Introducing the iRMX® Operating Systems

RadiSys Corporation
5445 NE Dawson Creek Drive
Hillsboro, OR   97124
(503) 615-1100
FAX: (503) 615-1150
www.radisys.com
07-00634-01
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:

✎ **Note**
Notes indicate important information.

⚠ **CAUTION**
Cautions indicate situations which may damage hardware or data.

Contents

1 Basic Concepts

2 Nucleus and Kernel Features

3 BIOS and EIOS Features

Figures

Basic Concepts 1

i Introducing…

R Real-time operating systems that support applications which require deterministic response time, high reliability, and synchronization, with

M Multitasking capabilities that handle multiple external interrupts and internal events occurring asynchronously. They provide multiuser and multiprogramming support, and the

X eXecutive scheduler provides preemptive, priority-based, and predictable response time to interrupts and events.

Real-time applications can be:

Measurement and monitoring	of seismic activity for analysis of patients in medical procedures
Process control	of high-speed machinery or robots of research and lab activities
Data acquisition and distribution	of automatic test data to vendors or clients over networks or satellite links

The iRMX® Operating Systems run on single-board computers and microcomputers built around the Intel386 , Intel486 , Pentium® and Pentium Pro microprocessors. There are three versions of the iRMX Operating System (OS).

iRMX for PCs — Runs on PC platforms with or without DOS, but with no interoperability between the OSs. It can use the PC's ROM BIOS to interact with peripherals or it can directly interact with these peripherals,

DOSRMX — Runs on PC platforms concurrently with DOS; it provides interoperability between the iRMX OS andDOS. Provides access to off-the-shelf DOS software and tools.

iRMX III OS — Runs on Multibus I and II platforms. Provides maximum configurability, including booting the OS with an embedded application.

iRMX Operating System Features

The iRMX OS offers a broad range of real-time, object-based functions and features:

- Real-time processing to monitor and control events
  - Multitasking
  - Preemptive priority-based scheduling
  - Interrupt processing
  - Predictable response time (determinism)
  - Multiprogramming
- Objects to simplify application design and programming and to control resources
  - Intertask coordination and communication
  - Shared memory and dynamic memory allocation
  - System calls that manipulate objects and control the computer
  - Configurable layers of the OS, each with its own system calls
- Example code and demonstration programs that illustrate how to use iRMX objects and system calls
- Network support for file access between iRMX and other OSs
- 32-bit architecture
  - 4 Gbyte physical addressing and 4 Gbyte segments
  - Protection features for improved reliability and multiuser support
- Industry-standard bus support

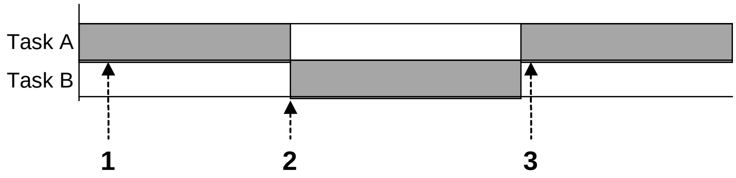
Real-time Processing

Real-time processing requires these capabilities:

- Multitasking, which means switching between threads of execution or tasks
- Preemptive, priority-based scheduling, to determine which task needs to run immediately and which can wait
- Interrupt processing, to respond to external interrupts that occur during processing
- Predictable response time (determinism) so tasks execute before their deadlines expire
- Multiprogramming, so more than one application can run at a time

Multitasking

Multitasking means the computer stops doing one task and starts doing another, as shown in this figure. An iRMX *task* is a thread of execution, similar to a UNIX process.



1. The processor executes Task A.
2. An event happens and a task switch occurs. The processor then executes Task B.
3. When Task B finishes, Task A becomes the running task again.

Figure 1-1. Task Switching in a Multitasking Environment

The iRMX OS manages task switching; it saves the old task's *state* (context) on the old task's stack and loads the new task's state before starting execution.

Multitasking and modular programming go hand-in-hand. You start by breaking down a large, difficult application problem into successively smaller and simpler problems, grouping similar problems where you can. Finally, you solve the small problems in separate program modules. In the iRMX multitasking environment, each module is a task.

