF0H	C000H to FFF0H

Handling Exceptions Inline

You can write tasks that handle exceptions inline.

Each system call has `except_ptr` as its last parameter. After a system call, the Nucleus returns the resulting condition code to this parameter. By checking this parameter after each system call, you can determine if the call was successful or which exceptional condition occurred. This information can sometimes enable the task to recover. In other cases, more information is needed.

If a system call returns an exception code to indicate an unsuccessful call, all other output parameters of that system call are undefined.

See also: Condition codes for each system call, *System Call Reference*

Assigning an Exception Handler

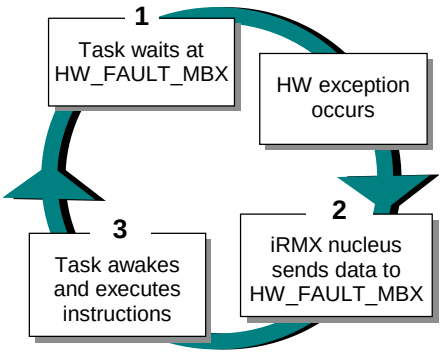
Use the **set_exception_handler** system call to enable a task to use its own exception handler and exception mode. Otherwise, the task inherits the exception handler and mode of its job.

You can also use the **rqe_set_exception_handler** system call to set or modify the exception handler or exception mode for the current task's job or for the system.

Exception handlers execute in the context of the task that caused the problem.

OS-Assisted Handling of Hardware Exceptions

As an alternative to writing a custom hardware exception handler, the iRMX nucleus creates a system-wide data mailbox during initialization and catalogs it in the root job's object directory with a name of HW_FAULT_MBX. If the active system hardware exception mode is either Delete the Job, Delete the Task, or Suspend the Task, then the Nucleus handles hardware exceptions as shown in this figure:



1 An application task waits at the HW_FAULT_MBX for a message.

2 When a hardware exception occurs and Soft-Scope is inactive, the iRMX nucleus sends a message to this mailbox in the format listed in the tagFaultInfo structure.

3 When a message arrives, the task awakes and executes its instructions.

You can set up hardware exception handling in these ways:

- **System-wide**: When a message arrives at HW_FAULT_MBX, you can call rqgettype to determine the system hardware exception mode, and thus determine the state of the offending task:
  - If the task and job are valid, then the offending task is suspended.

- If the job is valid but the task is not, then the offending task has been deleted.
- If both the task and the job are invalid, then the offending task's job has been deleted.

- **Job-specific**: You can set up your application to create its own data mailbox and catalog it in its object directory with a name of HW_FAULT_MBX. Then the iRMX nucleus sends the message to this mailbox as well as to the system-wide HW_FAULT_MBX. When the message arrives, you can determine the offending job's state as described above and, since you are in the job whose member just experienced the hardware fault, you can instruct the program to remedy the situation.

TagFaultInfo structure

If the system hardware exception handler is set to Delete Job, Delete Task, or Suspend Task, and a hardware exception handler is encountered, the iRMX nucleus sends a message to the HW_FAULT_MBOX with this form:

```
typedef struct  tagFaultInfo {
    TOKEN    task;
    TOKEN    job;
    UINT_16  exception;
    UNIT_32  faulting_eip;
    TOKEN    faulting_cs;
} FAULT_INFO;
```

Where:

`task` Is the TOKEN of the task which encountered the hardware fault.

`job` Is the job that contains the task which encountered the hardware fault.

`exception` Is a WORD that contains the hardware fault exception as follows:

EH_ZERO_DIVIDE	8100H	Divide by zero error
EH_SINGLE_STEP	8101H	Single step trap
EH_NMI	8102H	Non-maskable interrupt
EH_DEBUG_TRAP	8103H	Debug interrupt
EH_OVERFLOW	8104H	Overflow error
EH_ARRAY_BOUNDS	8105H	Array bounds error
EH_INVALID_OPCODE	8106H	Invalid op code error
EH_DEVICE_NOT_PRESENT	8107H	No numerics device error
EH_DOUBLE_FAULT	8108H	Double fault error
EH_DEVICE_ERROR	8109H	Device error
EH_INVALID_TSS	810AH	Invalid TSS error
EH_SEGMENT_NOT_PRESENT	810BH	Segment not present error

EH_STACK_FAULT	810CH	Stack fault error
EH_GENERAL_PROTECTION	810DH	General protection error
EH_PAGE_FAULT	810EH	Page fault error
EH_RESERVED_INT15	810FH	Reserved
EH_DEVICE_ERROR1	8110H	Device 1 error
EH_ALIGNMENT_CHECK	8111H	Alignment check error

`Faulting_eip` Is the OFFSET of the instruction which caused the hardware exception.

Faulting_cs Is the SELECTOR of the instruction which caused the hardware exception.

Writing Your Own Exception Handler

You need to consider several things when you write your own exception handler. For example, 32-bit code requires 32-bit exception handlers, and 16-bit code requires 16-bit exception handlers. The only time this is not true is if the exception handler deletes the offending job, deletes the offending task, or suspends the offending task.

Another consideration is the type of exception you are processing. As of release 2.2 of the iRMX OS, you can write exception handlers that process hardware traps. This means that your handler can process three groups of errors:

- Hardware traps
- Numeric Processor Extension (NPX) exceptions
- All other programming and environmental conditions

Also, the exception handler executes in the context of the task that caused the problem. Because of this, deleting the task will kill the exception handler.

Finally, if you set the system's default exception handler in the ICU on the (NUC) Nucleus screen by setting DSH equal to "User", your exception handler module must have these characteristics:

- The public entry point must be named `rqsysex`
- It must be 32-bit code
- It must be compiled as Near using Intel OMF386 tools (iC-386, PL/M-386, or ASM386)

Handler Prototype

You can create your exception handler to determine the type of problem and act accordingly by creating a FAR, typed procedure that follows this prototype definition:

```
UINT_8 _Fparam far my_exception_hndlr(
            (UINT_16) err_code,
            (UINT_16) param_num,
            (UINT_16) param_1,
            (UINT_32) param_2);
```

Where:

`err_code` Indicates the type of error. Values from 8100H through 8111H represent hardware traps. A value of 8007H represents an NPX exception. The ranges shown in Table 9-1 on page 118 represent all other programming errors and environmental conditions.

`param_num` Represents the offending parameter number of the call that caused the problem. `param_num` is not valid for hardware traps or NPX exceptions.

`param_1` For hardware traps, `param_1` is the selector part of the pointer to the CPU_FRAME_STRUCT structure (see CPU_FRAME_STRUCT later in this chapter). For all other exceptions, `param_1` is meaningless.

`param_2` For hardware traps, `param_2` is the offset part of the pointer to the CPU_FRAME_STRUCT structure (see CPU_FRAME_STRUCT later in this chapter). For NPX exceptions, `param_2` is an NPX status. For all other programming errors and environmental conditions, `param_2` is meaningless.

Handler Contents

The first task your new handler (and all existing user-written handlers) must perform is to examine the value of `err_code`. Next, the handler must perform one of the following, based on the type of error:

- Use the BUILDPTR function to build a pointer (`frame_p`) out of `param_1` and `param_2` that points to the CPU_FRAME_STRUCT if the exception is a hardware trap.
- Derive the NPX status from `param_2` if the exception is from the Numeric Processor Extension.
- Ignore `param_1` and `param_2` if the error is a programming error or environmental condition.

Once your handler determines the type of exception and casts the parameters to the right types, it must process the error. Usually this involves correcting, logging, or reporting the condition. However, for hardware exceptions you have two choices after processing the error:

- Return to the task that caused the exception. If you write the handler to do this your handler must also fix the task's problem.
- Prevent the task from running again by either deleting or suspending it. Because the operating system already has handlers that delete tasks, delete jobs, and suspend tasks, you can write your handler to return to the appropriate system exception handler.

The value returned in the AX register by your typed exception handler procedure determines which of these two options is taken. A returned value of 0 causes the exception handler dispatcher to call the currently active system hardware trap handler to deal with the offending task. Returning a value of 0FFH causes the exception handler dispatcher to return to the offending task at the code segment (CS:EIP) present in the CPU_FRAME_STRUCT structure.

The following pseudo-code example shows how to handle any exceptional condition. The first conditional handles NPX conditions. The second conditional handles hardware traps. The default condition handles all other programming and environmental exceptions.

```
if err_code is 8007H then {
   derive NPX status as follows:
      NPX_status = (UINT_16) param_2 ;
   Correct, log, or report the condition ;
   return ;
}
if err_code ranges from 8100H through 8111H then {
   generate pointer using the built_in BUILDPTR as follows:
      frame_p = BUILDPTR((selector)param_1),
                          (void near *)param_2) ;
   Log or report the condition ;
   If calling active system handler then {
      return (0) ;
   }
   else if returning to the offending task then {
      return (0FFH);
   }
}
else exception is env/prog error, handle normally {
   Correct, log, or report the condition ;
   return ;
}
```

Compiling Your Exception Handler

If you are writing your own exception handlers, compile them as far procedures by EXPORTING the procedure with the PUBLIC attribute.

Parameters Used With Hardware Traps

When the value for the `err_code` parameter is in the range 8100H to 8111H, a hardware trap has occurred. As shown earlier, your handler must generate the pointer `frame_p` that points to the CPU_FRAME_STRUCT structure when processing a hardware trap. The type definition of this structure is as follows:

```
typedef      struct{
       SELECTOR     running_task;
       UINT_16      fill0;
       UINT_32      reg_cr2;
       SELECTOR     reg_gs;
       UINT_16      fill1;
       SELECTOR     reg_fs;
       UINT_16      fill2;
       SELECTOR     reg_es;
       UINT_16      fill3;
       SELECTOR     reg_ds;
       UINT_16      fill4;
       SELECTOR     reg_ldt;
       UINT_16      fill5;
       UINT_32      reg_edi;
       UINT_32      reg_esi;
       UINT_32      reg_ebp;
       UINT_32      reg_esp;
       UINT_32      reg_ebx;
       UINT_32      reg_edx;
       UINT_32      reg_ecx;
       UINT_32      reg_eax;
       UINT_32      error_code;
       UINT_32      ret_eip
       SELECTOR     ret_cs;
       UINT_16      fill6;
       UINT_32      eflags;
       UINT_32      ret_esp;
       SELECTOR     ret_ss;
       UINT_16      fill7;
} CPU_FRAME_STRUCT;
```

where:

`fill<`*n*`>` Reserved

`running_task`
: The token of the task whose CPU register state is being provided.

`reg_cr2` The task's CR2 register; `reg_cr2` is only valid in the context of an exception handler.

`reg_gs` The task's GS register.

`reg_fs` The task's FS register.

`reg_es` The task's ES register.

`reg_ds` The task's DS register.

`reg_ldt` The task's LDTR register.

`reg_edi` The task's EDI register.

`reg_esi` The task's ESI register.

`reg_ebp` The task's EBP register.

`reg_esp` The task's ESP register.

`reg_ebx` The task's EBX register.

`reg_edx` The task's EDX register.

`reg_ecx` The task's ECX register.

`reg_eax` The task's EAX register.

`error_code`
: The error code returned by the processor; `error_code` is only valid in the context of an exception handler.

`ret_eip` The task's EIP register.

`ret_cs` The task's CS register.

`eflags` The task's EFLAGS register.

`ret_esp` The task's ESP register.

`ret_ss` The task's SS register.

Exception Handler System Calls

These are the system calls that relate directly to exception handlers.

get_exception_handler
set_exception_handler
rqe_get_exception_handler
rqe_set_exception_handler

Table 9-2 lists common operations on exception handlers and the system calls that perform the operations.

Table 9-2 Exception Handler System Calls

Operation	Description
set handler	**Set_exception_handler** sets the exception handler and exception mode attributes of the calling task. **Rqe_set_exception_handler** sets or modifies the exception handler and exception mode for any task, job, or the system.
get handler attributes	**Get_exception_handler** returns to the calling task the current task's exception handler and exception mode attributes. **Rqe_get_exception_handler** returns to the calling task the current exception handler and mode for any task, job, or the system.

See also: Nucleus system calls, *System Call Reference*

System Accounting

Several system calls allow you to check on the state of tasks, the CPU, and other high-level system information. These calls can be useful at any time but are particularly useful after exceptions occur. The calls allow you to:

- Return information about the execution state and CPU registers of a task
- Return information about when a task was created and how long it has run
- Enable and disable tracking of CPU use by the operating system

Enabling and Disabling CPU Tracking

Use the **system_accounting** system call to enable or disable tracking of CPU usage by the operating system. Accounting must be enabled to use the **get_task_accounting** call.

See also: Nucleus System Calls, *System Call Reference*

Returning Information About a Task

Use the **get_task_info** system call to return high-level information such as task priority, exception handler, the containing job, and execution state. For a more detailed look at the state of a task, use the **get_task_state** system call. This call returns information about the state of any task in the system, including such items as the execution state and the CPU registers for the task's execution context.

> **Note**
> CPU context is only available for tasks that are suspended by a task other than itself.

See also: Nucleus System Calls, *System Call Reference*

Returning Task Creation and Duration Statistics

Use the **get_task_accounting** system call to find out when a task was created and how long it has run. This call can be useful in debugging a system when exceptions cause a task to be suspended.

See also: Nucleus System Calls, *System Call Reference*

System Accounting System Calls

These are the system calls that relate directly to system accounting.

get_task_info
get_task_state
get_task_accounting
system_accounting

Table 9-3 lists the type of system accounting you can perform with these calls.

Table 9-3 Accounting System Calls

Operation	Description
get high-level task information	**Get_task_info** returns high-level information about the target task.
get CPU information	**Get_task_state** returns some high-level information about the target task. This call also returns CPU register context for suspended tasks.
get accounting information	**Get_task_accounting** returns accounting information for the target task.
enable or disable accounting	**System_accounting** enables or disables tracking of CPU use.

See also: Nucleus system calls, *System Call Reference*

❑❑❑

Interrupts 10

How Do Interrupts Work?

Many different events can cause an interrupt. An interrupt which signals the occurrence of an external event, triggers an implicit call using an address supplied in the IDT. This directs control to an interrupt handler.

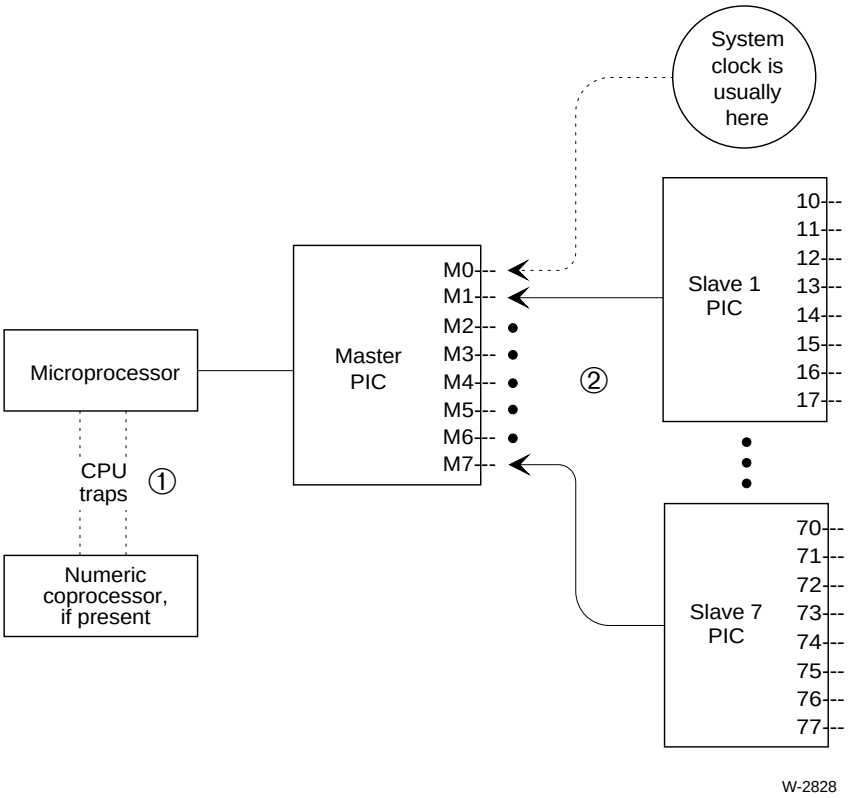
If handling the interrupt takes little time and requires no system calls other than certain interrupt-related system calls, the interrupt handler can process the interrupt itself; the interrupt handler executes in the context of the task running when the interrupt occurred. Otherwise, the handler should invoke an interrupt task to finish processing the interrupt. Interrupt tasks have their own context and are not dependent on the context of the task that was interrupted.

After the interrupt has been serviced by either the interrupt handler or the interrupt task, control returns to the interrupted task.

Interrupt Controllers and Interrupt Lines

External interrupts pass through programmable interrupt controllers (PICs) such as the 8259A PIC. The master PIC can manage interrupts from as many as eight external sources, one being the system clock. But the iRMX OS supports a cascaded environment in which up to seven input lines of one master PIC are connected to slave PICs, each with eight input lines.

A cascaded environment in native mode (non-PC architecture) lets the OS manage interrupts from up to 56 external sources as well as the system clock in native mode. Figure 10-1 on page 132 illustrates the concept.



1. If your system includes an Intel387 numeric processor, do not connect the NPX to a PIC. The Intel386 processor uses CPU interrupt traps 7 and 16 to communicate directly with the Intel387 numeric processor.

2. The interrupt lines on the master PIC are numbered M0 through M7. The interrupt lines on the slave PICs are numbered *n*1 through *n*7. You can connect a master PIC input line either to an external interrupt or to a slave PIC, but not to both.

Figure 10-1. Processor and PIC Interrupt Lines in Native Mode

PC-compatible Mode

In PC-compatible mode, attach the keyboard to M1, attach the only slave PIC to level M2 and attach the NPX to line 5 on the slave PIC.

See also: PIC, *ICU User's Guide and Quick Reference*

Interrupt Levels

The interrupt lines of the master and slave PICs are associated with interrupt levels. An interrupt level names an interrupt line and indicates the priority of the line: the lower the number, the higher the priority.

Lower-numbered lines like M2 (or lines from the slave PIC connected to it) have higher priority than higher-level lines like M5 (or lines from the slave PIC connected to it). If two interrupts occur simultaneously, the PIC informs the CPU of the higher-priority interrupt first. The Nucleus often disables low-priority interrupts to service high-priority interrupts.

Interrupt Descriptor Table

The processor uses the IDT entry as a pointer to the interrupt handler to execute for the specific interrupt. Each IDT entry contains the physical address of the interrupt handler.

The hardware assigns a number to the cause of each interrupt and gives it an entry in the IDT. The IDT is composed of up to 256 entries, numbered 0-255. In an ICU-configurable system, you specify the number of IDT entries your application needs using the NIE parameter. You will probably not need more than 128 entries. If, for example, your system has only the 8259A master PIC with no slaves, the first 64 entries are enough. The Nucleus does not use entries 128-255. The entries are allocated as shown in Table 10-1 on page 134.

See also: IDT, user's guide for your microprocessor;
NIE parameter, *ICU User's Guide and Quick Reference*

Table 10-1. Allocation of Interrupt Entries

IDT Entry	Description
0	divide by zero
1	single step (used by the SDM monitor)
2	power failure (nonmaskable interrupt, used by the SDM monitor)
3	one-byte interrupt instruction (used by the SDM monitor)
4	interrupt on overflow
5	run-time array bounds error
6	undefined opcode
7	NPX not present/NPX task switch
8	double fault
9	NPX segment overrun
10	invalid TSS
11	segment not present
12	stack exception
13	general protection
14-15	reserved
16	NPX error
17-55	reserved
56-63	8259A PIC master (external interrupts)
64-127	8259A PIC slaves (external interrupts)
128-255	unused

Assigning Interrupt Levels to External Sources

To assign interrupt levels to external sources, use these guidelines:

- Assign the system clock to a master interrupt level, usually M0.
- Assign the most critical interrupts to the lowest-numbered levels. To provide preemptive, priority-based scheduling, the Nucleus usually disables less-important interrupts.
- You cannot attach both an interrupting device and a slave PIC to the same master level. Suppose you physically attach a device to level M3: entry 59 decimal of the IDT contains the address of the interrupt handler for the device; entries 88 through 95 decimal of the IDT (the slave level entries that correspond to master level M3) will not be available.

- You cannot connect a slave PIC to M0 if an interrupting device connects directly to any other master level. If you assign the system clock to level M0, you can connect seven slave PICs. If you assign the system clock to another interrupt level, you can connect at most six slave PICs to the master PIC.
- The Intel387 NPX does not require a dedicated interrupt line in native mode. In PC-compatible mode, it uses M2, level 25.

See also: PIC, *ICU User's Guide and Quick Reference*

Interrupt Handlers and Interrupt Tasks

Whether an interrupt handler services an interrupt level by itself or invokes an interrupt task to service the interrupt depends on the system calls (these are limited) and the amount of time needed. An interrupt signal disables all interrupts; they remain disabled until the interrupt handler either services the interrupt and exits, or invokes an interrupt task. Invoking an interrupt task enables higher priority interrupts (and in some cases, the same priority interrupts) to be accepted.

See also: Random access support for interrupt-driven devices for examples of interrupt tasks, *Driver Programming Concepts*

System Calls and Interrupt Handlers

When writing an interrupt handler, you need to keep these points in mind:

- Interrupt handlers can make only the Nucleus **enter_interrupt**, **exit_interrupt**, **get_level**, **disable**, and **signal_interrupt** system calls. If you need other system calls to service the interrupt, create an interrupt task.
- Interrupt handlers should not use C library calls that perform high-level I/O operations such as **printf()**. These types of C library calls may be unsafe for use by handlers because they use signaling or blocking objects or they use high-level I/O.
- Interrupt handlers can use system calls that signal the Kernel such as **KN_send_units**. However, the handler must take steps to prevent a task switch.

  See also: Using iRMX Kernel Calls in iRMX Interrupt Handlers later in this chapter

Writing an Interrupt Handler

Interrupt handlers are generally written as C or PL/M interrupt procedures, but they can be written in assembly language. If you use assembly language, you must save and restore all register values.

An interrupt handler uses the stack of the interrupted task.

If an interrupt handler services interrupts for a given level without invoking an interrupt task, it must do these things:

1. Save all register contents (C and PL/M do it for you when the procedure is given the `INTERRUPT` attribute).
2. If the handler can load its own DS register with the data segment selector, do so. If the handler requires a special data segment, call **enter_interrupt**.
3. Service the interrupt.
4. Call **exit_interrupt**. This sends an end-of-interrupt (EOI) signal to the hardware.
5. Restore all register contents.
6. Return using an IRETD instruction.

See also: Designing an Application, *Programming Techniques*; examples in */rmx386/demo/c/int* directory

Using set_interrupt With a Handler Only

Before an interrupt handler can service an interrupt level, a task must invoke the **set_interrupt** system call to bind the handler to the interrupt level. **Set_interrupt** places a pointer to the first instruction of the handler in the appropriate entry in the IDT.

These are the parameters you supply in **set_interrupt**:

- Use the `interrupt_handler` parameter to specify the starting address of the interrupt handler. When an interrupt of that level occurs, control automatically transfers through the IDT to the handler.
- Set the `interrupt_task_flag` parameter to 0, to specify that there is no interrupt task for the level.
- Set the `interrupt_handler_ds` parameter to null to specify that the handler loads its own data segment. (Interrupt handlers written in PL/M, including COMPACT model, have their DS registers loaded automatically on invocation.)

What the OS Does With a Handler Only

1. When an iRMX application system starts running, all interrupt levels are disabled.
2. When **set_interrupt** binds an interrupt handler to a level, the Nucleus enables the level immediately.
3. When an interrupt occurs, the processor automatically transfers control to the handler. The handler executes in the context of the interrupted task with all interrupts disabled.
4. When the handler calls **exit_interrupt**, this sends an end-of-interrupt (EOI) signal to the hardware. Control returns to the interrupted task when the handler issues an IRETD instruction.

Use **reset_interrupt** to cancel the assignment of a handler by clearing out the appropriate entry in the IDT. The call also disables the specified level.

Using an Interrupt Handler and an Interrupt Task

If an interrupt handler invokes an interrupt task, it must do these things.

1. Save all register contents (C and PL/M do it for you when the procedure is given the `INTERRUPT` attribute).
2. If the handler needs to pass information to the interrupt task in a special data segment, call **enter_interrupt**. Usually the interrupt handler and interrupt task are in the same system and share the same data areas.
3. Possibly begin servicing the interrupt.
4. Do one of these:

   Call **signal_interrupt** to start the interrupt task and enable higher (and possibly equal) priority interrupts.

   or

   Call **exit_interrupt**. This sends an end-of-interrupt (EOI) signal to the hardware.
5. Restore all register contents.
6. Return using an IRETD instruction.

An interrupt task must perform these functions in the indicated order, although the first two functions may be interchanged:

1. Do any required task initialization, such as preloading variables.
2. Call **set_interrupt**.
3. Enter a loop which:
    a. Calls **rqe_timed_interrupt** or **wait_interrupt**.
    b. Services the interrupt when notified by a **signal_interrupt** call from the handler.
    c. Returns to step a.

An interrupt task, once initialized, is always in one of two modes: either servicing an interrupt or waiting for notification of an interrupt. However, the Nucleus does not enable the level or any lower levels until the task invokes the **wait_interrupt** or **rqe_timed_interrupt** system call.

The interrupt task has its own resources and runs in its own environment. The interrupt task can use exception handlers, whereas the interrupt handler always handles exceptions inline.

Using set_interrupt With a Handler and Task

These are the parameters you supply in **set_interrupt**:

- Use the `interrupt_handler` parameter to specify the starting address of the interrupt handler.
- Set the `interrupt_task_flag` parameter to not 0, to specify that there is an interrupt task for the level and to indicate how many pending interrupts can be queued before E_INTERRUPT_SATURATION occurs: the interrupt limit.
- Use the `interrupt_handler_ds` parameter to specify the data segment for the interrupt task. The interrupt handler can later load this data segment into the DS register by calling **enter_interrupt**. In most cases, an interrupt handler and an interrupt task are in the same subsystem and share the same data areas. (Interrupt handlers written in high-level languages that have a FAR interface, have their DS registers loaded automatically on invocation.)

  See also: EXPORT control for PL/M Compact subsystems

While the interrupt task is processing, the Nucleus disables all lower interrupt levels. The associated interrupt level is either disabled or enabled, depending on the `interrupt_task_flag` parameter.

If the number of pending interrupts is less than the interrupt limit specified, the associated interrupt level is enabled. All **signal_interrupt** calls that the handler makes (up to the limit specified) are logged.

If the associated interrupt level is disabled (the number of pending interrupts is equal to the pending interrupt limit) while the interrupt task is running, the call to **wait_interrupt** enables that level.

Using rqe_timed_interrupt or wait_interrupt

You should call **rqe_timed_interrupt** or **wait_interrupt** from interrupt tasks immediately after initializing and immediately after servicing interrupts. These calls suspend the interrupt task until the interrupt handler for the same level resumes it by invoking **signal_interrupt**.

If the number of pending interrupts from **signal_interrupt** calls is greater than 0 when the interrupt task calls **rqe_timed_interrupt** or **wait_interrupt** the task is not suspended. Instead, it continues processing the next **signal_interrupt** request.

Shared Interrupts

The PCI bus architecture as used in the PC allows less flexibility to the system designer regarding how interrupts are routed and separated from other sources. While it may be possible on a system consisting of a PCI Local Bus architecture to separate PCI device interrupts from each other, as soon as a bridge is added, and more buses connected, then it is almost impossible to avoid the possibility that different devices may be sharing the same interrupt line.

The iRMX III.2.3 OS provides support to allow different devices to share the same interrupt line by providing generic interrupt handlers internally, which call out to user-supplied, device-specific handlers. When an interrupt occurs on such a line, all the handlers installed for that hardware interrupt level are called in sequence, starting with the first installed. This results in a little extra overhead over standard handlers, so use of this feature should be restricted to situations where you cannot guarantee exclusive access to an interrupt line.

Another consequence of interrupt sharing is that the interrupt signals are level-triggered. That is, the PIC is programmed to recognize an interrupt condition whenever the interrupt line is asserted. Non-PCI (non-shared) interrupts are edge-triggered; the PIC recognizes when the interrupt line changes state to active. The practical consequence of this level triggering is that if the device is not instructed to de-assert its interrupt line before the CPU exits from the handler, then the interrupt condition is still asserted, and the handler is immediately re-entered. Against that, remember that while the CPU is executing an interrupt handler, no other interrupt will occur until the CPU interrupt mask is cleared. Thus we want to minimize the

time spent in each shared interrupt handler, but do sufficient work to remove the interrupt condition.

To install a shared handler, use the **rqe_set_interrupt** system call, and specify that this is to be a shared interrupt handler by setting bit 15 of the hardware interrupt level parameter.

As you write an interrupt handler that will be used with a shared interrupt level, take into account that some interrupt system calls will have no effect. For example, **signal_interrupt** is ignored for shared handlers, since this action is taken automatically after all handlers have been called.

Shared interrupt handlers should not call **get_level** since the return value is meaningless in shared handler context. The encoded interrupt level is supplied as a parameter to the handler. Calling **enter_interrupt** is also superfluous since the process context is already set up on shared handler entry.

Interrupt Task Priorities

When a task becomes an interrupt task by calling **set_interrupt**, the Nucleus assigns a priority to it according to the interrupt level to be serviced. Table 10-2 on page 141 shows the relationship between the encoded level (the value used for the `level` parameter of **set_interrupt**), the Master and Slave interrupt levels, the IDT slot and the priorities of tasks that service those levels.

Note

If an interrupt task's priority exceeds the maximum priority attribute of its job, the interrupt task fails to set up and the Nucleus returns an exceptional condition code. Prevent this by increasing the job's maximum task priority using **rqe_set_max_priority**

Table 10-2. Interrupt Level and Task Priority Information

iRMX Encoding	PIC Level Master	PIC Level Slave	IDT Slot	Interrupt Task Priority	iRMX Encoding	PIC Level Master	PIC Level Slave	IDT Slot	Interrupt Task Priority
00H		00	64	4	40H		40	96	68
01H		01	65	6	41H		41	97	70
02H		02	66	8	42H		42	98	72
03H		03	67	10	43H		43	99	74
04H		04	68	12	44H		44	100	76
05H		05	69	14	45H		45	101	78
06H		06	70	16	46H		46	102	80
07H		07	71	18	47H		47	103	82
08H	M0		56	18	48H	M4		60	82
10H		10	72	20	50H		50	104	84
11H		11	73	22	51H		51	105	86
12H		12	74	24	52H		52	106	88
13H		13	75	26	53H		53	107	90
14H		14	76	28	54H		54	108	92
15H		15	77	30	55H		55	109	94
16H		16	78	32	56H		56	110	96
17H		17	79	34	57H		57	111	98
18H	M1		57	34	58H	M5		61	98
20H		20	80	36	60H		60	112	100
21H		21	81	38	61H		61	113	102
22H		22	82	40	62H		62	114	104
23H		23	83	42	63H		63	115	106
24H		24	84	44	64H		64	116	108
25H		25	85	46	65H		65	117	110
26H		26	86	48	66H		66	118	112
27H		27	87	50	67H		67	119	114
28H	M2		58	50	68H	M6		62	114
30H		30	88	52	70H		70	120	116
31H		31	89	54	71H		71	121	118
32H		32	90	56	72H		72	122	120
33H		33	91	58	73H		73	123	122
34H		34	92	60	74H		74	124	124
35H		35	93	62	75H		75	125	126
36H		36	94	64	76H		76	126	128
37H		37	95	66	77H		77	127	130
38H	M3		59	66	78H	M7		63	130

Using iRMK Kernel Calls in iRMX Interrupt Handlers

The Nucleus assigns priorities to iRMX interrupt tasks based on the handler's interrupt level. Less important interrupts are disabled when an interrupt task is running and can be missed. If this is a problem for your application, you can use iRMK calls to signal an ordinary or non-interrupt task. This enables you to control the task's priority. When you use iRMK Kernel calls in an iRMX interrupt handler, you need to create the service task, cause the service task to perform specific functions, and cause the handler to perform specific functions.

Creating the Service Task

You need to create a task to handle the interrupt. When creating the task, set the task priority so that it will not disable lower-level interrupts.

Things to do from the Service Task

From the service task you need to do the following:

1. Use **KN_create_semaphore** or **KN_create_mailbox** to create a Kernel semaphore or mailbox. Store the token in your application's global memory referenced by the DS.
2. Call **set_interrupt** with `interrupt_task_flag` set to 0. This indicates there is no associated iRMX interrupt task.

   Specify in `interrupt_handler_ds` whether the handler's DS is self-loaded (null selector) or loaded using **enter_interrupt**.
3. Enter an infinite loop in which you wait at the Kernel semaphore or mailbox for notification of an interrupt. Process the interrupt then wait again, etc.

Things to do from the Handler

From the interrupt handler you need to do the following:

1. Load your own DS or use **enter_interrupt** to load DS from which you access the Kernel semaphore or mailbox which will signal your service task.
2. Obtain a scheduling lock by using **KN_stop_scheduling** prior to signaling the task. This prevents a task switch from immediately occurring as a result of the signaling call. If a task is waiting, it will be made ready but will not run immediately.
3. After doing the required handler-level processing use **KN_send_unit** or **KN_send_data** to signal the task that handles the interrupt.
4. Send an End of Interrupt (EOI) to the interrupt controller by using the **exit_interrupt** call.

5. Release the scheduling lock by using **KN_start_scheduling**. This call resumes normal scheduling. Under normal scheduling the highest priority ready task runs. If **KN_start_scheduling** causes an immediate task switch, the remainder of **KN_start_scheduling** and the rest of the handler code will not execute until the originally interrupted task gets to run again. For this reason, you should place the **KN_start_scheduling** call just prior to the interrupt return (IRET).

Example Using iRMK Kernel Calls in iRMX Interrupt Handlers

The following code shows how you can use iRMK Kernel Calls in iRMX interrupt handlers:

```
void interrupt IntHdlr(void)
{
UINT_16 local_status;

/*
* Call RQ_enter_interrupt if the handler requires access to a
* specific application Data Segment.  The segment is specified in the
* call to RQ_set_interrupt, which establishes the handler.
*/

/*
* perform any handler level interrupt processing here
*/

/*
* The handler will now signal an ordinary iRMX task which is waiting at
* a Kernel semaphore.
* Get scheduling lock prior to making the signaling call.
*/

KN_stop_scheduling();

/*
* The KN_stop_scheduling call prevents a task switch from immediately
* occurring as a result of KN_send_unit.
* If a task is waiting at knsemaphore, it will be made ready but will
* not run immediately.
*/

KN_send_unit(knsemaphore);   /* signal ordinary task */

/*
```

```
* The rq_exit_interrupt call sends an End of Interrupt (EOI) to the
* interrupt controller.
*/

rq_exit_interrupt(IntLevel,&local_status);

/*
* Release the scheduling lock and resume normal scheduling.
* At this point the highest priority ready task will run, possibly
* even before the return from KN_start_scheduling.
*
* If KN_start_scheduling causes an immediate task switch, the remainder
* of KN_start_scheduling and the rest of the handler code will not be
* executed until the *interrupted* task gets to run again.  For this
* reason, the KN_start_scheduling call should be the very last call in
* the handler, just prior to the interrupt return (IRET).
*/

KN_start_scheduling();

}
```

Interrupt Servicing Patterns

Figure 10-2 on page 145 illustrates the relationships between the servicing patterns of interrupt handlers and interrupt tasks.

The handler performs the simple, less time-consuming functions; it signals the interrupt task to perform more complicated functions. The handler sends information to the task in data buffers. The number of pending interrupts influences when and how interrupts are disabled.

An interrupt handler might call an interrupt task sometimes, but not every time. For example, an interrupt handler may put characters entered at a terminal into a buffer. If the character is an end-of-line character, or if the character count maintained by the interrupt handler indicates the buffer is full, the interrupt handler calls **signal_interrupt** to activate the interrupt task to process the contents of the buffer. Otherwise, the interrupt handler calls **exit_interrupt** and then returns control to application tasks.

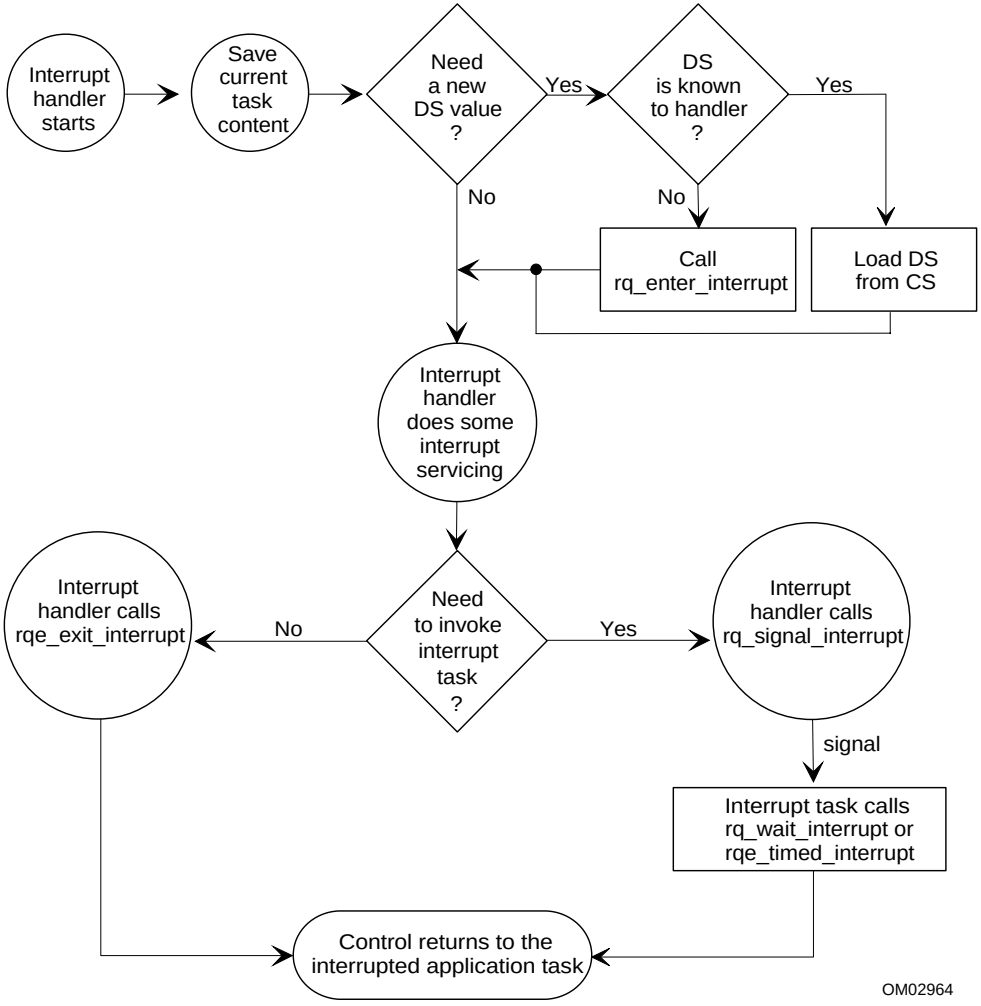


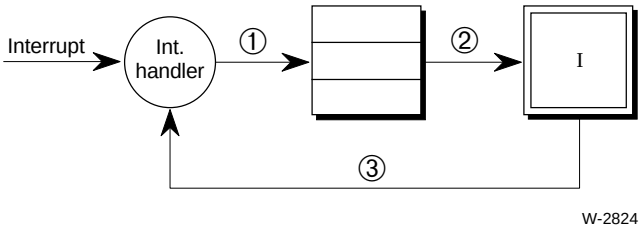
Figure 10-2. Flow Chart of Interrupt Handling

Single Buffer Example

An interrupt handler that reads data from an external device, character by character, and places the characters into a buffer is an example of a single-buffer interrupt handler. When the buffer fills, the handler calls **signal_interrupt** to signal an interrupt task to further process the data. There is only one buffer for the data, so the interrupt level associated with the interrupt task must be disabled while the task is processing.

Because the task called **set_interrupt** with `max_interrupts` equal to 1, the OS automatically disables the interrupt level when the handler invokes **signal_interrupt**.

This prevents the interrupt handler from destroying the contents of the buffer by continuing to place data into an already full buffer. Figure 10-3 illustrates single buffering.



1. The handler places data into the buffer.
2. When the buffer is full, the handler calls **signal_interrupt** to start the task.
3. Upon completion, the task calls **wait_interrupt** or **rqe_timed_interrupt**.

Figure 10-3. Single-Buffer Interrupt Servicing

If you require only single buffering in interrupt servicing, specify 1 for the `interrupt_task_flag` parameter in **set_interrupt**.

Multiple Buffer Example

In this example, the interrupt handler and the interrupt task provide the same functions as in the previous example, but they use multiple buffers. In this case, the interrupt level associated with the task need not always be disabled while the task runs. Instead, the task can process a full buffer while the handler continues to accept interrupts. When the handler fills a buffer, it calls **signal_interrupt** to start the interrupt task, as in the first example. However, because the `interrupt_task_flag` is greater than 1, the interrupt level is not disabled. Instead, the handler continues to accept interrupts, placing the data into the next empty buffer.

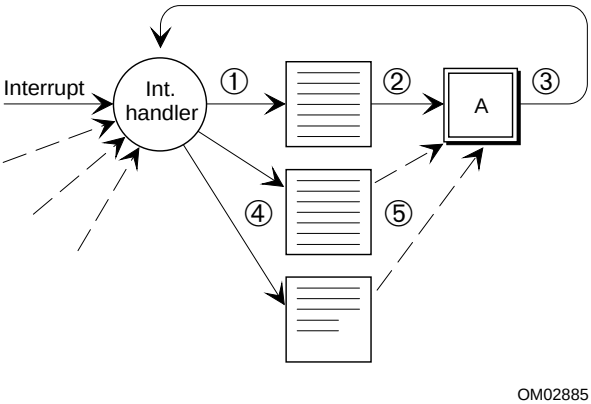
While this occurs, the interrupt task processes the full buffer. When the task completes the processing, it calls **wait_interrupt** or **rqe_timed_interrupt** to indicate that it is ready to accept another **signal_interrupt** request (another full buffer) and to indicate that the buffer it just finished processing is available for re-use by the handler.