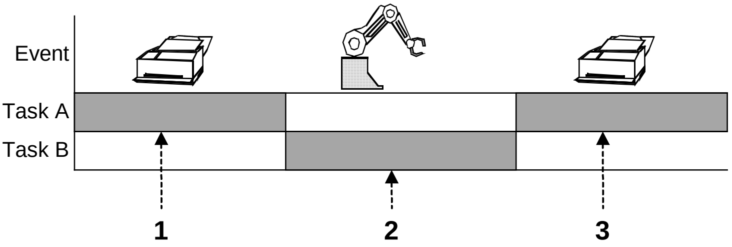
Multitasking simplifies building an application. When you need a new function, you just add a new task.

See also: Tasks, Chapter 2 in this manual;
tasks, *System Concepts*

When you combine multitasking with preemptive, priority-based scheduling, your application can switch from relatively unimportant tasks, to important tasks, to critical tasks and back again, as appropriate.

Preemptive, Priority-based Scheduling

In a *preemptive, priority-based* system, some tasks are more critical than others. Critical tasks run first and can preempt less critical tasks, as shown in this figure.



1. Task A, a low-priority task, prints data accumulated from a robotic arm in report form.
2. Task B, a high-priority task, controls the robotic arm. If the arm needs to move while Task A runs, Task B preempts the print task, then starts and moves the arm.
3. After Task B repositions the arm, Task A finishes printing.

Figure 1-2. Multitasking and Preemptive, Priority-based Scheduling

Multitasking enables your application to respond to *internal events* and *external interrupts*, such as clock ticks from the system clock or receiver ready from a serial device, based on how critical they are. You determine the priority of tasks in your application; the iRMX OS provides the task scheduling algorithms.

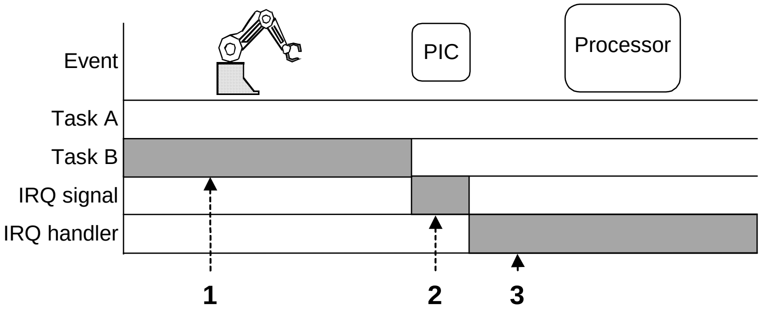
See also: Scheduling tasks, Chapter 2 in this manual;
scheduling, *System Concepts*

When you add interrupt processing to multitasking and preemptive, priority-based scheduling, your application can respond to interrupts as they occur. Your application becomes *event-driven*; it does not waste time polling for interrupts.

Interrupt Processing

Interrupts are signals from devices such as a malfunctioning robot or interactive terminal. You connect interrupt sources to the processor through Programmable Interrupt Controllers (PICs).

With interrupt processing, your application can handle interrupts occurring at random times (*asynchronously*) and can handle multiple interrupts without losing track of the running task, or those tasks waiting to run. Interrupts can occur while the processor is executing either an unrelated task or a related task, as shown in the next figure.



1. Task B, the running task, repositions the robotic arm.
2. The robotic arm malfunctions and sends an interrupt signal through the PIC.
3. As soon as it receives the signal, the microprocessor stops the running task and starts an interrupt handler. The interrupt handler runs in the context of Task B. No new task is loaded; Task B's state does not need to be saved. It remains loaded in RAM until the scheduler runs it again. Task A, the print task, is still waiting to run.

Figure 1-3. Interrupt Handler Interrupting a Task

Typically, you have numerous sources of interrupts in your application. Some of them, like the malfunctioning robotic arm, are critical; some of them are not. You assign *interrupt levels* (which map directly to priorities) to the interrupt sources by the order in which you connect your external sources to the PIC. The iRMX OS handles more critical interrupts first, and keeps track of which interrupts occurred, the order in which they occurred and which ones have not been handled.

Interrupt handlers can perform very limited operations, so you typically write an interrupt handler to signal an *interrupt task*. The interrupt task's priority is automatically assigned, based on the interrupt level of the external source.

Multitasking and interrupt processing simplify expanding an application. Because of the one-to-one relationship between interrupts and tasks, you just add a new task when you need to respond to a new interrupt. Interrupt processing is also more efficient, since your system spends all of its time running tasks, not polling for interrupts.

The Kernel offers a different model for interrupt handling.

See also: Nucleus interrupt and exception handling, Chapter 2 in this manual;
Kernel features, Chapter 2 in this manual;
interrupts, *System Concepts*;
interrupt_task, *Driver Programming Concepts*

Predictable Response Time

The iRMX OS provides *deterministic* response: there is a predictable, worst-case response time to a high-priority interrupt. The OS does this in two ways: *interrupt response* time and *task switch* time.

Interrupt Response Time. This is the time between a physical interrupt happening and the system beginning to execute the interrupt handler. A predictable worst-case response time to interrupt processing ensures that incoming data is handled before it becomes invalid.

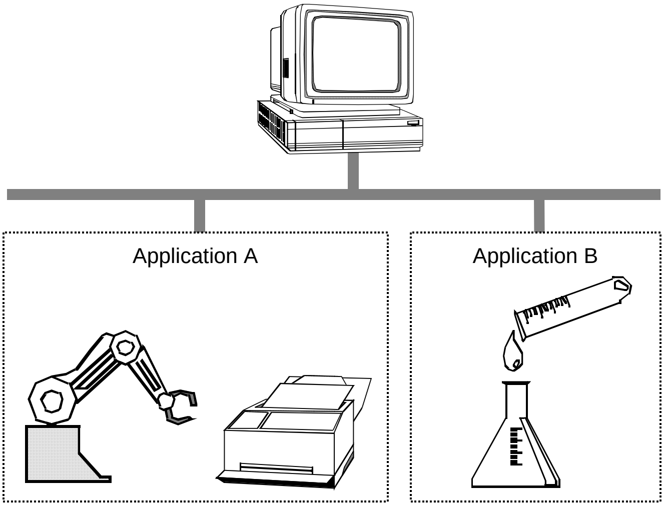
Task Switch Time. A task switch is saving the data registers, stack and execution pointers (the *task state*) of one task, and starting another task. Minimized task switch time also provides a predictable response time to a high-priority task.

Since the typical response to an interrupt is invoking a handler and then performing a *task switch* to an interrupt task, the deterministic response time includes both the interrupt response and task switch times.

Real-time response does not mean instantaneous execution. A high-priority task that is very long and performs many calculations will use as much processor time to execute on a real-time system as on any other system. The length of time instructions take to execute is a function of processor speed.

Multiprogramming

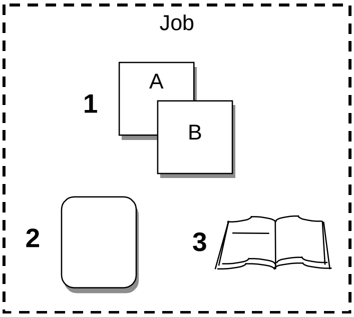
The iRMX OS supports *multiprogramming*: running several unrelated applications on a single system, as shown in the next figure.



1. Application A contains all the tasks that relate to the robotic arm, including the print task. It may also contain tasks that control other devices on the factory floor.
2. Application B contains all the tasks relating to another application that controls a chemical mixing system in another part of the factory.

Figure 1-4. Multiprogramming

To take full advantage of multiprogramming, you provide each application with a separate environment: separate memory, files, and other resources. The iRMX OS provides this kind of isolation in a *job*. Typically, a job includes a group of related tasks and the resources they need, as shown in the next figure.



Typically a job includes:

1. A group of related tasks.
2. The memory the tasks need.
2. An object directory where you can catalog task resources.

Figure 1-5. Resources in a Job

You decide what jobs to include in your system. The iRMX OS coordinates the use of resources within and between jobs so independently-developed applications do not cause problems for each other.