Because the handler and the task are running somewhat independently, the handler may fill a buffer and call **signal_interrupt** before the task has finished processing the previous buffer. To prevent the **signal_interrupt** request from becoming lost, the OS maintains a count of pending interrupt requests. Each time the handler calls **signal_interrupt**, the count of pending interrupts is incremented by one. Each time the task calls **wait_interrupt** or **rqe_timed_interrupt**, the count of pending interrupts decrements by one. You can use the SDB **vt** command to view an interrupt task and the count of pending interrupts.

See also: **vt** command, *System Debugger Reference*

If the count of pending interrupts is still greater than 0 after the interrupt task calls **wait_interrupt** or **rqe_timed_interrupt**, the task does not wait for the next **signal_interrupt** to occur before resuming execution. Instead, it immediately starts processing the next full buffer. Neither the interrupt task nor the interrupt handler has to wait for the other. The interrupt handler can continually respond to interrupts without having the task disable the interrupt level. The interrupt task can continually process full buffers of data without waiting for the handler to call **signal_interrupt**.

Figure 10-4 illustrates this multiple buffering handler.



1. The interrupt handler starts filling the empty buffer.
2. The handler calls **signal_interrupt** to start the task on processing the full buffer.
3. The interrupt task processes the full buffer, then calls **wait_interrupt** or **rqe_timed_interrupt** to wait for the next full buffer.
4. The handler keeps filling buffers.
5. The task keeps processing them and calling **wait_interrupt** or **rqe_timed_interrupt**.

Figure 10-4. Multiple-Buffer Interrupt Servicing

Table 10-3 describes the actions of the handler and the task. The table is divided into three parts: actions of the interrupt handler, actions of the interrupt task, and the count of pending interrupts specified in the `interrupt_task_flag` parameter of **signal_interrupt**. The count is set to three. The table shows the actions of both the handler and the task through time, and the change in value of the count.

Table 10-3. Handler and Task Interaction through Time

Time	Interrupt Handler	Interrupt Task	Count
		Task A calls **set_interrupt** to set handler and task for level, setting interrupt_task_flag to 3.	0
		A calls **wait_interrupt** to wait for first request from handler.	0
Intrpt	Handler processes interrupt, starts filling first buffer.		
Intrpt	Process interrupt. Buffer is full. Call **signal_interrupt**.		
		A starts processing 1st full buffer.	1
Intrpt	Process interrupt. Start filling next buffer.		
Intrpt	Process interrupt. Buffer is full. Call **signal_interrupt**.		2
Intrpt	Process interrupt. Start filling next buffer.		2
Intrpt	Process interrupt. Buffer is full. Call **signal_interrupt**. Count is 3. Interrupt level is disabled.		3
		Call **wait_interrupt**. Start processing next buffer.	2
Intrpt	Process interrupt. Buffer is full. Call **signal_interrupt**.		3
		Call **wait_interrupt**. Start processing next full buffer.	2

The interrupt task, when it initially calls **set_interrupt**, specifies the number of pending interrupt requests in the `interrupt_task_flag` parameter. When the interrupt handler calls **signal_interrupt**, causing the number of pending interrupts to be incremented to the maximum:

- The interrupt level is disabled; the handler won't receive further interrupts until the interrupt task makes a **wait_interrupt** or **rqe_timed_interrupt** call, which reduces the number of pending interrupts below the maximum. The OS then enables the level.
- The E_INTERRUPT_SATURATION condition code returns from **signal_interrupt** to the handler, indicating that the number of pending interrupts limit has been reached. The only exception to this rule is if the **set_interrupt** call limit is 1; then **signal_interrupt** will not return the E_INTERRUPT_SATURATION condition code. The level is disabled until the task calls **wait_interrupt** or **rqe_timed_interrupt** and decrements the number of pending interrupts below the limit specified in `interrupt_task_flag` in **set_interrupt**. The interrupt level is enabled, allowing the handler to resume accepting interrupts.

Always set `interrupt_task_flag` equal to the number of buffers that the task and handler use. If the task sets `interrupt_task_flag` larger than the number of buffers, the handler will accept interrupts when no buffers are available and data will be lost. If the task sets `interrupt_task_flag` smaller than the number of buffers, there will always be empty buffers and space will be wasted.

For example, if you need one buffer, set `interrupt_task_flag` to one. In this case, the Nucleus disables the interrupt level while the task is processing the buffer. If you need two buffers, set `interrupt_task_flag` to two. Then, the handler can fill one buffer while the task is processing the other. Additional buffers require correspondingly higher limits. However, if the task sets the limit to 0, the interrupt handler operates without an interrupt task.

Disabling Interrupts

The Nucleus masks less important interrupts automatically while the interrupt task is running. Occasionally you may want to prevent interrupt signals from causing an immediate interrupt at the task's own level. For example, in a device driver finish procedure, you may want to disable interrupts from the device before deleting resources an interrupt handler or task would require. You can disable each interrupt level except the system clock. You disable a level by using the **disable** system call. Or you can set the `interrupt_task_flag` parameter in **set_interrupt** to 1.

If the level is disabled, the interrupt signal is blocked until the level is enabled, at which time the signal is recognized by the CPU. However, if the signal is no longer emanating from its source, it is not recognized and the interrupt is not handled.

If the associated interrupt level is disabled while the interrupt task is running and the number of outstanding **signal_interrupt** requests is less than the limit you specified in `interrupt_task_flag`, the call to **rqe_timed_interrupt** or **wait_interrupt** enables that level.

An interrupt level can be disabled in these ways:

- A task can disable a specific interrupt level by calling **disable**, then re-enable the level by calling **enable**.
- The number of pending interrupts received can reach the limit you set in the **set_interrupt** system call. Whenever this happens, the OS automatically disables the interrupt level until the number of pending interrupts falls below the maximum.
- When a task calls **reset_interrupt** to cancel the assignment of a particular interrupt handler to a particular interrupt level, the OS automatically disables that interrupt level. If there is an interrupt task for the level, **reset_interrupt** deletes it. **Delete_task** does not delete interrupt tasks.
- To provide preemptive priority-based scheduling, the OS can automatically disable or re-enable some interrupt levels whenever a task begins running, depending on the priority of the new running task and the priority of the interrupt level. This enables high-priority tasks to run faster, without interrupts from lower-priority external devices. Table 10-4 on page 152 shows the correlation between the levels disabled and the priority of the running task.

Table 10-4. Interrupt Levels Disabled for Running Task

Task Priority	Disabled Levels Slave	Master	Task Priority	Disabled Levels Slave	Master
0-2	00 - 77	M0 - M7	65-66	40 - 77	M4 - M7
3-4	01 - 77	M1 - M7	67-68	41 - 77	M5 - M7
5-6	02 - 77	M1 - M7	69-70	42 - 77	M5 - M7
7-8	03 - 77	M1 - M7	71-72	43 - 77	M5 - M7
9-10	04 - 77	M1 - M7	73-74	44 - 77	M5 - M7
11-12	05 - 77	M1 - M7	75-76	45 - 77	M5 - M7
13-14	06 - 77	M1 - M7	77-78	46 - 77	M5 - M7
15-16	07 - 77	M1 - M7	79-80	47 - 77	M5 - M7
17-18	10 - 77	M1 - M7	81-82	50 - 77	M5 - M7
19-20	11 - 77	M2 - M7	83-84	51 - 77	M6 - M7
21-22	12 - 77	M2 - M7	85-86	52 - 77	M6 - M7
23-24	13 - 77	M2 - M7	87-88	53 - 77	M6 - M7
25-26	14 - 77	M2 - M7	89-90	54 - 77	M6 - M7
27-28	15 - 77	M2 - M7	91-92	55 - 77	M6 - M7
29-30	16 - 77	M2 - M7	93-94	56 - 77	M6 - M7
31-32	17 - 77	M2 - M7	95-96	57 - 77	M6 - M7
33-34	20 - 77	M2 - M7	97-98	60 - 77	M6 - M7
35-36	21 - 77	M3 - M7	99-100	61 - 77	M7
37-38	22 - 77	M3 - M7	101-102	62 - 77	M7
39-40	23 - 77	M3 - M7	103-104	63 - 77	M7
41-42	24 - 77	M3 - M7	105-106	64 - 77	M7
43-44	25 - 77	M3 - M7	107-108	65 - 77	M7
45-46	26 - 77	M3 - M7	109-110	66 - 77	M7
47-48	27 - 77	M3 - M7	111-112	67 - 77	M7
49-50	30 - 77	M3 - M7	113-114	70 - 77	M7
51-52	31 - 77	M4 - M7	115-116	71 - 77	None
53-54	32 - 77	M4 - M7	117-118	72 - 77	None
55-56	33 - 77	M4 - M7	119-120	73 - 77	None
57-58	34 - 77	M4 - M7	121-122	74 - 77	None
59-60	35 - 77	M4 - M7	123-124	75 - 77	None
61-62	36 - 77	M4 - M7	125-126	76 - 77	None
63-64	37 - 77	M4 - M7	127-128	77	None

Enabling Interrupt Levels from within a Task

Sometimes, an interrupt task may finish with a buffer of data before it finishes its processing. An example of this is a task that processes a buffer and then waits at a mailbox, possibly for a message from a user terminal, before calling **wait_interrupt**. If other buffers of data are available to the handler (the number of pending interrupts has not reached the limit), this does not present a problem. The handler can continue accepting interrupts and filling empty buffers. However, if the interrupt task is processing the last available buffer (i.e., the limit has been reached), the interrupt handler will not receive further interrupts because the interrupt level is disabled. This may be an undesirable situation if the interrupt task takes a long time before calling **wait_interrupt**.

To prevent this situation, the interrupt task can call **enable** immediately after it processes the buffer, enabling its associated interrupt level. This means that while the task engages in its time-consuming activities, the interrupt handler can accept further interrupts and place the data into the buffer just released by the task. You can use this technique whenever the limit is 1, whether or not you use a buffer.

However, if the interrupt handler fills the buffer and calls **signal_interrupt** before the task calls **wait_interrupt**, these events occur:

- The count of outstanding **signal_interrupt** requests is incremented, causing it to exceed the limit you specified.
- The condition code E_INTERRUPT_OVERFLOW is returned to the interrupt handler to indicate this overflow.
- The interrupt level is again disabled. The interrupt task cannot explicitly enable the level again until the count falls to or below the limit.

If the interrupt task calls **enable** when the count is below the limit, nothing happens and no exception code is returned. However, if the interrupt task tries to enable the interrupt level when the count is greater than the limit, the **enable** system call returns the E_CONTEXT condition code.

If a task other than an interrupt task tries to enable the level, one of three events may occur:

- If the level is already enabled, the **enable** system call returns the E_CONTEXT condition code.
- If the noninterrupt task tries to enable the level (presumably following a **disable**) and the interrupt task is not running (i.e., the interrupt task has called **wait_interrupt** and is waiting for a service request), the level is enabled immediately.
- If the interrupt task is running, the enable does not take affect until the interrupt task next invokes **wait_interrupt**.

Handling Spurious Interrupts

When a PIC receives a signal from an interrupting device, it informs the iRMX OS of the interrupt level. If the interrupting device sends interrupt signals of short duration (that is, the input line is active for very short periods), the interrupt signal might be gone when the PIC tries to determine the interrupt level. If this happens, the PIC cannot determine the interrupt level and thus treats the interrupt as a spurious interrupt.

Each time the PIC detects a spurious interrupt, it responds as if a level 7 interrupt had occurred. Thus, if a master PIC detects a spurious interrupt, it responds as if the interrupt occurred on level M7. If a slave PIC detects a spurious interrupt (for example, a slave connected to master level M3), it responds as if the corresponding level 7 interrupt occurred (in this case, level 37).

A spurious interrupt indicates a problem; the PIC detected an interrupt signal but was unable to determine the level.

Your application system should provide some means of isolating spurious interrupts to prevent further problems (such as falsely responding to a level 7 interrupt). This involves judiciously selecting interrupt levels and adding code to all level 7 interrupt handlers (handlers that service master level M7 or slave levels *x*7, where *x* ranges from 0 through 7). Once the spurious interrupt has been isolated, the level 7 interrupt handler can either attempt to correct the problem or ignore the spurious interrupt and resume system processing.

In either case, before the handler returns control it should call **exit_interrupt** to clear the hardware.

These sections describe several options for isolating spurious interrupts.

Calling get_level

One way that a level 7 interrupt handler can check for spurious interrupts is by invoking the **get_level** system call as soon as the handler starts running. **Get_level** returns the level of the highest priority interrupt that a handler has started but not yet finished processing. If the level returned is not the level associated with the interrupt handler, the interrupt is spurious.

This method is simple to implement, but it does take more handler time to execute **get_level**. Your handlers may have speed requirements that prohibit the use of **get_level**.

Judicious Selection of Interrupt Levels

Another way to isolate spurious interrupts is to avoid connecting devices to level 7 interrupts (master level M7 and slave levels *x*7, where *x* ranges from 0 to 7). If you have no devices connected to these levels, and thus no handlers servicing them, spurious interrupts will not affect system operation. Instead, each time a spurious interrupt occurs, the PIC reacts as if a level 7 interrupt had occurred and sends control to the appropriate interrupt table entry. Because no handler is associated with level 7, that entry contains a pointer to the default handler, which returns control to the interrupted task.

Examining the In-service Register

Another way that a level 7 interrupt handler can check for spurious interrupts is by immediately reading the ISR (In-Service Register) of the corresponding PIC. If the BYTE value obtained from that register does not have a 1 in the high-order bit, the interrupt is spurious. To read the value, the handler must know the port address of the ISR. In PL/M, these lines perform this check when placed at the beginning of the interrupt handler:

```
if ((inbyte (port address of ISR)) & 0x80) == 0
        interrupt is spurious
```

Only use this method of isolating spurious interrupts as a last resort. It requires the handler to know the address of the ISR, which may vary from system to system.

Interrupt System Calls

These are the system calls that relate directly to interrupts.

set_interrupt
rqe_set_interrupt
reset_interrupt
exit_interrupt
signal_interrupt
rqe_timed_interrupt
wait_interrupt
enable
disable
get_level
enter_interrupt

Table 10-5 lists common operations for interrupts and the system calls that perform the operations.

Table 10-5. Interrupt System Calls

Operation	Description
assign handler	**Set_interrupt** assigns an interrupt handler and, if desired, assigns an interrupt task to an interrupt level.
assign shared handler	**Rqe_set_interrupt** assigns an interrupt handler and, if desired, assigns an interrupt task to an interrupt level which is being shared by multiple devices.
remove interrupt level	**Reset_interrupt** cancels the assignment made to a level by **set_interrupt** and, if applicable, deletes the interrupt task for that level.
send EOI	**Exit_interrupt** sends an EOI signal to the PICs. *
invoke task	**Signal_interrupt** invokes interrupt tasks and sends an EOI signal to PICs. *
put task to sleep	**Rqe_timed_interrupt** puts the calling interrupt task to sleep for a specified time. The task awakens either when the specified time elapses or **signal_interrupt** is called.
suspend task	**Wait_interrupt** suspends the calling interrupt task until it is called by an interrupt handler using **signal_interrupt**.
enable level	**Enable** enables an external interrupt level.
disable level	**Disable** disables an external interrupt level.

continued

***** If the interrupt is on a slave, this call sends the EOI to the slave and the master.

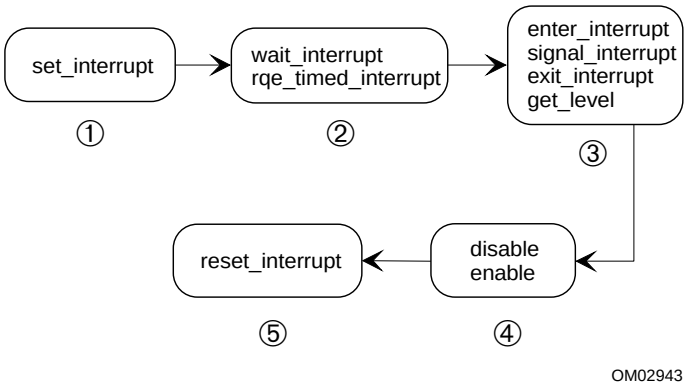
Table 10-5. Interrupt System Calls (continued)

Operation	Description
get current level	**Get_level** returns the interrupt level of highest priority for which an interrupt handler has started but has not yet finished processing.
set up segment	**Enter_interrupt** sets up a previously designated data segment base address for the calling interrupt handler.

See also: Nucleus system calls, *System Call Reference*

How to Use Interrupt System Calls

Figure 10-5 shows the order in which you make interrupt system calls.



1. Make this call from the interrupt task.
2. Make these calls from the interrupt task.
3. Make these calls from the interrupt handler.
4. Make these calls from the interrupt task.
5. Make this call from the interrupt task.

Figure 10-5. Interrupt System Calls

□□□

Descriptors 11

What is a Descriptor?

The Nucleus assigns each object a descriptor when it is created. Each descriptor is an entry in the Global Descriptor Table (GDT); it contains the physical base address, the access rights, and the segment size of a given segment. The descriptors are managed by the OS, which uses them to address an area of memory. Every segment must have at least one descriptor, or the segment is not addressable. Figure 11-1 shows how the 16-bit selector indicates an entry in the Global Descriptor Table. The descriptor contains a base address, to which the processor adds the offset part of the logical address, forming an address in physical memory.

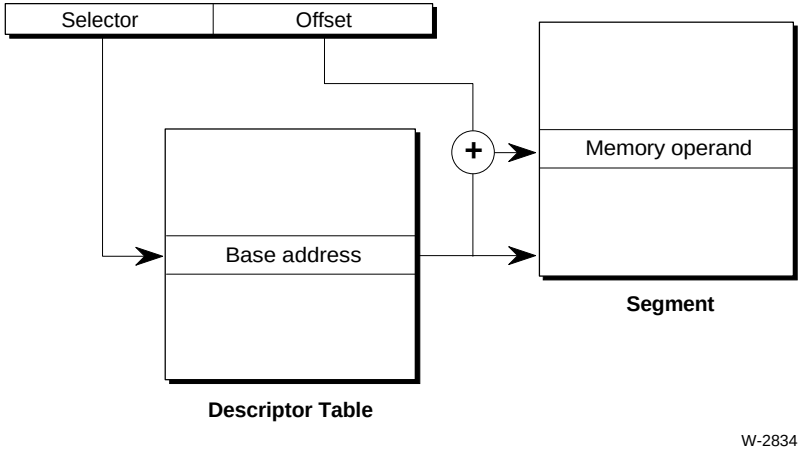


Figure 11-1. Descriptor and Offset Used To Access a Segment's Physical Memory

Note

The paging mechanism that you use with flat model applications forms addresses differently than in the previous figure.

See also: Segments, in this manual

Advanced Uses for Descriptors

The OS enables you to access physical memory anyplace you want to.

> ⚠ **CAUTION**
>
> Descriptors are very powerful. If misused, they can affect the integrity of the entire OS and can corrupt the interaction between tasks in an application system. Do not use descriptors unless you are an experienced programmer with full understanding of iRMX addressing.

You can create, change, and delete descriptors just like segments. To the OS, they look like segments. If you call **get_type** and specify a descriptor, the type code returned is for a segment.

Advanced uses of descriptors are:

- To address areas of memory that are not defined when the system is configured and are therefore excluded from the OS. You might do this for a VGA controller. A device driver can access the controller using a descriptor you create.
- To create aliases for existing segments. Aliases enable you to define a different segment type or different access rights for the same segment. You might have a piece of code that requires changing in the course of processing. You could create a read/write segment that you write a change to in the course of processing. After the change, you could use a descriptor to create an alias for the segment so you could execute it.

Descriptors for Undefined Memory

When you configure the OS, you specify which areas of memory the it uses. The memory pools for dynamic allocation to jobs comes from this memory. You can use descriptors to address areas of memory that were not defined when the system was configured. These memory areas are not allocated from the Free Space Manager (FSM), or from the job's memory pool. When you create them, they do not reduce the size of the job's memory pool, nor do they reduce the size of free space. When you delete them, only the GDT entry is affected; the memory that was referenced by the descriptor remains outside of the control of the OS.

See also: Memory pools, in this manual

Descriptors with Aliases

You can use descriptors to alias existing segments. Aliases provide segments with alternate names and access rights.

Using rqe_create_descriptor

CAUTION

Only use **rqe_create_descriptor** when you need to alias memory already allocated to a job as an object, usually a segment, or when you need to access memory outside the FSM.

Be careful! You can create a descriptor for any physical address; if you make an error in calculating the address, you may corrupt system and user data and overwrite program code.

You specify the full 32-bit physical base address and the segment size in **rqe_create_descriptor**. The segment can lie anywhere in available memory, even outside the range managed by the OS. The memory can overlap that contained in other segments, if desired. The OS automatically sets up the new segment as a data segment with read/write access.

When you create a descriptor, the Nucleus assigns a slot in the GDT with the physical address and marks the object as a descriptor.

Using rqe_delete_descriptor

When you call **rqe_delete_descriptor**, the Nucleus removes the association between the GDT slot and the memory but does not delete the memory addressed by the descriptor. The system call returns the GDT slot to the OS for re-use.

Using rqe_change_descriptor

This system call is intended for system programs that need to access areas of memory in special ways. You can use **rqe_change_descriptor** to access areas of memory that are not part of the OS and to alias segments, giving you the ability to change segments that were originally read/write segments to execute segments.

Descriptor System Calls

These are the system calls that relate directly to descriptors.

rqe_create_descriptor
rqe_delete_descriptor
rqe_change_descriptor

Table 11-1 describes operations on descriptors and what the related system calls are.

Table 11-1. Descriptor System Calls

Operation	Description
return token for segment	**Rqe_create_descriptor** places a descriptor, including the base physical address and segment size, in the GDT.
delete descriptor	**Rqe_delete_descriptor** removes the association between a GDT slot and an area of memory and returns the slot to the OS for re-use.
change address or segment size	**Rqe_change_descriptor** changes the base address contained in the GDT and/or the size of the segment described.

See also: Nucleus system calls, *System Call Reference*

❑❑❑

Other Nucleus Features 12

Date and Time Subsystem

The iRMX III.2.3 Nucleus provides a Year 2000 compliant date/time system call that allows applications the ability to set and get time and date information from both the local CPU clock as well as from some external Real-Time Clock (RTC) devices.

See: **rqe_time** system call, *System Call Reference*;

Live Insertion Support

The iRMX OS supports the live insertion capability of Multibus II systems. Live insertion allows you to replace or add a Multibus II board in a system with the power on and with minimal disruption to the other boards.

Multibus II live insertion requires a particular Central Services Module (CSM) and backplane. If you are uncertain whether your hardware supports live insertion, contact the manufacturer(s) or the Multibus Manufacturer's Group for information.

In a Multibus II live insertion environment, if an OS on one board depends on another board (such as a file server) or communicates with another board, it must know if that board fails or is reset. The iRMX OS contains mechanisms to detect these conditions and notify your application. Once the application is notified of board failure or reset, it can take action based on recovery procedures specific to your requirements.

Watchdog Timer

The watchdog timer is the main component of iRMX live insertion support. The watchdog timer detects when another board fails or is reset and informs applications of the event.

The watchdog timer on each board performs the following functions:

- Periodically broadcasts an existence message to inform other boards in the system that this board exists.
- Monitors the existence messages of other boards in the system to determine when they fail or are reset.
- After receiving an existence message from a board, sets an alarm period and waits for the next existence message. If the board's alarm period expires before

the expected existence message arrives, the watchdog timer assumes that the remote board has failed and notifies applications on its own board.

- Examines each incoming existence message to determine the slot ID of the sender. If the sending board has an alarm associated with it, the watchdog timer deletes the alarm before it expires. The watchdog timer checks the incarnation number in the existence message to determine if the remote board has been reset. If the remote board has been reset, the watchdog timer notifies applications on its own board and creates an alarm for the remote board.

Existence messages include the board ID (slot ID) and incarnation number. The incarnation number gives the watchdog timer enough information to determine if a remote board has been reset since the last existence message.

Reconfiguration Mailboxes

Reconfiguration mailboxes let your application receive notification of board failure or reset in the system. You can design a recovery task in your application to act on the type of failure.

To create a reconfiguration mailbox, first create a data mailbox with the **rq_create_mailbox** system call. Then use the **rq_add_reconfig_mailbox** system call to assign it as a reconfiguration mailbox.

See also: **rq_add_reconfig_mailbox**, **rq_create_mailbox** system calls, *System Call Reference*;
Reconfiguration Mailboxes; Chapter 3

Failure Handling

The watchdog timer on one board can detect that any board in the Multibus II system has either failed or been reset, if those other boards have also enabled a watchdog timer.

If the watchdog timer detects a failure, it informs all reconfiguration mailboxes on its board with the appropriate message:

- For a remote host failure, it sends out a WD_HOST_FAILURE message to all reconfiguration mailboxes on its own board. The WD_HOST_FAILURE message indicates that the alarm expired without receiving an existence message from the remote board.

- For a remote host reset, it first sends out a WD_HOST_FAILURE message to all reconfiguration mailboxes for all incarnations of that board starting with the last known incarnation and up to but not including the current incarnation number. Following this message is a WD_HOST_RESET for the current incarnation. The WD_HOST_RESET indicates that the incarnation number in the existence message is not the same as previously received from the remote board.

See also: **rq_add_reconfig_mailbox**, *System Call Reference*

Internal Failure Recovery

The operating system has internal procedures to handle WD_HOST_FAILURE messages and WD_HOST_RESET messages. Currently, the ATCS 279/ARC server and client(s) use this mechanism.

Application Failure Recovery

You can assign any data mailbox as a reconfiguration mailbox by using the **rq_add_reconfig_mailbox** system call. Write a recovery procedure to wait at each reconfiguration mailbox. When the watchdog timer sends a message to reconfiguration mailboxes on a host, your recovery procedure can respond as required.

For example, suppose you have a client weather station that receives weather data from a number of server collection stations. You could write a procedure that would keep your client from asking for data from a server that was not operating, and that would begin asking for data from that server again after it came back on line. The following pseudocode example shows how you might create a reconfiguration mailbox and use it to begin such a recovery procedure.

```
monitor_task()
{
RQ_TOKEN              mbox;
UINT_32               msg_size;
RQ_RECONFIG_MSG_STRUC message;

      mbox=RQ_create_mailbox(0x20,&exception);
      RQ_add_reconfig_mailbox(mbox,&exception);

      /*wait for failure or reset message*/
      FOR (;;)
      {
      msg_size=RQ_receive_data(mbox,&message,RQ_WAIT_FOREVER,
               &exception);

        /* If get failure message, perform server failure
               procedure*/
        IF (message.msg_type==RQ_HOST_FAILURE)
               server_fail(message.host);

        /* If get reset message, perform server recovery
               procedure*/
        ELSE IF (message.msg_type==RQ_HOST_RESET)
               server_recover(message.host);
      }
}
```

See also: **rq_add_reconfig_mailbox** system call, *System Call Reference*; Reconfiguration Mailboxes, Chapter 3

Configuring the Watchdog Timer

You set up the watchdog timer on the (MBII) Multibus II hardware screen of the ICU. From this screen you can:

- Enable or disable the watchdog timer.
- Specify the number of reconfiguration mailboxes that can be in use simultaneously. Allow enough for any ARC server and each ARC client on your board in addition to the number needed by your application.
- Set the time the board waits between each broadcast of its existence message.
- Set the time the board waits for the next existence message from other boards before notifying the reconfiguration mailboxes that the other board failed. Broadcasts of existence messages must occur more often than wait periods. A good ratio to use is two broadcasts for every wait period.

See also: MBII screen, *ICU User's Guide and Quick Reference*

What is Interconnect Space?

Interconnect space is a collection of 512 one-byte registers on every board in a Multibus II system. The registers contain information about the board: the manufacturer, model number, memory configuration, and other board-specific information. The first 32 interconnect registers of every board have an Intel-specified format and are called the *header record*. The hardware specification for the board defines the format of the rest of the interconnect registers.

See also: Architecture of interconnect space, *Multibus II Interconnect Interface Specification*;
the hardware reference manual for your board

How the OS Uses Interconnect Space

The OS uses interconnect space to automate board identification on the parallel system bus (PSB) at system start-up. The interconnect registers configure a board dynamically, replacing many functions previously handled by onboard jumpers. The OS uses interconnect space to determine the available resources and load system utilities as necessary. Most registers are set during system initialization and remain unchanged until the board is reset.

The OS also uses interconnect space when the watchdog timer has been configured into the system. The watchdog timer detects board failures and resets by monitoring certain interconnect space registers.

How an Application Uses Interconnect Space

CAUTION

The interconnect registers are not intended for general run-time communication. Using the interconnect registers during normal system operations may have a severe impact on the system response.

You can corrupt the operation of the board or the system by specifying incorrect values in interconnect registers.

You can read the interconnect registers to determine current board configuration and set them to modify identification, configuration, and diagnostic information. The registers are organized modularly. A group of contiguous registers, called a *record*, describes a single function. To access registers at the record level, you access each register in a record individually. The Nucleus Communication Subsystem (NCS) provides direct read and write access to individual interconnect registers in the system. The NCS provides mutual exclusion on the access to any single interconnect register.

If you want to read or write a series of registers arranged as a record, you must provide mutual exclusion by using a semaphore or region. You must access multiple interconnect registers in a well-known record format.

Referencing Interconnect Space

You reference interconnect space for each board using the board's slot ID in the PSB backplane; the slot number of a host is equal to its host ID. Using slot 31 specifies the host of the calling task, so a task can access registers on its board without knowing the slot number. You also specify the register number.

Reading and Writing Interconnect Space

You can read or write interconnect space from the command line using the **ic** command. This command performs several functions, such as displaying the slot ID and product code for each board in the system and displaying register contents.

See also: **ic** command, *Command Reference*

Interconnect Register System Calls

These are the system calls that you use to access interconnect registers.

set_interconnect
get_interconnect

Table 12-1 lists operations on interconnect registers and what the related system calls are.

Table 12-1. Interconnect Register System Calls

Operation	Description
get settings	**Get_interconnect** gets the value of a specified interconnect register on a host in the specified slot number.
change settings	**Set_interconnect** sets the value of a specified interconnect register on a host in the specified slot number. It will not write a read-only register, but will not return a condition code. Check the general status register, 24, for results.

See also: Nucleus system calls, *System Call Reference*

❑❑❑

OS Extensions and Type Managers 13

How Do You Add a System Call?

If more than one job in your application system requires a function that is not supplied by the OS, you can add the function in these ways:

- Write the function as a procedure and add it to the OS. Invoke the function with a system call you write. This is called extending the OS; the procedures you add are *OS extensions*. This alternative is the subject of this chapter.
- Write the function as a procedure and place it in a library, using the LIB386 librarian utility. After compiling each job that requires the function, bind the library to the object module for the job.
- Write the function as a task and allow application tasks to invoke the function through a mailbox interface.

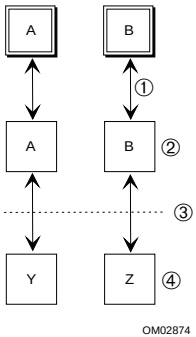
Table 13-1 compares the ways of adding functions.

Table 13-1. Comparing Techniques for Creating System Calls

	OS Extension	Library	Task
Difficulty	Simple	Simple	Complex
Performance	Fair (slow functions) Good (slow functions)	Good (all functions)	Poor (quick functions) Fair (quick functions)
System calls	Both asynchronous and synchronous	Both asynchronous and synchronous	Asynchronous only
Programmer	System	Application	Application
Duplicate code	Avoided automatically	Hard to avoid	Easy to avoid
Relinking	Not required	Required	Not required
New objects	Supported	Not supported	Not supported

Creating an OS Extension

Every OS extension consists of an interface and a function procedure. An entry procedure is optional. Figure 13-1 shows the simplest arrangement of an extension. The figure shows two OS extensions, each containing one system call. There is no entry procedure.



1. The application tasks are linked to the interface procedures.
2. The interface procedures are part of the application software.
3. The interface procedures pass control to the function procedures by using a call gate.
4. The function procedures are part of the system software.

Figure 13-1. OS Extension Using Interface and Function Procedures

Call gates redirect flow within a task from one code segment to another. Each system call uses a call gate to transfer control to the requested function. This makes it possible to go directly from the interface procedure to the function procedure. In an ICU-configurable system, you can specify the GDT slot reserved for call gates using the GSN parameter; in iRMX for PCs and DOSRMX, use the OSX parameter in the *rmx.ini* file. For compatibility between the OSs, use consecutive slots starting with 440.

See also: GSN parameter, *ICU User's Guide and Quick Reference*;
OSX parameter, *System Configuration and Administration*

Interface Procedures

An interface procedure connects your application code to an OS extension call gate. Since they are very small, you can provide an interface procedure for each supported compilation model. The OS provides a library of interface procedures for various compilation models of the Intel iC-386, Watcom C, Microsoft C, and PL/M compilers.

For example, to issue a **new_function** system call, your task executes a statement like

```
new_function (......);
```

This is a call to an interface procedure, named **new_function**, which transfers control to the OS. For each system call in your OS extension, you must write a reentrant interface procedure.

1. The interface procedure uses a call gate to transfer control from the task that invoked the call to a function procedure.

   For example, when transferring control to a function procedure whose call gate number is 441H, the interface procedure is bound to a *.GAT* file produced by BLD386 and then calls GATE 0441, which is the PUBLIC name for this gate. You can find a gate's PUBLIC name in the *mp2* file generated by BLD386.

2. If an entry procedure exists, the interface procedure must give a code to the entry procedure that identifies the function procedure to call. The interface procedure does this by loading the code into a previously designated register or onto the stack of the calling task.

3. The entry procedure, when invoked, extracts the code from this register or the stack.

See also: *Assembly Language Reference*

Function Procedures

The duties of the function procedure are mainly to do what the calling task asks. One function procedure is required for each customized system call. If there is no entry procedure, the function procedure should inform the interface procedure of the system call's exception status by setting CX and DL. Function procedures should be reentrant and can be written in any high-level language or in assembly language.

These are the ways to specify a call gate:

- Using the *.GAT* file created by BLD386
- Using an assembly language macro

See also: Developing applications in assembly language, OS extension example, *Programming Techniques*

Entry Procedures

The entry procedure is associated with a call gate. Each OS extension with multiple system calls assigned to it must include a reentrant entry procedure. Its main purpose is to route the call from the interface procedure to the appropriate function procedure. This procedure is optional.

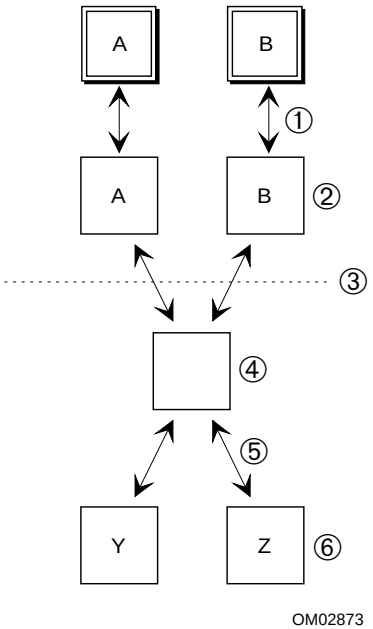
Write the entry procedure in assembly language so you can directly access the stack and the registers. This gives you access to the input parameters passed by the calling task and the interface procedure. It also enables you to set the CX and DL registers in the event of an exceptional condition.

The entry procedure must send a code identifying the function procedure called by the task. The interface procedure does this by loading the code into a previously designated register or onto the stack of the calling task.

Other possible functions of entry procedures are:

- To set up exception handling for the OS extension, if this is needed.
- To perform a routine common to all system calls in this OS extension.
- To transmit the exception incurred by the function procedure back to the interface routine in the CX and DL registers.

Figure 13-2 shows a single OS extension with an entry procedure.



1. The application tasks are linked to the interface procedures.
2. The interface procedures are part of the application software.
3. The interface procedures pass control to the entry procedure by using a call gate.
4. There is one entry procedure for the OS extension.
5. The entry procedure passes control to the designated function procedure.
6. The function procedures are part of the system software.

Figure 13-2. OS Extensions with Entry Procedure

Figure 13-3 summarizes, in algorithmic form, what the procedures do.

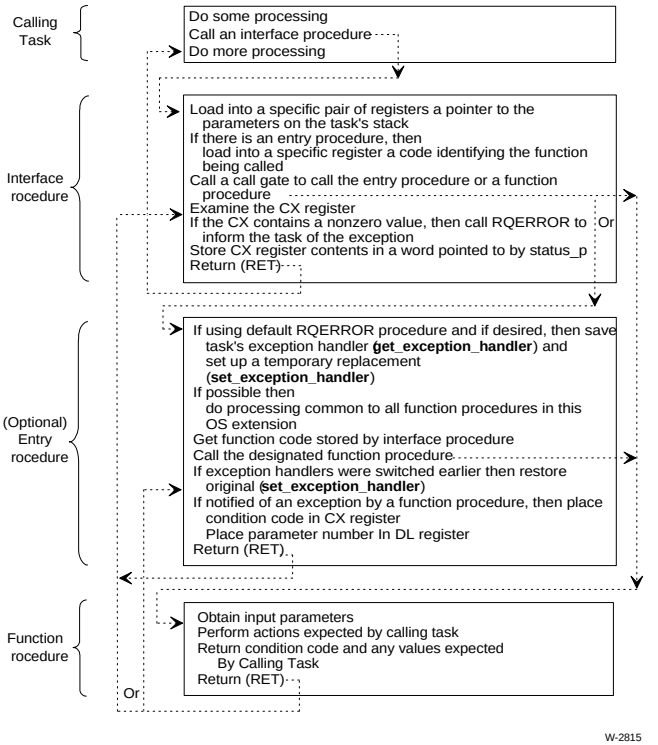


Figure 13-3. Summary of Duties of Procedures in OS Extensions

Exception Handling for Custom System Calls

Exception handling for custom calls usually results in the OS extension calling iRMX system calls. This section lists the appropriate calls.

The interface procedure must inform the calling task (or its exception handler) of any exceptional conditions that occurred:

1. The function procedure places the condition code in the CX register and the number of the parameter that caused the error in the DL register.
2. The interface procedure then checks the CX register for the condition code. If this register contains a value other than 0 (E_OK), an exceptional condition occurred.

3. The interface procedure calls RQERROR, NUCERROR, or a custom exception handler you write, or it handles exceptions inline.

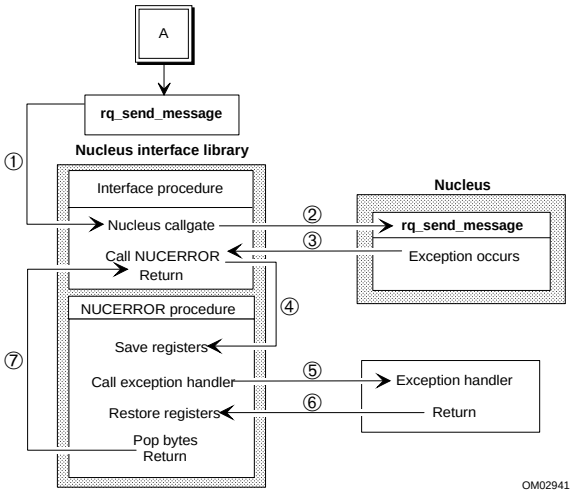
RQERROR and NUCERROR Procedures

RQERROR is a procedure in the iRMX OS that is called by the interface procedures of all iRMX layers except the Nucleus. For example:

1. If a task calls **create_file** and incurs an exceptional condition, the I/O System returns control to the I/O System interface library linked to that task.
2. The interface procedure in that library calls RQERROR to process the error.
3. RQERROR gets the condition code and parameter number from the CX and DL registers and then makes a **signal_exception** system call to inform the calling task (or its exception handler) of the exception.
4. When **signal_exception** returns to the RQERROR procedure, RQERROR restores CX and DL with the condition code and parameter number and places a value of 0FFFFH in the AX register.

   You should link RQERROR to your tasks to ensure that their exception handlers are called when exceptional conditions occur.

NUCERROR performs the same functions for Nucleus interface procedures as RQERROR, except it does not call **signal_exception**. Instead, when a Nucleus system call returns with an exceptional condition, the stack contains extra UINT_16s used to process the exception. They include the exception mode and a pointer to the exception handler. If the mode specifies calling the handler, NUCERROR calls the exception handler directly. Figure 13-4 on page 180 shows the flow of control from an application task to an exception handler when the task incurs an exception.



1. The task makes a call which goes through the interface procedure and call gate.
2. The function procedure is called.
3. An exception occurs and control passes to NUCERROR.
4. NUCERROR saves the CX and DL registers.
5. NUCERROR calls the exception handler to process the exception.
6. The exception handler returns. NUCERROR restores CX and DL and places 0FFFFH in AX.
7. NUCERROR returns, cleaning the stack.

Figure 13-4. Handling Exceptions with an iRMX Exception Handler

Writing Your Own RQERROR or NUCERROR Procedure

If you do not want to use the default RQERROR or NUCERROR procedure provided by the OS, you can write your own. Your procedure can do any functions needed to inform the application task of the exceptional condition, as long as you do this:

- Your RQERROR procedure should place 0FFFFH in AX and then issue a RETURN, returning control directly to the application task to avoid the task's normal exception handler.
- You must always clear three stack words (12 bytes for 32-bit code and 6 bytes for 16-bit code) on return.
- To ensure that your procedure instead of the default version is called, link it directly to the interface procedure or include it in a library with the rest of your interface procedures. When linking modules together, this library should always precede the Nucleus interface library in the link sequence.