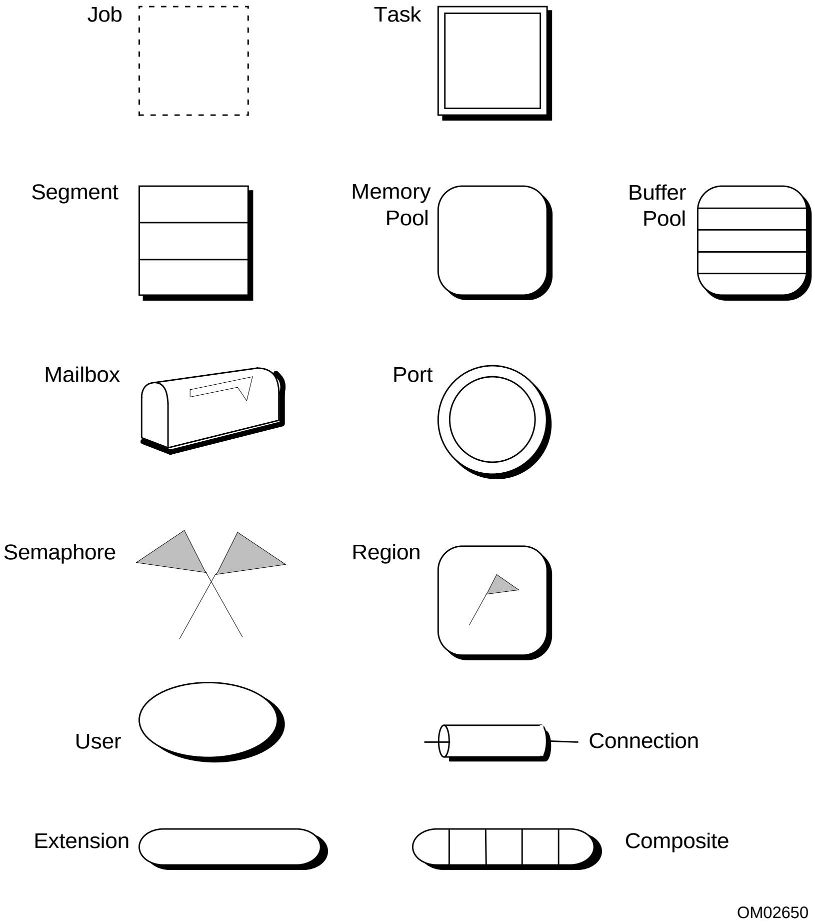
See also: Jobs, Chapter 2 in this manual;
jobs, *System Concepts*

Multiprogramming simplifies adding new applications; you can modify your system by adding new jobs (or removing old ones) without affecting other jobs. In addition, you get more from your hardware investment by running several applications on it.

Objects and System Calls

The iRMX OS provides basic objects and maintains the data structures that define these objects. The basic objects are building blocks that application programs manipulate. The characteristics of the objects are easy to learn and use; they are well-defined and consistent.

This figure shows the iRMX-provided object types.



The icons for the object types shown in this illustration appear in all illustrations in this manual.

Figure 1-6. Object Types

Each object type, such as a mailbox, has a specific set of *attributes* or characteristics. Once you learn the attributes of a mailbox, you know how to use all mailboxes. Each object also has its own set of related system calls.

See also: Object types, Kernel objects, Chapter 2 in this manual;
I/O system objects, Chapter 3 in this manual;
Individual object chapters, *System Concepts*

Object-based programming, which concentrates on the objects and the operations performed on them, is compatible with modular programming. Typically a single task performs only a few related functions on a few objects.

Intertask Coordination and Communication

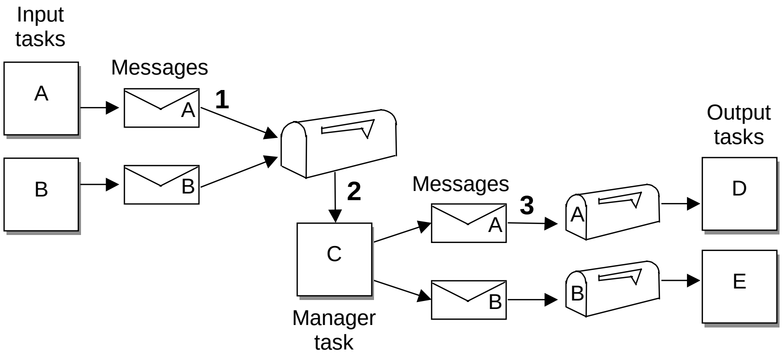
iRMX *exchange* objects are mailboxes, semaphores, regions, and ports. They enable asynchronous tasks, which run in a random order, to coordinate and communicate with one another by:

- Passing messages
- Synchronizing with each other
- Mutually excluding each other from resources

Messages. Tasks may need to exchange data, messages, or *object tokens* (object addresses and attributes).

For instance, one task may accumulate input from a terminal until it receives a carriage return. It uses an exchange object to send the entire line of input as data to another task that decodes the input.

This figure summarizes how you can solve a problem that requires routing several types of input into several types of output using a mailbox object. One mailbox and one manager task can handle messages from multiple input and output tasks.

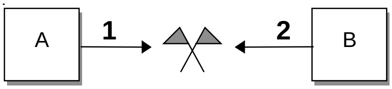


1. System calls move data from input tasks A and B to a waiting mailbox.
2. Task C, the manager task, waits at the mailbox and determines which messages go to which output tasks. If another message arrives during processing, the message waits in the mailbox queue until the manager task can handle it.
3. The individual output tasks receive their data at their mailboxes and process it.