The function procedure must change the exception handler from that of the calling task to an exception handler for the OS extension. To make this change:

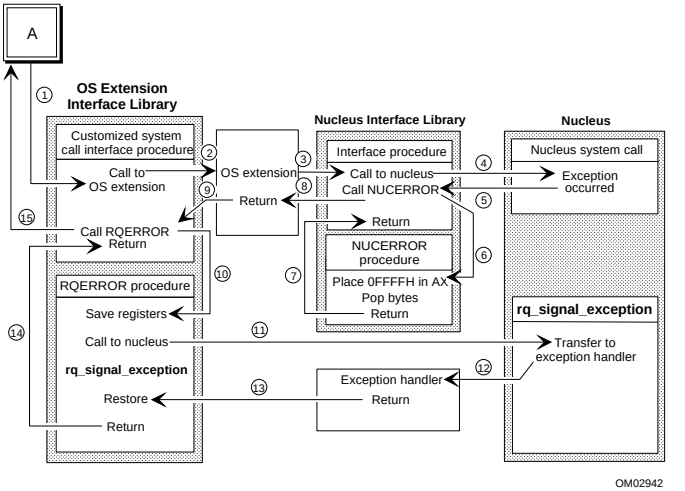
1. The function procedure should first call **get_exception_handler** or **rqe_get_exception_handler** to get and save the task's exception handler address and exception mode.
2. It should call **set_exception_handler** or **rqe_set_exception_handler** to set new values for these entities.
3. Just before returning control to the interface, the function procedure should call **set_exception_handler** or **rqe_set_exception_handler** to restore the original values. In the case of an entry procedure, the entry procedure saves and restores the exception handler and mode.

Handling Exceptions Inline

If you want the OS extension to handle exceptions inline, you can follow the above steps, calling either **set_exception_handler** or **rqe_set_exception_handler** with the `exception_mode` parameter set to NEVER. This is the simplest and most straightforward method. However, it uses the three Nucleus calls listed above upon entry and exit from the function procedure.

Another way of handling exceptions inline is to link your OS extension to your own version of RQERROR or NUCERROR. The RQERROR procedure may simply place 0FFFFH in the AX register (so that 0FFFFH is returned for system calls that are invoked as functions) and then do a RETURN, to return control directly to the interface library. The interface library then returns control to your OS extension, allowing the OS extension to process the exception inline.

Figure 13-5 illustrates the flow of control for an OS extension that incurs an exceptional condition, processes the exception inline, and then returns an exception to the application task that called it. Notice that both the OS extension and the application task, although not linked together, are linked to interface libraries and an RQERROR procedure. The RQERROR procedure linked to the OS extension returns control to the OS extension.



Follow the numbered arrows. These are descriptions of some steps.

6. NUCERROR places 0FFFFH in AX.

7. NUCERROR clears three stack words on return.

10. RQERROR saves the CX and DL registers.

13. RQERROR restores CX and DL places 0FFFFH in AX.

Figure 13-5. Control Flow for Handling Exceptions Inline

Overriding NUCERROR

To override NUCERROR with your own procedure, return from your version of the NUCERROR procedure by popping three stack words using RET 12 for 32-bit or RET 6 for 16-bit code. These words were placed on the stack to use for the call to **rq_signal_exception**.

Even though your OS extension processes its own exceptions inline, you should return exceptions to tasks (or other OS extensions) that invoke the custom system calls. The function procedure of your OS extension should place the condition code

and parameter number in CX and DL, and return to the interface linked to the application task.

Overriding RQERROR

You can provide your own RQERROR routine and bind it to your programs.

> **Note**
> Your routine must contain a public procedure named RQERROR and you must bind the routine to application code before binding the UDI or RMX interface library.

In the BND statement, place the name of the file containing your RQERROR routine before the name of the interface library. This causes your RQERROR routine to be bound in place of the default routine. Your RQERROR routine must adhere to the model of segmentation you used in the application program itself.

The source code of the default UDI RQERROR routine is available in the */rmx386/udi* directory. You can use this source code as an example when building your own RQERROR routine. The file *UCERR.A38* applies only to COMPACT applications.

When the RQERROR procedure invokes **signal_exception**, control can pass to an exception handler. If the default exception handler is in effect, it displays the appropriate error message at the console and can terminate the application.

Establish your own exception handler by calling **rq_set_exception_handler** or **dq_trap_exception**. The new exception handler will be called whenever you invoke **rq_signal_exception**.

After an exceptional condition occurs and before your exception handler gains control, the iRMX OS:

1. Pushes the condition code on the stack of the program that made the system call generating the condition code.
2. Pushes the number of the parameter that caused the exception on the stack (1 for the first parameter, 2 for the second, etc.).
3. Pushes a UINT_16 on the stack (reserved).
4. Pushes a UINT_16 for the NPX on the stack.
5. Initiates a far call to the exception handler.

If the exceptional condition was not caused by an erroneous parameter, the responsible parameter number is 0. If the condition code is E_NDP_ERROR, the fourth item pushed onto the stack is the NPX status word. The NPX exceptions are cleared.

Custom Condition Codes

When you add your own system calls, you may need to add your own exceptional conditions and condition codes. You can use values 4000H to 7FF0H for environmental conditions and 0C000H to 0FFF0H for programmer errors.

Linking the Procedures

For each OS extension, you should produce one library of interface procedures for each segmentation model in which the calling task can be written. Within each library, you should have one interface procedure for each custom system call. Each module in your system should be linked to the appropriate interface library for each OS extension that it calls.

For each OS extension, link all the function procedures (and the entry procedures, if any) along with any OS interface libraries that the procedures need. Do not link them to any application code because they are connected to the application tasks with call gates.

Any RQERROR or NUCERROR procedure that you write should be linked to the appropriate routines:

- To inform the application task of an exception, place your RQERROR procedure in the interface library you create.
- To process exceptions that your OS extension incurs, link your RQERROR or NUCERROR procedure directly to the function procedures.

- Link the iRMX OS interface library, and the interface libraries for any of the other subsystems that you use, to the application task and/or the OS extension, whichever uses these subsystems. If you provide your own RQERROR or NUCERROR procedure, either for your interface procedures to call or to process exceptions in your OS extension, this procedure must precede the iRMX OS interface library in the link sequence.

Including OS Extensions

Before an interface procedure can successfully transfer control to an OS extension, you must establish an entry point. You can add your OS extension to the OS at build time using the ICU or you can add it at boot time using the **sysload** command from the *:config:loadinfo* file or you can add it dynamically.

- For ICU-configurable systems:
  - To only reserve the gate number when you configure the system, enter the next available OS extension slot in the GSN parameter and leave the EPN field blank.
  - To have the OS assign your OS extension to a call gate at build time, fill in both the GSN and EPN parameters.
- For non-ICU-configurable systems, use the OSX loadtime parameter in the *rmx.ini* file to reserve your OS extension slot.
- Use the system call **rqe_set_os_extension** to include extensions dynamically. When you invoke the call, enter the gate number and the start address of the first instruction of your entry or function procedure. You cannot use the same call gate for more than one OS extension simultaneously.

CAUTION

Always reset the OS extension with a null value in the `start_address` parameter first. Then issue the call again with the desired `start_address`. Otherwise, the system will not initialize on a warm reset.

See also: GSN and EPN parameters, *ICU User's Guide and Quick Reference*; OSX loadtime parameter, *System Configuration and Administration*

System Calls for OS Extensions

These system calls are used extensively by OS extensions:

rqe_set_os_extension
signal_exception

Table 13-2 lists operations on OS extension system calls and what the related system calls are.

Table 13-2. OS Extension System Calls

Operation	Description
attach call gate	**Rqe_set_os_extension** attaches the entry point address of the OS extension to a call gate.
signal error	**Signal_exception** advises a task that an exceptional condition has occurred in an OS extension.

See also: Nucleus system calls, *System Call Reference*

Protecting Objects From Deletion

Normally, you delete an object by a call to the **delete_** system call corresponding to the object's type. However, you can use the **disable_deletion** system call to make the object immune to this kind of deletion. A subsequent call to **enable_deletion** removes the immunity. You can use deletion immunity anywhere in your application, not just in OS extensions.

An object can have its deletion disabled more than once. An object's *disabling depth* is the number of times the object has had its deletion disabled.

Each call to **disable_deletion** must be countered by a call to **enable_deletion** before the object can be deleted.

Usually, an object cannot be deleted until its disabling depth is 0. The only exception is that a call to **force_delete** deletes objects whose disabling depth is one. Also, calling **enable_deletion** for an object whose deletion depth is 0 results in the E_CONTEXT condition code.

CAUTION

When you attempt to delete an object whose disabling depth is too high to permit deletion, the deleting task goes to sleep. The task remains asleep until the object's deletion depth becomes small enough to permit deletion. At that time, the object is deleted and the deleting task is awakened. Because these circumstances can cause system deadlock, be careful when deleting objects and when disabling deletion.

Never disable deletion in applications that rely on <Ctrl-C> for program termination.

System Calls for Deletion Immunity

These system calls are used for deletion immunity:

force_delete
enable_deletion
disable_deletion

Table 13-3 lists deletion immunity operations and what the related system calls are.

Table 13-3. Deletion Immunity System Calls

Operation	Description
delete object	**Force_delete** deletes objects whose disabling depths are 0 or 1.
increase disabling	**Disable_deletion** increases the deletion disabling depth of an object by one.
enable deletion	**Enable_deletion** removes one level of deletion disabling from an object.

Type Managers and Custom Objects

Some applications require both custom objects and system calls for manipulating them. A type manager is an OS extension that provides these services. If you require custom objects, you must write a manager for each type. The duties of type managers are:

- Creating objects of the new type.
- Deleting objects of the new type.
- Optionally providing the system calls that your tasks can invoke to create, manipulate, and delete objects of the new type.

This section describes creating and deleting objects of a new type.

See also: Appendix A for an example that creates and deletes objects of a new type;
Extending iRMX, *Real-Time and Systems Programming for PCs* by Christopher Vickery

Creating New Objects

Creating custom objects requires:

- Creating the type
- Creating objects of that type

In **create_extension**, you specify the type code for the new object and whether you want a deletion mailbox. If you specify a mailbox, **delete_extension** and **delete_job** will send composite objects to the mailbox for the type manager to delete. Otherwise, **delete_extension** and **delete_job** will delete composite objects. The **create_extension** system call returns a token for the new type. The token represents a license to create objects of the new type.

The **create_composite** system call creates objects of the new type; it accepts the token returned from **create_extension** as a parameter. **Create_composite** also accepts a list of tokens for the component objects that will compose the new object. It returns a token for the new object, called a *composite object*. Figure 13-6 on page 190 shows the order for creating composite objects.

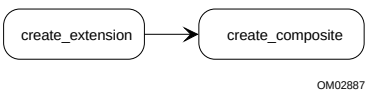


Figure 13-6. Composite Object System Call Order

When you create a composite object:

- Its component objects are all iRMX objects, either provided by the iRMX OS or by other objects you have created.
- No structure is imposed on composite objects of a given extension type. Two objects of the same extension type can be completely different in structure or in the number of component objects they comprise. This feature allows for maximum flexibility in the creation of new objects.

Once a type manager creates a new object type by calling **create_extension**, the type manager owns the type. Only the type manager can create composite objects of that type. In addition, when it creates composite objects, the type manager can request that a token for the composite object be sent back to the type manager when the object has to be deleted.

Deleting Composite Objects and Extension Types

Delete_composite deletes a particular composite object, but not its components. **Delete_extension** deletes a specified extension type, and either deletes all composites of that type or sends them to a deletion mailbox, in which case the type manager must delete them.

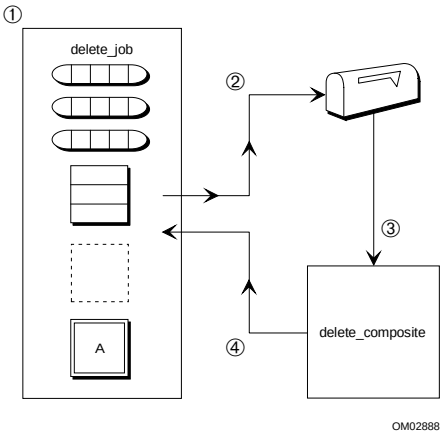
Delete_job, also deletes composite objects as a part of its processing. Although **delete_job** cannot delete extension types (it returns an exception code if the job contains any extension objects), it can delete composites or send tokens for them to deletion mailboxes where their type managers delete them.

Using delete_job

When a task calls **delete_job**, the Nucleus normally deletes every object in the job. However, if the job contains a composite object whose extension has a deletion mailbox, the Nucleus sends the token for the composite object to the deletion mailbox. The Nucleus then waits until the type manager calls **delete_composite** before continuing the deletion process. In that case:

1. The type manager must wait at the deletion mailbox to receive the tokens for the objects to be deleted.
2. It must perform any special processing required to delete the composite object. For example, it might want to wait until all tasks have stopped using the composite.
3. It has the option of deleting those component objects not contained in the job being deleted. It cannot, however, delete any objects contained in the job being deleted or it will incur an exceptional condition. (This is not a problem because the objects in the job being deleted will automatically be deleted during the **delete_job** call.)
4. It must call **delete_composite**, which deletes the composite object (but not the component objects) and informs the Nucleus that the type manager has finished the special processing that deletes the composite object. After the type manager calls **delete_composite**, the Nucleus resumes the **delete_job** processing. If the type manager fails to call **delete_composite**, the **delete_job** system call will not finish processing.

Figure 13-7 shows the type manager's involvement in the **delete_job** process.



1. The task calls **delete_job**.
2. The Nucleus sends the composites to the deletion mailbox.
3. The type manager waits at the mailbox. It performs any cleanup required and calls **delete_composite**.
4. Control returns to **delete_job**.

Figure 13-7. Type Manager Involvement in Delete_job

The type manager is not required to delete all objects. The Nucleus sends the tokens for all other composite objects to their own deletion mailboxes, where their type managers are responsible for deletion. Therefore, all the component objects are eventually deleted, as long as they are in the job being deleted.

In the course of **delete_job**, the Nucleus deletes any Nucleus objects in the job. It sends the tokens for any I/O System, EIOS, or Human Interface objects to their respective deletion mailboxes, where the subsystems themselves delete the objects.

Using delete_extension

You can call **delete_extension** to delete an extension type. Use this call when you no longer need to create composite objects of a given extension type. When you call **delete_extension** and the extension has a deletion mailbox, the Nucleus sends the tokens for all composite objects of that extension type to the deletion mailbox. Then the Nucleus waits until the type manager calls **delete_composite** before sending the next composite to the mailbox. The type manager has responsibilities during **delete_extension** similar to **delete_job**:

1. First, it waits at the deletion mailbox for the objects' tokens.
2. Then, it handles any special processing necessary to delete the object.
3. Finally, it calls **delete_composite** to delete the composite. The type manager must call **delete_composite** for each token it receives at the deletion mailbox. If it does not, the **delete_extension** system call will not finish processing.

However, unlike **delete_job** processing, the type manager has the choice during **delete_extension** of whether or not to delete individual component objects. If it wishes to delete the component objects, the type manager must explicitly delete them. **Delete_extension** does not delete any component objects.

Deleting Nested Composites

A composite object can contain objects of any type, and as a result, some of its component objects may be composite objects themselves. This can cause problems for type managers when they delete the composite objects if the type manager for any of the composite objects depends on the existence of any of the other composite objects to complete its processing.

For example, suppose objects A and B are composites in the same job. They have different extension types, and B is a component of A. Each composite has a type manager that performs special cleanup functions before it can delete the corresponding composite. If neither type manager requires information contained in the other composite to perform its special processing, the deletion process can proceed without difficulty.

However, if the type manager for composite A requires some information contained in composite B to complete its processing, the deletion process becomes more complex. For this deletion scheme to work, you must guarantee that composite A will be deleted before composite B. Thus, you must know the order in which the **delete_job** call deletes objects and sends composites to deletion mailboxes, so that you can set up your composites correctly.

Delete_job deletes composite objects before it deletes non-composite objects. It deletes composite objects on a last-in-first-out basis; that is, in the reverse order from which they were created. Therefore, a type manager can depend on receiving the tokens for composite objects that it creates before the Nucleus deletes the component objects contained in them. The only exception is when a composite (composite A) is created before another composite (composite B), and composite B is inserted as a component into composite A using **alter_composite**. In this case, composite B will be deleted first, and the type manager of composite A cannot rely on the existence of composite B when it receives composite A's token for deletion.

Writing a Type Manager

A type manager consists of two parts:

- The initialization part creates the type and optionally creates a deletion mailbox to which the system can send tokens for objects when deleting either jobs or the type itself.
- The service part provides system calls so tasks can create and manipulate objects of the type.

Because the initialization phase must be completed before any task attempts to obtain tokens for objects, you should execute the initialization part early in the life of the system.

- In an ICU-configurable system, the task should be part of the initialization task of a first-level user job to ensure early execution.
- In non-ICU-configurable systems, make the type manager part of the first sysloaded job. Add the -w option (synchronization), to the **sysload** command and implement the necessary synchronization steps in the type manager's initialization module.

Write the service part of the type manager as an OS extension.

See also: USERJ screen, *ICU User's Guide and Quick Reference*;
sysload command, *Command Reference*

Type Manager System Calls

These are the system calls that you use to manipulate extensions and composite objects:

create_extension
delete_extension
create_composite
inspect_composite
alter_composite
delete_composite

Table 13-4 lists operations on extensions and composite objects and what the related system calls are.

Table 13-4. Type Manager System Calls

Operation	Description
create extension	**Create_extension** creates an extension object that you may use as a license for creating composite objects.
delete extension	**Delete_extension** deletes an extension object and optionally, sends all composite objects of that extension type to the associated deletion mailbox.
create composite	**Create_composite** creates a composite object of a specified extension type.
list components	**Inspect_composite** returns a list of the component object tokens contained in a composite object.
replace component	**Alter_composite** replaces a component in a composite object with either a null or another object.
delete composite	**Delete_composite** deletes a composite object.

See also: Nucleus system calls, *System Call Reference*

□□□

iRMX Kernel Programming Concepts 14

The iRMX Kernel is a part of the Nucleus that provides high performance task and time management and message passing; it enhances the OS. This chapter describes how to use the Kernel within the iRMX OS.

The Kernel does not provide the protection and validation features available in the Nucleus:

- Kernel system calls do not validate parameters. Use Nucleus system calls instead, if you need parameter validation.
- The Kernel assumes that all memory reference pointers it receives are valid.
- Kernel objects are not protected against unexpected deletion.
- The Kernel uses the flat, 4 Gbyte addressing capabilities of the microprocessor. It does not use segmentation.

Use the Kernel in these situations:

- Only for very well-tested code
- For isolated parts of the application
- When performance is critical

It is a good idea to write, test, and debug your application using Nucleus system calls. When the application is correct, substitute Kernel system calls where appropriate.

What Does the Kernel Provide?

Object management	Includes creating, deleting, and manipulating object types defined by the Kernel. You must provide memory for Kernel objects and may allocate memory beyond the Kernel's needs to store application-specific state information associated with the object.
Time management	Includes a real-time clock, alarms that simulate timer interrupts, and the ability to put tasks to sleep.

Task management	Includes scheduling locks that protect the currently running task from being preempted and task handlers, which perform additional functions during task creation, deletion, and transition.
Memory management	Implements memory pools from which it allocates memory in response to application requests.

Kernel Object Management

Each Kernel object type has its own set of operations and unique attributes. The Kernel defines these object types:

- Semaphores
- Mailboxes
- Memory Pools

Note

A Kernel object is not the same as an iRMX object. You cannot use iRMX calls with Kernel objects. Conversely, you cannot use Kernel calls with iRMX objects.

You can create objects anywhere in the application's memory space. You must provide sufficient memory to contain the data structures that define the object. Literals declared in the Kernel's include files specify the amount of memory the Kernel needs for each object type. You may allocate extra memory for the object, to be used by the application.

When you create an object, the Kernel returns a 32-bit kn_token that identifies the object. Thereafter you access the object by passing the kn_token to the appropriate system call.

The Kernel provides system calls to delete objects that you no longer need. Deleting an object removes the Kernel's association of state data with an object.

CAUTION

The Kernel does not protect itself against unexpected object deletion. Do not attempt to access objects either while they are being deleted or after they have been deleted.

Kernel Semaphores

The kinds of Kernel semaphores are:

- FIFO semaphores, which enable tasks to queue in First-In, First-Out order.
- Priority semaphores, which enable tasks to queue in priority order.
- Region semaphores, which are a special type of semaphore with priority adjustment capabilities. Region semaphores are useful for mutual exclusion.

FIFO and Priority semaphores are general-purpose semaphores that can have up to 65,535 units.

See also: Semaphores, Chapter 4

Creating and Deleting Semaphores

Create semaphores with the **KN_create_semaphore** system call and delete them with the **KN_delete_semaphore** system call. To create a semaphore specify:

- The memory area for the semaphore object.
- The kind (priority-based or FIFO task queue, or region semaphore).
- The initial number of units in the semaphore, 0 or 1. If a region is created with 0 initial units, the creating task is the owner of the region and the region cannot be used by other tasks until the creating task sends a unit.

To provide additional units to a semaphore after creation, use the **KN_send_unit** system call once for each additional unit you need. Region semaphores cannot accept more than one unit.

If a semaphore is deleted, all tasks in the semaphore's task queue are awakened with an E_NONEXIST status code.

Sending and Receiving Semaphore Units

Request a unit from a semaphore with the **KN_receive_unit** system call; you must repeat the call for each unit you need. If the semaphore contains units, the count of units decrements by one and the task proceeds. If the semaphore has no units and the task is willing to wait, the task goes to sleep in the semaphore's task queue. Send a unit to a semaphore with the **KN_send_unit** system call. If tasks are waiting at the semaphore, the task at the head of the queue is awakened.

Using Region Semaphores

A region semaphore can provide mutual exclusion and synchronization. If a task must get a unit from a region before entering a critical section, and if it returns the unit when leaving the area, only one task will ever execute in the critical section at a time. Region semaphores contain a maximum of one unit and support priority adjustment.

Kernel region semaphores are similar to iRMX regions. iRMX regions additionally protect a task inside the region from being suspended or deleted.

See also: Regions, Chapter 5

Priority Adjustment

The same priority bottleneck and inversion problems may arise when using non-region Kernel semaphores as Nucleus semaphores. Region semaphores, like Nucleus regions, provide dynamic priority adjustment to avoid blocking a high priority task.

Kernel Semaphore System Calls

These are the system calls you use to manage Kernel semaphores:

KN_create_semaphore
KN_delete_semaphore
KN_receive_unit
KN_send_unit

Table 14-1 lists operations on semaphores and what the related system calls are.

Table 14-1. Kernel Semaphore System Calls

Operation	Description
create semaphore	**KN_create_semaphore** creates a semaphore of the specified type with 0 or 1 initial units.
delete semaphore	**KN_delete_semaphore** deletes the specified semaphore.
request unit	**KN_receive_unit** requests a unit from the specified semaphore.
return unit	**KN_send_unit** adds a unit to the specified semaphore.

Mailboxes

When you create a Kernel mailbox, you may reserve one of the slots in the mailbox message queue for a high priority message. This enables the mailbox to accommodate at least one high priority message even if the queue is full.

Creating and Deleting Mailboxes

Create mailboxes with the **KN_create_mailbox** system call and delete them with the **KN_delete_mailbox** system call. To create a mailbox specify:

- The memory area for the mailbox.
- The message size for the mailbox.
- The maximum number of messages the mailbox can hold.
- Whether the task queue will be priority based or FIFO.
- Whether the mailbox will reserve a slot for a high priority message.

If you delete a mailbox and there are tasks waiting for messages, all tasks are awakened with an E_NONEXIST status code and all messages queued at the mailbox are lost.

Sending and Receiving Mailbox Messages

Send ordinary messages to a mailbox with the **KN_send_data** system call. If a task is waiting at the mailbox, it receives the message. Otherwise, the message is queued. If the mailbox is full, an exception returns.

Send high priority messages with the **KN_send_priority_data** system call. If a task is waiting at the mailbox, it receives the message. Otherwise, the message is placed at the head of the queue. If the mailbox is full, an exception returns.

Specify the actual message size, which must be less than or equal to the maximum message size for the mailbox. The maximum message size is the size you specified when creating the mailbox. The **KN_send_data** and **KN_send_priority_data** system calls return a status value indicating either that the message was accepted or the mailbox was full.

Receive data from a mailbox with the **KN_receive_data** system call. If the mailbox contains at least one message, the message at the head of the queue is returned to the caller. This is either the oldest message or the latest high-priority message. You must provide a message area equal to the maximum message size of the mailbox for the task.

The call returns the actual size of the received message and a status value indicating:

- The task received a message.
- The time limit expired while the task was waiting.
- The mailbox was deleted while the task was waiting.

If no messages are available and the task is willing to wait, the task is put to sleep in the task queue.

Handling Mailbox Overflow

If a mailbox contains its maximum number of messages when a message is sent, the Kernel returns an exception stating that the mailbox limit was exceeded. The mailbox enforces flow control by rejecting messages when the queue is full. Depending on your application, there are several ways to handle mailbox overflow:

- Design the application so mailboxes never overflow.
- Consider mailbox overflow a fatal system error.
- Abort the activity causing the overflow.
- Send the message again, if you know that a task received a message, creating room in the message queue.

If you reserved a slot for a high priority message, mailbox overflow may be indicated when sending an ordinary message, even though the mailbox can still accept a high priority message.

Kernel Mailbox System Calls

These are the system calls you use to manage mailboxes:

KN_create_mailbox
KN_delete_mailbox
KN_receive_data
KN_send_data
KN_send_priority_data

Table 14-2 lists operations on mailboxes and what the related system calls are.

Table 14-2. Kernel Mailbox System Calls

Operation	Description
create	**KN_create_mailbox** creates a mailbox in the specified area.
delete	**KN_delete_mailbox** deletes the specified mailbox.
get message	**KN_receive_data** requests a message from the specified mailbox.
send message	**KN_send_data** sends a message to the specified mailbox. **KN_send_priority_data** sends a high priority message to the specified mailbox.

Kernel Time Management

The Kernel provides time management calls that allow tasks to create alarms (virtual timers) and to sleep for a specified amount of time. The Kernel also provides a real-time clock.

Using the Kernel Tick Ratio

The Kernel uses the Nucleus to provide an external source of periodic signals to implement its time management facilities. All time values in the Kernel are specified in units of clock ticks. You decide the frequency of the clock ticks.

You can use the Kernel Tick Ratio (KTR) parameter to configure support for timed events at a granularity of less than 10 milliseconds (the Nucleus clock tick interval). To use this feature, you must adhere to these rules:

1. You must use Kernel calls when you need an event granularity of < 10 ms. The Nucleus event granularity is still 10 ms.
2. When mixing Kernel and iRMX system calls, remember that a Nucleus tick interval may not equal a Kernel tick interval.
3. The smaller the Kernel tick interval, the higher the system overhead for handling clock interrupts. This does not affect average and maximum interrupt latency.

See also: KTR parameter, *ICU User's Guide and Quick Reference* and *System Configuration and Administration*;
RQSYSINFO structure, *System Call Reference*, to programmatically get the KTR value

The KTR parameter sets the ratio of the Nucleus tick interval (10 milliseconds) to the Kernel tick interval.

KTR	Kernel tick
01	10 milliseconds (default)
02	05 milliseconds
05	02 milliseconds
10	01 millisecond
20	500 microseconds

Do not change the default value of KTR unless you need a Kernel tick interval smaller than 10 milliseconds. The KTR parameter only affects the tick interval in Kernel system calls. The value of KTR does not affect timed wait operations using Nucleus calls.

The Kernel provides a real-time clock by counting clock ticks. The **KN_get_time** and **KNE_get_time** system calls return the current value of the counter. The value of

the real time clock is set to 0 at initialization. You may set the count to any value by using the **KN_set_time** or **KNE_set_time** system calls.

You can measure elapsed time by reading the real-time clock with the **KN_get_time** or **KNE_get_time** system calls at the beginning and end of the interval to be measured. By subtracting, you determine the elapsed time.

Using Alarms

The Kernel lets you to create alarms to simulate timer interrupts. Alarms invoke alarm handlers that you write. Alarm handlers operate in similar fashion to iRMX interrupt handlers. You cannot make blocking calls from them. Alarm handlers run in the context of the timer interrupt handler. Your alarm handler should be as short as possible since it is called with interrupts disabled and scheduling stopped.

Two kinds of alarms exist: single-shot alarms and repetitive alarms. A single-shot alarm invokes its alarm handler once when its time interval elapses. The alarm then becomes inactive and its memory can be re-used. A repetitive alarm invokes its alarm handler after its time interval elapses and then resets itself for the same time interval. It continues to invoke its handler until the alarm is explicitly deleted.

Note

You cannot write alarm handlers in applications that use the flat memory model.

Create an alarm with the **KN_create_alarm** system call and delete it with the **KN_delete_alarm** system call. To create an alarm specify:

- The memory area in which the alarm object will exist.
- Whether it is a single shot or a repetitive alarm.
- The time interval for which the alarm is set.
- A pointer to the application handler that the alarm invokes when the time period elapses.

After a task calls the **KN_delete_alarm** system call, the handler associated with that alarm will no longer be invoked and the memory that the alarm occupies can be re-used. Alarms may be deleted whether or not they have invoked the associated alarm handler. This means that deleting an alarm does not have to be synchronized with the expiration of the alarm.

The **KN_reset_alarm** system call resets an alarm, returning it to its initial state. **KN_reset_alarm** uses the alarm's kn_token; the alarm parameters are not required. Both single shot and repetitive alarms can be reset. Regardless of whether the alarm's time limit has expired, resetting an alarm returns it to its creation state and starts it

running as if it were just set. Resetting a single shot alarm after it has gone off is equivalent to setting the alarm again.

Using Sleep

The Kernel enables tasks to sleep for a specified time, using the **KN_sleep** system call. The amount of time the task will be in the asleep state can vary from no time to forever. If the specified time is KN_DONT_WAIT, the task will not go to sleep. A KN_DONT_WAIT time limit gives the processor to another task of equal priority, if one exists.

KN_WAIT_FOREVER means the task will never wake up. When a task sleeps forever, it is effectively deleted, but its memory is not released. Use the KN_WAIT_FOREVER literal only with blocking system calls, indicating that the task will wait until an event occurs to wake it. For example, a task might wait forever until a message arrives at a mailbox.

Time Management System Calls

These are the system calls you use to manage time:

KN_create_alarm
KN_reset_alarm
KN_delete_alarm
KN_get_time
KN_set_time
KN_sleep
KNE_get_time
KNE_set_time

Table 14-3 lists operations on alarms and time and their related system calls.

Table 14-3. Time Management System Calls

Operation	Description
create alarm	**KN_create_alarm** creates and starts a virtual alarm clock.
delete alarm	**KN_delete_alarm** deletes an existing alarm.
get elapsed time	**KN_get_time** returns the number of clock ticks that have occurred. **KNE_get_time** is an extended version that allows use of 32-bit data types.
reset alarm	**KN_reset_alarm** returns an existing alarm to its creation state.
reset time	**KN_set_time** sets the counter that the Kernel uses to count the clock ticks that have occurred. **KNE_set_time** is an extended version that allows use of 32-bit data types.
put task to sleep	**KN_sleep** puts the calling task in the asleep state for the specified number of clock ticks.

Kernel Task Management

The Kernel uses the same scheme of preemptive, priority-based scheduling as the Nucleus. In addition, it provides ways for controlling or monitoring task switches.

You can protect the currently running task from being preempted by performing a scheduling lock. A scheduling lock provides protection for the running task; the running task can make other tasks ready without losing control of the processor. The Kernel delays a task switch to the running state until the task releases the scheduling lock. Then the running task may be preempted.

Note

Scheduling locks delay the preemptive, priority-based scheduling of the OS. Only use them if absolutely necessary and be very careful.

To lock scheduling, use the **KN_stop_scheduling** system call. Calling **KN_stop_scheduling** multiple times causes multiple scheduling locks to be in effect. Any scheduler task state transitions that would move a task from the running state to the ready state are delayed until scheduling is resumed. **KN_stop_scheduling** does not prevent a task switch if the running task becomes blocked or calls one of the rescheduling system calls.

To resume scheduling, call **KN_start_scheduling**. You must call **KN_start_scheduling** once for each lock in effect. Normal task switching resumes after you remove all scheduling locks. Tasks signaled into operation by interrupt handlers won't run until scheduling is resumed.

Note

A scheduling lock does not prevent task switching in all cases. A system call that causes blocking or rescheduling can initiate a task switch even if there is a scheduling lock.

Disabling interrupts and locking scheduling can cause the system to be less responsive.

Controlling Task State Transitions

Before making a system call, you must determine whether a task switch is appropriate and perform a scheduling lock if necessary. Kernel system calls can be classified into four categories, based on their ability to cause state transitions:

- Non-scheduling system calls never cause state transitions. **KN_create_semaphore** is an example of a non-scheduling system call.
- Signaling system calls can put tasks into the ready queue and potentially cause state transitions. **KN_send_unit** is an example of a signaling system call. If invoking such a system call would cause a higher priority task to become ready, the running task can use a scheduling lock to keep control of the processor.
- Blocking system calls will cause the Kernel to put the running task to sleep (block) and thus initiate a state transition. **KN_receive_unit** is an example of a blocking system call. When you use **KN_receive_unit**, rescheduling occurs unless the unit is actually available. A scheduling lock does not prevent a system call in this category from causing rescheduling.
- Rescheduling system calls always cause rescheduling or will cause rescheduling when invoked on the running task, regardless of a scheduling lock. For example, **KN_sleep** always causes rescheduling.

Using Task Handlers

You may configure the Kernel to invoke application procedures, called *task handlers*, which you write to perform additional functions during these situations:

- Task creation
- Task deletion
- Task switching

Your handlers can enhance the Kernel operations. By writing these procedures, you can add functionality and/or handle error situations. For example, if your application requires a hierarchical structure for tasks, you can write a task handler to implement the setup when the task is created.

These are the task handlers you can write:

- create_task_handler
- delete_task_handler
- task_switch_handler

Task handlers may invoke non-blocking Kernel system calls to perform various functions. The Kernel expects these procedures to be as correct as one of its own internal calls.

Note

Incorrect task handler code can impact performance and can corrupt application operation. The duration of these handlers can adversely affect system performance and interrupt latency.

The Kernel invokes task handlers when the task makes a system call such as the Nucleus system calls **rq_create_task** or **rq_delete_task**, or when a system call causes a task switch or change in priority of a task. All handlers are invoked with interrupts disabled and scheduling locked. Handlers cannot enable interrupts or unlock scheduling.

Figure 14-1 illustrates the interrelation between the Kernel and task handlers. The application has installed task handlers. When any task creation, deletion or switching occurs after the handlers have been installed, the Kernel calls the appropriate handler.

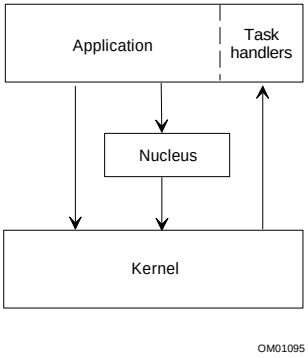


Figure 14-1. Kernel Invoking of Task Handlers

Installing and Removing Task Handlers

You install task handlers dynamically with the **KN_set_handler** system call. You may install multiple handlers of each type. The **KN_reset_handler** system call dynamically removes your task handler.

> **Note**
> Multiple task handlers degrade the performance of your system; remove them using **KN_reset_handler** when they are not needed.

This example describes Kernel operation using task creation handlers.

1. With no task creation handlers installed, the application calls the Nucleus system call **rq_create_task**. No handlers are invoked.
2. Using **KN_set_handler**, the application installs two task creation handlers (`createA_hdlr` and `createB_hdlr`).
3. Then, when an application calls the **rq_create_task** system call, the Kernel initializes the new task. Before the task is allowed to execute, the Kernel calls `createA_hdlr`, then `createB_hdlr`. Finally, it enables the task to execute.
4. Next, the application calls **KN_reset_handler** to remove `createA_hdlr`. When **rq_create_task** is next called, the Kernel initializes the new task. It calls `createB_hdlr`, then enables the task to execute.

5. Now the application reinstalls `createA_hdlr` using **KN_set_handler**. When **rq_create_task** is called, the Kernel initializes the new task, calls `createB_hdlr`, then `createA_hdlr`, then enables the new task to execute.

6. Finally, the application removes both task creation handlers using **KN_reset_handler**. When **rq_create_task** is called, the Kernel performs only its standard **create_task** functions.

See also: Kernel system calls and handlers, *System Call Reference*

Task Management System Calls

These are the system calls you use to manage tasks:

KN_start_scheduling
KN_stop_scheduling
KN_set_handler
KN_reset_handler

Table 14-4 lists Kernel task operations and their related system calls.

Table 14-4. Task Management System Calls

Operation	Description
restart scheduling	**KN_start_scheduling** cancels one scheduling lock imposed by **KN_stop_scheduling.** When it cancels the last outstanding schedule lock, the Kernel carries out all delayed task state transitions.
lock scheduling	**KN_stop_scheduling** temporarily locks (or places an additional lock on) scheduling for the running task.
install task handler	**KN_set_handler** dynamically installs your task handler. You may install multiple task handlers of each type by invoking **KN_set_handler** multiple times.
remove task handler	**KN_reset_handler** dynamically removes your task handler.

iRMX Memory Management for Kernel System Calls

In an iRMX system, you can obtain memory for the Kernel to use as a memory pool in these ways:

- Call **rq_create_segment**, which returns a token for a 16-byte aligned memory segment. This method gives the best performance since the memory specified in the Kernel `area_ptr` parameter should be aligned on a 4-byte boundary.
- Exclude the memory from free space memory using the ICU or the *rmx.ini* file and then create a descriptor for the excluded memory.

In either case, supply a pointer to the memory, `token:0`, and use this pointer in Kernel object creation system calls, which require an `area_ptr` parameter. If you use the iRMX OS to manage memory, specify:

```
area_ptr = token:0
```

or

```
area_ptr = malloc (size)
```

If you use your application to manage memory, specify:

```
mem_array [n]  UINT_8
area_ptr = &mem_array
```

Aligning Application or malloc Allocated Memory

If you provide memory directly from your application's data segment or using **malloc**, you may need additional steps to align the memory, for these reasons:

- The size literals supplied by the Kernel in the literal declarations files are specified in units of bytes, causing the areas to be declared as byte arrays.
- Compilers do not necessarily align byte arrays that appear in the data segment.

To force the compiler to align arrays on 4-byte boundaries, declare memory allocations as integer arrays. An integer is 4 bytes, so you should declare one-fourth the number of array elements. For example, when declaring memory to be used by an alarm object for the **KN_create_alarm** call, you might use these statements in C:

```
int   alarm_area [KN_ALARM_SIZE/4];
KN_create_alarm (alarm_area,...)
```

To align an 80-byte array that you need to access in byte values rather than in integer values, you might use these statements:

```
int    y[20];
char   *x;
x = y;
```

This guideline of using an integer declaration works for all compilers. There are other methods, such as declaring the array at the beginning of a structure, or testing the alignment of the pointer and adjusting it. If you already use another technique to align memory, make sure it still works if you change compilers.

Using malloc

If you use **malloc**, you will need to test the alignment of the pointer and adjust it yourself. To do so, request a size 3 bytes larger than you need for a particular `area_ptr`. Then adjust the `area_ptr` to be 4-byte aligned using code similar to this:

```
char          *array;
UINT_32       align_factor;
UINT_32       kn_sema_t;

array = malloc (KN_SEMAPHORE_SIZE + 3);

align_factor = ((long)(near *) array) & 3;

kn_sema_t = KN_create_semaphore(
       UINT_32 *) &array[align_factor],        /* area_ptr */
       (KN_FIFO_QUEUEING | KN_ZERO_UNITS));    /* flags */
```

Demo Files for the Kernel

There are two files installed with the OS that create a demo program for the Kernel:

- A makefile to use with the **make** command to generate the demo.
- *sr.c*, the C language demo source code.

Make requires that you load *clib.job*. If it is not already loaded (for example, by your *:config:loadinfo* file), enter this command:

```
sysload /rmx386/jobs/clib.job <CR>
```

To generate the demo enter:

```
cd /rmx386/demo/c/rmk/src <CR>
make  <CR>
```

These commands generate the executable file *sr*. The *sr* program performs a send/receive semaphore test, first using the Nucleus and then using the Kernel. Use this syntax to run the demo:

```
sr <priority> <iteration_count> [k]
```

where:

<*priority*> is the priority of tasks in the demo

<*iteration_count*>
is the count of iterations of sends and receives.

[k] indicates that both the iRMX and iRMX semaphore functions are used. If you don't specify that both types of functions are used, only iRMX semaphore functions will be used by the demonstration program.

For example, to run the demo, enter:

```
- sr 128 100000 K  <CR>
```

Include Files for the Kernel

The files in this table are for compatibility with existing code that makes Kernel calls. For example, if your C code already includes the files listed under *rmk.h* in Table 14-5, you need not include file *rmk.h* (it includes the other files itself). Compilers automatically include only the code needed from include files.

Table 14-5. Kernel Include Files

PL/M	C	Assembler
	rmk.h	
rmk_type.lit	rmk_type.l	rmk_type.equ
rmk_ex.lit	rmk_ex.equ	
rmk_base.lit	rmk_base.l	rmk_base.equ
rmk_base.ext	rmk_base.h	rmk_base.edf

For C applications, also include *rmk_ex.l* for definitions of exception codes.

Kernel Memory Management

This section is provided for compatibility with existing Kernel applications. It is not necessary that you create Kernel pools and areas to use the Kernel system calls for object, time, and task management.

The Kernel Memory Manager defines and implements memory pools, providing Kernel applications with a physical memory management facility.

CAUTION

The Kernel Memory Manager does not protect memory areas from unauthorized access. Any task could ignore the rules and access memory given to another task, sometimes with disastrous results.

Creating Memory Pools and Areas

Use the **KN_create_pool** system call to create a memory pool in a specific range of memory. Specify where in memory to create the memory pool object and the size of the pool, including overhead.

To use the memory in a memory pool, invoke the **KN_create_area** system call. Specify the memory pool's kn_token and the size of the requested area, including area overhead. If the requested space is available in the pool, the Kernel Memory Manager returns a pointer to the area. Use this pointer to access the area, to create a segment descriptor to the area, or to create a memory sub-pool from the area. If the request cannot be filled, **KN_create_area** returns a null pointer.

Note

If a memory pool is created on a 4-byte boundary, all areas created from that pool will be on a 4-byte boundary. To align the memory, the pool can be the start of a Builder-defined segment or a large array of integers defined statically in your application.

Deleting Memory Pools and Areas

When the application is through using an area, call **KN_delete_area**, specifying the area to be released and the pool from which the area came. The **KN_delete_area** system call returns the memory to the memory pool, making it available for re-use.

When an application no longer needs a memory pool, call the **KN_delete_pool** system call. The **KN_delete_pool** call does not require all of the areas to be returned in order to delete a pool. However, if an area is still in use when the pool is deleted, there is a chance that the same memory could be used simultaneously for two purposes, with undefined results.

Note

When using memory pools, do not access memory within the pool except for areas allocated by the **KN_create_area** system call. Do not invoke memory pool system calls on a memory pool after invoking the **KN_delete_pool** system call on it.

Pool and Area Overhead

A memory pool occupies exactly the size specified when it is created. There is a minimum size that can be requested, represented by the literal KN_MINIMUM_POOL_SIZE. This size is the minimum number of bytes that the Kernel requires for a memory pool. It includes overhead data structures whose memory cannot be allocated from the pool. The usable space for a pool is actually the requested size minus the pool overhead. The literal KN_POOL_OVERHEAD defines the number of bytes in the overhead. To create a pool of size *n*, the total number of bytes required would be *n* + KN_POOL_OVERHEAD.

The literal KN_MINIMUM_AREA_SIZE designates the smallest area that can be allocated from a memory pool. If an application requests an area smaller than the minimum size, the memory manager rounds the requested size up to the minimum size. There is also an overhead associated with each area created from a memory pool. The literal KN_AREA_OVERHEAD defines this amount. Thus, if an area of size *n* is desired, *n* + KN_AREA_OVERHEAD bytes are required.

Performance Issues

You gain the highest level of performance from a memory pool if you allocate memory areas of the same size. In addition to minimizing wasted space, the times to allocate and deallocate fixed-size areas are less.

Allocating memory areas on 4-byte boundaries enables Kernel system calls to execute faster because the objects created in the areas are also aligned on 4-byte boundaries. Memory pool properties provide that, if a memory pool is created aligned on a 4-byte boundary, all areas allocated from within that pool are also aligned on 4-byte boundaries.

To create a pool that can allocate exactly *n* areas all of size *m*, the area required is as follows:

```
n * (m + KN_AREA_OVERHEAD) + KN_POOL_OVERHEAD
```

If *m* is less than KN_MINIMUM_AREA_SIZE, replace *m* with KN_MINIMUM_AREA_SIZE in the expression.

Figure 14-2 shows the relationship between a memory pool and memory areas. Although areas may be different sizes, access to the areas is more efficient if all areas in a pool are the same size.

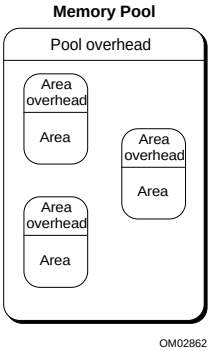


Figure 14-2. Memory Pools and Areas

Getting Information about a Pool

Using the **KN_get_pool_attributes** system call, you can get this information about a specific memory pool:

- The size of the pool
- The total available space in the pool
- The largest contiguous available area in the pool

Allocating Memory in an Interrupt Handler

In general, managing memory from within an interrupt handler is unwise because it impacts performance. The **KN_create_area** and **KN_delete_area** system calls use an internal semaphore for mutual exclusion and may cause tasks to go to sleep. Interrupt handlers may safely use these system calls on a pool only if you perform all operations on the memory pool (by either the interrupt handler or any other procedure) with interrupts disabled. This ensures that the memory pool will always be accessible when the interrupt handler invokes a system call on it.

Kernel Memory Management System Calls

These are the system calls you use to manage memory:

KN_create_area
KN_delete_area
KN_create_pool
KN_delete_pool
KN_get_pool_attributes

Table 14-6 lists operations on memory and the related system calls.

Table 14-6. Management System Calls

Operation	Description
create area	**KN_create_area** allocates an area of memory of specified size from a specified memory pool.
create pool	**KN_create_pool** creates a memory pool in a specified range of memory.
delete area	**KN_delete_area** returns an area to the memory pool it was allocated from.
delete pool	**KN_delete_pool** deletes a memory pool.
get available space	**KN_get_pool_attributes** returns the size of the pool, the total size in the pool, and the largest contiguous available area in the pool.

❑❑❑

I/O SYSTEMS PROGRAMMING CONCEPTS

This section describes the Basic I/O System, the Extended I/O System, and the Universal Development Interface.

See also: System call descriptions, *System Call Reference*;
I/O System, and UDI overviews, *Introducing the iRMX Operating Systems*

I/O System Basic Concepts 15

This chapter introduces concepts which apply to both the Basic I/O System (BIOS) and the EIOS (EIOS), as well as those that apply only to the BIOS or the EIOS.

See also: BIOS and EIOS, *Introducing the iRMX Operating Systems*

The concepts presented in this chapter are:

- System programming (BIOS only)
- Synchronous and asynchronous calls
- Device controllers and device units
- Volumes
- Files
- Communication between tasks and device units
- Logical Names
- `Path_ptr` parameters and default prefixes (EIOS only)
- I/O Jobs (EIOS only)

System Programming (BIOS)

There are two programming roles associated with the iRMX OSs: *application programming* and *system programming*.

System programming affects the performance and security of the entire system; application programming has a more limited effect because it involves individual jobs. Although the roles have different names, separate people are not required. One individual can perform both roles.

The BIOS system call descriptions include notes for system calls that, if misused, can have serious consequences for an application system. These system calls should be used by the designated system programmer.

Synchronous and Asynchronous Calls

The I/O System provides *synchronous* and *asynchronous* system calls. Both the BIOS and the EIOS provide synchronous calls; only the BIOS provides asynchronous calls.

Synchronous calls begin running as soon as the application invokes them and continue running until they detect an error or complete. While a synchronous system call is running, the calling task cannot run. It resumes running only after the synchronous call has either failed or succeeded. Synchronous calls act like subroutines.

Asynchronous calls complete their operation by using tasks that run concurrently with the application. The application can accomplish some work while the BIOS accesses disk drives or tape drives, for example.

Each asynchronous system call has two parts: one *sequential* and one *concurrent*.

- The sequential part behaves in much the same way that synchronous system calls do. It verifies parameters, checks conditions, and prepares the concurrent part of the system call. If any problem is detected during the sequential part, an exception code returns to the caller and the concurrent part does not start. If no error is detected, an E_OK condition code returns to the caller and the concurrent part starts.
- The concurrent part runs as an iRMX task. This task is readied by the sequential part of the call and runs only when the priority-based scheduling of the OS gives it the processor. The concurrent part also returns a condition code as part of an *I/O Request/Result Segment* (IORS) sent to the response mailbox specified in the asynchronous call.

See also: BIOS and EIOS Layer Specific Information, *System Call Reference*

Asynchronous Call Order of Operations

This example shows how an application can use an asynchronous call to retrieve some information stored on disk. Figure 15-1 on page 217 illustrates how the sequential and concurrent parts of the call relate.

1. The application issues **a_read** and specifies a response mailbox for communicating with the concurrent part of the system call.

2. The sequential part of **a_read** begins to run. This part checks the parameters for validity. These operations execute in context of the application code. Figure 15-1 on page 217 labels this area as “sync”.

3. The sequential part returns a condition code. If it is E_OK, the BIOS readies the concurrent part of the call to perform the read; otherwise, it does not.

4. The application receives control and tests the sequential condition code. If it is E_OK, the application continues running until it needs the information from disk. Now, the application can take advantage of the asynchronous and concurrent behavior of the BIOS to perform other tasks. Figure 15-1 on page 217 labels the asynchronous area as “async”.

   For example, the application can implement multiple buffering by issuing other **a_read** calls while waiting for the first call to complete. Alternatively, the application can use this overlapping processing to perform computations.

   For the balance of this example, assume that the sequential part of the system call returned E_OK. (If the sequential condition code is not E_OK, the application must respond appropriately.)

5. Before taking the information from the buffer, the application verifies that the concurrent part of **a_read** ran successfully. There are three ways the task can do this.

   One way is to issue a **receive_message** call to check the response mailbox specified in **a_read**. In this case, the application obtains an IORS that contains a condition code for the concurrent part of the system call. If it is E_OK, the application can get the data from the buffer. Otherwise, the application should analyze the code to determine why the read was not successful.

   See also: IORS, *System Call Reference*;
   Accessing the IORS, *Programming Techniques*

Another way, which can be used only after **a_read**, **a_write**, or **a_seek**, is to issue **wait_io**, which passes a token for the response mailbox to the application. In this way, the application can receive the condition code directly for the concurrent part of the system call. In addition, if the concurrent condition code is E_OK, the application also receives the number of bytes successfully read. Otherwise, the number of bytes returned has no significance.

The final way is used for flat applications. This way calls **wait_iors**. You could use either of the previous methods with a flat application but you can't do anything practical with the returned IORS structure.

See also: **wait_iors** call, *System Call Reference*

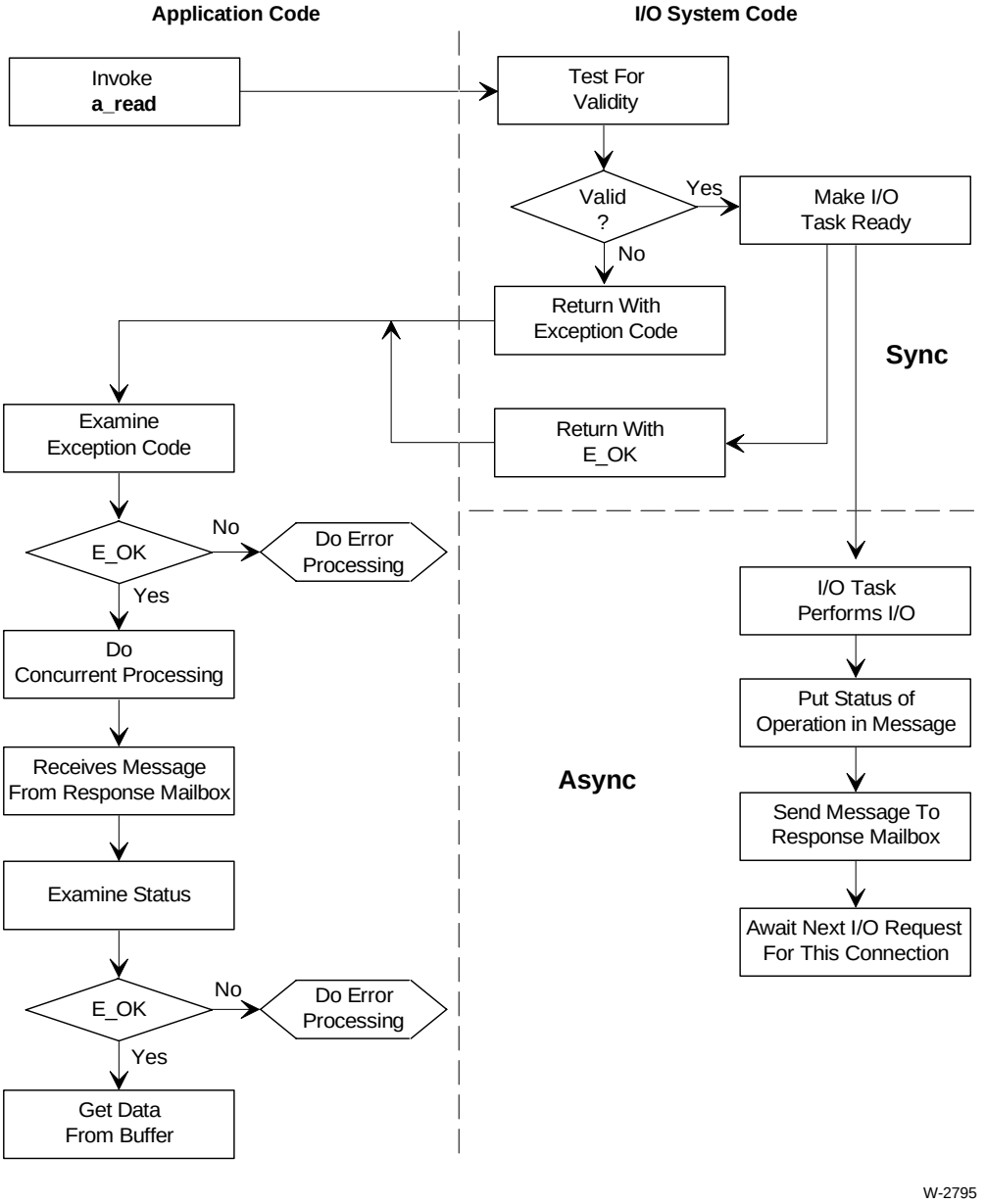


Figure 15-1. Behavior of an Asynchronous System Call

Using Asynchronous Calls

These explanations apply to all asynchronous calls.

- All of the asynchronous system calls consist of two parts: one sequential and one concurrent. The BIOS activates the concurrent part only if the sequential part runs successfully and returns E_OK.
- Every asynchronous system call requires a response mailbox for communication with the concurrent part of the system call. Use the **create_mailbox** system call to create a message mailbox.
- Whenever the sequential part of an asynchronous system call returns a condition code other than E_OK, the application should not attempt to receive a message from the response mailbox nor should it call **wait_io**. Doing so can cause the application to wait indefinitely. The BIOS cannot run the concurrent part of the system call.
- Whenever the sequential part of an asynchronous system call returns E_OK, the BIOS runs the concurrent part of the system call. The application can take advantage of the concurrency by doing some processing before receiving the message at the response mailbox or calling **wait_io**.
- After the concurrent part of a system call runs, the BIOS signals its completion by sending an object to the response mailbox. The precise nature of the object depends upon which system call the application invoked. Use **receive_message** to receive the message. The application can determine the returned object type by calling **get_type**.
- The application, with one exception, must delete the IORS when it is no longer needed. The BIOS uses memory for such segments so if the application fails to delete the IORS, it might run short of memory. Use **delete_segment** to delete the IORS.

  The exception is when the application calls **wait_io**. The application does not have access to the IORS and cannot delete it. This enables the BIOS to maintain a supply of IORSs that it can use repeatedly. Because most I/O-related operations are reads, writes, or seeks, this means a significant performance enhancement for the application.

Condition Codes for Asynchronous Calls

For those system calls that require a response mailbox parameter, the BIOS returns a condition code for the sequential portion of the call to the word pointed to by the `except_ptr` parameter and a condition code for the concurrent portion of the call to the `status` field of the IORS.

See also: IORS, *System Call Reference*;
Accessing the IORS, *Programming Techniques*

Some calls can return a connection instead of an IORS. If a sequential exceptional condition occurs, the BIOS either returns control to the calling task or passes control to an exception handler. It does not process the asynchronous portion of the call. If a concurrent exceptional condition occurs, the calling task must signal the exception handler or process the exceptional condition inline.

If the application handles the exception inline, use the Nucleus **get_type** system call to obtain the type of object returned, for example an IORS.

See also: **get_type**, *System Call Reference*;
exception handling, in this manual

Creating I/O Buffers

A_read, **s_read_move**, **a_write**, and **s_write_move** each require a buffer to read from or write to while performing I/O. When you create these buffers, these restrictions apply:

- The memory segments used for the I/O buffers must have the appropriate access rights: be readable for read operations or writable for write operations.
- Once the I/O operation has been invoked, the application tasks should not change the contents of the buffer until the BIOS finishes the operation.
- Do not delete an iRMX segment used as a buffer while an I/O operation is in progress.

  Using segments from one job as buffers for I/O operations in a different job can cause unintentional deletion. If Job A owns an iRMX segment, that segment is automatically deleted when the job is deleted. If Job B uses this segment as a buffer for I/O, the buffer will be deleted even if Job B has I/O in progress.

Device Controllers and Device Units

The iRMX OS distinguishes between *device units* and *device controllers*.

A device unit is a hardware entity that tasks use to read or write information, or both. Device units include diskette drives, hard disk drives, tape drives, printers, and terminals.

A device controller is a hardware entity that talks directly with iRMX software and controls device units. Typically, a device controller enables iRMX applications to communicate with several device units. For example, a 2215 SCSI Disk Controller acts as an interface between an application program and several disk drives (device units).

Setting Mass Storage Device Granularity

When information is stored on a mass storage device, space is allocated in *granules* and the block size is called *granularity*. If your device supports multiple device granularities, selecting the larger value usually gives higher performance, but you may waste storage space due to large granules containing only a few bytes of data.

See also: Granularity, *Introducing the iRMX Operating Systems*

Use these guidelines when setting granularity:

- For diskettes, always set the volume granularity equal to the device granularity, unless you plan to store many large files on the volume. Don't select a volume granularity larger than 1 Kbyte.
- For hard disks, set the volume granularity equal to the device granularity, unless the device granularity is less than 1 Kbyte. Then set the volume granularity to 1 Kbyte.
- For sequential file access, larger granularity sizes generally improve access time. Each access can handle more data.
- For random file access, smaller granularity sizes generally improve access time. Each access handles only that data that is needed, thereby spending less time transferring needless data.
- When creating a large file, assign a large file granularity to minimize the number of noncontiguous blocks that make up the file. This decreases the fragmentation of the volume.
- For smaller files, set the file granularity equal to the volume granularity to minimize wasted space on the volume.

File Granularity Example

This example uses only one small mass storage unit containing a file of 20,010 bytes. It illustrates how performance interacts with use of space. Performance may not be critical if you do not use the device often enough for the data transfer rate to have much impact.

1. If the granularity is 10,000 bytes, the file occupies three granules. The first two granules are full and the third contains only 10 useful bytes.

   Although this file wastes 9,990 bytes of storage space, the data transfer rate is quicker than with a similar file of smaller granularity.

2. If the file granularity is 200 bytes, the file occupies 101 granules. Each of the first 100 granules is full, while the last granule contains only 10 useful bytes.

   The file now wastes only 190 bytes of storage space, but the data transfer rate is slower than with a granularity of 10,000 bytes.

If the application system has many mass storage units and space is readily available, a large file granularity will give faster average transfer rates and shorter access times, at the expense of device space.

Volumes

A *volume* is the medium used to store the information on a device unit. For example, if the device unit is a diskette drive, the volume is a diskette; if the device unit is a multi-platter hard disk drive, the volume is the disk pack; if the device unit is a tape drive, the volume is the cartridge tape.

File Types

The I/O System defines a file to be information, not a device. The BIOS and EIOS provide these types of files:

- Named files allow random access, hierarchical file structure, and access control.
- EDOS (Encapsulated DOS) and DOS files are DOS files accessible to DOSRMX and iRMX for PCs applications using EIOS and BIOS system calls.

> **Note**
> The EDOS and DOS file drivers are mutually exclusive. DOSRMX provides the EDOS file driver. iRMX for Pcs and the iRMX III OS provide the DOS file driver.

- Remote files are named files that exist on another system and are accessed on an iNA 960/iRMX-NET network.
- Network File System (NFS) files are files that exist on another system and are accessed on a TCP/IP network using NFS. NFS files are accessible between systems using different operating systems.
- Physical files allow more direct hardware control over a device.
- Stream files allow one task to write to a file while another reads it.

Each kind of file has characteristics that make it unique. Regardless of the kind of file, the BIOS and EIOS provide information to applications as a string of bytes, rather than as a collection of records.

See also: Named Files, Physical Files, and Stream Files chapters in this section for more information on files;
remote files, *Network User's Guide and Reference*;
NFS chapters, *TCP/IP and NFS for the iRMX Operating System*

Communication Between Tasks and Device Units

Several layers of software and hardware must be bound together before communication between application tasks and device units can occur. Figure 15-2 shows these layers.

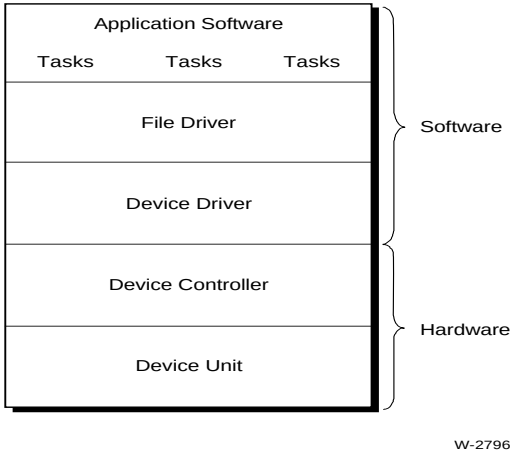


Figure 15-2. Hardware and Software Layers Between Tasks and a Device

There are several kinds of bonds:

- The bond between the application tasks and the file driver is supplied during the linking or binding process. A file driver provides the interface between the BIOS and a device driver. The information needed to perform the binding process is specified at configuration time. Loadable file drivers provide almost the same function as using the ICU.
- The bond between a device driver and a device controller is data residing in a data structure called a Device Unit Information Block (DUIB). Data for DUIBs is specified at configuration time. Critical data involves the DUIB parameters `update_timeout` and `fixed_update`. Loadable device drivers provide almost the same function as using the ICU.

- The bond between the device controller and the device units is a physical bond, typically wires or cables.

See also: Loadable file and device drivers, *System Configuration and Administration*;
File and device drivers and DUIB data structure definition, *Driver Programming Concepts*

The tasks access files and devices through connections. Two kinds of system calls produce connections: one kind produces a *device connection* and the other produces a *file connection.* Before a task can use a file, it must invoke both of these kinds of calls.

See also: I/O Jobs and Connections, in this section

Device connections are like conduits (pipes); file connections like wires through the conduits. These descriptions apply to device and file connections.

- Device connections extend from the application software to the individual device units and each passes through only one file driver.
- There is only one device connection to each connected device. However, multiple file connections can share the same device connection.
- There is only one device connection through the stream file driver, because one logical device contains all stream files.
- Unconnected device units are not connected to the application software.
- Different device units with the same controller can be connected by different file drivers.
- Tasks can share access to the same device unit through the physical file driver and they can share access to the same files on the same device unit through the named file driver.

An application task must attach a device before accessing the files on that device and must establish a connection to the file before accessing the data in that file.

Logical Names

You can use logical names to identify file connections or device connections. A logical name is an iRMX STRING of 12 or fewer characters with a unique syntax.

Every I/O job has three distinct object directories in which objects can be cataloged. When looking up a logical name, the EIOS searches these directories in this order and stops when it finds the name.

- The object directory of the *local job*
- The object directory of the *global job* for a user session
- The object directory of the *root job*

See also: Logical names, *Command Reference*;
Connections and Objects in this section

Path_ptr Parameters and Default Prefixes (EIOS)

Some EIOS calls refer to files rather than to connections. All such calls require a `path_ptr` parameter to identify the file to be attached, created, or otherwise used.

One aspect of the `path_ptr` parameter applies to all kinds of files. If the parameter is set to null, or if it points to a null String (an iRMX STRING containing 0 characters), the EIOS selects the file based on the *default prefix* of the calling task's job.

The default prefix is an attribute of an I/O job and it is a logical name for a device or a file connection. It is cataloged under the name *$* in either the local or the global object directory for the job. Whenever a task invokes a system call but does not specify a logical name, the EIOS looks up the default prefix and uses the associated connection.

The complete interpretation of the `path_ptr` parameter depends upon the kind of file being accessed.

See also: Named Files, paths, prefixes and subpaths in this section

I/O Jobs (EIOS)

Any job using EIOS calls must be an I/O job. The advantage of using EIOS calls is that they perform many functions automatically, making them simpler to use than BIOS calls.

I/O jobs can be created when programs are running and, for ICU-configurable systems, when the system is initialized. An I/O job must have:

A global job	A token for the user session's global job must be cataloged in the I/O job's object directory under the name *rqglobal*.
A default prefix	The default prefix is a connection cataloged under the name *$* in either the local job object directory or the global job object directory.
A default user object	This user object is required to access named files using EIOS calls and must be cataloged in the I/O job's object directory under the name.

See also: Named Files, default prefix, default user object in this section

For ICU-configurable systems, specify the characteristics of I/O jobs that are created when the system is initialized.

❑❑❑

I/O Jobs and Connections 16

Creating I/O Jobs

I/O jobs differ from other jobs in these ways.

- Many of the parameters required by the Nucleus' **create_job** system call are not required by the EIOS job creation system calls. Instead, some of these values are specified at system configuration time. These parameters include:

  ```
  directory_size
  param_object
  max_objects
  max_tasks
  max_priority
  ```

- The EIOS calls automatically initialize the new job with a default user object, global job for that user session, and default prefix, inherited from the parent job.
- The EIOS system calls allow the new job to send a termination message to the parent job.
- The **rqe_create_io_job** system call creates I/O jobs while the system is running and reserves memory for the job's memory pool.

  See also: EIOS calls, *System Call Reference*

  Any task that invokes this system call must be running within an I/O job. For ICU-configurable systems, you can create the initial I/O job during system configuration. For iRMX for PCs and DOSRMX systems, the initial I/O job is already configured into the system.

Creating Device Connections

These system calls apply to device connections:

a_physical_attach_device (BIOS)	**logical_attach_device** (EIOS)
a_physical_detach_device (BIOS)	**logical_detach_device** (EIOS)

The device connection is the application's only pathway to the device. There can be only one device connection between a device unit and all of the application tasks that need to use that device.

See also: Named Files, Physical Files, Stream Files, and call sequences in this section

Using BIOS System Calls

To attach a device for BIOS calls, use **a_physical_attach_device**, which:

- Creates a device connection that represents the device.
- Identifies the owner of the device connection, to prevent other users from detaching devices that they do not own.

Use this call only once for each device because devices cannot be attached multiple times. Only one or a few selected tasks should call **a_physical_attach_device**. These tasks can be in one these forms:

- An initialization task can create all of the device connections and catalog them in the root object directory. Then all required device connections are available to all application tasks that need them.
- Several tasks can make the device connection available to selected application tasks by sending the connection to certain mailboxes or by cataloging it in certain object directories.

Use **a_physical_detach_device** to delete the device connection when the device is no longer needed by the application.

The OS keeps track of the number of tasks using the device. It does not detach the device until it is no longer being used by any task.

Using EIOS System Calls

To attach a device for EIOS calls, use **logical_attach_device**. This system call

- Creates a device connection that represents the device.
- Catalogs a token for the connection under the specified logical name, which the EIOS uses to access the device.
- Identifies the owner of the device connection, to prevent other users from detaching devices that they do not own.

Use this call only once for each device because devices cannot be attached multiple times.

Logical_attach_device calls **a_physical_attach_device**, but may not do so immediately. Instead, physical attachment occurs transparently during processing of any system call that references the logical device object. This timing can be an issue when BIOS system calls use logical device objects, as described in the next section.

When the device is no longer needed by the application, use **logical_detach_device** to delete the device connection.

The OS keeps track of the number of tasks using the device. It does not detach the device until it is no longer being used by any task.

Using a Logical Device with BIOS System Calls

You can assign a logical name to any device with **logical_attach_device**. Typically, you use these logical device objects with EIOS calls. However, BIOS calls also permit the `prefix` parameter to be a logical device object; it is a shorthand way to traverse the directory structure.

When you use a logical device object in BIOS calls, the BIOS examines the logical device object to determine the device connection. In such cases, you could receive the E_DEV_OFF_LINE condition code. If the device is online, the device has not yet been physically attached with **a_physical_attach_device**.

You can correct this situation by invoking at least one EIOS system call that refers to the logical device by its logical name. The calling task must reside in an I/O Job before it can invoke EIOS system calls.

Creating File Connections

When an application task is ready to use a file, it establishes a connection to that file. These system calls apply to file connections:

a_attach_file (BIOS)	**s_attach_file** (EIOS)
a_create_file (BIOS)	**s_create_file** (EIOS)
a_open (BIOS)	**s_open** (EIOS)
a_seek (BIOS)	**s_seek** (EIOS)

Unlike device connections, there can be multiple file connections to a single file. This allows different tasks, if necessary, to have different kinds of access to the same file at the same time.

Using BIOS System Calls

Use **a_create_file** to obtain a file connection:

- When the task does not know if the file already exists.
- When the task knows that the file does not yet exist.

If the file already exists, use **a_attach_file**.

In either case, the I/O System returns a connection to the physical file.

CAUTION

It is possible to use **a_create_file** to obtain a file connection for a file that already exists, however the file will be truncated to 0 length in the process. Other tasks having other connections to that file will lose access to data because the end-of-file marker will have moved to the beginning of the file.

The distinction between the file creation and the file attachment system calls enables the application to work with named files as well as physical files.

After receiving a file connection, use **a_open** to open the connection. Use the `mode` parameter to specify if the connection is open for reading only, for writing only, or for both reading and writing. Use the `share` parameter to specify if other connections to the file can be opened for reading only, for writing only, or for both reading and writing.

Using EIOS System Calls

Use **s_create_file** to obtain a file connection:

- When the task does not know if the file already exists.
- When the task knows that the file does not yet exist.

If the file already exists, use **s_attach_file**.

In either case, the I/O System returns a connection to the physical file.

CAUTION

It is possible to use **s_create_file** to obtain a file connection for a file that already exists, however the file will be truncated to 0 length in the process. Other tasks having other connections to that file will lose access to data because the end-of-file marker will have moved to the beginning of the file.

The distinction between the file creation and the file attachment system calls enables the application to work with named files as well as physical files.

After receiving a file connection, use **s_open** to open the connection. Use the `mode` parameter to specify if the connection is open for reading only, for writing only, or for both reading and writing. Also specify if other connections to the file can be opened for reading only, for writing only, or for both reading and writing.

Note

If a task in one job obtains a file connection that was created in a different job, the task cannot successfully use the connection to perform I/O operations. However, the task can catalog the connection under a logical name and use the logical name in **s_attach_file** to obtain a second connection that can be used without restriction.

A connection can be *open*, such as during read or write operations, or *closed*, such as during renaming or file status operations. Connections created by one I/O system can be used by the other as long as the connection is closed. For example, you can use an EIOS call to create a file and obtain a connection with the BIOS calls that rename a file or get a file's status. However, the connection cannot be used with a BIOS read, write, or truncate call, which require an open connection.

The same restriction applies if the BIOS creates the connection. The EIOS can use the connection as long as the system call does not require an open connection.

Moving File Pointers

The BIOS and EIOS maintain a file pointer for each open file connection to a random-access device unit. This file pointer tells the I/O System the logical address of the byte where the next I/O operation on the file is to begin. The logical addresses of the bytes in a file begin with 0 and increase sequentially through the entire file.

Normally the pointer for a file connection points to the next logical byte after the one most recently read or written. However, a task can modify the file pointer by invoking the EIOS **s_seek** or BIOS **a_seek** system call. This is useful when performing random-access operations on a file.

❑❑❑

Named Files 17

Named files are used with random-access, secondary storage devices such as disks and diskettes. Named files provide several features that are not provided by physical or stream files. These features include:

- Multiple files on a single device or volume
- Hierarchical file names
- Access control
- Extension data
- Disk integrity

Named files are useful in systems that support more than one application and in applications that require more than one file.

iRMX named files can also reside on remote systems. You access remote named files in the same way as local named files, using iNA 960 and/or iRMX-NET.

Named files can also reside on the DOS partition of DOSRMX systems and iRMX for PC systems. You access DOS files using the Encapsulated DOS (EDOS) file driver if you are using DOSRMX. You can access DOS files using the DOS file driver if you are running the iRMX OS on a PC that does not run DOS.

See also: Accessing EDOS Files, in this chapter;
Accessing DOS Files, in this chapter;
Remote Files, *Network User's Guide and Reference*;
EDOS, *Programming Concepts for DOS and Windows*

Using Prefixes, Subpaths and File Paths in System Calls

You designate named files in system calls by specifying their path. There are two components to a path: the *prefix* and *subpath*. A prefix is a logical name for a device or the name of a directory file or data file. A subpath is a data-file name or a sequence of directory names optionally followed by a data filename.

You can represent the character string that designates a path for a named file with an iRMX string. To represent a string of *n* characters, you must use 1+*n* consecutive bytes. The first byte contains the character count. The next *n* bytes contain the ASCII codes for the characters, in the same order as the string. This string is a pathname.

Use a pointer to this pathname as the `subpath` parameter in the system call and use the file or device connection as the `prefix` parameter in the system call.

Subpaths

The subpath ASCII string is a list of filenames separated by slashes, terminating with the desired file. A file name can be 1-14 ASCII characters, including any printable ASCII character except the / (slash), ↑ (up-arrow) or ^ (circumflex). These special characters are reserved for use in designating directory levels or dividing components in a pathname. The subpath can also be null or can point to a null string, in which case the prefix indicates the desired connection.

This subpath is an example of the most common form:

A/B/C/D

Where:

A, *B*, *C* Are the names of directory files.

D Is the name of either a directory or data file.

This example causes the I/O System to start at the default directory and descend to directories *A*, *B*, and *C* in order. Then it acts on file *D*.

An example of a less common form of subpath is:

↑*A/B/C/D*

Where:

↑ or ^ Tells the I/O System to ascend one level in the hierarchy of files; then descend to directories *A*, *B*, and *C* in order; then act on file *D*.

The I/O System also accepts consecutive up-arrows. For example:

↑↑*A/B/C*

This construction causes the I/O System to start with the directory indicated by the default prefix and ascend two levels before interpreting the remainder of the subpath.

A subpath can begin with a / (slash). For example:

/A/B/C

Whenever the I/O System detects a slash at the beginning of a subpath, the I/O System starts interpreting the remainder of the subpath at the root directory of the device indicated by the prefix.

Prefixes

A prefix is a logical name for a connection to either a device, a named directory file, or a named data file. The device may be either a local or remote device. The files may also be either local or remote files. The prefix is the only component that distinguishes a local connection from a remote connection. The prefix tells the I/O System where to begin interpreting the subpath:

- If the prefix is a connection to a local device, the I/O System begins scanning the subpath at the root directory of the device.
- If the prefix is a connection to a remote device, the I/O System begins scanning the subpath at the virtual root directory of the device.
- If the prefix is a connection to a local or remote named directory file, the I/O System begins scanning the subpath at the specified directory.
- If the prefix is a connection to a local or remote named data file, the I/O System checks to see if the subpath is null. If it is, the I/O System uses the file indicated by the prefix. If the subpath is not null, the I/O System returns a condition code indicating that the application program is attempting to use a data file as though it were a directory file.

All other syntax applies to both local and remote files.

Using the Default Prefix

Within one iRMX job, most references to a named file tree are generally confined to one branch of the tree.

For a file, a *default prefix* is a connection to a directory at the head of the most commonly used branch in the named file tree. To use the default prefix, set the `prefix` parameter to null. The I/O System keeps track of a job's default prefix by using the job's object directory.

You can specify one default prefix for each iRMX job. A default prefix provides a job with two advantages. First, it enables the application to use subpath names instead of pathnames. If your tree is several levels deep, this can save programming time during development. Second, a default prefix provides a means of writing generalized application code that can work at any of several locations within a tree.

For example, suppose that an assembler (implemented as an iRMX job) uses a default prefix to find a location in a named file tree. The assembler could then use a subpath name of *temp* to find or create a temporary work file. Before an application invokes the assembler, it sets the default prefix of the assembler job to a directory in the application's named file tree. This enables more than one job to invoke the assembler concurrently without the risk of sharing temporary files.

Specifying Paths in System Calls

System calls referring to named files need a path (prefix and subpath) to locate the file. If you specify a null prefix, the default is used. Specify a token to override the default.

You can specify paths in these forms:

Prefix	**Subpath**	**Designated Connection**
null	pointer to a null string	Connection is the default prefix.
null	pointer to an ASCII string	ASCII string defines a path from the default prefix to the target connection.
token	pointer to a null string	Prefix parameter contains a token for a connection and overrides the default prefix. Since the subpath is null, acts on the directory or file specified in the prefix.
token	pointer to an ASCII string	Prefix parameter contains a token for a connection and overrides the default prefix. The ASCII string defines a path from that connection to the target connection.

If the ASCII string begins with a slash, the prefix merely designates the tree and the subpath is assumed to start at the root directory of the tree associated with the prefix.

Named files can also be addressed relative to other files in the tree, using ↑ (up arrow) or ^ (circumflex) as a path component. These two symbols have the same meaning. (Some terminals do not have the up-arrow key.) The ↑ or ^ refers to the parent directory of the current file in the path scan.

Those system calls that require paths have a `path_ptr` parameter. You can use this `path_ptr` parameter, along with the default prefix, to specify the file to be used. This parameter is a pointer to an iRMX STRING that must be in one of these forms:

Null string If the STRING is 0 characters long, the I/O System will act on the file indicated by the default prefix of the calling task's job.

Logical name only

If the STRING consists only of a logical name enclosed in colons (such as *:g:* for the *Dept1* directory) the I/O System will look up the logical name and obtain the associated connection. Then, because the subpath is empty, the I/O System will act on the data file or directory file indicated by the connection.

Subpath only

The STRING can consist of a subpath without a prefix. The I/O System interprets such subpaths by starting at the directory indicated by the default prefix of the calling task's job. Then the I/O System follows the subpath from directory to directory until it reaches the final component of the subpath. This final component is the file on which the I/O System acts.

Whenever the STRING contains a subpath without a logical name, the default prefix must be a logical name for a connection to a device or to a named directory file. If the default prefix represents a connection to a named data file, the I/O System returns a condition code indicating that your task is attempting to use a data file as a directory.

Logical name and subpath

The application code can use a STRING with a logical name in colons followed immediately by a subpath. For example:

:g:tom/test_data/batch_1

The I/O System interprets this example as follows. First, it looks up the logical name *:g:* in the object directory of the local job, or if necessary, the global or root job. Then it follows the subpath from the directory associated with the connection. So in the example, the I/O System would find the directory associated with :g: and it would step through directories tom and test_data. Finally, the I/O System would act on file batch_1.

Using Connections

Once you have a connection to a particular file, you can use it as the `prefix` parameter of any system call by setting the `subpath` parameter to null. The I/O System will ignore the subpath and use only the prefix to find that particular file.

Suppose the application has a connection to directory *dept1/tom*. Use the connection to directory *dept1/tom* as the prefix, and use a pointer to a filename as the subpath. For example, if the subpath name is *test_data/batch_1*, the specified file is *dept/tom/test_data/batch_1*.

A file connection obtained in one job cannot be used as a connection by another job. However, a file connection can be used as a prefix by other jobs in any call requiring `prefix` and `subpath` parameters. The only exceptions to this rule are that the other jobs cannot use the connection as a prefix while specifying a null subpath in calls to **a_change_access**, **s_change_access**, **s_delete_file**, or **a_delete_file**. This means that a file connection can be passed to another job and the other job can obtain its own connection to the same file by calling **a_attach_file**, with the passed file connection being used as the `prefix` parameter in the call.

However, if the connection was created by a task in a different job, your task should not use the connection in any of these system calls. Rather, your task should first obtain a new connection to the same file by performing these steps:

1. Catalog the current connection in the object directory of your task's job. This establishes a logical name for the current connection.
2. Using the newly-defined logical name, invoke **s_attach_file** to obtain another connection to the same file.

If your task does attempt to use a connection created in another job, the I/O System will return a condition code rather than performing the requested function.

Controlling File Access

In environments where files are shared among multiple users and operating systems, you may need to control user access and the level of user access to files. The iRMX OS provides this control by identifying users with user IDs and embedding access rights for these IDs into the files. This section describes the user ID and file access along with the mapping process used for NFS files.

Users

The iRMX OS defines all entities, such as people or iRMX jobs, that use named files in your system as *users*. If you want all of these entities to be able to access any file, consider them as a single user. However, if different entities require different accesses, you must divide the entities into subsets, each of which is a separate user.

Alternatively, if the application does not interact with people (or enables only one person to interact), you might consider each iRMX job as a user. This setup would enable the application to control the files that each job can access.

User Ids

A user ID is a 16-bit number that represents any individual or collection of individuals requiring a separate identity for the purpose of gaining access to files.

Two user IDs have special meaning. One is the number 0 (the *system manager* or Super user). The other is the number 0FFFFH (the *World* user). If specified during system configuration, user ID 0 represents the system manager. When the system manager creates or attaches files, the resulting file connection automatically has read access to data files and list access to directory files, even if a file's access list does not contain ID 0. The system manager can also change any file's access list.

The user ID 0FFFFH represents World (all users in the system). Placing the ID for World in the list of user IDs for every user object enables the application to set aside certain files as public files, giving everyone limited access to a series of utilities, such as compilers. The HI follows this convention by ensuring that all users who log on dynamically have the World ID in their user object.

Note

Including the World ID in every user object, lets anyone modify the access list of a file whose owner ID is 0FFFFH (World).

See also: **permit** command, *Command Reference*
Accessing NFS Files, later in this chapter

Figure 17-1 shows the relationship between a user and the user ID.

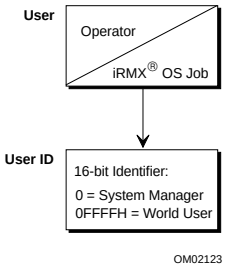


Figure 17-1. User and User ID Relationship

User Objects

The I/O System uses a *user object* when determining access rights to files. A user object contains a list of one or more user IDs. When a task attempts to use a file, it must supply the token for a user object. To determine access, the OS compares the IDs in the supplied user object with information contained in the file itself.

Most I/O operations performed within a particular iRMX job are performed on behalf of one user object. The I/O System enables the application to designate a default user object for each job, which defines the access rights for all tasks in that job.