Figure 1-7. Object-based Solution for Message Passing

Synchronization. One task may need to run before another task. It can use an exchange object to signal the second task when it has completed. For example, the task that creates the transaction summary in an automated teller application shouldn't run until after the tasks that handle withdrawals and deposits have run. The transaction summary task must be synchronized with the other tasks.

The iRMX OS provides several objects for synchronization that can accommodate a wide variety of situations. The next figure illustrates using a semaphore to send a signal to another task.



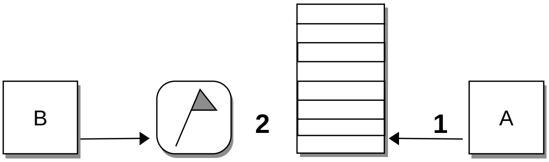
Task A, the running task, preprocesses some data. Task B needs to use the data after Task A finishes.

1. WhenTask A finishes, it sends a signal (not data) to the semaphore.
2. When Task B receives the signal, it processes the data.

Figure 1-8. Tasks Using a Semaphore for Synchronization

Mutual Exclusion. With the iRMX OS, multiple tasks can concurrently access the same file. This is useful in a multitasking system, such as a transaction processing system where a large number of operators concurrently manipulate a common database.

If each terminal is driven by a distinct task, the only way to implement an efficient transaction system is to have the tasks share access to the database file. Occasionally, when tasks are running concurrently, the situation in this figure arises.



1. Task A, the running task, reads some data from the database and does computations based on the data.
2. Task B tries to preempt Task A and update the data while Task A works on it. Mutual exclusion, provided by a region, prevents two tasks from accessing the same data concurrently.

Figure 1-9. Multitasking and Mutual Exclusion

The iRMX OS includes regions to provide mutual exclusion. Unless Task B is prevented from modifying the data until after Task A has finished, Task A may unknowingly use some old data and some new data to produce an invalid computation. It should, however, read and compute the new data after Task B updates it. Your code can use a region to protect the data from being accessed by both tasks at the same time. This is called mutual exclusion.

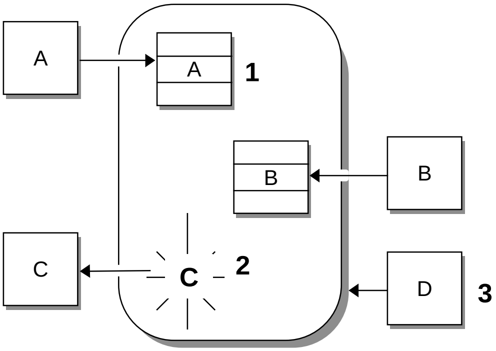
Memory Pools and Memory Sharing

Memory pools are the basis of iRMX memory management. The *initial* memory pool is all the memory available to the application (that is, *free space memory*). It is managed by the OS and allocated to the application on request.

The initial memory pool is subdivided into smaller pools as application jobs and segments are created. Each job in the application has its own subpool, with a minimum and a maximum size. Once the minimum pool is allocated to a job, it is not available to other jobs. As tasks in the job create and delete objects, the job pool may approach its maximum size, then return to its minimum. This provides dynamic memory allocation of the memory pool as jobs require less memory.

Dynamic memory allocation enables jobs to take advantage of one job freeing memory while another needs more. When you delete a job, its memory returns to the initial memory pool.

Dynamic memory allocation is also useful within a job. All the tasks in a job compete for memory in the job's memory pool. Some tasks periodically need lots of memory to improve efficiency, for example a task that allocates large buffers to speed up input and output operations. These tasks can release memory for other tasks when they complete, as shown in this figure.



1. Tasks A and B use memory in the job's memory pool for objects they create.
2. Task C completes, and then deletes its objects and releases its memory to the job's memory pool.
3. Task D requests memory.

Figure 1-10. Dynamic Memory Allocation between Tasks

Virtual Memory Allocation With the Paging Subsystem

If you use a non-Intel compiler that produces flat model applications instead of segmented applications, you include the Paging Subsystem in the OS. This job manages the processor in paging mode, which treats memory differently from the segmented model used historically by iRMX applications. With paging enabled, flat model applications allocate *virtual memory*, which is indirectly accessed through page tables instead of in separate segments.