The I/O System uses the job's object directory to keep track of the job's default user object, which is named *r?iouser*. Consider *r?iouser* to be a reserved name and do not use it.

Whenever the application invokes a BIOS call on behalf of the default user object, the application can use a null selector as the token for the `user` parameter. Use a null selector to designate the default user in BIOS system calls.

For ICU-configurable systems, you set up the default user objects for your initial EIOS I/O jobs (which start running immediately upon system initialization). Later, when a task creates an I/O job, the new I/O job inherits the default user object of its parent I/O job. The EIOS automatically catalogs the parent job's user object in the new I/O job's object directory under the name *r?iouser*.

File Access List

For each named file (data or directory), the I/O System maintains an access list which defines the users who have access and their access rights. Each access list is a collection of up to three ordered pairs with each pair having the form:

```
ID, ACCESS MASK
```

The ID portion is a user ID. The list of user IDs defines the users who can access the file. For systems that use NFS, the three iRMX user IDs map to NFS user IDs as described earlier.

The access mask portion defines the kind of file access that the corresponding user has. An access mask is a byte in which individual bits represent the various kinds of access permitted or denied that user. When a bit is set to 1, it signifies that the associated kind of access is permitted. When set to 0, the bit signifies that the associated kind of access is denied.

iRMX-NET uses a slightly different access mask for *remote* files than is used for *local* files. A file is local if it resides in the same physical system to which the terminal is connected. A file is remote if it resides on another system accessible through a network.

See also: Remote files, *Network User's Guide and Reference*;
File access attributes in this chapter for DOS and EDOS;
permit command, *Command Reference*

Note

NFS file access is mapped to the iRMX OS file access scheme. For information on this mapping see Accessing NFS Files, in this chapter.

The association between the bits of the access mask and the kinds of access they control are as follows:

Bit	Data Files	Directory Files
3	Update	Change Entry
2	Append	Add Entry
1	Read	List
0	Delete	Delete

The remaining bits in the access mask have no significance.

For example, an access list for a data file might look like this:

```
5B31        00001110
9F2C        00000010
```

The ID numbers (left column) are in hexadecimal and the access masks (right column) are in binary. This means that the ID number 5B31 has update, append, and read access rights, while the ID number 9F2C has the read access right.

The first entry in the file's access list is placed there automatically by the I/O System when it creates the file. The ID portion of that entry is the first ID number in the user object specified in the call that creates the file. The first ID is the owner ID for the file. The access rights portion is supplied as a parameter in the same call. The owner ID has full (unlimited) access to the file.

The system calls to add or delete ID-access pairs, or change the access rights of IDs already in the access list are **a_change_access** or **s_change_access**.

Note

Only the system manager and the file's owner can change the file's access list without being granted explicit permission to do so.

Computing Access for File Connections

Whenever a task creates a directory or creates or attaches a file, the I/O System constructs an access mask and binds it to the file connection object returned by the call. This access mask is constant for the life of the connection, even if the access list for the file is subsequently altered. When the connection is used to manipulate the file, the access mask for the connection determines how the file can be accessed. For example, if the computed access rights for a connection to a data file do not include appending or updating, that connection cannot be used for writing.

When a task uses BIOS calls to create a directory or file, the access mask for the connection is the same as the access mask that the task supplies in the `access` parameter of the system call. When a task uses EIOS calls to create a directory or file, the EIOS supplies an access mask that grants full access to the connection.

However, when a task attaches a file, the I/O System compares the user object specified in the `user` parameter with the file's access list and computes an aggregate mask.

Figure 17-2 on page 243 illustrates the algorithm that the I/O System uses during a call to attach a file. As the figure shows, the OS compares the IDs in the default user object with the IDs in the file's access list. The access masks corresponding to matching IDs are logically ORed, forming an aggregate mask.

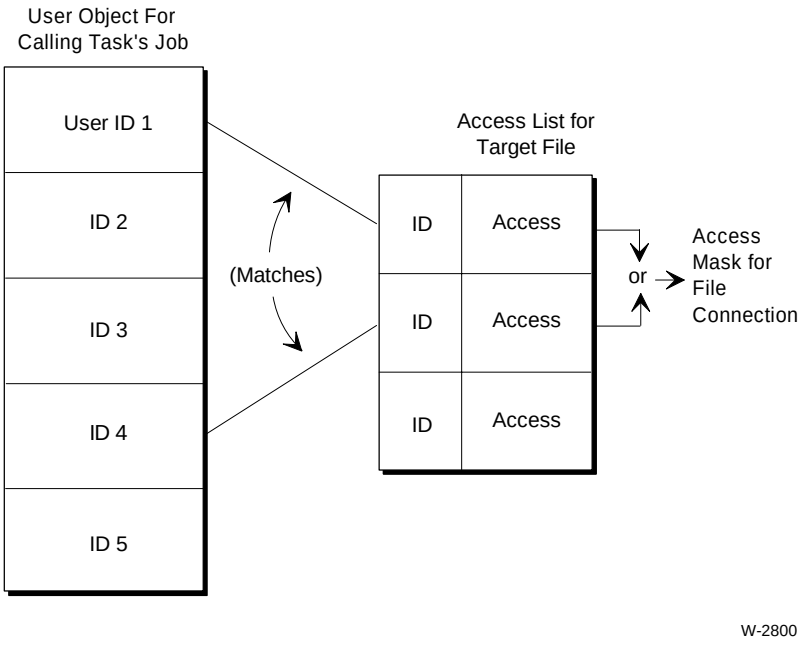


Figure 17-2. Computing the Access Mask for a File Connection

Normally, the I/O System uses the aggregate access mask embedded in the connection to determine a task's ability to access a file. However, there are two circumstances in which the I/O System computes access again: during **a_change_access** or **s_change_access**, and during **a_delete_file** or **s_delete_file**. When a task invokes one of these system calls, the I/O System computes the access to the target file (or to the data file or directory specified in the `prefix` parameter, if the subpath portion is null). If the user object specified in the system call does not have appropriate access rights, the I/O System denies the task the ability to delete the file or change the access.

Note

When computing access, the I/O System checks the access only to the last file in the specified subpath and to the parent directory of the last file. It does not check the access to any other directory files specified in the path. If the subpath is null, the BIOS checks the access to the file indicated by the `prefix` parameter.

File Access Rights Example

This example illustrates using IDs, access masks, access lists, and user objects to permit each user in a system to have exactly the kinds of access that you want that user to have.

This example shows that one ID number can give certain access rights to an individual and that another ID number can give different access rights to a collection of individuals. Here are the individuals and their access rights:

- Tom is to have full access to the file *batch_1*
- Bill is to have read and append access only
- Members of Department 2 are to have read access only

Tom (or whoever creates *batch_1*) can arrange for these kinds of access by doing:

1. Create a number of user objects, one for Tom, one for Bill, and one for each of the members of Department 2 (George, Harry, and Sam). When creating the user objects, assign unique owner IDs for each user: 4000H for Tom and 8000H for Bill. Assign unique owner IDs for each of the members of Department 2, but also include a common user ID, F000H, as an additional ID in each of their user objects.
2. Use **a_create_file** to create the file *batch_1*. Use the token for the user object containing the 4000H ID number and specify the access mask 00001111B. This call returns a file connection that gives Tom full access to *batch_1*. The access list for *batch_1* has just one ID-access mask pair.
3. Use **a_change_access** to add an ID-access mask pair to the access list of *batch_1*: ID 8000H and access mask 00000110B. This gives Bill read and append access to *batch_1*. Now the access list has two ID-access mask pairs.
4. Use **a_change_access** to add a third pair to the access list of *batch_1*: ID F000H and access mask 00000010B. This gives the people in Department 2 read access to *batch_1*.

Bill can read the contents of *batch_1* and append new information to it, if he knows the prefix and subpath that are needed to attach *batch_1* and he creates a user object with the ID 8000H. He specifies that user object when attaching *batch_1*.

The members of Department 2 can read the contents of *batch_1*, if they know the prefix and subpath that are needed to attach *batch_1* and they create a user object with the ID F000H. They specify that user object when attaching *batch_1*.

When Bill attaches *batch_1*, he receives a file connection that he can use to read the file. He also can write, provided that the file pointer for that connection is at the end of the file.

When a member of Department 2 attaches *batch_1*, he receives a file connection that he can use in calls to read the file.

Getting and Setting Extension Data

For each named file on a random access volume, the BIOS creates and maintains a file descriptor on the same volume. The first portion of the descriptor contains information for the BIOS. The last portion, called extension data, is available to your OS extension. You specify the number (from 0 to 255, inclusive) of bytes of extension data for each named file on the volume, when formatting the volume with the **format** command.

See also: **format** command, *Command Reference*

The BIOS system calls that enable you to record special information in the trailing portion of the file's descriptor and to access this data when it is needed later are **a_get_extension_data** and **a_set_extension_data**.

Maintaining Disk Integrity

The BIOS has several features that enable programs to maintain disk integrity and determine whether files or volumes have been corrupted. The next sections outline these features.

Attach Flags

The BIOS maintains flags that can indicate the integrity of named volumes and named files. When you attach a named volume, the BIOS sets a flag in the volume label to indicate that the volume is attached. When you attach a named file, the BIOS sets a flag in the *fnode* (file descriptor node) file to indicate that the file is attached. When you detach a volume or file, the BIOS clears the associated flag, indicating that the file or volume was successfully detached.

You can check the condition of a volume by invoking **a_get_file_status** or **s_get_file_status**. You can write your own programs to check the file flag, or you can use the Disk Verification Utility to examine the fnode file.

The Disk Verification Utility (DVU) enables you to inspect, verify, and correct the data structures of named or physical volumes. You can use the DVU to reconstruct the fnode file, the volume label, the fnode map, the volume free space map, and the bad blocks map of the volume.

See also: Disk Verification Utility, *Command Reference*

Fnode Checksum Field

The BIOS uses the fnode file to keep track of every named file on a volume. The fnode file lists such information as the file name, the creation and last modification dates, and the location of every disk sector that makes up the file. When you access a file, the BIOS uses the fnode file to determine the file's location on the volume. When you create, modify, or delete a file, the BIOS modifies the fnode file to match the changes you made.

When the last connection to the file is deleted, the BIOS writes to the fnode file, and calculates a checksum and writes that value in one of the fields of the fnode file. This checksum can be used to determine whether any data errors occurred when the BIOS wrote the fnode file. Your programs can use the checksum field to determine whether the fnode file has become corrupted. Using the **shutdown** command helps prevent fnode corruption; use the **diskverify** command to repair damaged files.

Getting and Setting the Bad Track/Block Information

It is not uncommon for a hard disk to have a few sectors or tracks that cannot reliably store information. Many of these disks have a record of these bad tracks written on the second-highest cylinder of the disk. When the BIOS formats a disk, it uses this bad track/sector information to assign alternate tracks or sectors for the bad tracks/sectors listed. The **a_special** system call also has the ability to retrieve and set the bad track/sector information on a volume. One subfunction enables you to retrieve the current list of defective tracks or sectors. Another subfunction enables you to set up a new bad track/sector list.

Bad tracks and bad blocks are different. Bad tracks are handled by the device drivers in conjunctions with the hardware, whereas bad blocks are handled by the Basic I/O System. The Disk Verification Utility mainly deals with bad blocks. It can view bad track information with **getbadtrackinfo** but the **format** command must be used to change it.

Note

Use the iRMX ability to read and set bad track and block information only with ST506 drives. Drive electronics on newer SCSI and IDE drives handle this mapping.

Accessing Remote Files

The I/O System supports the iRMX-NET local area network (LAN) by providing the Remote File Driver (RFD) and the **encrypt** system call. Remote (public) files are accessed by the RFD, which is similar to the Named File Driver.

The **encrypt** call encrypts passwords. You can use this system call to enable remote file access through iRMX-NET or in any application that needs to perform password encryption. No password decryption or data decryption facilities are provided in the iRMX OSs.

Systems that Include iRMX-NET

iRMX systems can be networked together using iRMX-NET. iRMX-NET gives you access to the files on hard disks of other systems on your network. The root directory of a remote device is referred to as a *virtual root*. The remote system selects the directories and files to be made accessible by using the **offer** command. Not all files and directories on a remote system are automatically accessible.

A file owner specifies what kind of access will be given to other users using the **permit** command. In iRMX-NET, giving or denying network file access is called making files *public* or *private*. Use the **offer** command to make files public. Files retain the same file permissions even when they are made public. Making directories public has the effect of making all files below that directory public.

Figure 17-3 illustrates public and private files on two networked systems. User Bob, working on the system shown on the right of the figure, is able to access the public data files on *system1*. Bob's files are not accessible from *system1*, because none of his files are public.

See also: **offer** command, *Command Reference*

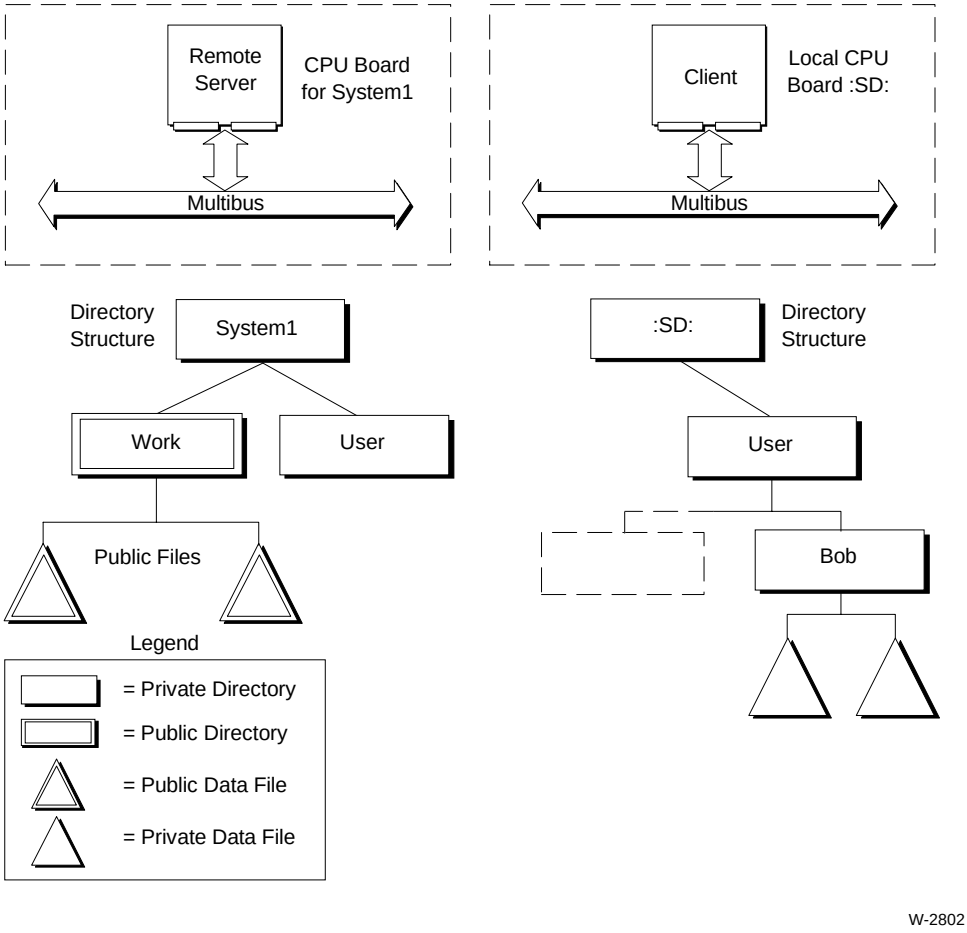


Figure 17-3. Example of Public and Private Files in an iRMX-NET System

Dynamic Logon and iRMX-NET

In a system that supports the dynamic logon facilities of the Human Interface or iRMX-NET, a *User Definition File* (UDF) lists the user name, password (in encrypted form), user ID, and other information about everyone who is allowed to log on to the iRMX system. The EIOS provides **get_user_ids** so that you can look up the permitted user ID of any user whose user name you know. This system call is useful for tasks that need to set up user objects based on the information listed in the UDF.

The EIOS also helps control remote file access through **verify_user**. This system call validates user names and passwords to ensure file security. As a result, the EIOS enables users to access remote files when logged on to dynamic terminals controlled by the Human Interface.

Access rights to remote files are slightly different than for named files.

See also: **permit**, *Command Reference*

Accessing NFS Files

On a TCP/IP network you can use NFS for transparent file access between systems. The NFS file driver enables application programs and users to access files on an NFS-shared resource. Before using NFS from a client system, you must define the files as NFS-shared on the server system. This section describes how file characteristics are mapped between operating systems when you use NFS. The NFS client or server software running on a non-iRMX OS (DOS or UNIX) is responsible for mapping file characteristics to or from files on the non-iRMX system during NFS file operations.

See also: **attachdevice** and **permit**, *Command Reference*
Using NFS, *TCP/IP and NFS for the iRMX Operating System*

Volume Names

The volume name for NFS files is the hostname. The number of free files is not returned to the iRMX OS when you access files using NFS.

File Names

NFS filenames cannot be longer than 14 characters. If you try to access a file whose name exceeds 14 characters, the system displays a truncated version of the name and marks it as “file not found”.

Non-iRMX hosts can further restrict filename lengths. For example, DOS machines limit filenames to eight characters followed by a three-character suffix.

NFS supports case-sensitive filenames. For hosts whose filenames are case-insensitive, the filename is converted to comply with the host. For example, if you use NFS to copy the file *My_Stuff.txt* to a DOS machine, it is saved as the DOS file *MY_STUFF.TXT*. Copying the same file to a UNIX host results in the file *My_Stuff.txt*. You need to reference files using the same case as they appear in the directory.

File Ownership

File ownership mapping occurs between iRMX, DOS, and UNIX files when using NFS. The following list describes the mapping:

- When you use NFS between two iRMX systems, file owners are maintained on a one-to-one basis.
- When you use NFS between an iRMX system and a UNIX system, the following mapping occurs regardless of which OS is the NFS client:

iRMX	UNIX
First owner in access list	"owner"
Second owner in access list	"group"
Third owner in access list	(ignored)
World	Owner is user ID 60000 and Group is user ID 1 (other)
Super	Owner and group user IDs are 0 (root)

Note

You can modify iRMX to UNIX file ownership mapping values for the World user by setting parameters in the */etc/stune.ini* file.

See also: Tunable Parameters, *TCP/IP and NFS for the iRMX Operating System*

- When you are the Super user on an iRMX client and you copy files to an NFS-shared file system on a UNIX host and the host does not allow root access, the files get an owner ID of 60001 (nobody) and a group ID 1 (other).
- When you use NFS between an iRMX system and a DOS system, file ownership mapping does not apply. This is because DOS has no concept of file owners. The NFS package you use on a DOS system may make certain assumptions. For example, a DOS-based NFS product might translate a file owned by user ID 0 (Super) as read-only from the DOS side. See the documentation for your non-iRMX NFS product for such details.

User ID Translation

User IDs map one-to-one across NFS except as noted for the Super and World users between iRMX and UNIX systems described in the previous section.

When you use NFS between two machines that happen to have different user accounts with the same user ID, the file's ownership is determined by the client's account. For example, assume that a file on an NFS server is owned by Sam with a user ID of 33. User Sarah on an NFS client also has a user ID of 33. If Sarah accesses the file on the NFS server through NFS, the user IDs map one-to-one. However, Sarah's access rights to the file will be whatever rights Sam has for the file on the server machine. Also, if Sarah lists the directory that contains the file, the owner will appear as Sarah, not Sam.

This user ID mechanism works similarly between iRMX systems or between iRMX and UNIX systems.

File and Directory Creation

When an iRMX user creates a file or directory across NFS, the default access rights are as follows:

UNIX Access Rights	DOS Access Rights
"owner" = "rwx"	read/write
"group" = "---"	(not applicable)
"other" = "---"	(not applicable)

Your NFS software on the non-iRMX host can further define these default access rights.

File Access Rights

When you change file access permissions programmatically or with the **permit** command from an iRMX client, the access rights are mapped through NFS as follows:

Setting any of these bits on an iRMX Client		**Results in all of these bits being set on iRMX, UNIX, and DOS Servers**		
	iRMX	**iRMX**	**UNIX**	**DOS**
Files	D-AU -R--	D-AU -R--	-w- r-x	read/write read-only
Directories	D-AC -L--	D-AC -L--	-w- r-x	read/write read-only

For example, if you set only the Delete (D) bit from an iRMX client system, this is translated across NFS to mean `D-AU` access on an iRMX server, `-w-` (write) access on a UNIX server, and read/write access on a DOS server.

When you change access permissions from another OS through NFS, the access permissions on an iRMX server are set as follows:

Setting any of these bits on UNIX and DOS Clients			**Results in all of these bits being set on an iRMX Server**
	UNIX	**DOS**	**iRMX**
Files	`-w-` `r-x`	read/write read-only	`D-AU` `-R--`
Directories	`-w-` `r-x`	read/write read-only	`D-AC` `-L--`

For example, if you set the read (`r`) bit or the execute (`x`) bit from UNIX, it results in a file with the "`-R--`" access on the iRMX server.

Accessing EDOS Files

The EDOS file driver enables application programs to access files on a DOS partition and uses DOS as a file server. Before using any DOS partition or diskette, attach the drive or the diskette using **attachdevice**.

See also: **attachdevice**, *Command Reference*

Directories

Users cannot rename a DOS directory or file to another subdirectory (such as renaming *dir1* to *dir3/dir1*). DOS directory files can only be read a multiple of 16 bytes at a time on 16-byte boundaries.

File Attributes

DOS file access attributes include read-only and read/write permission; iRMX access attributes include read, change-entry, delete, update, add-entry, and append. An iRMX user that has any of delete, change, update, add, or append permission has write permission for DOS files.

The DOS user always has read (list) access to DOS files and directories; write (delete, append, update, add-entry, and change-entry) access is optional. The DOS user must have write access to the file to rename it or to delete a connection to it. DOS and the iRMX OS have different ways for handling invisible files.

See also: Invisible files, *iRMX Programming Concepts for DOS and Windows*

File Names

DOS filenames must be eight characters or less in length, with a three character (or less) extension. DOS truncates iRMX filenames, which may be up to 14 characters and may contain one or more . (period). Any DOS filename is a valid iRMX filename, but the converse is not true.

Time Stamps

The `create_time`, `access_time`, `and modify_time` elements are not valid for DOS files. The only time stamp for DOS files is creation time or last-modified time.

File Ownership

`Owner_access` does not apply to DOS files because DOS does not support multiple file owners. EDOS files have only one user, which is World.

Accessing DOS Files

The DOS file driver enables application programs to access files on a DOS partition and uses DOS as a file server. Before using any DOS partition or diskette, attach the drive or the diskette using **attachdevice**.

See also: **attachdevice**, *Command Reference*

Directories

DOS directory files can only be read a multiple of 16 bytes at a time on 16-byte boundaries.

File Attributes

DOS file access attributes include read-only and read/write permission; iRMX access attributes include read, change-entry, delete, update, add-entry and append. An iRMX user that has any of delete, change, update, add, or append permission has write permission for DOS files.

The DOS user always has read (list) access to DOS files and directories; write (delete, append, update, add-entry and change-entry) access is optional. The DOS user must have write access to the file to rename it or to delete a connection to it. DOS and the iRMX OS have different ways for handling invisible files.

See also: Invisible files, *Programming Concepts for DOS and Windows*

File Names

DOS filenames must be eight characters or less in length, with a three character (or less) extension. DOS truncates longer iRMX filenames, which may be up to 14 characters and may contain one or more . (period). Any DOS filename is a valid iRMX filename, but the converse is not true.

Time Stamps

The `create_time`, `access_time`, `and modify_time` elements are not valid for DOS files. The only time stamp for DOS files is creation time or last-modified time.

File Ownership

`Owner_access` does not apply to DOS files because DOS does not support multiple file owners. DOS files have only one user, which is World.

Accessing CDROM Files

The CDROM file driver enables application programs to access files on a CDROM. Before using any CDROM, attach the drive using **attachdevice**.

See also: **attachdevice**, *Command Reference*

Directories

CDROM directory files can only be read a multiple of 16 bytes at a time on 16-byte boundaries.

File Attributes

CDROM files are read-only.

File Names

CDROM filenames must conform to ISO9660. The CDROM file driver cannot read any other format currently. Filenames must be eight characters or fewer in length, with a three character (or fewer) extension.

File Ownership

`Owner_access` does not apply to CDROM files.

Using Nucleus System Calls for the Default User and Default Prefix

Several system calls provided by the Nucleus allow you to specifically manipulate user objects and prefix objects.

- **catalog_object**
- **uncatalog_object**
- **lookup_object**

The default user and default prefix for each I/O job are cataloged in the job's object directory.

System Calls for Named Files

Some system calls are useful for both data and directory files, some for only one kind of file, and some (such as **create_user**) do not relate directly to either kind of file. Generally, system calls that relate to named files also relate to remote files and DOS files.

The brief descriptions in Tables 17-1 through 17-**Error! Reference source not found.** on pages 259 through **Error! Bookmark not defined.** are grouped by function, not alphabetically. Where a prefix is not used, an **a_** prefix is required for BIOS system calls and an **s_** prefix for EIOS system calls. For example, the full syntax for the BIOS system call for **create_file** is **rq_a_create_file**.

BIOS and EIOS System Calls for Named Files

Table 17-1. Getting and Deleting Connections

Call	Target	Used To
create_file	data	Create a new data file and automatically add an entry in the parent directory. Obtain a connection to an existing data file.
create_directory	directory	Create a new directory file and automatically add an entry in the parent directory.
attach_file	data and directory	Obtain a connection to an existing data or directory file.
delete_connection	data and directory	Delete a file connection, not a device connection.
***a_physical_attach_device**	device	Obtain a connection to a device.
a_physical_detach_device	device	Delete a connection to a device.
***s_logical_attach_device**	device	Obtain a connection to a device and catalog the logical name for the device in the object directory of the root job.
s_logical_detach_device	device	Delete a connection to a device and remove the logical name of the device from the object directory of the root job.
hybrid_detach_device	device	Delete a connection to a device. Does not remove the device's logical name from the object directory of the root job. Use to temporarily attach a device in a different manner.

* For **a_physical_attach_device** and **s_logical_attach_device**, the device connection can be used as the prefix for the root directory of the device.

Table 17-2. Getting and Setting Default Prefixes

Call	Target	Used To
***set_default_prefix**	job	Set the default prefix for any iRMX job and catalog the connection under the name *$* in the job's object directory.
***get_default_prefix**	job	Determine the default prefix for any iRMX job.

***** These system calls do not require a prefix of **a**_ or **s**_.

Table 17-3. User Objects

Call	Target	Used To
*create_user	user object	Create a user object and return a token to the calling task.
*delete_user	user object	Delete an existing user object.
*inspect_user	user object token	Return the ID list in an existing user object to the calling task.
*set_default_user	user object	Establish a default user for any existing iRMX job.
*get_default_user	user object	Determine or change the default user for any existing iRMX job.

* These system calls do not require a prefix of **a**_ or **s**_.

Table 17-4. Using Data

Call	Target	Used To
open	data and directory	Open a connection to the file.
close	data and directory	Close the file connection.
seek	data	Position the file pointer of the file connection. Tells the BIOS the location in the file where the read, write or truncate operation is to take place. Requires that the file connection be open.
a_read **s_read_move**	data and directory	Read file data from the location indicated by the file pointer and place the data in a memory buffer. Use the **seek** system call to position the file pointer. Requires that the file connection be open. Requires that the segment to which you copy the data be writable.
a_write **s_write_move**	data	Copy information from a memory buffer and place it in the file at the position indicated by the file pointer. Use **seek** to position the file pointer. Requires that the file connection be open. Requires that the segment from which you copy the data be readable.
a_truncate **s_truncate_file**	data data	Drop information from the end of the file. Use **a_seek** to position the file pointer at the first byte to be dropped. Requires that the file connection be open.
*__wait_io__	file	Receive the concurrent condition code of the prior system call and the number of bytes read or written. Use after **a_read**, **a_write**, or **a_seek.**
*__a_update__	BIOS	Transfer data remaining in internal buffers immediately to the files on a device. Use to ensure that all files on removable volumes (such as diskettes) are updated before removal.

* These system calls do not require a prefix of **a**_ or **s**_.

For **close**, the application can elect to leave the file open, letting the BIOS close it when the connection is deleted, but when a connection is shared between two or more applications, some of the applications can place restrictions on the manner of sharing. For instance, an application can specify sharing with writers only. By closing connections, the application can improve the likelihood that the connections can be used by other applications. A connection is not closed until all pending I/O requests have been handled.

Each entry in a directory consists of 16 bytes. The first two bytes contain a 16-bit file descriptor number corresponding to the file descriptor number associated with **get_file_status**. The remaining 14 bytes are the ASCII characters making up the name of the file to which the directory entry points. A file's name is the last component of a pathname. Using **read** to read a directory lets the application obtain several entries with one operation.

Table 17-5. Getting Status

Call	Target	Used To
get_file_status	data and directory	Get file status.
get_connection_status	data and directory	Get connection status.
*__get_logical_device_status__	device	Retrieve information about devices.

* This system call does not require a prefix of **a**_ or **s**_.

Table 17-6. Reading Directory Entries

Call	Target	Used To
a_read	directory	Get contents of the directory.
a_get_directory_entry	directory	Read directory entries; can be used without opening a connection.

Note: These system calls are for the BIOS only.

Table 17-7. Deleting and Renaming Files

Call	Target	Used To
delete_file	data and directory	Delete files or empty directories.
rename_file	data and directory	Rename files or directories. Add entries to directories.

Deleting a file involves two steps. First, call **a_delete_file**. This marks the file for deletion. The second step, actual deletion, is performed by the BIOS. The BIOS deletes marked files only after all connections to the file have been deleted.

For **rename_file**, the application can move the file to any directory in the same named file tree. For example, you can rename *A/B/C* to be *A/X/C*. This example moves file *C* from directory *B* to directory *X*. This means that the application can change every component of a file's pathname except the root directory.

See also: Accessing DOS and EDOS Files, in this chapter;
rename_file system call, *System Call Reference*

Table 17-8. Changing Access

Call	Target	Used To
change_access	data and directory	Change the file's access list, or change access rights of files in a directory, when used by only the owner of a file or a user with change entry access to the directory containing the file.

Table 17-9. Identifying a File's Name

Call	Target	Used To
a_get_path_component	data and directory	Find out the last component of a file's pathname. Use repeatedly to obtain the entire pathname for a file.

Note: This system call is for the BIOS only.

Table 17-10. Changing Extension Data

Call	Target	Used To
a_set_extension_data	data and directory	Writes extension data. Use even if the file connection is not open.
a_get_extension_data	data and directory	Reads extension data. Use even if the file connection is not open.

Note: These system calls are for the BIOS only.

When you format a volume to accommodate named files, you have the option of allowing each file to include extension data.

Table 17-11. Detecting Changes in Device Status

Call	Target	Used To
a_special	device	Perform functions that are device dependent, such as formatting a disk or setting terminal characteristics.

Note: This system call is for the BIOS only.

Table 17-12. Deleting Connections

Call	Target	Used To
s_delete_connection	data and directory	Delete a file connection, not a device connection.

Note: This system call is for the EIOS only.

Table 17-13. Using Logical Names

Call	Target	Used To
s_catalog_connection	object directory	Create a logical name by cataloging a connection in the object directory of a job.
s_lookup_connection	object directory	Accept a logical name from an application task, look up the name in the object directories of the local, global, and root jobs (in that sequence), and return a token for the first connection found.
s_uncatalog_connection	object directory	Delete a logical name from the object directory of a job.

Note: These system calls are for the EIOS only.

Table 17-14. Creating and Deleting I/O Jobs

Call	Target	Used To
***create_io_job**	I/O job	Create an I/O job while the system is running. Available for compatibility with the iRMX I OS. The memory pools associated with those I/O jobs cannot exceed 1 Mbyte. Specify if you want the initial task to start running automatically, or wait until **start_io_job**.
***rqe_create_io_job**	I/O job	Create an I/O job while the system is running. The memory pools can be up to 4 Gbytes for iRMX III systems. Use this system call (instead of **create_io_job**) for all new applications, because it takes full advantage of iRMX features.
***start_io_job**	I/O job	Start the initial task in an I/O job.
***exit_io_job**	I/O job	Terminate an I/O job and inform the parent job of the termination.

Note: These system calls are for the EIOS only and do not require a prefix of **a_** or **s_**.

These EIOS system calls perform operations that do not fit into any other category.

Table 17-15. Miscellaneous Functions

Call	Target	Used To
s_special	file connection	Perform functions that are device dependent, such as formatting a disk or setting terminal characteristics.
s_get_directory_entry	filename	Look up the name of any file in a directory.
s_get_path_component	filename	Look up the name of a file as it is known in the file's parent directory.

Note: These system calls are for the EIOS only.

See also: BIOS and EIOS calls, *System Call Reference*

Call Sequence for Named Files

System calls for named files cannot be used in arbitrary order. The following figure shows the sequence for the most frequently used I/O System calls. Start with the leftmost box and follow the arrows. Any path that you can trace is a legitimate sequence of system calls. Figure 17-4 on page 267 is not a complete list of all sequences.

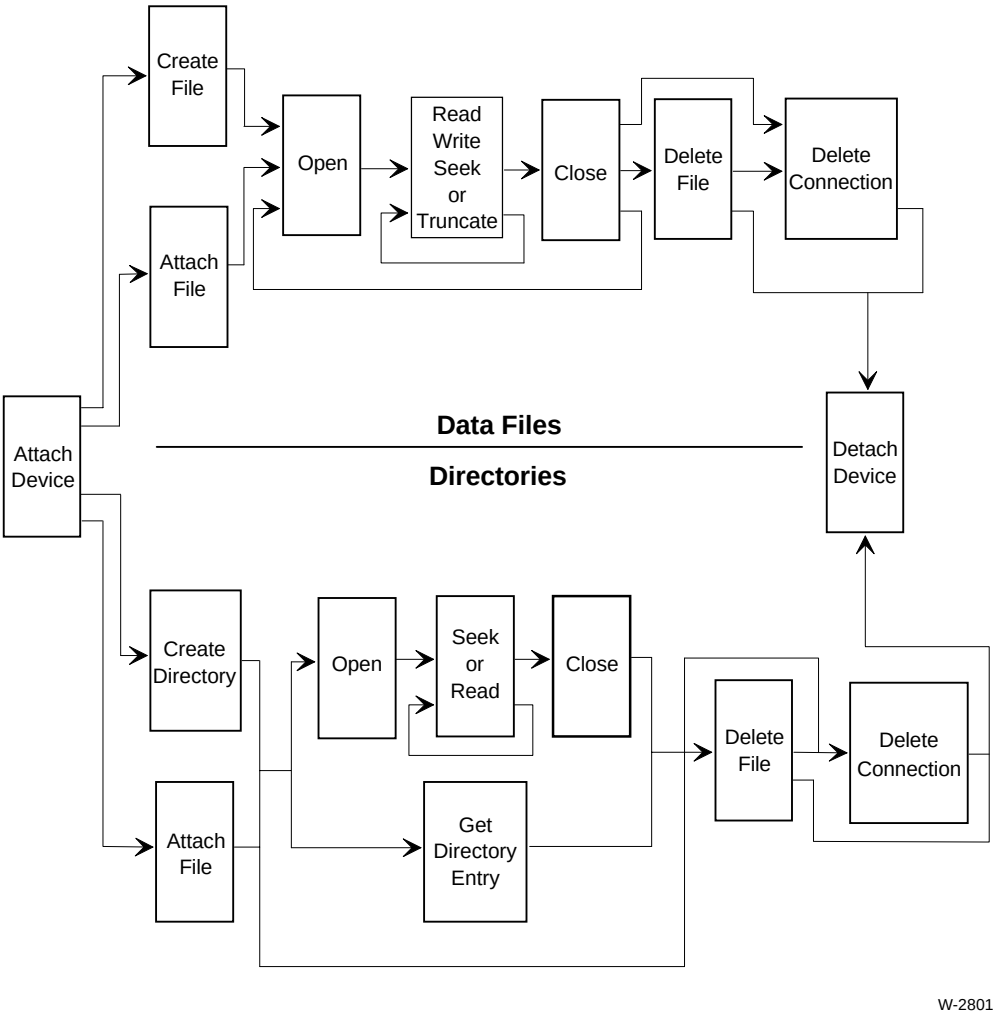


Figure 17-4. Sequence of Frequently Used System Calls for Named Files

❑❑❑

Physical Files 18

Physical files enable applications to read or write strings of bytes from or to a device. A physical file occupies an entire device or the device's entire volume; the I/O System provides applications with the ability to access the device driver directly.

Although there is a one-to-one correspondence between the bytes on a device and the bytes of a physical file, the device connection is different from the file connection.

Situations Requiring Physical Files

Physical files are useful when the application system uses sequential devices, such as line printers, display tubes, plotters, and magnetic tape units.

Physical files are also useful to communicate with random access devices, such as disk drives and diskette drives, in these situations:

- When formatting volumes, the task accesses every byte on the volume. Only physical files provide this kind of access.
- When using volumes in formats other than the iRMX format, you must use physical files. Tasks will have to interpret information such as labels and file structures, but a physical file can provide tasks with access to the raw information.
- When implementing your own file format, such as a structure different from iRMX named files, you can build a custom file structure using a physical file as a foundation.

Maintaining Physical File Independence

To allow application tasks to use stream or named files in addition to physical files, create two tasks: one to obtain a connection to the file and one to use the connection to perform I/O. By maintaining this separation, the second task can work with any kind of file.

To use this two-task approach, be sure that both tasks are in the same job. This avoids passing a file connection from one job to another.

BIOS Calls for Physical Files

1. Obtain a device connection to tell the I/O System that the file is a physical file and which device contains the file using

   a_physical_attach_device

2. Obtain a file connection using

   a_create_file or **a_attach_file**

   - For **a_create_file**, use the device connection token as the `prefix` parameter to tell the BIOS which device you want as the physical file.
   - For **a_attach_file**, use the device connection for the device, or use an existing file connection to the file as the `prefix` parameter in the system call.

3. Open the file connection using

   a_open

4. Use the file. There are four system calls that can read, write, or otherwise use the physical file.

   These system calls read and write information from or to the physical file:

   a_read and **a_write**

   This call moves the file connection's file pointer if the device is a random access device such as a disk or diskette.

   a_seek

   This call requests device dependent functions from the device driver.

   a_special

   Tasks can use this call to format a disk for use with the iRMX OS. The kinds and number of functions supported depends upon the device and device driver. Using special functions generally prevents a task from being device independent.

5. Close the file connection using

   a_close

   This is important if the connection share mode restricts the use of the file through other connections. The application can repeat steps 2, 3, 4, and 5 any number of times.

6. Delete the connection using

   a_delete_connection

   This is only necessary if the tasks of the application are completely finished using the file.

7. Detach the device when the task no longer needs the device using

   a_physical_detach_device

   See also: BIOS calls, *System Call Reference*

EIOS Calls for Physical Files

1. Obtain a device connection to tell the I/O System that the file is a physical file and which device contains the file using

   logical_attach_device

   The application program must use the device name that was assigned to the device during system configuration. This system call obtains a device connection and catalogs the connection under the specified logical name. Other tasks wishing to use the device connection can look up the connection by using the logical name.

   See also: **attachdevice**, *Command Reference*;
   for ICU-configurable systems, *ICU User's Guide and Quick Reference*

2. Obtain a file connection using

   s_create_file or **s_attach_file**

   - For **s_create_file**, use the `path_ptr` parameter to point to an iRMX STRING containing the device's logical name enclosed in colons, as in *:F0:*. This tells the EIOS which device you want as the physical file.
   - For **s_attach_file**, use the `path_ptr` parameter of the call to point to an iRMX STRING containing the device's logical name enclosed in colons, as in *:F0:*, or use the `path_ptr` parameter of the call to point to an iRMX STRING containing the connection's logical name enclosed in colons, as in *:database:*.

3. Open the file connection using

   s_open

   The task must also specify how many buffers the EIOS can use when reading from or writing to the file.

4. Use the file. There are four system calls that can read, write, or otherwise manipulate the physical file.

   These system calls read and write information from or to the physical file:

   s_read_move or **s_write_move**

   This call moves the file connection's file pointer if the device is a random access device such as a disk or diskette.

   s_seek

   If you are writing a device driver for a magnetic tape unit, you can design it to support **s_seek**.

   This system call requests device dependent functions from the device driver.

   s_special

   Tasks can use these calls to format a disk for use with the iRMX OS. The kinds of and number of functions supported depends upon the device and device driver. Using special functions generally prevents a task from being device independent.

5. Close the file connection using

   s_close

   This is important if the connection share mode restricts the use of the file through other connections. The application can repeat steps 2, 3, 4, and 5 any number of times.

6. Delete the connection using

   s_delete_connection

   This is only necessary if the tasks of the application are completely finished using the file.

7. Detach the device when the task no longer needs the device using

   logical_detach_device

   See also: EIOS calls, *System Call Reference*

Call Sequence for Physical Files

You can use several system calls with physical files. Figure 18-1 shows the system call sequence for physical files. To use the figure, start with the leftmost box and follow the arrows. Any path that you can trace is a legitimate sequence of system calls. The steps on the next pages provide a brief description of how an application can use a physical file.

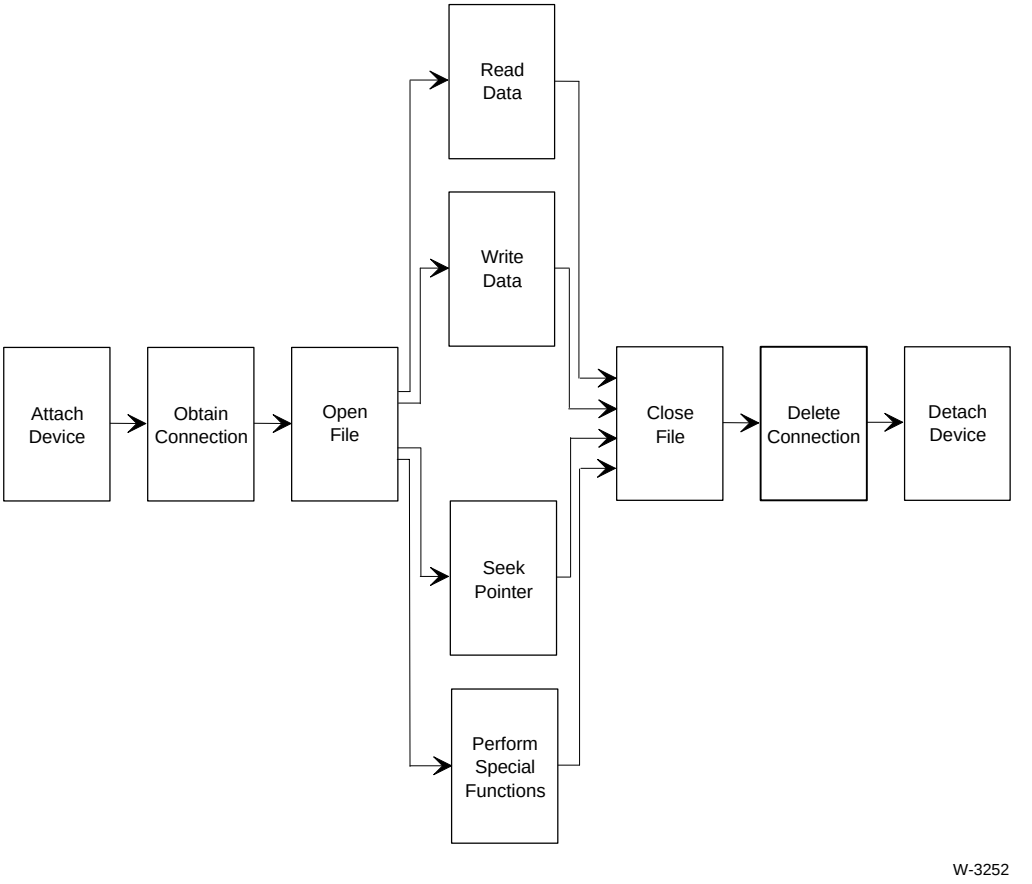


Figure 18-1. Sequence of System Calls for Physical Files

□□□

Stream Files 19

Stream files enable one task to send large amounts of information to a second task, even when the two tasks are in different jobs. The first task communicates with the second task as though the second task were a device. This extends device independence to include tasks.

Stream files are only one of several techniques for job-to-job communication.

Maintaining Stream File Independence

Two tasks, the reading task and the writing task, are always involved in using a stream file. To allow your reading and writing tasks to use named files or physical files in addition to stream files, add a third task to the application: creating the file. This enables both the reading and writing tasks to be independent of the kind of file being used.

Creating the File

The creating task obtains a device connection to the stream file device and creates the stream file. It also catalogs the file connection under a logical name so the reading and writing tasks can attach the file. This task is not device independent; it works only for stream files.

BIOS Calls for Creating Stream Files

1. Obtain a connection to the stream file device using

   a_physical_attach_device

   Use the configured stream file name, typically *stream*, for the `dev_name_ptr` parameter. For stream files, there is only one device.

2. Create the stream file and obtain a token for a file connection using

   a_create_file

   Use the token for the device connection as the `prefix` parameter, to tell the BIOS to create a stream file.

   A_create_file examines the device connection to determine what kind of file to create.

3. Pass the file connection to the reading task.

   There are several ways of doing this, including using object directories and mailboxes.

EIOS Calls for Creating Stream Files

1. Create a stream file using

   s_create_file

   Use a `path_ptr` parameter pointing to an iRMX STRING of this form:

   :stream_filename:

   Where:

   stream_filename is the logical name for the stream file device connection.

   S_create_file returns a connection to the newly created stream file.

   The logical name for the stream file device is a configuration parameter. During system initialization, the EIOS attaches the stream file device and catalogs the device connection under that logical name. Your tasks can use the logical name to obtain the device connection.

2. Catalog the file connection under a unique logical name for each specific stream file using

   s_catalog_connection

   The reading and writing tasks can then use the logical name to attach the file.

Writing the File

The writing task obtains a device connection to the stream file device and opens the file for writing. It also closes and removes the connection. Figure 19-1 on page 281 illustrates the file writing process.

BIOS Calls for Writing Stream Files

1. Open the file for writing using

   a_open

   Use the token for the file connection as the `connection` parameter. Set the `mode` parameter for writing; set the `share` parameter for sharing only with readers.

2. Write information to the stream file using

   a_write

   Use the token for the file connection as the `connection` parameter. Use multiple invocations of **a_write** if necessary. In this case, the BIOS uses the concurrent part of the call to synchronize the writing and reading tasks. The BIOS sends a response to each invocation of **a_write** only after the reading task has finished.

3. Close the connection using

   a_close

   The writing task can repeat steps 1, 2, and 3 as many times as needed.

4. Delete the connection using

   a_delete_connection

EIOS Calls for Writing Stream Files

1 Obtain a connection to the stream file using

   s_attach_file

   Set the `path_ptr` parameter of the system call to point to an iRMX STRING containing the file connection's logical name, enclosed in colons as in *:sf23:*.

2. Open the file connection for writing using

   s_open

   Use the token of the file connection for the `connection` parameter and set the `mode` parameter to write.

3. Write information to the stream file using

   s_write_move

   Use the token for the file connection as the `connection` parameter.

4. Close the connection when finished writing to the stream file using

   s_close

   The writing task can repeat steps 2, 3, and 4 any number of times.

5. Delete the connection using

   s_delete_connection

Reading the File

The reading task obtains a device connection to the stream file device and opens the file for reading. It also closes and removes the connection. Figure 19-1 on page 281 illustrates the file reading process.

BIOS Calls for Reading Stream Files

The reading task performs these steps to successfully read the information written by the writing task:

1. The reading task must have a different file pointer than the writing task. Create a file connection for the stream file using

   a_attach_file

   Set the `prefix` parameter to the token for the original file connection.

   The reading task can also use **a_create_file** to obtain the new connection to the same stream file. If the specified `prefix` parameter is a device connection, the BIOS will create a new file and return a connection for it. If the specified parameter is a file connection, the BIOS will just create another connection to the same file.

2. Open the file connection for reading using

   a_open

   Use the token of the file connection for the `connection` parameter. Set the `mode` and `share` parameters to read and sharing with all connections to the file.

3. Read information from the stream file using

   a_read

   Read the file until reading is no longer necessary or until an end-of-file condition is detected. Use the token for the file connection as the `connection` parameter.

4. Close the connection when finished reading from the stream file using

   a_close

   The reading task can repeat steps 2, 3, and 4 any number of times.

5. Delete the connection using

   a_delete_connection

   The writing task deletes the old connection, and, as soon as both connections have been deleted, the BIOS deletes the stream file.

EIOS Calls for Reading Stream Files

The reading task performs these steps to successfully read the information written by the writing task:

1. The reading task must have a different file pointer than the writing task. Create a file connection for the stream file using

   s_attach_file

   Set the `path_ptr` parameter to point to an iRMX STRING containing the file connection's logical name enclosed in colons, as in *:sf23:*.

   The reading task can also use **s_create_file** to obtain the new connection to the same stream file. If the specified `prefix` parameter is a device connection, the EIOS will create a new file and return a connection for it. If the specified parameter is a file connection, the EIOS will just create another connection to the same file.

2. Open the file connection for reading using

   s_open

   Use the token of the file connection for the `connection` parameter. Set the `mode` and `share` parameters to read and sharing with all connections to the file.

3. Read information from the stream file using

   s_read_move

   Read the file until reading is no longer necessary or until an end-of-file condition is detected. Use the token for the file connection as the `connection` parameter.

4. Close the connection when finished reading from the stream file using

   s_close

   The reading task can repeat steps 2, 3, and 4 any number of times.

5. Delete the connection using

   s_delete_connection

6. Delete the file's logical name created by the creating task using

   s_uncatalog_connection

   Do not delete the logical name for the stream file device.

7. Delete the file connection created by the creating task using

   s_delete_connection

   The reading task deletes the file connection that the creating task obtained. Once this connection is deleted, the EIOS automatically deletes the stream file.

See also: BIOS and EIOS calls, *System Call Reference*

Call Sequences for Stream Files

Figure 19-1 on page 281 illustrates three tasks: one each for creating, writing, and reading the file. The writing task can create the file before it performs the write, but this forces the writing task to use only stream files.

This figure shows the system call sequence for stream files. To use the figure, start with the leftmost box and follow the arrows. Any path that you can trace is a legitimate sequence of system calls. The sequences of steps on the next pages work even if the three tasks are in different jobs. They also work regardless of the order in which they are executed.

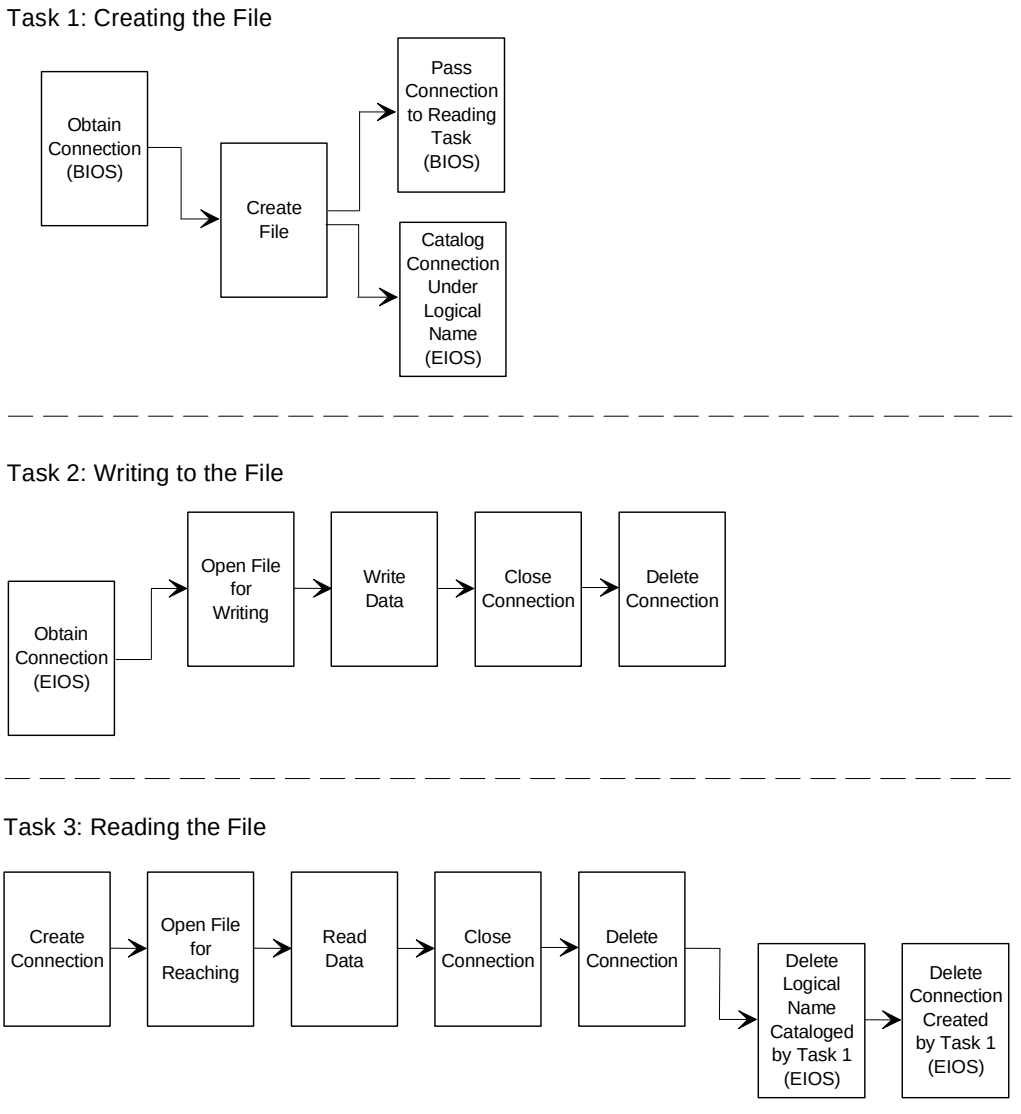


Figure 19-1. Sequence of System Calls for Stream Files

❑❑❑

Connections and Objects 20

Cataloging Connections

Use **s_catalog_connection** to control which directory the connection is cataloged in, depending on the specified job token.

- To share a connection with tasks in the same job, but not other jobs, catalog the token for the connection under a logical name in the local object directory.
- To share connections among tasks in several jobs, designate one global job for a user session. Then catalog tokens for shared connections in the global job object directory.
- To share certain connections with all tasks in the system, catalog tokens for the connections in the root job's directory.

Note

Before an I/O job exits, it must uncatalog any tokens it cataloged in other directories (global or root). If it does not and the logical name and token remain even though the connection is deleted, other tasks referring to the logical name or attempting to use the connection will receive an error.

Cataloging Objects

The EIOS catalogs entries in the object directory of the system's root job and each I/O job. This is a list of the names that the EIOS uses.

rqglobal — The EIOS uses this name to identify the user session's global job for each I/O job. Whenever you create an I/O job, the EIOS automatically catalogs the token for the global job in the object directory of the I/O job. You may redefine this name, but doing so may alter the interpretation of any logical names that are cataloged in the object directory of your job's global job.

r?iojob — Whenever you create an I/O job, the EIOS catalogs an object under this name in the object directory of the I/O job. **Do not redefine this name!**

r?message — Whenever you create an I/O job, the EIOS catalogs an object under this name in the object directory of the I/O job. **Do not redefine this name!**

r?iouser — Whenever you create an I/O job, the EIOS catalogs an object under this name in the object directory of the I/O job. **Do not redefine this name!**

$ — The EIOS uses this name to catalog the default prefix for each I/O job. If you modify the definition associated with this name by invoking **catalog_object**, you change the job's default prefix. If you catalog an object other than a device connection or a file connection under this name, the EIOS generates a condition code whenever you attempt to use the default prefix.

With the exception of *rqglobal* and *$*, do not use **catalog_object** to modify any of the definitions described here. If you do, the results will be unpredictable.

The EIOS uses object directories for two other purposes:

- Whenever you use **catalog_connection** to define a logical name for a connection, the EIOS catalogs the connection in the object directory of the job that you specify.
- Whenever you use **logical_attach_device**, the EIOS catalogs the device connection in the object directory of the system's root job.

❑❑❑

UDI Basic Concepts and System Calls 21

The Universal Development Interface (UDI) is a set of system calls compatible with multiple OSs. If an application program makes only UDI system calls, it can be transported between OSs. You can use the UDI as an alternative to the iRMX I/O Systems; if you do so, you should only use UDI calls for I/O operations.

Figure 21-1 illustrates the relationship between application code, processing hardware, and layers of software. The downward arrows represent command flow and data flow from the application code to the hardware. All interaction between the application code and the OS is through the UDI software.

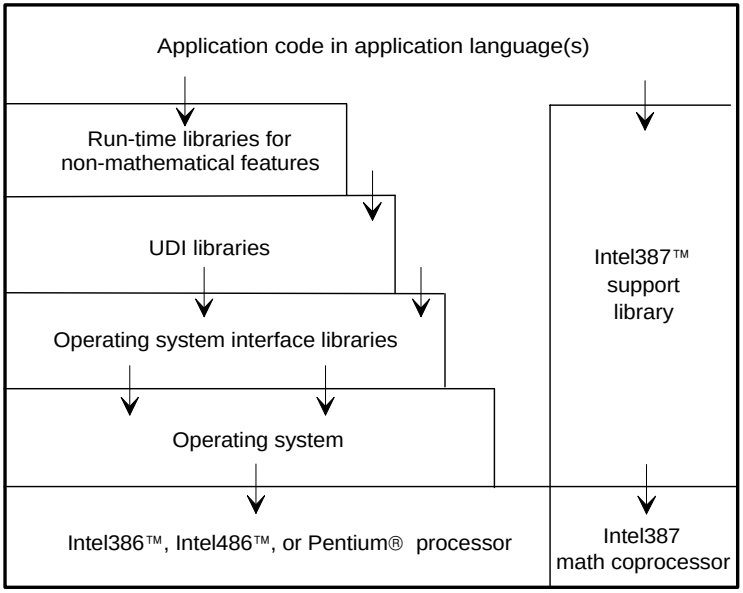


Figure 21-1. The Application Software-Hardware Model

To make an application transportable between OSs, you need a UDI library for each OS. All libraries present the same interface to applications. UDI OS interfaces, however, are designed for specific operating systems, including the iRMX, iNDX, UNIX, and XENIX OSs.

The UDI system calls, while presenting a standard interface to user programs, behave somewhat differently when used in different OS environments. This is because different OSs have unique characteristics.

UDI System Calls

The calls are divided into functional groups.

UDI Memory Management System Calls

When iRMX OSs load and run a program, the program is allocated memory. The portion of memory not occupied by code and data, the free space pool, is available dynamically while the program runs. The OS manages this memory as segments that programs can obtain, use, and return.

Programs can use the UDI system calls **dq_allocate** and **dq_mallocate** to get memory segments from the pool. They can use the system calls **dq_free** and **dq_mfree** to return segments to the pool. Programs can also call **dq_get_size** and **dq_get_msize** to receive information about allocated memory segments.

You can reserve memory for the I/O System by using the system call **dq_reserve_io_memory**. This ensures that the OS allocates memory to accommodate the buffers needed to open files.

Dq_reserve_io_memory is particularly useful to an application that has used all of its allocated memory and must open a temporary file to store data. The system call reserves additional memory for this purpose. If an application program has not invoked **dq_reserve_io_memory** and is out of memory, the OS returns an E_MEM condition code when the application tries to create a temporary file.

A program obtains a connection by calling **dq_attach** (if the file already exists) or **dq_create** (to create a new file). **Dq_detach** deletes the connection. To delete both the connection and the file, use **dq_delete**.

Once a program has a connection, it calls **dq_open** to prepare the connection for I/O operations. The program performs input or output operations using **dq_read** and **dq_write**. It can move the file pointer associated with the connection by calling **dq_seek**. It can truncate the file by calling **dq_truncate**.

When the program finishes input and output to the file, it closes the connection by calling **dq_close**. The program closes the connection, not the file. Unless the program deletes the connection, by calling **dq_detach**, it can continue to open and close the connection as necessary.

If a program calls **dq_delete** to delete a file, the file cannot be deleted while other connections and I/O requests exist. In that case, the file is marked for deletion but is not actually deleted until the last of the connections is deleted. During the time that the file is marked for deletion, no new connections or I/O requests to the file may be issued.

Using Program Control Calls

UDI provides two system calls for program control: **dq_exit** and **dq_overlay**.

Dq_exit terminates a program, closing all open files and freeing allocated resources. You should always include this system call as the last statement in your program.

Dq_overlay lets you take advantage of the overlay support provided by the OS. This system call loads an overlay into memory.

Note

Prepare the overlay with the BND binder and the OVL286 overlay generator.

Using Utility and Command-parsing Calls

UDI provides system calls for command parsing, date stamping, time stamping, and system identification. The system calls are **dq_get_time**, **dq_decode_time**, **dq_get_system_id**, **dq_get_argument**, and **dq_switch_buffer**.

Dq_get_time and **dq_decode_time** return the date and time information maintained by the OS. Both calls provide the same kind of information, but **dq_get_time** is provided for compatibility with previous releases. Use **dq_decode_time** instead of **dq_get_time** when possible.

Dq_get_system_id returns a string that identifies the name of the OS. This system call is useful for programs that need to perform operating-system-specific functions.

Dq_get_argument and **dq_switch_buffer** enable programs to retrieve parameters from the command line (or from any other program buffer). **Dq_switch_buffer**

switches to a new buffer so that the next time you call **dq_get_argument**, you will retrieve a parameter from the new buffer.

Dq_get_argument parses the command line, returning the next parameter in the sequence. The parameters are separated by delimiters, which include the space, <CR>, ASCII character values ranging from 1 through 20H and 7FH through 0FFH, and these:

```
,  )  (  =  #  !  %  +  -  &  ;  <  >  [  ]  \  '  |  ~
```

Using Condition Codes and Exception-handling Calls

Every UDI call (except **dq_exit**) returns a numeric condition code specifying the result of the call. Each condition code is equated with a label. For example, the code 0 has the name E_OK. E_OK indicates that a call has been successful. Conditions may also indicate a problem or require a response (exceptional conditions). The **dq_decode_exception** returns the mnemonic description of any condition code generated by a UDI system call.

See also: Condition codes, *System Call Reference*

A routine in the UDI interface library called **rq_error** handles UDI exceptional conditions. This routine is called whenever a condition code is generated by a UDI system call. **Rq_error** performs these operations:

- If an environmental condition occurs (device error, incorrect file reference, insufficient memory, etc.), the condition code is returned to the calling program. The calling program handles the exceptional condition inline.
- If a programmer error occurs, **rq_error** invokes the Nucleus system call **signal_exception**. The action that **signal_exception** takes depends on the Nucleus exception mode. If the exception mode is `never` (the default) or `environ`, **signal_exception** passes control back to the calling program so that it can process the exceptional condition inline. If the exception mode is `all` or `program`, **signal_exception** passes control to the exception handler that is in effect at the time the exception occurs.

See also: **signal_exception**, Nucleus calls, *System Call Reference*

Overriding the <Ctrl-C> handler

UDI provides a method for a program to handle <Ctrl-C> characters entered while the program is running. The system default <Ctrl-C> handler terminates any program that is active when <Ctrl-C> is entered. However, a program can override the default handler for the duration of its execution by calling **dq_trap_cc** and supplying a long pointer to a new <Ctrl-C> handler. The OS will call this new <Ctrl-C> handler whenever a <Ctrl-C> is typed at the terminal. The new handler remains in effect until the program calls **dq_exit**, or until it establishes another handler by calling **dq_trap_cc** again.

Writing Portable Programs Using the UDI

Not all programs making UDI calls are portable across all UDI-supported OSs. Employ these techniques to ensure that the programs you write are as portable as possible:

- Never examine filenames (and pathnames) in your program. The rules for forming pathnames are OS dependent.
- Modify filename strings only by calling the UDI procedure **dq_change_extension**.
- Work only with pathnames supplied by the user, pathnames created by calling **dq_change_extension**, or predefined filenames.
- Always check the condition code to see if a call failed.
- When handling condition codes, create the necessary file connections in the initial part of programs or make a **dq_reserve_io_memory** call before making any other UDI system call.

Call Sequence for File-Handling System Calls

Figure 21-2 shows how file-handling calls are related. A program needing to access a file obtains a token for a connection to the file. It then uses the connection to perform operations. Other programs can simultaneously have connections to the same file. Each program having a connection to a file uses its connection as if it had exclusive access.

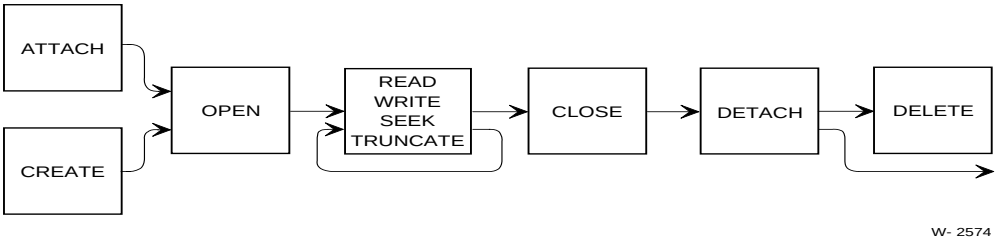


Figure 21-2. Sequence of System Calls for UDI

□□□

Application Loader Programming Concepts

This section describes the Application Loader subsystem. The AL loads programs from secondary storage into memory under the control of iRMX tasks or tasks that are part of application programs. The AL enables:

- Programs to run in systems with insufficient memory to accommodate all programs at one time.
- Seldom used programs to reside on secondary storage rather than in memory.

These are the chapters in this section:

Application Loader Basic Concepts 22

This chapter defines terms used in Application Loader (AL) system calls and the AL concepts described in this section.

These terms are used in the AL concepts and system call descriptions:

- Object code, object module, and object file
- Synchronous and asynchronous system calls
- I/O job
- Overlay, root module, and overlay module
- Device independence
- Configurability

Object Code

Object code may be:

- Output of a translator (for example, PL/M and ASM).
- Output of the BIND command.

An *object file* contains object code. An *object module* is the output of a single compilation, a single assembly, or a single invocation of the BIND command.

Synchronous and Asynchronous System Calls

The AL provides both synchronous and asynchronous system calls. While a *synchronous* system call is running, the calling task cannot run. The calling task resumes running only after the loading operation has either failed or succeeded.

While an *asynchronous* system call is running, the calling task runs concurrently. To explicitly overlap processing with loading operations, use asynchronous system calls.

See also: Asynchronous and synchronous calls, in this manual

Situations Requiring an I/O Job

Some of the system calls provided by the AL use the EIOS. These system calls must be part of an I/O job: if a task is not in an I/O job, it cannot successfully use system calls that require the EIOS. The AL creates the I/O job when one is required.

See also: I/O jobs, in this manual

Overlays

Overlays are logically independent subsections of a program which need not all be present in memory at the same time during program execution. Using overlays can reduce the memory space required for a program to execute, as these designs of a data processor illustrate.

- If the data processor is structured as a monolithic program that resides on secondary storage, the entire collection of object code will be loaded into RAM when needed.
- If the data processor is an overlaid program, pieces (overlays) of the data processor reside on secondary storage; individual overlays are loaded as needed. In this way, the data processor can run in a much smaller area of memory because different overlays are alternately loaded into the same memory space. The data processor might be slower if it uses overlays, depending on how it uses the time when the overlays are being loaded.

To implement an overlaid program using the AL, create a program with a *root module* and one or more *overlay modules*. A root module is an object module that controls the loading of overlays. When you invoke an overlaid program, the application system loads the root module. The root module then loads overlay modules as needed.

Overlays are supported in OMF86 and OMF286 programs; they are not supported in OMF386 programs.

See also: Overlays, root modules, and overlay modules, *Intel386 Family Utilities User's Guide*

Device Independence and the AL

The AL can load object code from any mass storage device supported by the BIOS.

Configuring the AL

For ICU-configurable systems, you can configure the kind of load function required by your system. Your system may be configured for:

- Load job, which includes all the AL system calls.
- Load, which includes only **a_load**.

If you choose all AL system calls, the ICU will incorporate the EIOS into your system.

You can configure the read buffer size to optimize loading time: a smaller buffer size may cause a longer load time.

You can configure the memory pool minimum size used by the AL to create an I/O job for newly loaded programs. If you specify 0 in the pool minimum parameter, the Application Loader computes the required size.

See also: *ICU User's Guide and Quick Reference*

□□□

Preparing Code for Loading 23

To process your code so that the AL can load it:

- Use an Intel386 translator or assembler (PL/M-386, ASM386, or iC-386) to produce object modules that you can bind. COMPACT is the only supported compilation model for Intel386 translators. Then use BND386 to produce a load file. Use the RCONFIGURE control. The load file must be an OMF-286 Single Task Loadable (STL) object file with LODFIX records.

  STL format is the only supported object code format. LODFIX records enable the AL to replace each selector in the object file, with the new GDT selector assigned at random by the iRMX OS, at load time. Use the **debug** command to determine which GDT slots were allocated for your program.

- Use a non-Intel compiler to produce your application. Some of these third-party tools produce flat model (non-segmented)_ applications. The AL recognizes and can load a flat model application.

See also: **debug**, *Command Reference*;
porting code, *Programming Techniques*;
Third-party Compilers, Flat Model, *Programming Techniques*

Specifying Pool Sizes for I/O Jobs

There are two ways to specify memory requirements for the I/O job's memory pool. Both involve setting the BND386 RCONFIGURE control when you create the object file. You can:

- Let the AL decide how large a memory pool to allocate to the new I/O job.
- Manually set the pool size.

The AL determines the size of the I/O job's memory pool using this information:

- The `pool_min` parameter, as a number of 16-byte paragraphs.
- The `pool_max` parameter, as a number of 16-byte paragraphs.
- The DMP ICU configuration parameter specifying the default dynamic memory requirements.
- Memory requirements specified in the target file with the RCONFIGURE parameter.

If the AL allocates the memory pool, it uses the requirements of the target file and the configured DMP parameter to make this decision. Unless you have unusual requirements, choose this option. Make sure the values specified by the RCONFIGURE parameters provide more than enough memory for the program.

If you override the AL's decision on pool size, the AL uses the `pool_min` parameter or `pool_max` parameter specified in the system call to decide how large a memory pool to allocate. If the value you enter in `pool_min` is less than what is required to load the file, the AL ignores your input and sets `pool_min` to the minimum amount of memory required by your file. If you set `pool_max` to `max_pool_size`, the created I/O job can borrow unlimited memory from its parent.

The pool size parameters in AL system calls are specified in 16-byte paragraphs. However, the pool parameters in the RCONFIGURE control of BND386 are entered in BYTES.

See also: Pool sizes for I/O jobs in this manual

Producing an STL Object File

This example illustrates how to produce an STL object file. The directory attached as *:lang:* contains the PL/M runtime libraries. The source code for the program is located in a PL/M file named *my_prog.plm*. This source is common to both the 16- and 32-bit OSs. The program uses COMPACT model. The 16-bit version is linked to the compatibility interface library, *rmxifc.lib*. The 32-bit version is linked to the COMPACT library, *rmxifc32.lib*. Use this sequence to produce an object module from *my_prog.plm*. The SEGSIZE control and DYNAMICMEM option of the RCONFIGURE control are described after the example.

The 16-bit version runs on 16-bit systems.

```
PLM286 MY_PROG.PLM COMPACT
BND286 &
    MY_PROG.OBJ, &
    :LANG:PLM286.LIB, &
    /RMX386/LIB/RMXIFC.LIB  &
    OBJECT(MY_PROG_16) SEGSIZE(STACK(+500H)) &
    RCONFIGURE(DYNAMICMEM(5000H, 10000H))
```

The 32-bit version is:

```
PLM386 MY_PROG.PLM COMPACT WORD16
BND386 &
    MY_PROG.OBJ, &
    :LANG:PLM386.LIB, &
    /RMX386/LIB/RMXIFC32.LIB  &
    OBJECT(MY_PROG_32) SEGSIZE(STACK(+500H)) &
    RCONFIGURE(DYNAMICMEM(5000H, 10000H))
```

Binary compatibility support enables the 16-bit version to run on a 16- or 32-bit system. The WORD16 compiler control tells the compiler that a WORD is a 16-bit quantity in the source; this enables the source modules to be truly common.

Upon completion, the object module *my_prog_16* or *my_prog_32* is ready for loading.

Specifying Stack Requirements with SEGSIZE Control

The SEGSIZE control specifies the stack requirements for your program and the stack requirements for the highest iRMX layer used. Table 23-1 lists the stack requirements for each layer. The value given as the minimum stack size for an individual layer includes the requirements of all lower layers. For example, if you use the Nucleus, BIOS and EIOS, add 550 bytes to the stack. Then add your program's stack requirements.

Table 23-1. OS Stack Sizes

OS Layer	Minimum Stack Size
Nucleus	250 bytes
BIOS	350 bytes
EIOS	550 bytes
Application Loader	700 bytes
Human Interface	1500 bytes
UDI	1750 bytes

When any task is created in the iRMX OS, the Nucleus ensures that it has at least a 1 Kbyte stack unless you have specified a size with the SEGSIZE control. Sixteen bit tasks need appropriate stack padding so they run properly with the iRMX OS. If you use the SEGSIZE control, make sure to specify at least a 1 Kbyte stack.

Specifying Dynamic Memory Allocation with DYNAMICMEM Option

BND386 enables you to specify the amount of memory your program will allocate dynamically, so that your program has enough dynamic memory once it is loaded and running. The value specified by `pool_min` is always available for your program, while the value specified by `pool_max` enables your program to borrow from its parent. `Pool_min` and `pool_max` apply only for programs that are loaded as I/O jobs.

□□□

Application Loader System Calls 24

The AL system calls divide into two categories:

- I/O job and non-I/O job system calls
- Synchronous and asynchronous system calls

AL System Calls Requiring an I/O Job

The AL creates an I/O job and loads a program within it when one of these system calls is issued:

a_load_io_job
rqe_a_load_io_job
s_load_io_job
rqe_s_load_io_job

The AL task which loads the job is a task in the new job. Once the code is loaded, the AL task terminates itself, unless the new program contains overlays. If so, the AL task waits for requests to load new overlays.

Specify the pool size parameters in AL system calls in 16-byte paragraphs; enter the pool parameters in the RCONFIGURE control of BND386 in BYTES.

a_load Does Not Require an I/O Job

A_load is the only system call that does not create an I/O job. Instead, the AL task that loads the program runs in the context of the caller's job.

The AL places the loaded code in memory; it does not create a task for it. If you want this code to run, explicitly create a task for it using the Loader Result Segment (LRS) that the caller receives on completion of loading. Because no I/O job is created, you can use **a_load** in systems configured without the EIOS layer.

The LRS contains two fields, `code_seg_base` and `stack_seg_base`, that list the tokens of the segments (up to 255) created by loading a file. These fields let you call **a_load** while loading OMF286 programs that use MEDIUM and LARGE models. Only COMPACT model OMF386 programs are supported for calls to **a_load**.

Note

The system call a_load is not supported in flat-model applications.

See also: LRS, *System Call Reference*

Synchronous System Calls

The synchronous system calls are:

rqe_s_load_io_job
s_load_io_job
s_overlay

If the system call returns to the calling routine after the service has completely finished, an E_OK condition code returns, using the specified exception pointer. If the system call terminates due to an error, an exception condition code is returned.

Using rqe_s_load_io_job and s_load_io_job

These two system calls load the specified file and create an I/O job as the environment for the loaded code.

Either call can immediately start or delay execution of the loaded code, depending on the `task_flags` parameter. If you specify delayed execution, call **start_io_job** after the AL has successfully returned and you are ready to start the program.

The `resp_mbox` parameter specifies the exit mailbox for the newly created I/O job. The EIOS sends an exit message to this mailbox when the loaded program, contained within the newly created I/O job, terminates using **exit_io_job**.

See also: **create_io_job**, **start_io_job**, and **exit_io_job**, *System Call Reference*

Loading Overlays with s_overlay

To create OMF286 overlaid programs on an Intel system, use OVL286 to produce the object files. The AL assumes that you adhered to these rules when writing the overlaid program.

- The root is always present in memory.
- No overlay, except the root, is present in memory unless its parent is also present.
- The only possible request from any given overlay is to load a descendent overlay.
- Any previously loaded sibling is no longer accessible once an overlay has been loaded.
- No assumptions are made about the preservation of data across multiple requests to load the same overlay.

Use **s_overlay** whenever the loaded program requires that a new overlay be present in memory. This call can be used only by an overlaid program. It can be issued by any overlay (including the root) to load any of its descendants.

Although **s_overlay** is synchronous, it can be used in conjunction with the asynchronous AL system calls. When you invoke an overlaid program, use **a_load_io_job** or **s_load_io_job** to load the root module. The root module then uses **s_overlay** to load overlay modules as needed.

Asynchronous System Calls

The asynchronous system calls are:

rqe_a_load_io_job
a_load_io_job
a_load

The concurrent part of the call runs as an iRMX task. The task is readied by the sequential part of the call and runs only when the priority-based scheduling of the OS gives it control of the processor. The concurrent part also returns a condition code as part of an LRS sent to the response mailbox specified in the asynchronous AL call.

See also: Synchronous and asynchronous calls, in this manual

Asynchronous Call Order of Operations

This example shows how an application can load a program stored on disk. The application issues **a_load** to have the AL load the program into memory.

1 The application issues **a_load** and specifies a response mailbox for communication with the concurrent part of the system call.

2. The sequential part of **a_load** begins to run and checks for valid parameters.

3. The iRMX OS returns a sequential condition code. It then returns control to the application. If the condition code is E_OK, the AL readies the AL task; otherwise, it does not ready the AL task.

4. The application receives control and tests the sequential condition code. If the code is E_OK, the application continues running. At this point, the application can take advantage of the asynchronous and concurrent behavior of the AL to perform computations.

   If the sequential condition code is not E_OK, the AL did not ready a task to perform the function and the application must respond appropriately.

   For the balance of this example, assume that the sequential part of the system call returned an E_OK sequential condition code.

5. Before using the loaded program, the application verifies that the concurrent part of **a_load** ran successfully. The application issues a **receive_message** system call to check the response mailbox specified in **a_load**.

6. After receiving the LRS indicating successful loading, the application uses **rq_create_task** (using the entry point, data segment, and stack segment specified in the LRS) to activate the loaded program.

7. When the loaded program is no longer required, the application can delete all the segments that the AL created for this program by using the segment list in the end of the LRS. The LRS itself can then be deleted.

See also: Asynchronous and synchronous calls in this manual;
Application Loader calls, *System Call Reference*

Response Mailbox Functions

All AL system calls except **overlay** have a response mailbox parameter. The response mailbox has two different functions, depending on the system call used.

When you invoke an asynchronous system call, this mailbox enables the AL to notify the caller that the concurrent part of the system call is finished. The AL sends an LRS to this mailbox on completion of the loading process.

In general, the LRS indicates the result of the loading operation. The format of an LRS depends upon which system call was invoked.

See also: LRS, *System Call Reference*

For **s_load_io_job** and **rqe_s_load_io_job**, this mailbox also receives the exit message from the loaded I/O job. The EIOS sends the exit message when the loaded program terminates using **exit_io_job**.

Therefore, you can wait at the same mailbox two times: first for the LRS and then for the exit message, in this order.

Avoid using the same response mailbox for more than one concurrent invocation of asynchronous system calls because the AL may return LRSs in an order different from the order of invocation. However, it is safe to use the same mailbox for multiple invocations of asynchronous system calls if these conditions are met:

- One task invokes the calls.
- The task always obtains the result of one call via **receive_message** before making the next call.

❑❑❑

HUMAN INTERFACE PROGRAMMING CONCEPTS

This section documents the Human Interface (HI) layer of the iRMX OSs. This section is intended for the programmers who write application programs that can be loaded and executed using keyboard commands. It is also for system administrators who use the HI command lines to configure the system.

This documentation assumes that you are familiar with the C or PL/M programming language.

See also: *iC-386 Compiler User's Guide*;
PL/M-386 Programmer's Guide

These are the chapters in this section.

Human Interface Basic Concepts 25

The HI provides features to aid both console operators and programmers. These features include:

- A set of HI commands, such as general utilities and file, volume, and device management commands.
- An initial program, the Command Line Interpreter (CLI), with its own set of commands.
- A logon facility to validate users and set up their environment.
- Multiuser support.
- A recovery/resident user for ICU-configurable systems that enables access to the system if it does not initialize properly.
- Wildcard pathname support.
- A group of system calls to aid programmers in writing application-specific commands.

Sample Code

Code fragments illustrating HI concepts are included in the *demo* directory. Filenames for the programs are listed in the respective chapters.

Resident HI Commands

You can use resident HI commands with any application system that includes the HI. Here are some of the commands:

- File management commands such as **copy**, **delete**, and **backup**.
- Device and volume management commands such as **attachdevice**, **format**, and **diskverify**.
- General utility commands such as **debug** and **date**.

See also: HI commands, *Command Reference*

CLI: The Initial Program

The initial program is the first program to run when a user logs on. An initial program typically reads commands from the terminal and executes the commands based on that terminal input. The iRMX-supplied initial program is called the HI CLI. The CLI reads input from the terminal, enables the user to edit that input if necessary, and executes commands (either CLI or HI) based on the input. Some CLI commands are **alias**, **history**, and **submit**.

The CLI provides a number of additional features such as aliasing, background processing, and recalling of previously entered command lines.

Loading Other Initial Programs

The initial program does not have to be the HI CLI; it can be almost anything from an editor, to a BASIC interpreter, to a loadable command interface that you write. The system manager determines which initial program runs when a user logs on when he adds new users to the system. There can be a separate initial program for each user.

⚠ **CAUTION**

Unloading jobs that contain interrupt handlers using **sysload -u** or <Ctrl-C> will cause unpredictable results.

See also: **path** command, *Command Reference*

Logon

Logon validates terminal users and sets up their environment.

Validation

On some terminals, typically those used by a single user, the logon and validation process is invisible. On other terminals, typically those used by several users, logon and validation requires entering a name and password. The kinds of terminals are:

- Static terminals
- Dynamic terminals

Static terminals are configured to service a specific user. The static terminal's attributes are usually taken from the user configuration files during logon. The logon process is automatic and invisible to the user. When the HI starts running, it has information about the user such as user ID, the amount of memory available to this user, and the user's priority. The only way to change the HI's assumptions about static terminals is to change the OS's user configuration files and restart the OS.

See also: Configuration files, *System Configuration and Administration*

Dynamic terminals are configured to service many different users on a request-by-request basis. The HI requests a logon name and a password before allowing the user to access the system. The HI verifies that the information entered is valid by checking user configuration files set up by the system manager. Then it sets up the terminal based on the information listed in those files.

Unlike static terminals, dynamic terminals have dynamic memory partitions. That is, the HI does not assign any memory to the terminal at system startup. Instead, it assigns the memory when a user logs on. When the user logs off a dynamic terminal, the memory goes back into the general free space memory pool. If there is no free memory left in the system, a user will be notified of this condition and will not be able to log on.

The amount of memory assigned varies depending on the user's requirements, as listed in the user configuration files. The advantage of dynamic terminals is that the memory available to users varies depending on the needs of the user.

See also: Dynamic terminals, Static terminals, *System Configuration and Administration*

Environments

The HI creates a job for each user that logs on. This job furnishes the application environment by assigning:

- Memory for the user to use for running commands.
- The initial program for the user.

Any commands that the user invokes use the assigned area of memory. If there is not enough memory in the system to initialize a user, the system assigns whatever memory is available at the time and issues a warning message to the terminal.