Flat model applications use a different set of system calls to allocate memory than segmented applications. However, the job still manages a pool of memory, which the individual tasks can use as described above. With the Paging Subsystem, you can still run existing segmented applications (or write new ones) along with flat model applications. Flat and segmented applications must reside in different jobs.

See also: Paging, Flat model, *Programming Techniques*
paging.job, *flat.job*, *System Configuration and Administration*

System Calls

Each iRMX object has an associated set of *system calls*: programmatic interfaces you use to manipulate objects or control the computer's actions. System calls for one object type cannot manipulate objects of another type. This protects the objects from inappropriate modification.

Most system calls have parameters, such as values and names, that you can set to tailor the call's performance. Some of the functions you can do with system calls are:

create_mailbox	Creates a new mailbox
set_priority	Sets a task's priority
send_message	Sends a token or data to a mailbox
a_read	Reads a data file or the contents of a directory

Most system calls validate the parameters you enter; a *condition code* returned by the call indicates whether you used invalid parameters. There are also condition codes for trying to read or write segments that you do not have access to and for trying to write past the end of a segment.

You can add new objects and the system calls that manipulate them to the iRMX OS if your application requires them. Objects and system calls you create can be shared by jobs.

See also: System call summary tables in Chapter 1 of *System Call Reference*;

Operating System Layers

The iRMX OS is a collection of layers, or subsystems. All layers have their own system calls. Some layers have their own objects; some use the objects from other layers.

Each layer builds on the capabilities of the previous ones. These layers are all included in iRMX for PCs and DOSRMX. The Bootstrap Loader is included with the iRMX III OS and iRMX for PCs.

On ICU-configurable systems, you choose which layers you need to create a tailored OS. These are the layers and their functions:

iRMK Kernel
A part of the Nucleus that provides high-performance task and time management and message passing. The Kernel does not provide the protection that the Nucleus does. The Kernel has its own objects.

Nucleus
The heart of the OS, built on the Kernel, and is the only required layer. It provides most of the objects that the other layers use. It schedules jobs and tasks, controls access to system resources, communicates between tasks and enables the system to respond to interrupts.

Basic I/O System (BIOS)
Includes file and device drivers and data tables that define I/O devices. This layer performs I/O functions; creates and deletes files, and controls file access. The BIOS controls the Connection and User objects.

Extended I/O System (EIOS)
A higher-level I/O system that provides all the BIOS features plus I/O buffering and overlapping, logical names for files and devices, and automatic reattachment of devices. The EIOS controls the I/O job object.

Application Loader (AL)
Loads programs from secondary storage into memory.

Human Interface (HI)
Provides multiuser support with logon/logoff and user job creation. It is the parent job for applications in DOSRMX and iRMX for PCs. It provides commands for controlling application systems from terminals. You can create additional commands for your application.

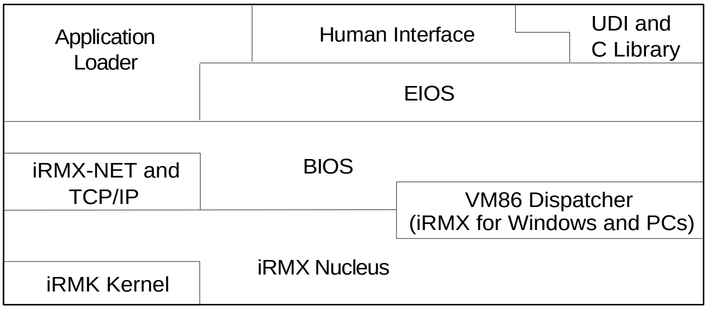
Universal Development Interface (UDI)
A high-level interface between the OS and the application. It enables compilers, language translators, and other software development tools to run.

Shared C Library — Provides functions and macros that enable ANSI-standard C programs to perform common operations concurrently. The Shared C Library provides support for some non-Intel C compilers.

Paging Subsystem — Supports flat model (non-segmented) applications by providing virtual memory in a flat address space; it manages the processor in paging mode.

Bootstrap Loader — Loads the initial OS and the application's initial jobs from secondary storage into RAM on the hosts in the system, then starts the system.

This figure illustrates the layers and the relationships between them. The UDI also accesses the Application Loader.



iRMX is a registered trademark of Intel Corporation.

OM02727

Figure 1-11. The Layers of the iRMX Operating Systems

Example Code and Demonstration Programs

The /rmx386/*demo* subdirectory in the iRMX directory structure contains many programming examples in C, PL/M and assembler. You should use and modify the example code when you develop your applications.

There are examples of:

- Basic multitasking, creating and cataloging objects
- Message passing
- Interrupt handling
- Managing terminal I/O
- Programmatic network access

- Third-party compiler support

See also: *Programming Techniques* for an explanation of some example programs;
Real-Time and Systems Programming for PCs, for additional programming examples

Network Support

The iRMX OS software supports networking of independent computers connected together to exchange information. iRMX-NET enables iRMX-based computers to communicate with OpenNET servers and clients running other OSs, as shown in this figure.

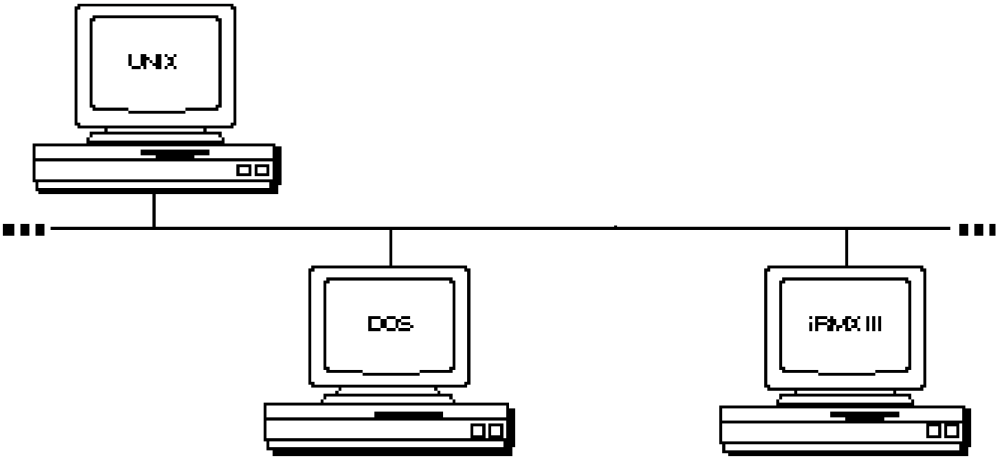


Figure 1-12. Networked Systems

The basic networking software is iRMX-NET, a part of Intel's family of OpenNET Local Area Network (LAN) products and TCP/IP, which enables interoperability with most other OSs.

See also: Networking, Chapter 7 in this manual;
Network User's Guide and Reference;
TCP/IP and NFS for the iRMX Operating System

32-Bit Architecture, Addressing, and Protection

The iRMX OS supports 4 gigabytes (Gbyte) physical addressing, 32-bit addresses and data types, and the addressing modes shown in this figure.

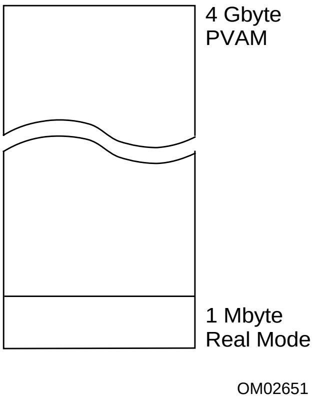


Figure 1-13. Real Mode and PVAM

Real mode	Enables addressing 1 Mbyte of physical memory. Real mode is the initial operating or bootload mode for supported Intel microprocessors. During iRMX initialization, the microprocessor is switched into protected virtual address mode (PVAM).
Protected virtual address mode (PVAM)	Enables the iRMX OS to take advantage of 32-bit architecture, the full 4 Gbytes of memory, and advanced protection features. It also enables real mode programs to run concurrently in Virtual 86 (VM86) mode. DOSRMX runs in protected mode, but switches to VM86 mode to run DOS and DOS applications.
Paging mode	A flat model application uses near pointers only, in a virtual segment that can span the entire 4 Gbyte address range.

The protection features of the microprocessor includes:

- Memory segment length protection that prevents segment accesses from crossing segment boundaries. Writing too much data will not corrupt the system.
- Access-rights protection that enables tasks to define access to a segment and prevents access to those segments in other than the defined mode.
- When the Paging Subsystem is enabled, the processor manages memory through page tables. In this mode, the OS provides protection by isolating memory between jobs with an unallocated page table entry. An attempt to access beyond the allocated memory results in a processor-generated page fault.
- Stack-overflow detection that prevents out-of-control tasks from overflowing the stack and overwriting important information.
- Invalid-selector detection that prevents tasks from referring to segments of undefined memory.