Users can use CLI commands (**alias**, **background**, etc.) which are executed in the interactive job or HI commands (**copy**, **format**, etc.) which run as child jobs of the user's interactive job.

This table shows the process of entering CLI and HI commands. Either of these commands can be entered with optional parameters.

CLI	**HI**
Invoke by command name	Invoke by pathname/command name
Interpret command	CLI loads command into main memory from secondary storage
Execute command	Create a child job of the interactive job for the command Execute command

Some commands are available from both the HI and the CLI. In this case, CLI commands are executed before HI commands. For example, if you enter **submit**, the CLI version of the **submit** command is executed, not the HI version.

Network Access

If the system is set up as a workstation on an iRMX-NET communications network, any user who logs onto the system on a dynamic terminal automatically becomes a verified user of the network and can access remote files using the iRMX-NET network.

See also: iRMX-NET environment, *Network User's Guide and Reference*

If the system has NFS enabled and has files or directories defined as NFS-shared, users can access these files and directories using Human Interface commands as if the files and directories were local.

See also: Using NFS, *TCP/IP and NFS for the iRMX Operating System*

Logging Off

When users of dynamic terminals finish accessing the OS, they should use the **logoff** command to terminate their sessions. Other users can then log onto the same terminals.

Multiuser Support

Multiuser support enables multiple users to communicate with the OS. The BIOS supports multiple terminals by providing device drivers that communicate with multiple-terminal hardware. The HI supports multiple users by providing identification and protection of users based on logon names and user IDs. The multiuser HI also enables a programmer in the development environment to execute commands, run development programs (editors, compilers, etc.), and run other application programs.

The system manager must first set up the proper directory structure and provide several files containing information about the users that can access the system. However, you can still tailor your system to meet your individual needs by selecting, for each user, the initial program that runs when that user accesses the HI. The user configuration files maintained by the system manager identify this choice to the HI.

Figure 25-1 on page 311 shows how the HI handles multiple users.

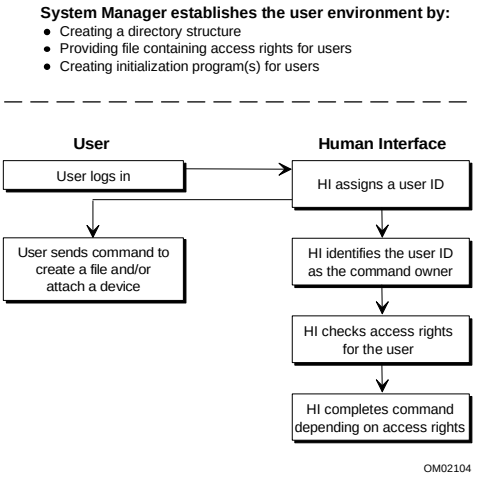


Figure 25-1. Multiuser Support under the HI

Recovery/Resident User

The recovery/resident user is available in ICU-configurable systems only. The recovery/resident user only gains control if an initialization error occurs in the configuration files and the system cannot initialize. The recovery/resident user (and the associated terminal) is defined during ICU configuration.

User attributes are defined in HI memory during the configuration process and are loaded with the system. A resident user does not use any of the system configuration files and is not presented with a logon prompt. Because this user is only active if an initialization error occurs, the user is typically configured as the system manager (user ID 0).

Wildcards

The HI supports using wildcard characters in filenames. This provides a shorthand method of specifying several files in a single reference. The wildcard characters supported by the HI are:

? Matches any single character

* Matches any sequence of characters (including no characters)

See also: Wildcard characters, *Command Reference*

Programmers who write their own HI commands do not have to provide special code to support wildcard pathnames as long as they use the HI system calls **c_get_input_pathname** and **c_get_output_pathname** to obtain the file names from the command line.

See also: Writing and Parsing Commands, in this manual

Human Interface System Calls

The HI provides a set of system calls that you can use in writing custom commands for applications. These categories of HI system calls are available:

- Command parsing system calls

  These calls provide the ability to parse the command line, enabling you to isolate and identify the parameters in a command line. They also enable you to determine the command name and parse other buffers of text.

  See also: Writing and Parsing Commands, in this manual

- I/O and message processing system calls

  These calls enable you to establish connections to input and output files, communicate with the terminal, and format condition codes into a ready-to-display form.

  See also: Communicating with the Operator, in this manual

- Command processing system calls

  These calls enable you to invoke interactive HI commands programmatically.

  See also: Invoking HI Commands Programmatically, in this manual

- Program control system call

  This call enables you to override the default <Ctrl-C> handler task provided by the HI.

  See also: Writing a <Ctrl-C> Handler, in this manual

Human Interface Operations

When the HI begins running, it:

1. Initiates a logon process that validates users.
2. Displays an initialization error on the terminal if an initialization error occurs.
3. Creates an iRMX job for each user logged into the HI.
   a. Assigns an area of memory for the user to use for running commands.
   b. Starts an initial program which is the user's interface to the OS.

□□□

The Command Line Interpreter 26

The HI Command Line Interpreter (CLI) is invoked by the HI when the user logs on. The CLI provides the user with:

- Line-editing
- Alias facilities
- Background processing
- Session history
- Terminal definition
- Execution of its own set of commands

For ICU-configurable systems, the HI can also operate with a user extension, which enables you to add customized features to the CLI.

You can also write a loadable command interface to use as an initial program instead of the CLI. This chapter lists the rules for writing a loadable command interface.

See also: **password** command, *Command Reference*;
User definition files, Terminal configuration files, *System Configuration and Administration*

CLI Features

The CLI provides a number of features that make it a useful tool in a development environment:

Line-editing	enables the user to re-edit input.
Aliasing	enables the user to abbreviate commonly used commands and assign parameters to them. CLI commands: **alias**, **dealias**, **logoff**
Background processing	enables the user to run jobs in a background environment while continuing to invoke commands at the terminal. The user is notified when a background job starts and finishes. It is possible to request a list of the active background jobs or cancel a background job. CLI commands: **background**, **jobs**, **kill**, **logoff**
Session history	Displays the last 40 commands and enables the user to select lines for re-editing. CLI command: **history**
I/O redirection	enables standard input and output to be directed somewhere other than the user's terminal. CLI command: **submit**
CLI environment	enables the user to perform online changes to certain CLI attributes, such as the prompt and the background memory pool size. CLI commands: **set**, **super**

Note

CLI commands such as **alias**, **submit**, and **super** do not recognize continuation characters.

If the CLI satisfies the needs of your application, you can assign it to each user as an initial program.

See also: CLI commands, *Command Reference*

Initializing the CLI

The CLI can be invoked during either static or dynamic logon. During initialization, the CLI performs these operations:

- Initializes the CLI environment
- Calls CLI extensions, if necessary
- Displays a sign-on message
- Creates a command connection object where it places information received from the terminal

  See also: Invoking HI Commands Programmatically, in this manual
- Attaches the user's directory
- Submits the file for processing (if it exists)

After this initial processing, the CLI displays the - (HI default) prompt and reads input from the terminal. Input from the terminal can be a CLI command, an HI command, or a user application program that is to be executed.

Invoking and Executing Commands

The CLI begins executing a command after a user enters a <CR> or an <Esc>. However, before execution, the CLI enables the user to edit the input line or recall previously entered lines. When input stops, the CLI performs these operations:

1. Reads the command line from the terminal into a CLI buffer.
2. Determines if the command is a CLI or an HI command.
3. Expands all aliases.
4. Handles any I/O redirection that may be necessary.
5. Passes control to the user extension procedure `CLI$user$process`, if applicable.

   See also: User Extensions, in this manual
6. Searches for CLI or HI commands.

   If the CLI encounters a CLI command, it executes the action requested.

   If the CLI encounters an HI command or any user application program, it:

   a. Loads the file containing the command

   b. Passes the parameters to the command

For long commands, it may be necessary to continue an HI command. The CLI recognizes the & (ampersand) mark at the end of a command line as a continuation character, and displays a ** (double asterisk) on the continuation line.

The user can recall either the complete continuation line or only part of it. A double asterisk on the screen indicates that a continuation line is being recalled. The user can then edit the relevant section of the line. However, after the section has been edited, the entire command line is executed if the user presses <Esc> to terminate input.

The CLI displays error messages for each command if the user does not invoke the command properly or if the CLI cannot execute the command as requested.

See also: Continuation character, and specific CLI command error messages, *Command Reference*

Adding User Extensions to the CLI

Only ICU-configurable versions of the OS enable the CLI to be extended to include customized functions.

With this feature, you can create an initial program that takes advantage of the CLI features, such as line-editing and aliasing, and still meets your precise needs. This section explains how to extend the CLI to include user extensions that parse commands differently or implement your own commands using CLI user extensions.

Creating User Extensions

The CLI is a 16-bit application that uses 16-bit user extensions. Creating an extension involves writing three procedures:

- An initialization procedure
- A processing procedure
- An epilog procedure

You can combine these procedures, described in the next sections, into one module. An empty default PL/M module called (located in) provides you with null instances of the three procedures. The CLI has three entry points to the user extensions, one before each procedure. You can make a local copy of the example module to develop your CLI extension.

Initialization Procedure

When the CLI is initialized, it first defines its own alias tables (the memory area where user-defined aliases are stored) and data structures. It then calls your user-supplied initialization procedure one time only. If you have tables or data structures to add during initialization, they should be part of the initialization procedure. The CLI enters the user extension by calling:

```
CALL CLI$USER$INIT(except_ptr);
```

You can bind this procedure to the CLI library supplied with the HI. Examples of how to do this are given later in this section.

Processing Procedure

After each command line (entered either from a terminal or in a submit file), the CLI translates all aliases, and checks again for user extensions. At this point, you can change a command, perform additional functions before execution, or process the command. To access your user extension, the CLI calls:

```
cont_flag = CLI$USER$PROCESS(command_ptr, except_ptr);
```

Where:

`command_ptr`	A pointer to a STRING containing the expanded command ready for execution.
`cont_flag`	A BYTE indicating whether the CLI should continue executing the command line modified by the user extension, or ignore it and continue to the user extension epilog procedure.

Epilog Procedure

After the CLI executes an HI, CLI, or user-supplied command, it calls the epilog procedure. This procedure handles error conditions or performs any other functions that cannot be performed until the command has been executed. The epilog procedure is called by:

```
CALL CLI$USER$EPILOG(except_ptr);
```

Bind this procedure to the CLI library as shown in the example given later in this section.

Error Handling

Each of the three user extension procedures returns a condition code in the exception pointer, `except_ptr`. If the procedure returns anything other than E_OK, the CLI outputs an error message in addition to the message issued by **c_send_command** or the CLI command.

The CLI catalogs the condition code generated by the last command under the name *r?error* in the global directory before executing the epilog procedure. You can access this value and use it in your application. However, any changes to *r?error* are not recognized by the CLI.

This PL/M code enables you to access the value in *r?error*.

```
DECLARE error_t   TOKEN,
        error     BASED error_t WORD,
        except    WORD;
error_t = RQ$LOOKUP$OBJECT (SELECTOR$OF(NIL),
                            @(7,'R?ERROR'),0,@except);
```

After execution of this system call, the `error` field will contain the condition code that the last command sent to *r?error*.

This C code also enables you to access the value in *r?error*.

```
main ()
{
selector            error_t;
unsigned short      excep;

       error_t = rq_lookup_object ((selector)
                           NULL, "\07R?ERROR", 0, &excep);
       print_error (5);
}
```

Demonstration Program - User Extension

A PL/M example, which is installed with the iRMX OS, shows how to create a user extension using the CLI initialization, process, and epilog procedures described above. This user extension enables you to measure the time required to execute a CLI command, an HI command, or any application program. The PL/M code shown is a straightforward example. Many special cases have been omitted.

Binding a User Extension

The CLI is a 16-bit PL/M application. Use BND286 to bind the user extension to the CLI library. This section provides an example of the bind process.

You can combine the three user extension procedures into one module, but this is not necessary.

Binding your extensions as shown below creates a CLI with your user extension. You can add this newly created CLI to the application boot file using the ICU. Then this new CLI will be called by its pathname, *mycli*, as a nonresident CLI during the logon process.

If you want the default resident CLI to include user extensions, specify the pathname of the user extension module during configuration.

See also: For ICU-configurable systems, *ICU User's Guide and Quick Reference*

If you have named your user extension module *myext.p28*, you can use this example exactly as it is written. Otherwise, replace *myext.obj* with the name of the object module you wish to bind.

```
:LANG:BND286            &
MYEXT.OBJ,              &
:RMX:HI/HCLI.LIB(HCLI), &
:RMX:HI/HCLI.LIB,       &
:RMX:HI/HI.LIB,         &
:RMX:LIB/RMXIFC.LIB,    &
:RMX:HI/HUTIL.LIB,      &
:LANG:PLM286.LIB        &
RENAMESEG(CODE TO CLI_CODE,DATA TO HI_DATA, &
HI_CODE TO CLI_CODE,HI_DATA TO CLI_DATA)    &
OBJECT(MYCLI) NOLOAD NODEBUG SEGSIZE(STACK(2400H)) &
RC(DM(1000H,0FFFFH))
```

Where:

`MYCLI` is the name you use to invoke this CLI.

Creating a Loadable Command Interface

If the CLI, with or without a user extension, does not meet your needs, you can provide your own loadable command interface. Your loadable command interface may be a completely different kind of program from the CLI. For example, you could write a loadable command interface that enables access to files in selected directories only. This would prevent a user from accidentally modifying other files.

Use the selections of static or dynamic terminal type, password or no password required, and a loadable command interface, to create the user environment needed for your application. For example, you can define a static terminal using the file. Then, use the **password** command to assign your application program as the initial program. By deleting all other users except Super (again, using the **password** command), you would have created a system running only your application (with or without a password requirement, depending on your needs).

If you provide your own loadable command interface, the program must obey these rules:

- It must select the initial program for each user, and specify the selection in the user configuration files maintained by the system manager.
- It must initialize its own data segment. The HI does not set the DS register for the CLI.
- It must perform input and output using logical names *:ci:* (console input) and *:co:* (console output).
- If it requires the ability to run HI commands, it must create a command connection object using the **c_create_command_connection** system call.

  If the loadable command interface does not create a command connection, it (and any other application tasks) cannot use these HI system calls:

  c_get_input_pathname
  c_get_output_pathname
  c_get_input_connection
  c_get_output_connection
  c_send_co_response
  c_send_eo_response
  c_send_command
  c_set_control_c
  c_delete_command_connection

- If it does not create a command connection, it must first invoke the **c_set_parse_buffer** system call before using the HI system calls **c_get_parameter**, **c_get_char**, and **c_backup_char**.
- It must invoke the EIOS call **exit_io_job** to terminate processing. It must not use the PL/M or ASM `RETURN` statement for this purpose.

See also: HI system calls and **exit_io_job**, *System Call Reference*; **path** command, *Command Reference*

Alternatively, if you want a particular user to use only BASIC-language programs, a BASIC interpreter might be the initial program for that user.

CAUTION

Unloading jobs that contain interrupt handlers using **sysload -u** or <Ctrl-C> will cause unpredictable results.

□□□

Writing and Parsing Commands 27

This chapter deals only with HI command parsing. HI commands are handled differently than CLI commands.

When the user invokes a command, the OS places the command's parameters into a parsing buffer. One of the first things that the invoked command must do is read the parsing buffer, identify the individual parameters, and determine the correct action to take, based on the number and meaning of the parameters.

See also: The Command Line Interpreter, in this manual;
CLI commands, *Command Reference*

The HI provides several system calls to parse command lines that follow a standard structure. It also provides other system calls to process nonstandard formats. This chapter:

- Defines the standard structure of command lines.
- Describes the system calls used to parse standard commands.
- Discusses how to switch from one parsing buffer to another parsing buffer.
- Discusses wildcards used in input and output pathnames.
- Describes system calls used to parse nonstandard commands.
- Describes the **c_get_command_name** system call used to obtain the command name the user used when invoking a command.

Standard Command-line Structure

The standard structure of an HI command line consists of elements separated by spaces. Your commands should follow this structure to enable parsing by the HI system calls.

See also: For different command structures,
Parsing Nonstandard Command Lines in this chapter

Command-line Structure Parameters

In this example, square brackets [ ] indicate optional portions of the standard structure.

```
command [inpath-list [preposition outpath-list]] [params] <CR>
```

Where:

`command` Pathname of the file containing the command's executable object code. The pathname may include a prefix and a subpath. A prefix is a logical name of a directory and is unique if it is not duplicated in one of the directories in the command search sequence defined during configuration.

See also: Pathnames, logical names, *Command Reference*

`inpath-list`
One or more pathnames of files, separated by commas, that the HI reads as input during command execution. Individual pathnames can contain wildcard characters to signify multiple files. Use the **c_get_input_pathname** system call to process this inpath-list.

See also: Wildcard characters, *Command Reference*

`preposition`

Tells the HI how to handle the output. The standard structure supports these prepositions. Use the **c_get_output_pathname** system call to process the preposition.

to The HI writes the output to a new file indicated by the output pathname. If the file already exists, the HI asks if you want to overwrite the file.

Answering with a Y (uppercase or lowercase) causes the HI to overwrite the existing file with the new output. An R tells the HI to continue overwriting existing files without prompting for permission. An R causes the HI to proceed with the next pair of input and output files.

over The HI writes the output to the file indicated by the output pathname. It overwrites any information that currently exists in the file.

after The HI appends the output to the end of the file indicated by the output pathname.

`outpath-list`

One or more pathnames of files, separated by commas, that receive the output. The total number of pathnames in this list and the number of wildcards used depends on the inpath-list. Use the **c_get_output_pathname** system call to process the outpath-list.

See also: Pathnames, *Command Reference*

`params` Parameters that cause the command to perform additional or extended services during command execution.

See also: Command-line Structure Parameter Formats

`<CR>` and `<LF>`

Line terminator characters. The <CR> and the <LF> are both line terminators.

These examples show how to enter an HI command using the command structure described above.

```
- copy :home:file1 to /tmp/file2 <CR>
- format :f: files=300 interleave=1 bs <CR>
```

See also: HI commands, *Command Reference*

Command-line Structure Parameter Formats

The standard structure supports parameters with these formats:

`value-list`

One or more groups of characters (called values) separated by commas. When `value-list` is present, the command performs the service indicated by the values.

See also: **permit** command, access value, *Command Reference*

`keyword` Predefined keyword functions without added user values.

See also: **format** command, `force` parameter, *Command Reference*

`keyword = value-list` or `keyword (value-list)` A keyword with an associated value or value-list. The keyword portion identifies the kind of service to perform, and each value supplies further information about the service request.

See also: **format** command, FILES=num, **diskverify**, **kill** commands, *Command Reference*

`keyword value-list`

A keyword with an associated value or value-list. The keyword portion identifies the kind of service to perform and each value portion provides more information about the service. However, the keyword must be identified to the command as a preposition. Use the **c_get_parameter** system call to process the parameter.

See also: HI call **c_get_parameter**, *System Call Reference*

See also: Parsing Other Parameters in this manual

Command-line Structure Special Characters

The HI supports these special characters:

& (continuation character)	Continuation characters are recognized by all HI commands found in. When using an & (ampersand) in the command line as the last character before the line terminator, the HI assumes that the command continues on the next line. If the CLI (or any loadable command interface that uses **c_send_command** to invoke commands) processes the user's command entry, the & and the line terminator that follows are edited out of the parsing buffer. Then the continuation line is read and appended to the parsing buffer. This process continues until the user enters a line terminated by a <CR> without a continuation character. Therefore, when the command receives control, its parsing buffer contains a single command invocation, without intermediate continuation characters or line terminators.
; (comment character)	The HI considers this character and all text that follows it on a line to be a non-executable comment. If the CLI (or any loadable command interface that uses **c_send_command** to invoke commands) processes the user's command entry, all comments are edited out of the parsing buffer. Therefore, individual commands do not have to search for and discard comments.

" and " or ' and ' (quoting characters)	Two ' (single-quote) or " (double-quote) characters remove the semantics of special characters they surround. Use the same character for both the beginning and ending quote.
	
	If a command line contains quoted characters, HI system calls that invoke the command and parse the command line do not perform any special functions associated with the surrounded characters. For example, the "&" (ampersand surrounded by double quotes) is interpreted as a single ampersand and not a continuation character.
	
	The quotes do not remove the semantics of characters that are special to other layers of the OS, such as :, /, and ^, which are special to the I/O System.
	
	To include the quoting character in the quoted string, the user must specify the quoting character twice or use the other quoting character. For example:
	`'can''t'`
	is read in the command line as
	`can't`

Parsing the Command Line

The HI maintains a pointer for a parsing buffer, which initially points to the first parameter used when invoking a command. Table 27-1 lists system calls used in parsing command lines and their functions.

Table 27-1. Parsing System Calls

Call Name	Function
c_get_input_pathname	gets input pathname
c_get_output_pathname	gets output pathname
c_get_parameter	parses command line by parameter
c_backup_char	traverses backward by character in a command line
c_get_char	traverses forward by character in a command line
c_set_parse_buffer	changes parsing buffer from the HI to the one in the command line
c_get_command_name	obtains command pathname

Use any of the HI system calls in Table 27-2 to read the parameters from the parsing buffer.

Table 27-2. Parsing Buffer System Calls

Call Name	Reads	Understands Quotes	Moves Pointer
c_get_input_pathname	parameter	yes	to next parameter
c_get_output_pathname	parameter	yes	to next parameter
c_get_parameter	parameter	yes	to next parameter
c_backup_char	character	no	back one character
c_get_char	character	no	to next character

Note: System calls **c_get_input_pathname**, **c_get_output_pathname**, and **c_get_parameter** remove the special meaning from quoted characters and discard the quote characters.

CAUTION

Because **c_backup_char** and **c_get_char** move the pointer character by character, not parameter by parameter, ensure that they leave the pointer pointing at the beginning of a parameter (or at blank characters which immediately precede the parameter) before invoking any of the other system calls.

Parsing Input and Output Pathnames

Use the system calls **c_get_input_pathname** and **c_get_output_pathname** to identify the input and output pathnames in the command line. For command lines that contain multiple pathnames, invoke these system calls several times to obtain all the pathnames. These calls return the pathnames in the form of iRMX STRINGs. If **c_get_input_pathname** returns a 0-length string (that is, the first byte is 0), there are no more pathnames to obtain.

The first call to **c_get_input_pathname**:

1. Reads the entire inpath-list (the list of pathnames separated by commas) into a buffer.
2. Moves the parsing pointer to the next parameter.
3. Returns the first input pathname to the command.

The first call to **c_get_output_pathname**:

1. Identifies the preposition (`to`, `over`, or `after`).
2. Reads the entire outpath-list into a buffer.
3. Moves the parsing pointer to the parameter after the outpath-list.
4. Returns the first output pathname to the command.

Succeeding **c_get_input_pathname** and **c_get_output_pathname** calls return additional pathnames from the buffers created previously, but they do not move the parsing pointer to the next parameter.

This example illustrates parsing the buffer. The parsing buffer contains:

```
A,B to C,D
```

The call sequence to this buffer and the associated results are listed below:

Call Sequence	Result
c_get_input_pathname	Obtains input pathnames (A and B) Returns A to the caller Positions the pointer at the preposition "to"
c_get_output_pathname	Obtains output pathnames (C and D) Returns C to the caller
c_get_input_pathname	Returns B to the caller
c_get_output_pathname	Returns D to the caller

Note

Use the system calls **c_get_input_connection** and **c_get_output_connection** to obtain input and output file connections so the necessary I/O operations can be performed.

See also: **c_get_input_connection** and **c_get_output_connection** system calls, Communicating with the user, in this manual

File Connection Demo Programs

There are two demo programs (one written in C, the other in PL/M) installed with the OS that use **c_get_input_pathname** and **c_get_output_pathname** in their command-line parsing; they also use **c_get_input_connection** and **c_get_output_connection** to obtain connections to the files. These programs are a partial example of a **copy** command that you could implement.

Wildcard Characters In Input/Output Pathnames

The **c_get_input_pathname** and **c_get_output_pathname** system calls automatically handle pathnames that contain wildcard characters. They treat a wildcarded pathname as a list of pathnames.

See also: Wildcard characters, *Command Reference*

C_get_input_pathname matches wildcards. When called, it compares the wildcarded component with the files in the specified directory and returns the pathname of a file that matches.

C_get_output_pathname generates wildcards. Each time you call it, it compares the wildcarded output pathname with the wildcarded input pathname and with the most recent pathname returned by **c_get_input_pathname**. Then it generates a corresponding output pathname based on that information. The output pathname could refer to an existing file or to a file that does not yet exist. A query is issued when an existing file will be overwritten.

When both **c_get_input_pathname** and **c_get_output_pathname** use wildcard characters, obey these rules:

1. Call **c_get_input_pathname** first to obtain the input pathname and then call **c_get_output_pathname** so there is a corresponding output pathname. The identity of the output pathname depends on the identity of the input pathname.
2. Always alternate multiple calls to **c_get_input_pathname** and **c_get_output_pathname**. This is necessary to handle wildcard characters and lists of pathnames.

If you invoke two calls to **c_get_input_pathname** without an intermediate call to **c_get_output_pathname**, you will not be able to obtain the first output pathname.

If you invoke two calls to **c_get_output_pathname** without an intermediate call to **c_get_input_pathname**, the second call returns invalid information.

Parsing Other Parameters

You can also use the **c_get_parameter** system for parsing standard command lines in these instances:

- To parse parameters which appear after the input and output pathnames.
- To parse all parameters, if the command does not use input and output files.
- To parse the input and output pathnames, if the command requires a preposition other than `to`, `over`, or `after`.

Note

If you use **c_get_parameter** to parse input and output pathnames, you must provide additional code to handle wildcard characters that may appear in the command line. This call does not wildcard characters automatically.

For example, a command line contains the pathname *file**. If you use **c_get_parameter** to parse this parameter, the system call returns the value literally as `file*`.

It does not know that the characters represent a pathname, nor does it know that the asterisk represents a wildcard.

When called, **c_get_parameter** parses a single parameter and moves the pointer of the parsing buffer to the next parameter. The parameter returned as a result of this call is one of these:

`value-list` One or more groups of values separated by commas. The system call returns the entire list in the form of a string table. It places each of the values in the value list in a separate string.

See also: String table and string, *System Call Reference*

Individual parameters are separated by spaces.

C_get_parameter returns each listed value as a string in a string table. However, an individual value can itself consist of a value-list. If a group of values separated by commas is enclosed in parentheses, **c_get_parameter** treats the values as a single value, returning them in a single string. For example, consider this value-list:

```
A,(B,C,D),E
```

C_get_parameter recognizes three values: `A,` the group `B,C,D,` and `E`.

See also: Command-line Structure Parameter Formats in this manual

There are two demo programs (one written in C, the other in PL/M) installed with the OS that use **c_get_parameter** in their command-line parsing.

See also: Examples in */rmx386/demo/c/hi* directory

Parsing Nonstandard Command Lines

The next sections discuss two kinds of nonstandard command lines: one that is similar to the standard and one that is completely different.

Variations on the Standard Command Line

If you want to structure your commands so that other parameters appear before the input and output pathnames, you can still use **c_get_input_pathname** and **c_get_output_pathname** to parse the input and output pathnames. However, ensure that your command knows which of the parameters contain the input and output pathnames. Two ways to do this are:

- Enforce a rigid structure on the command line. For example, suppose you want two parameters to appear before the input and output pathnames, such as:

  ```
  command  p1  p2  input-pathname  prep  output-pathname
  ```

These commands can parse the command line:

Command	**Parameter**
c_get_parameter	p1, p2
c_get_input_pathname	input-pathname
c_get_output_pathname	output-pathname

If you do this, `p1` and `p2` are position-dependent parameters which must be included whenever the command is invoked.

- Use a separate parameter as a switch to inform the command that the parameters that follow are input and output pathnames. This method requires more code to implement but it can enable you to make all your parameters (including the input and output pathnames) position-independent.

  This command line example shows how users can specify what they want to retrieve before they specify where to get the information. The example uses a hypothetical command called **retrieve** (which retrieves information from various data bases) and a parameter called `FROM`.

  ```
  retrieve names addresses phones from file1 to file2
  ```

  The parameter `FROM` signals that the next parameters are input and output pathnames. An example of how to process this command line follows:

  ```
  while not end-of-command line
        call c_get_parameter
        if parameter = FROM then
              call c_get_input_pathname
              call c_get_output_pathname
  end
  ```

Other Nonstandard Command Lines

In some instances, you might want your command line to look completely different from that described earlier in this chapter. For example, suppose you require a syntax in which these rules apply:

- Spaces have no significance and can be omitted between parameters.
- A prefix character must be before each parameter ($ indicates an input file, @ indicates an output file, and - indicates all other parameters).

With this kind of syntax, a user could invoke a command (in this example, **refine**) as follows:

```
refine $infile-medium@outfile <CR>
```

Where:

`infile` The file from which to read information.

`outfile` The file in which **refine** should place its output.

`medium` A parameter that further directs the processing.

If you require any nonstandard syntax, you must use the **c_backup_char** and the **c_get_char** system calls to parse the command line. Using calls requires you to provide the parsing algorithm in your own program, because they make no assumptions about the structure or order of parameters. However, by using these system calls, you can enforce any command syntax you choose.

Note

You cannot use **c_get_input_pathname**, **c_get_output_pathname**, and **c_get_parameter** to parse the individual parameters. Any of these system calls would return the entire parameter list as a single parameter.

Switching To Another Parsing Buffer

Some commands might require the ability to parse additional lines of text after the original command invocation, for example, an editor needs to parse individual editor commands. A command such as this cannot use the HI-provided parsing buffer because it has no way of placing information in the buffer, and because it cannot reset the parsing pointer to the beginning of the buffer.

Using the system call **c_set_parse_buffer** changes the parsing buffer from the one the HI provides to one that the command provides. This call also sets the parsing pointer to the beginning of the buffer.

Resetting the parsing pointer to the beginning of the buffer enables you to use one buffer for parsing many lines of text. For example, suppose your command has several sub-commands. Each time the user enters a sub-command, your command reads the sub-command into a buffer, calls **c_set_parse_buffer** to reset the parsing pointer, and parses the sub-command.

The `buff_p` parameter (in the **c_set_parse_buffer** system call) is a pointer to a buffer containing the text to be parsed. This buffer can contain text read from the terminal, text read from a file, or even text that you hard code into the command. After the call to **c_set_parse_buffer**, these command parsing system calls obtain information from the new parsing buffer:

c_get_parameter
c_get_char
c_backup_char

The other command parsing calls (**c_get_input_pathname** and **c_get_output_pathname**) are not affected by calls to **c_set_parse_buffer**. These calls always obtain pathnames from the command line parsing buffer.

The program flow for an operation like this could be:

1. Read the information from the terminal into a buffer (use **c_send_co_response**, **c_send_eo_response**, or an EIOS call).
2. Call **c_set_parse_buffer** to set the parsing buffer to the buffer containing the sub-command. This sets the parsing pointer to the beginning of the buffer.
3. Parse the sub-command using **c_get_parameter**, **c_backup_char** or **c_get_char** system calls.
4. Perform the operations requested by the sub-command.
5. Go back to step 1. Continue this loop until the user exits from the command.

Note

If you specify null or a 0 value for the `buff_p` parameter, the parsing buffer switches back to the original command line buffer which remains pointing at the next parameter in the command line. This enables you to parse part of the command line, switch buffers and parse a portion of another buffer, and switch back to the command line.

Every time you call **c_set_parse_buffer**, the parsing pointer moves to the start of the new buffer. However, **c_set_parse_buffer** returns, in its `offset` parameter, the previous position of the pointer in the new buffer. If you switch back to that buffer by again calling **c_set_parse_buffer**, you can use this value to move the pointer to its previous position in two ways:

- Use the **c_get_char** system call to move the parsing pointer back to its previous position in the new buffer. Call **c_get_char** the number of times specified in the `offset` parameter of the first **c_set_parse_buffer** call. This positions the pointer to its previous location. You can then continue parsing parameters from the point at which you left off.
- Treat your parsing buffer as an array of characters (called `CHAR`, for example). When you call **c_set_parse_buffer** the first time, specify the `buff_p` parameter to point to the first element of the array. Then, when you switch parsing buffers, **c_set_parse_buffer** returns, in the `offset` parameter, the number of bytes already parsed. When you switch back to the new parsing buffer, you can use this offset value as an index into the array.

Obtaining the Command Name

The HI places the invoked command name in a buffer. The **c_get_command_name** obtains the command's pathname.

C_get_command_name does not operate on the parsing buffer, nor is it affected by the **c_set_parse_buffer** system call. It can be called multiple times; each time it returns the same command name.

If the user enters the complete pathname of the command (including the logical name), the command-name buffer contains exactly what the user entered. However, if the user enters a command name without a logical name, the HI automatically searches a number of directories for the command. In this case, the command-name buffer contains not only the name the user entered, but also the directory containing the command (such as *:system:*, *:prog:*, or *:$:*).

Therefore, a command can use the value returned by **c_get_command_name** and the circumflex (^) pathname separator to access the directory in which it resides. For example, if `command-name` is the name received from **c_get_command_name**, a command could access its directory by using the pathname:

```
command-name^
```

It could access another file in the directory by specifying the pathname:

```
command-name^file
```

❑❑❑

Communicating with the User 28

This chapter discusses the HI system calls that:

- Establish connections to input and output files.
- Communicate with the user's terminal.
- Format condition codes into messages that can be sent to the user.

Establishing Input and Output Connections

The HI provides two system calls for establishing connections to input and output files: **c_get_input_connection** and **c_get_output_connection**. These system calls are structured so that you can use the output from other system calls as input to these system calls.

Using c_get_input_connection

Use the **c_get_input_connection** system call for establishing file connections:

1. Get the pathname for the file which will be connected (either through the **c_get_input_pathname** function or by directly specifying the pathname).
2. Use the pathname as one of the parameters for the **c_get_input_connection** system call.
3. Call **c_get_input_connection** to establish the connection to the file.

If **c_get_input_connection** cannot obtain a connection to the specified file, it returns a condition code and writes an error message to *:co:* (normally, the user's terminal). For example, if the specified input file does not exist, **c_get_input_connection** displays this message:

```
<pathname>, file not found
```

See also: **c_get_input_connection** HI system call, *System Call Reference*

Because **c_get_input_connection** returns messages to the user in the event of an exceptional condition, your command does not have to return additional messages unless you require them. The command must decide only whether to abort or to continue processing.

Using c_get_output_connection

Use the **c_get_output_connection** system call for establishing file connections:

1. Get the pathname for the file which will be connected (either through the **c_get_output_pathname** function or by directly specifying the pathname).
2. Use the pathname as one of the parameters for the **c_get_output_connection** system call.
3. Call **c_get_output_connection** to establish the connection to the file.

A second parameter in **c_get_output_connection** specifies the preposition used when writing to the output file (`to`, `over`, or `after`). This preposition governs how the output file is processed.

`to` **c_get_output_connection** prompts the user for permission to delete the existing file. This prompt appears as:

```
<pathname>, already exists, OVERWRITE?
```

A user's `Y` or `y` response (yes), causes the system call to obtain the connection to the existing file.

A `R` or `r` response (repeat), causes the establishes the connection to that existing file, and obtains any additional output connections, without prompting for permission to delete other existing files.

Any other response causes the system call to return a condition code without obtaining a connection to the file.

`over` If you specify the `over` preposition, **c_get_output_connection** obtains the connection without prompting the user for permission.

`after` If you specify the `after` preposition, **c_get_output_connection** obtains the connection without prompting the user for permission. It also sets the file pointer to the EOF before returning control. Thus, new information does not overwrite existing information.

This is unlike `to` and `over` which cause **c_get_output_connection** to leave the file pointer at the beginning of the file.

If the user does not have the proper access rights to the file, or if **c_get_output_connection** cannot obtain a connection to the file, the system call returns a condition code and displays a message at the user's terminal.

See also: **c_get_output_connection** HI system call, *System Call Reference*

A normal scenario for using **c_get_input_connection** and **c_get_output_connection** is shown in Figure 28-1.

```
DO WHILE more input and output files
       Obtain input pathname from command line with
              c_get_input_pathname
       Obtain output pathname from command line with
              c_get_output_pathname
       Obtain connection to input file with
              c_get_input_connection
       Obtain connection to output file with
              c_get_output_connection
       Read information from input file
       Perform command operations on information
       Write information to output file
       Delete connections to input and output files
END
```

Figure 28-1. c_get_input_connection and c_get_output_connection Example

Communicating With the User's Terminal

The HI provides two system calls that communicate with the user's terminal. They are **c_send_co_response** and **c_send_eo_response**. Each of these system calls combines into a single system call several operations that you would normally perform when communicating with the terminal.

c_send_co_response System Call

In its general form, **c_send_co_response** attaches and opens connections to *:ci:* and *:co:*. Depending on the values you choose as parameters for this system call you can:

- Send a message and receive a message (write to *:co:* and read from *:ci:*).
- Send a message without waiting to receive a message (read from *:ci:*).
- Receive a message without sending a message (write to *:co:*).

C_send_co_response deals specifically with the logical names *:ci:* and *:co:*. Therefore, its input and output can be redirected to files by changing the pathnames represented by these logical names. For example, when a user places a command in a submit file, **submit** assumes that *:ci:* is the submit file and that *:co:* is the output file specified in the **submit** command. Figure 28-2 on page 344 shows how to use **c_send_co_response**.

See also: **c_send_co_response** HI system call, *System Call Reference*

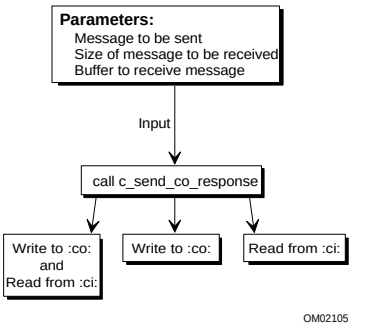


Figure 28-2. Using c_send_co_response

c_send_eo_response System Call

C_send_eo_response, although it performs the same operations as **c_send_co_response**, only reads information from and writes information to the

user's terminal. Input and output cannot be redirected. This system call is useful if you have multiple tasks communicating with a single terminal.

For example, if a task uses either of these system calls and requests a response from the terminal, no other output is displayed at the terminal until the user enters a response to the first system call. After the user responds, tasks can send further information to the terminal.

When used by all the tasks which communicate with the terminal, this prevents the user from receiving several requests for information before being able to respond to the first one.

See also: **c_send_eo_response** HI system call, *System Call Reference*

Formatting Messages Based on Condition Codes

Whenever you include OS calls in the code of a command that you write, it is possible for those system calls to encounter exceptional conditions, such as:

- Programming errors
- Environmental conditions

Even the most thoroughly debugged commands can encounter exceptional conditions. The exceptional conditions can arise from invalid user entries, lack of secondary storage space, media errors, and other problems over which the command has no control.

The HI provides a default exception handler to handle exceptional conditions in commands that you write. This exception handler receives control on all occurrences of exceptional conditions. It displays the condition code value and mnemonic at the user's terminal and aborts the command.

You can use the Nucleus system calls **get_exception_handler**, **rqe_get_exception handler**, **set_exception_handler**, and **rqe_set_exception_handler** to provide your own exception handling, either to pass additional information to the user or to enable the user another chance to enter correct information. You can also use these calls to cancel the effect of the default exception handler on some or all exceptions that occur in your command.

See also: **get_exception_handler**, **set_exception_handler**, **rqe_get_exception_handler**, and **rqe_set_exception_handler**, *System Call Reference*

c_format_exception System Call

When you perform your own exception handling, you can create messages that return to the user under specific exceptional conditions so they can correct the problem. The **c_format_exception** system call accepts a condition code value as input and returns a string whose contents describe the exceptional condition. You can use this string as input to a system call such as **c_send_co_response** to write the information to the user's terminal.

By using **c_format_exception**, you can return a message to the user for all exceptional conditions, but you do not have to enlarge your program by including the text of these messages in the code of your command.

The text portion of the string produced by **c_format_exception** consists of the condition code value and mnemonic in this format:

```
value : mnemonic
```

You can display this string as is, or you can place additional explanatory text in the string before displaying it.

This PL/M example shows you how to use **c_format_exception** to write an error message to the screen every time a procedure named `DoSomething` encounters an exception. You can declare a message as follows:

```
DECLARE
  error_msg    STRUCTURE(
                         length       BYTE,
                         char(80)     BYTE),
  failed(*) BYTE DATA(33,'DoSomething procedure failed ***
                    '),
  excep          WORD,
  local_excep    WORD;
```

Now, whenever `DoSomething` encounters an exception during execution, you can call **c_format_exception**, as shown below, to create the default message for the exception contained in the `excep` variable and concatenate it to the failed message you declared in the variable `failed`.

```
CALL MOVB(@failed, @error_msg, SIZE(failed));

CALL rq$C$format$exception(@error_msg, SIZE(error_msg),
                           excep, 1, @local_excep);
```

You can write the `error_msg` string to the screen. For example, if the `excep` variable contains 05H, the string contained in `error_msg` would be:

```
'DoSomething procedure failed *** 0005: E_CONTEXT'
```

See also: Examples in */rmx386/demo/c/hi* directory;
c_format_exception, *System Call Reference*

□□□

Invoking HI Commands Programmatically 29

When you write your own command, you might want to perform an operation that is already provided in another command (such as copying one file to another, displaying a directory, etc.). Instead of duplicating the code for this operation in your command, you can invoke HI system calls to issue the commands themselves. The effect of making these system calls is the same as that produced by a user entering an HI command at the terminal. The HI provides the three system calls in Table 29-1 to help process command invocations:

Table 29-1. Command Invocation System Calls

Call Name	Function
c_create_command_connection	Creates a command connection object to store the command invocation lines
c_send_command	Sends the command line to the command connection and invokes the command
c_delete_command_connection	Deletes the command connection

This chapter discusses these operations and provides an example of how the system calls appear in a program.

Creating a Command Connection

The **c_create_command_connection** system call creates the object and returns a token for it. The token can be used in calls to **c_send_command** (to send command lines to the object) and in calls to **c_delete_command_connection** (to delete the object after using it).

When you call **c_create_command_connection**, you also specify tokens for the connections that serve as command input and command output for the invoked command. This enables you to redirect input and output for the invoked command to secondary storage files. Or you can specify *:ci:* and *:co:*.

The command connection supports processing multiple-line commands without interference from other tasks. Without the command connections, the OS would be unable to determine which continuation line went with which command when many tasks were sending command lines to be processed. The command connection provides a place to store command lines until the command is complete.

Sending Command Lines to the Command Connection and Invoking the Command

The **c_send_command** system call sends command lines to a command connection and, when the invocation is complete, invokes the command. The format of the command line is the same as entering the command line at a terminal. The command can be any HI command or any command that you write. However, it cannot be a CLI command and it cannot use the alias feature of the CLI.

See also: HI commands, *Command Reference*

If the string specified as a parameter to **c_send_command** contains a complete command invocation then this takes place:

1. **C_send_command** places the command line in the command connection.
2. **C_send_command** invokes the command.

However, if the string does not contain the entire command invocation (that is, it contains the & as a continuation character), then this takes place:

1. **C_send_command** places the command line in the command connection without invoking the command.
2. **C_send_command** returns a condition code, E_CONTINUED, to inform the calling program that the command is continued.
3. The programmer calls **c_send_command** to combine continuation lines in the command connection with the command lines already there.
4. Repeat Step 3 until **c_send_command** encounters the end of the command invocation (a line without a continuation character).
5. **C_send_command** loads the command from secondary storage.
6. **C_send_command** invokes the command.

The **c_send_command** call that invokes the command does not return control until the invoked command finishes processing. Once the command finishes processing, you can use the command connection for invoking other commands.

The **c_send_command** system call contains two pointers to WORDs or unsigned shorts that receive condition codes. One of these points to a location that receives the status of the **c_send_command** system call. The other points to a location that receives the status of the invoked command.

Priority Considerations

Every command has a priority (usually based on the priority of the user who invoked the command) that determines when the command will be able to run in relation to the other tasks in the system. When commands are invoked using command connections, their priorities are lowered (numerically increased) by one. This ensures that the calling task (the one that created the command connection) retains control over the commands it invokes.

As a result, a command invoked directly at the terminal will have a higher priority (and possibly complete sooner) than the same command invoked using a command connection.

See also: **rqe_set_max_priority** command, *System Call Reference*

Deleting the Command Connection

After you have finished invoking commands programmatically, delete the command connection. The **c_delete_command_connection** system call performs this operation. You do not need to delete the command connection after each command invocation, because the command connection is reusable. However, delete the command connection after performing all **c_send_command** operations. This frees the memory used by the data structures of the command connection.

Command Connection Calls Demo Programs

There are two demo programs (one written in C, the other in PL/M) installed with the OS that use **c_create_command_connection**, **c_send_command**, and **c_delete_command_connection**. These programs invoke the HI **copy** command programmatically.

See also: Examples in */rmx386/demo/c/hi* directory

❑❑❑

Writing a <Ctrl-C> Handler 30

Normally, when an HI command is executing, a user cannot communicate with the system until the command requests input from the user. This can present problems if a user enters the wrong command or needs to access the system. However, there are a number of ways the user can abort command execution.

- If the command is executing interactively, the user can enter a <Ctrl-C> character to abort a command.
- If the command is running in the background environment, the user can enter the CLI commands **jobs** and **kill** to abort a job.

This chapter explains how to override the default <Ctrl-C> action by providing your own code to process a <Ctrl-C> character.

See also: Aborting background jobs, *Command Reference*

How the Default <Ctrl-C> Works

When the user enters a <Ctrl-C>, the OS sends a unit to a semaphore. In the default case, this is a semaphore established by the HI. An HI task waits at that semaphore to receive the unit. When it receives the unit, it aborts the command that is currently executing and returns control to the user. The HI task then waits at the semaphore for another unit.

This <Ctrl-C> facility enables users to cancel commands while the commands are executing. It can be used with your commands without requiring special implementation code.

Providing Your Own <Ctrl-C>

With some commands that you write, you might want to override the default <Ctrl-C> handling. For example, suppose you write a text editor. A user invokes the editor with an HI command and then specifies edit commands to enter text into a buffer and modify that text.

While using the editor, the user does not want a <Ctrl-C> character to abort the entire editing session, destroying text in the editing buffer that could have taken hours to create. Instead, the user might want a <Ctrl-C> to abort a single editor command only. In order to provide this facility, your HI command (the editor) must override the default <Ctrl-C> handling and provide its own code to handle <Ctrl-C> entries.

By changing the semaphore to one that you create, you can circumvent the default <Ctrl-C> task of the HI. You can use the HI system call **c_set_control_c** to replace the <Ctrl-C> semaphore. This system call changes the calling job's <Ctrl-C> semaphore to the semaphore you specify. There is only one parameter in this system call: `control_c_semaphore` which is a token for your new <Ctrl-C> semaphore. A single unit is sent to the new semaphore each time a <Ctrl-C> is entered from the terminal.

See also: HI system call **c_set_control_c**, *System Call Reference*

If you create an HI command that does not use the default <Ctrl-C> semaphore, that command must service the new <Ctrl-C> semaphore. It can do this by:

- Using inline code that periodically checks the semaphore for a unit.
- Creating a task that waits continually at your <Ctrl-C> semaphore for a unit.

In either case, when a unit is sent to the semaphore, the command (or the task) must perform the necessary <Ctrl-C> operation.

CAUTION

If you also include the UDI in your application, the <Ctrl-C> handler will revert to the UDI default handler unless you establish the new <Ctrl-C> handler in the UDI with the **dq_trap_cc** call.

Using Inline Processing

The program flow of such a command using inline processing would be:

1. Call **create_semaphore** to create the <Ctrl-C> semaphore.
2. Call **c_set_control_c** to switch the <Ctrl-C> semaphore to the one just created. Use the token for the semaphore you created in Step 1 as input.

3. Continue with command processing. Periodically check the semaphore (by calling **receive_units** with the `time_limit` parameter set to 0) to determine if it contains any units. If you obtain any units from the semaphore, perform the necessary <Ctrl-C> processing.

If your command services the <Ctrl-C> semaphore with inline code, you can perform any operation you want. You can branch to various locations, you can start new tasks running, you can abort the command, or you can perform any other function that you wish.

However, in order to service the <Ctrl-C> semaphore with inline code, check the semaphore periodically, to see if it contains a unit. When doing this, ensure that you place the checks inside all program loops that perform operations a user might want to abort. Also, because you can check the semaphore only periodically, you cannot always guarantee a quick response to the <Ctrl-C>.

Using a <Ctrl-C> Task

The program flow of such a command using a task would be:

1. Call **create_semaphore** to create the <Ctrl-C> semaphore.
2. Call **catalog_object** to catalog the token for the semaphore in an object directory.
3. Call **create_task** to start the <Ctrl-C> task.
4. Call **c_set_control_c** to switch the <Ctrl-C> semaphore to the one just created. Use the token for the semaphore you created in Step 1 as input.
5. Continue with command processing.

The program flow of the <Ctrl-C> task could be:

1. Call **lookup_object** to obtain the token for the semaphore.
2. Do forever:
    a. Call **receive_units** with the `time_limit` parameter set to 0FFFFH to obtain a unit from the semaphore.
    b. Perform the operation that must occur when the user enters a <Ctrl-C>.

If you use a <Ctrl-C> task, you can guarantee quick service because the task is always waiting at the semaphore. However, because a separate task services the <Ctrl-C>, you can perform only a limited number of operations in response to the <Ctrl-C>.

- The task can send a message to the command, but then the command would have to periodically check a mailbox. This has the same disadvantages as inline servicing with none of the advantages.
- The task can delete or suspend the command. However, the task has no way of knowing what operations the command was performing when the user entered the <Ctrl-C>. If the command was updating an internal table, deleting the command could corrupt your entire system. Suspending the command could enable the <Ctrl-C> task to interrogate the command's state. The <Ctrl-C> task could delete the command if appropriate, or it could enable the command to run until it was safe to be deleted.

Returning to the Default Handler

Once your command assigns a new <Ctrl-C> semaphore, that assignment remains until either:

- Your command invokes the HI **c_send_command** system call. Invoking this system call automatically reverts back to the default <Ctrl-C>. To continue using your own <Ctrl-C>, invoke **c_set_control_c** (to switch back to your <Ctrl-C> semaphore) immediately after invoking **c_send_command**.
- Your command is deleted. When this happens, the HI automatically reactivates its default <Ctrl-C> semaphore. For example, once the example text editor described earlier in this chapter terminates, the HI resets the semaphore so that <Ctrl-C> again becomes active.

<Ctrl-C> Task Demo Programs

There are two demo programs (one written in C, the other in PL/M) installed with the OS that are examples of a user-supplied <Ctrl-C>.

See also: Examples in */rmx386/demo/c/hi* directory

□□□

Creating Human Interface Commands 31

This chapter discusses the steps that you must perform to create your own HI commands. It discusses the necessary elements of a command as well as how to compile (or assemble) and bind your code.

You can make your application into an HI command and run it on an DOSRMX system. This requires these steps.

1. Program your application.
2. Give the application a command name and specify parameters, if any.
3. Provide for the command to parse its command line parameters, if any.
4. Provide for the command to terminate itself when finished. If you plan to use **sysload** to load it, use the **delete_job** system call. Otherwise, use **exit_io_job**.
5. Compile the command using the appropriate compiler.
6. Bind the command to the appropriate libraries to make a Single Task Loadable (STL) file. The `RCONFIGURE` control makes the command loadable.
7. Load the command manually (`x` in these examples), using one of these methods.

   - `sysload x parameter 1 parameter n <CR>`
     The job will continue to be available.

   - `background x parameter 1 parameter n <CR>`
     The command runs in the background. Redirect *:ci:* and *:co:* to log files.

   - `SS x parameter 1 parameter n <CR>`
     The command runs in the foreground; debug it using Soft-Scope.

   - `debug x parameter 1 parameter n <CR>`
     The command runs in the foreground and you can debug it.

   If you use **sysload** to load your application, that job will continue to be available.

Detailed instructions for steps 3, 4, 5, and 7 are described in sections which follow.

To perform the operations described in this chapter, you must have a system that includes the HI commands. The system must have an editor, the necessary compiler or assembler, and the appropriate binder, such as BND286 for 16-bit HI commands and BND386 for 32-bit HI commands.

Elements of a Human Interface Command

This section discusses the rules that every user-written command must obey. It also suggests some programming practices to make coding and using your commands easier.

Note

When coding your commands, avoid duplicating CLI command names such as, **alias** and **submit**. If you do name a new command with the same name as a CLI command, execute it with the full pathname, for example, *:utils:alias*. Otherwise, the CLI command will be executed instead of your command.

Parsing the Command Line

If you are going to enable the user to enter parameters when invoking the command, the first thing your command should do is parse the command line. To support lists of pathnames and wildcarded pathnames, the flow of a program that uses input and output files should be:

1. Call **c_get_input_pathname** to obtain the entire list of input pathnames.
2. Call **c_get_output_pathname** to obtain the preposition and the entire list of output pathnames.
3. Call **c_get_parameter** as many times as necessary to get all the parameters.
4. Do until no more input pathnames remain:
    a. Call **c_get_input_connection** to obtain a connection to the input file.
    b. Call **c_get_output_connection** to obtain a connection to the output file.
    c. Read the information from the input file, perform the command operations based on that input, and write the information to the output file.
    d. Call the EIOS **s_delete_connection** call to delete the connections to the input and output files.
    e. Call **c_get_input_pathname** and **c_get_output_pathname** to obtain the next input and output pathnames.

System Calls and Objects to Avoid

Although you can use any of the OS calls you require, some system calls are intended primarily for use in system-level jobs (those jobs that you configure into the OS rather than invoking as HI commands). The command descriptions for those calls describe when the calls should be avoided.

In particular, avoid objects (and their associated system calls) that, by their use, make your command immune to deletion. Regions and extension objects are examples of such objects. If your command becomes immune to deletion, a <Ctrl-C> that a user enters to cancel the command will have no effect; the user's terminal may also lock when the command finishes processing.

See also: Regions, extension objects, in this manual

Terminating the Command

When the user invokes a command, the OS loads the command into memory and creates an I/O job as the environment in which the command runs. The user can use the CLI **background** command to process commands in background mode, and at the same time continue processing another command in the foreground. In order to finish processing a foreground command correctly, any task in the command that exits must do so by calling **exit_io_job**. This system call causes the OS to delete the I/O job containing the command, therefore returning control to the user.

See also: I/O jobs, in this manual;
EIOS system call **exit_io_job**, *System Call Reference*

If the command running in the foreground omits the call to **exit_io_job**, the user might not be able to enter further commands. To terminate a command before it reaches its normal completion, the user should enter <Ctrl-C> to abort a command running in the foreground or the CLI **kill** command to abort a command running in the background environment.

Include Files

When writing the code for your commands, declare each OS call as an external procedure. Instead of writing these declarations yourself, you can use the `include` statement. Using `include` statements makes it possible to include code from an external file into your program. This information may be in an `include` file:

- External declarations of system calls
- Literal definitions of condition codes
- Common literal definitions that you declare

See also: Header files, *System Call Reference*

Producing a 16-bit Executable Command

After you have written the source code for your command, produce object code that can be executed in a 16-bit environment. Follow these steps:

> **Note**
> This section applies to object code developed using Intel tools only.
>
> See also: C Compiler-specific Information for information on building executable code with non-Intel tools, *Programming Techniques*

1. Compile (or assemble) the command using the appropriate translators. When you do this, ensure that the names you specify in `include` statements specify the correct devices and directories.
2. Using BND286, bind the code to the interface libraries (and any other libraries that you require) and produce a relocatable object module that the OS can load anywhere in memory. The format of the BND286 command is:

```
BND286                    &
  command-name,           &
  :RMX:LIB/RMXIF*.LIB     &
  :dir:other.lib,         &
      RCONFIGURE (DYNAMICMEM(min,max))  &
      OBJECT(output-pathname)           &
      SEGSIZE(STACK(stacksize))
```

Where:

`command-name`	The complete pathname of the file containing your compiled (or assembled) command. You can bind in several files or libraries at this point, if necessary.
`:dir:`	A generic logical name you create for directories containing miscellaneous libraries.
`other.lib`	Any other files or libraries that you need to bind with your command, for example, *plm286.lib*.
`*`	Replace this character with `C` if you are using COMPACT.
`output-pathname`	Complete pathname of the file in which BND286 places the command after binding.

`stacksize` Size, in bytes, of the stack needed by the command and any system calls that the command makes. The HI uses this value when it creates a job for the command. Be sure the stack is large enough to handle both user and system requirements.

See also: Stack requirements, *Programming Techniques*

`min,max` Minimum and maximum amount of dynamic memory, in bytes, required by the command.

The command uses this memory when it creates iRMX objects. The AL uses the `min` and `max` values when it loads a job for the command. Be sure that these values are large enough to satisfy the needs of your command and small enough to enable the command to be loaded into the user's memory partition.

For example, suppose a sort command requires at least 64 Kbytes of dynamic memory but can use any additional dynamic memory for buffers to increase performance. If you do not define a maximum memory parameter, all of your dynamic memory will be allocated to the sort command, preventing you from executing other commands at the same time. Therefore, assume that you want to limit the `max` value to 1 Mbyte. Specify:

```
RCONFIGURE(DYNAMICMEM(10000H,100000H))
```

Consider these factors when calculating the values for `min` and `max`.

- The value you give for the `min` field plus the memory required by the HI program must fit into contiguous memory. If there is not enough contiguous memory for them, you may not be able to load your command.
- The value for the `max` field should be large enough to ensure enough memory for commands that request memory dynamically.

The command is now ready for execution. A user can invoke the command by entering the pathname of the file containing the command (the `output-pathname` in the BND286 command).

Producing a 32-Bit Executable Command

After you have written the source code for your command, produce the object code. To generate a 32-bit command, use these steps. (16-bit commands can run on iRMX III and DOSRMX also.)

Note

This section applies to object code developed using Intel tools only.

See also: C Compiler-specific Information for information on building executable code with non-Intel tools, *Programming Techniques*

1. Compile (or assemble) the command using the appropriate translators. When you do this, ensure that the names you specify in `include` statements specify the correct devices and directories.
2. Using BND386, bind the code to the OS interface libraries (and any other libraries that you require) and produce a relocatable object module that the OS can load anywhere in memory. The format of the BND386 command is:

```
BND386                    &
  command-name,           &
  :RMX:LIB/RMXIFC32.LIB   &
  :dir:other.lib,         &
      RCONFIGURE (DYNAMICMEM(min,max))  &
      OBJECT(output-pathname)           &
      SEGSIZE(STACK(stacksize))         &
      RENAMESEG (CODE to CODE32, DATA to DATA32)
```

Where:

`command-name`	The complete pathname of the file containing your compiled (or assembled) command. You can bind in several files or libraries at this point, if necessary.
`:dir:`	A generic logical name you create for directories containing miscellaneous libraries.
`other.lib`	Any other files or libraries that you need to bind with your command, for example, *plm386.lib*.
`output-pathname`	Complete pathname of the file in which BND386 places the command after binding.

`stacksize` Stack size, in bytes, needed by the command and any system calls that the command makes. The HI uses this value when it creates a job for the command. Be sure the stack is large enough to handle both user and system requirements. The OS supports compact interface procedures.

See also: Stack requirements, *Programming Techniques*

`min,max` Minimum and maximum amount of dynamic memory, in bytes, required by the command.

The command uses this memory when it creates objects. The Application Loader (AL) uses the `min` and `max` values when it loads a job for the command. Be sure that these values are large enough to satisfy the needs of your command and small enough to enable the command to be loaded into the user's memory partition.

For example, suppose a sort command requires at least 64 Kbytes of dynamic memory but can use any additional dynamic memory for buffers to increase performance. If you do not define a maximum memory parameter, all of your dynamic memory will be allocated to the sort command, preventing you from executing other commands at the same time. Therefore, assume that you want to limit the `max` value to 1 Mbyte. Specify:

```
RCONFIGURE(DYNAMICMEM(10000H,100000H))
```

Consider these factors when calculating the values for `min` and `max`.

- The value you give for `min` and the memory required by the HI program must fit into contiguous memory. If there is not enough contiguous memory for them, you may not be able to load your command.
- The `max` value should be large enough to ensure memory for commands that request memory dynamically.

The command is now ready for execution. A user can invoke the command by entering the pathname of the file containing the command (the `output-pathname` in the BND386 command).

❑❑❑

INtime® 2.0 Compatibility and Interoperablilty 32

The iRMX III.2.3 OS includes components of the INtime 2.0 Windows NT Enhancement software. These components allow an iRMX III.2.3 system to function as a Remote INtime Client, thus allowing communications between a Windows NT Host and itself (as a Remote INtime Client), as well as running INtime RT software directly on the iRMX III.2.3 system. As a direct benefit of this communications mechanism (NTX) with an NT system, an updated version of Soft-Scope can be used on the NT Host that can communicate either serially (at up to 115KB) or via UDP/IP with the iRMX III.2.3 system to download and debug iRMX or INtime applications on the iRMX III.2.3 system (acting as a Remote INtime Client).

Becoming a Remote INtime Node

The following jobs/components must be running on the iRMX III.2.3 system to allow it to act as a Remote INtime Client, both for communications and application cross debug purposes:

- Paging Job
- Flat Job
- Remote INtime Personality Job
- Appropriate Remote INtime low level drivers
- serdrvr.job for serial interface
- ne.job, tulip.job, eepro.job, or eepro100.job for
- UDP interface
- New iRMX TCP/IP Stack components (TBD)
- Appropriate Channel Interface Module (CIM)
- rtcimcom.rta for serial communications
- rtcimudp.rta for UDP/IP communications
- NTX Proxy Job (ntxproxy.rta)

❑❑❑

Windows NT Host Cross-Development Environment 33

You can now develop iRMX Applications on a Windows NT Host and debug them on a Remote iRMX III.2.3 system using either serial or UDP/IP communications interfaces:

- To configure and generate an iRMX III.2.3 application system, you can run a DOS-hosted iRMX Interactive Configuration Utility (ICU) from a Windows NT Console (DOS Box).
- To develop an iRMX application, run the standard Intel/RadiSys OMF386 tools from a Windows NT Console (DOS Box).
- To download and debug an iRMX application on a remote iRMX system, use a Windows NT-hosted version of Soft-Scope.
- To drive OMF386 tools in the development of sample iRMX applications, use the various provided iRMX Demo Applications that use DOS-hosted make files.
- For easy setup of a Windows NT Host to communicate via NTX with a Remote INtime Client (iRMX III.2.3), use the provided Windows NT-hosted Windows NT-Link Configuration Utility. The Utility also produces working iRMX Configuration and batch (.CSD) files that you can use to set up the iRMX system as a Remote INtime Client.

□□□

OS Extension Example A

Ring Buffer Manager

This example (in PL/M) illustrates portions of a ring buffer manager and various parts of an OS extension. Be advised, however, that the example is incomplete and should be imitated with discretion. In particular, the example has these shortcomings:

- The issue of exception handling is not addressed. Clearly the code supporting a system call should examine each invocation for validity, but, for brevity, the ring buffer example does not do this.
- There are no safeguards against partial creation of an object. When creating a composite object, a type manager must first create the components of the object. Occasionally, after creating some of the components, the manager might be unable to create the others. A type manager should be able to recover from this situation, usually by deleting the components already created and returning an exception code to the caller. The example, again for brevity, does not do this.
- The entry routine does not check the entry code for validity.
- The potential for problems with deletion is ignored. For this reason, you should imagine that the environment of the example is constrained in at least two ways. First, only one task will ever try to delete a ring buffer and, when it does try, no other task will be using that buffer. Second, when a job containing a task that created a ring buffer is deleted, no tasks in other jobs are using that ring buffer.
- The example has been desk-checked, but the example has not actually been tested.
- The example ring buffer is limited to a maximum of 64 Kbytes in length.
- The example assumes use of version V3.1 or later of the desired PL/M compiler, i.e. PL/M-286 or PL/M-386.

A *ring buffer* is a block of memory in which bytes of data are placed at successively higher addresses. Byte removals are interspersed with byte insertions, with the restriction that the byte being removed must always be the byte that has been in the buffer for the longest time. Thus, data enters and leaves a ring buffer in a FIFO manner. Ring buffers are so named because the lowest address logically follows the highest address. That is, if the last byte placed in (or retrieved from) the buffer is at its highest address, then the next byte to be placed in it (or retrieved from it) is at the lowest address. As data enters and leaves the buffer, the portion containing data runs around the ring, with the pointer to the last byte out chasing the pointer to the last byte in. Figure A-1 illustrates these characteristics.

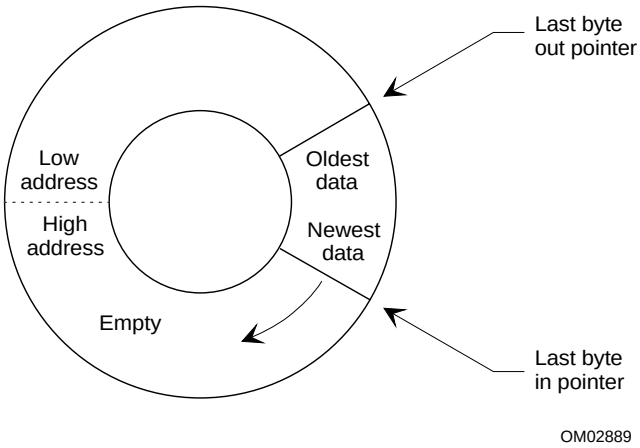


Figure A-1. A Ring Buffer

The main (service) part of the example consists of four procedures: CREATE_RING_BUFFER, DELETE_RING_BUFFER, PUT_BYTE, and GET_BYTE. The last two procedures are for placing a character in a ring buffer and for retrieving a character, respectively.

Note

The text description and the figures in this appendix use C-language syntax. However, these procedure examples are in PL/M-language syntax.

```
/*****************************************************************
 * NOTE:  The common literal file (COMMON.LIT) is included       *
 * in each of the PL/M portions of the example.  This include file *
 * uses conditional compilation.  The compilation switch 'R_32' be *
 * used when compiling with PL/M-386.                            *
 *****************************************************************/
$IF word16
       DECLARE WORD_32            LITERALLY 'DWORD';
       DECLARE WORD_16            LITERALLY 'WORD';
$ELSE
       DECLARE WORD_32            LITERALLY 'WORD';
       DECLARE WORD_16            LITERALLY 'HWORD';
$ENDIF
       DECLARE TOKEN       LITERALLY 'SELECTOR';
$IF r_32
       DECLARE SIZE$OF$OFFSET     LITERALLY 'WORD_32';
$ELSE
       DECLARE SIZE$OF$OFFSET     LITERALLY 'WORD_16';
$ENDIF
       DECLARE forever            LITERALLY 'WHILE 1';
       DECLARE indefinitely       LITERALLY '0FFFFH';
       DECLARE ASTR$STRUC  LITERALLY 'STRUCTURE(
                                   num$slots       WORD,
                                   num$components  WORD,
                                   seg             TOKEN,
                                   empty$ct        TOKEN,
                                   full$ct         TOKEN)';
       DECLARE POINTER$STRUC      LITERALLY 'STRUCTURE(
                                   off_set         SIZE$OF$OFFSET,
                                   selector        SELECTOR)';

       DECLARE SEGMENT$STRUC      LITERALLY 'STRUCTURE(
                                   size            WORD,
                                   head            WORD,
                                   tail            WORD,
                                   buffer(1)       BYTE)';
```

Initialization

The initialization task creates a region to protect data in ring buffers from being manipulated by more than one task at a time. This part of the OS extension also creates the required extension type and creates a deletion mailbox. In an ICU-configurable system, the OS extension call-gates are established during configuration. For this example, they are GDT slots 440H, 441H, 442H, and 443H. Finally, this part of the OS extension waits at the deletion mailbox. Code for the initialization task includes this:

```
$IF r_32
$COMPACT(-CONST IN CODE- has example)
$LARGE(other_libs EXPORTS ring$buffer$manager)
$ENDIF

example:
DO;
$INCLUDE(:RMX:INC/COMMON.LIT)     /* Declares common literals
                                     */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

       RING$BUFFER$MANAGER:  PROCEDURE EXTERNAL;
       END RING$BUFFER$MANAGER;
       DECLARE ring$buffer$type   TOKEN PUBLIC;
       DECLARE ring$buffer$region TOKEN PUBLIC;

  RING_BUFFER_INIT: PROCEDURE;
       DECLARE delete$object      TOKEN;
       DECLARE exception          WORD;
       DECLARE fifo               LITERALLY '0';
       DECLARE rb$code            LITERALLY '8000H';
       DECLARE deletion$mbox      TOKEN;
       DECLARE response$mbox      TOKEN;

       ring$buffer$region = RQ$CREATE$REGION (
              fifo,
              @exception);

       deletion$mbox = RQ$CREATE$MAILBOX (
              fifo,
              @exception);
```

```
            ring$buffer$type=R$$CREATE$EXTENSION (
                  rb$code,
                  deletion$mbox,
                  @exception);

      $IF rmx86
            CALL RQ$SET$OS$EXTENSION(
                  224,
                  @ring$buffer$manager,
                  @exception);
      $ENDIF

            CALL RQ$END$INIT$TASK;

            DO FOREVER;
                  delete$object = RQ$RECEIVE$MESSAGE (
                        deletion$mbox,
                        indefinitely,
                        @response$mbox
                        @exception);

/*****************************************************************
* If desired, delete the components of the composite object.  They *
* are not automatically deleted when DELETE$EXTENSION is called.   *
* See the DELETE$RING$BUFFER procedure, shown later, for the code  *
* that does this.                                                  *
*****************************************************************/

                  CALL RQ$DELETE$COMPOSITE (
                        delete$object,
                        @exception);
                  END;   /* FOREVER */
            END RING_BUFFER_INIT;
      END example;
```

The Interface Library

The user interface library consists of four small procedures, one for each of the system calls provided by the OS extension. The library supports application code written in the PL/M COMPACT model. If a different model had been used for compiling the application code, these interface procedures would be slightly different, reflecting the fact that, when making procedure calls in other models, the stack is used differently than in the COMPACT model.

See also: Interface libraries, *Programming Techniques*;
Interface libraries, *System Call Reference*

The interface procedures are as follows:

```
                                        ; define macro to allow
                                        ; both 16 and 32 bit
                                        ; usage
$IF (%r_32) THEN(%'
                                        ; 32 bit registers/data types
%define(ax) (eax)
%define(bx) (ebx)
%define(cx) (ecx)
%define(dx) (edx)
%define(si) (esi)
%define(di) (edi)
%define(bp) (ebp)
%define(sp) (esp)
%define(mov16) (movzx)
%define(pusha) (pushad)
%define(popa) (popad)
%define(pushf) (pushfd)
%define(popf) (popfd)
%define(iret) (iretd)
%define(dw) (dd)
%define(dd) (dp)
) ELSE (%'
```

```
                                        ; 16 bit registers/data types
%define(ax) (ax)
%define(bx) (bx)
%define(cx) (cx)
%define(dx) (dx)
%define(si) (si)
%define(di) (di)
%define(bp) (bp)
%define(sp) (sp)
%define(mov16) (mov)
%define(pusha) (pusha)
%define(popa) (popa)
%define(pushf) (pushf)
%define(popf) (popf)
%define(iret) (iret)
%define(dw) (dw)
%define(dd) (dd)
)FI%'

       CREATERB     PROC   NEAR
                    PUBLIC       CREATERB
%IF (NOT(%rmx86)) THEN (
                    EXTRN   GATE 440: FAR
)FI

                    PUSH    %BP
                    MOV     %BP, %SP
%IF (%rmx86) THEN (
                    LEA    %SI, SS:%BP+4 ; SS:SI contains
                                           location of first
                                           parameter
                    MOV    BX, 0          ; code for
                                           CREATE_RING_BUFFER
                    INT    224            ; call the
                                           OS-extension via a
                                           software interrupt
) ELSE (
                    PUSH    SS:%BP+4      ; parameter--the size
                                           of the ring buffer
                    CALL    GATE 440      ; call the
                                           OS-extension via a
                                           call-gate
```

```
)FI
                    POP     %BP                ; restore BP value
                    RET     2                  ; return clearing
                                                 passed parameter
       CREATERB     ENDP
       DELETERB     PROC    NEAR
                    PUBLIC  DELETERB
%IF (NOT(%rmx86)) THEN (
                    EXTRN   GATE 441: FAR
)FI

                    PUSH    %BP
                    MOV     %BP, %SP
%IF (%rmx86) THEN (
                    LEA    %SI, SS:%BP+4 ; SS:SI contains
                                           location of first
                                           parameter
                    MOV    BX, 1          ; code for
                                           DELETE_RING_BUFFER
                    INT    224            ; call the
                                           OS-extension via a
                                           software interrupt
) ELSE (
                    PUSH    SS:%BP+4      ; parameter--target
                                           ring buffer
                    CALL    GATE 441      ; call the
                                           OS-extension via a
                                           call-gate
)FI
                    POP     %BP           ; restore BP value
                    RET     2             ; return clearing
                                            passed parameter
       DELETERB    ENDP

       GETRBYTE     PROC    NEAR
                    PUBLIC  GETRBYTE
%IF (NOT(%rmx86)) THEN (
                    EXTRN   GATE 442: FAR
)FI

                    PUSH    %BP
```

```
                    MOV     %BP, %SP
%IF (%rmx86) THEN (
                    LEA     %SI, SS:%BP+4 ; SS:SI contains
                                            location of first
                                            parameter
                    MOV     BX, 2           ; code for GET_BYTE
                    INT     224             ; call the
                                            OS-extension via a
                                            software interrupt
) ELSE (
                    PUSH    SS:%BP+4        ; parameter--target
                                            ring buffer
                    CALL    GATE 442        ; call the
                                            OS-extension via a
                                            call-gate
)FI
                    POP     %BP             ; restore BP value
                    RET     2               ; return clearing
                                            passed parameter
      GETRBYTE    ENDP

      PUTRBYTE      PROC    NEAR
                    PUBLIC  PUTRBYTE
%IF (NOT(%rmx86)) THEN (
                    EXTRN   GATE 443: FAR
)FI

                    PUSH    %BP
                    MOV     %BP, %SP
%IF (%rmx86) THEN (
                    LEA     %SI, SS:%BP+4 ; SS:SI contains
                                            location of first
                                            parameter
                    MOV     BX, 3           ; code for PUT_BYTE
                    INT     224             ; call the
                                            OS-extension via a
                                            software interrupt
```

```
) ELSE (
                    PUSH    SS:%BP+6          ; parameter--character
                                                to write
                    PUSH    SS:%BP+4          ; parameter--target
                                                ring buffer
                    CALL    GATE 443          ; call the
                                                OS-extension via a
                                                call-gate
)FI
                    POP     %BP               ; restore BP value
                    RET     4                 ; return clearing
                                                passed parameters
      PUTRBYTE    ENDP
```

These interface procedures correspond to a set of external procedure declarations in the application PL/M code:

```
CREATERB:  PROCEDURE(size)                      TOKEN EXTERNAL;
       DECLARE size                             WORD;
END CREATERB;

DELETERB:  PROCEDURE(ring$buffer$token)         EXTERNAL;
       DECLARE ring$buffer$token                TOKEN;
END DELETERB;

GETRBBYTE:  PROCEDURE(ring$buffer$token)        BYTE EXTERNAL;
       DECLARE ring$buffer$token                TOKEN;
END GETRBBYTE;

PUTRBBYTE:  PROCEDURE(char, ring$buffer$token) EXTERNAL;
       DECLARE char                                    BYTE;
       DECLARE ring$buffer$token                       TOKEN;
END PUTRBBYTE;
```

The Create Ring Buffer Procedure

The sole function of the CREATE_RING_BUFFER procedure is to create a ring buffer for the calling task and to return to the task a token for the composite ring buffer object.

Each ring buffer consists of three objects: a segment and two semaphores. The supporting data structure, required by the iRMX OS for calls to **create_composite** and **inspect_composite**, has five fields:

- The number of slots available for tokens in this list of component object tokens. Because ring buffers are composed of three objects and no components will be added, the number of slots is set to three.
- The number of component objects actually in the composite object. In this case, the number of components is three.
- A token for a segment. The segment contains the ring buffer. The first WORD in the segment contains the size of the actual ring buffer. The second WORD of the segment is a POINTER to the most recently entered byte in the buffer. The third WORD points to the oldest byte in the buffer. The rest of the segment is used as the buffer itself. In the program, a structure reflecting the intended breakdown of the segment is superimposed on the segment.
- A token for a semaphore. This semaphore is used to keep track of the number of vacancies in the ring buffer. Thus, it is initialized to the size of the buffer.
- A token for a semaphore. This semaphore is used to keep track of the number of occupied bytes in the ring buffer. Thus, it is initialized to 0.

The CREATE_RING_BUFFER routine creates the components of the composite ring buffer object, initializes the appropriate fields, then creates the composite object, as follows:

```
$INCLUDE(:RMX:INC/COMMON.LIT)    /* Declares common literals */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE ring$buffer$type   TOKEN EXTERNAL;

CREATE_RING_BUFFER: PROCEDURE (size) TOKEN PUBLIC REENTRANT;
       DECLARE size              WORD;
       DECLARE seg$ptr           POINTER;
       DECLARE ptr$struc         POINTER$STRUC AT (@seg$ptr);
       DECLARE astr              ASTR$STRUC;
       DECLARE segment           SEGMENT$STRUC BASED seg$ptr;
       DECLARE exception         WORD;
       DECLARE ring$buffer       TOKEN;
       DECLARE priority          LITERALLY '1';

       astr.num$slots = 3;
       astr.num$components = 3;
       astr.seg = RQ$CREATE$SEGMENT (
              size+6,
              @exception);
       astr.empty$ct = RQ$CREATE$SEMAPHORE (
              size,
              size,
              priority,
              @exception);
       astr.full$ct = RQ$CREATE$SEMAPHORE (
              0,
              size,
              priority,
              @exception);

       ptr$struc.base = astr.seg;
       ptr$struc.off_set = 0;
       segment.size = size;
       segment.head = -1;
       segment.tail = 0;
```

```
        ring$buffer = RQ$CREATE$COMPOSITE (
              ring$buffer$type,
              @astr,
              @exception);
        RETURN ring$buffer;
END CREATE_RING_BUFFER;
```

The `segment.head` variable is set to -1 because the PUT_BYTE procedure (shown later) advances this pointer before placing a character in the buffer.

The Delete Ring Buffer Procedure

DELETE_RING_BUFFER, which can be called by any task, deletes a ring buffer.

```
$INCLUDE(:RMX:INC/COMMON.LIT)     /* Declares common literals
                                     */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

       DECLARE ring$buffer$type   TOKEN EXTERNAL;

       DELETE_RING_BUFFER:   PROCEDURE(ring$buffer$token)
                                REENTRANT PUBLIC;
             DECLARE ring$buffer$token  BASED  TOKEN;
             DECLARE astr               ASTR$STRUC;
             DECLARE exception          WORD;

             astr.num$slots = 3;
             CALL RQ$INSPECT$COMPOSITE (
                    ring$buffer$type,
                    ring$buffer$token,
                    @astr, @exception);
             CALL RQ$DELETE$COMPOSITE (
                    ring$buffer$type,
                    ring$buffer$token,
                    @exception);
             CALL RQ$DELETE$SEGMENT (
                    astr.seg,
                    @exception);
             CALL RQ$DELETE$SEMAPHORE (
                    astr.empty$ct,
                    @exception);
             CALL RQ$DELETE$SEMAPHORE (
                    astr.full$ct,
                    @exception);
       END DELETE_RING_BUFFER;
```

The Put Byte Procedure

PUT_BYTE places a character in the buffer by advancing the pointer to the front of the buffer then placing the character in the byte being pointed to.

```
$INCLUDE(:RMX:INC/COMMON.LIT)     /* Declares common literals
                                     */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

      DECLARE ring$buffer$type   TOKEN EXTERNAL;
      DECLARE ring$buffer$region TOKEN EXTERNAL;

      PUT_BYTE: PROCEDURE(char, ring$buffer$token)
                                 REENTRANT PUBLIC;
            DECLARE ring$buffer$token  TOKEN;
            DECLARE char         BYTE;
            DECLARE size         WORD;
            DECLARE seg$ptr      POINTER;
            DECLARE ptr$struc    POINTER$STRUC AT (@seg$ptr);
            DECLARE astr         ASTR$STRUC;
            DECLARE segment      SEGMENT$STRUC BASED seg$ptr;
            DECLARE exception    WORD;
            DECLARE units$left   WORD;

            astr.num$slots = 3;
            CALL RQ$INSPECT$COMPOSITE (
                  ring$buffer$type,
                  params.ring$buffer$token,
                  @astr,
                  @exception);
            units$left = RQ$RECEIVE$UNITS (
                  astr.empty$ct,
                  1,
                  indefinitely,
                  @exception);
            CALL RQ$RECEIVE$CONTROL (
                  ring$buffer$region,
                  @exception);
            ptr$struc.base = astr.seg;
            ptr$struc.off_set = 0;
            segment.head = ((segment.head + 1) MOD
                                 segment.size);
            segment.buffer(segment.head) = params.char;
```

```
            CALL RQ$SEND$CONTROL (
                  @exception);
            CALL RQ$SEND$UNITS (
                  astr.full$ct,
                  1,
                  @exception);
      END PUT_BYTE;
```

This procedure enters a region after obtaining the desired unit. To reverse these steps would create a deadlock situation, particularly if the same reversal occurs in the GET_BYTE routine.

The Get Byte Procedure

GET_BYTE removes the oldest byte in the buffer, then advances the `segment.tail` pointer.

```
$INCLUDE(:RMX:INC/COMMON.LIT)    /* Declares common literals
                                    */
$INCLUDE(:RMX:INC/NUCLUS.EXT)

DECLARE ring$buffer$type   TOKEN EXTERNAL;
DECLARE ring$buffer$region TOKEN EXTERNAL;

GET_BYTE: PROCEDURE(ring$buffer$token) BYTE PUBLIC REENTRANT;
       DECLARE ring$buffer$token  TOKEN;
       DECLARE size        WORD;
       DECLARE seg$ptr     POINTER;
       DECLARE ptr$struc   POINTER$STRUC AT (@seg$ptr);
       DECLARE astr        ASTR$STRUC;
       DECLARE segment     SEGMENT$STRUC BASED seg$ptr;
       DECLARE exception   WORD;
       DECLARE char        BYTE;
       DECLARE units$left  WORD;

       astr.num$slots = 3;
       CALL RQ$INSPECT$COMPOSITE (
              ring$buffer$type,
              ring$buffer$token,
              @astr
              @exception);
       units$left = RQ$RECEIVE$UNITS (
              astr.full$ct,
              1,
              indefinitely,
              @exception);
       CALL RQ$RECEIVE$CONTROL (
              ring$buffer$region,
              @exception);
       ptr$struc.base = astr.seg;
       ptr$struc.off_set = 0;
       char = segment.buffer(segment.tail);
       segment.tail = ((segment.tail + 1) MOD segment.size);
```

```
        CALL RQ$SEND$CONTROL (
              @exception);
        CALL RQ$SEND$UNITS (
              astr.e,pty$ct,
              1,
              @exception);
        RETURN char;
END GET_BYTE;
```

Epilogue

Any task in any job linked to these procedures may call any one of the procedures. The procedure names to be used in such calls are `CREATE_RB`, `DELETE_RB`, `GET_RB_BYTE`, and `PUT_RB_BYTE`. Application programs cannot manipulate either ring buffers or their component objects, except through these system calls. In fact, application programmers need not be aware that ring buffers are composed of several other objects. To them, ring buffers appear (except for the absence of "RQ" in the procedure names) to be standard iRMX objects.

□□□

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W