See also: *Programmer's Reference Manual* for your microprocessor for information on the internal hardware features of the microprocessor

Bus Architecture Support

The iRMX OS supports these industry-standard bus architectures: PC, Multibus I, and Multibus II.

PC architecture benefits and features for iRMX for PCs and DOSRMX systems include:

- A 32-bit wide (EISA or PCI) or 16-bit wide (ISA) data word across the bus
- A widely-installed base
- A low-cost platform for iRMX development and target systems

Multibus I architecture benefits and features include:

- A 16-bit wide data word across the bus
- A widely-installed base
- Intelligent board-to-board communications using hardware signals and similar signal handling techniques

Multibus II architecture benefits and features include:

- A 32-bit wide data word across the bus.
- Enhanced board-to-board communication through additional internal buses and a well-defined data transfer protocol. The internal buses enable virtual interrupt

processing so any board can communicate with any other, regardless of hardware limitations.

- Efficient bus use through a *data packet* (small portion of data) transfer scheme. Transferring data by packets prevents a slower device on the bus from monopolizing all the bus time: the bus is not limited in speed to the slowest device using it.
- Support of multi-processor systems for higher performance.

iRMX for PCs Special Features

iRMX for PCs can reside with DOS on a DOS partition, or it can reside on an iRMX partition.

iRMX for PCs provides these features:

- Can use ROM BIOS functions to control the system mass storage devices (diskettes, hard disks, and CD ROMs)
- Can use native iRMX drivers to control the system mass storage devices (diskettes, hard disks, and CD ROMs)
- Contains a native DOS file driver and can read/write any local DOS (FAT12/FAT16) drive (diskette, primary DOS partition, Extended DOS partition) without the presence of DOS
- Is a cost-effective solution for many applications.

See also: *Installation and Startup*

❑❑❑

Nucleus and Kernel Features 2

Nucleus functions include:

- Managing objects that control access to system resources and communicate between tasks
- Scheduling tasks based on priority
- Handling interrupts based on interrupt level

Kernel functions include:

- Controlling Kernel objects
- Managing tasks
- Managing time

Nucleus Objects

The Nucleus provides these objects.

- *Jobs* are the environments where tasks do their work. A *job* contains:
  - — Tasks
  - — Whatever objects the tasks create
  - — An *object directory* for cataloging objects
  - — A *memory pool* that provides memory for the tasks to do their work
- *Tasks* do the work of the system and respond to interrupts and events.
- *Memory segments* are addressable, contiguous blocks of memory that tasks can use for any purpose.
- *Buffer pools and Heaps* are holding areas for dynamically allocatable memory.
- *Exchange objects* that tasks use to pass information are:
  - — *Mailboxes* that enable tasks to pass messages and data
  - — *Semaphores* that enable tasks to synchronize
  - — *Regions* that provide mutual exclusion
  - — *Ports* that synchronize operations and pass messages
- *Extension objects* let you build new types of objects.
- *Composite objects* include the extension and whatever existing objects you need.

Objects are data structures that occupy memory. Each object type has unique attributes, described in the next sections. Each object has a *token* that uniquely identifies it. The maximum number of objects allowed in the system at any one time is 8192 (2000H).

See also: Object directories, tokens and descriptors later in this section;
I/O system objects, Chapter 3 in this manual;
Individual object chapters and OS extensions, *System Concepts*

Jobs

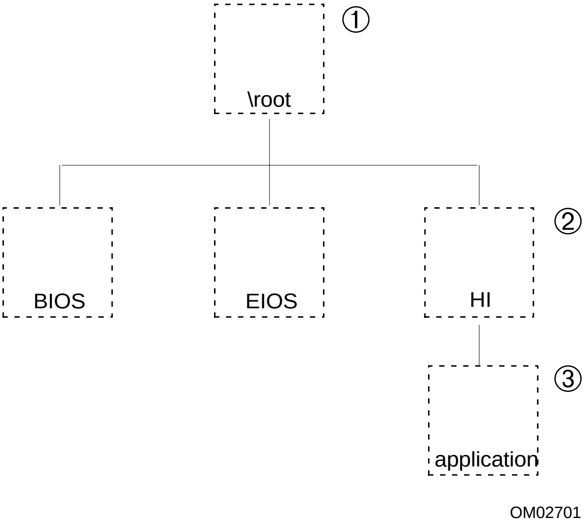
A job is an iRMX object that contains tasks and all the resources they need. iRMX jobs that you can create have these characteristics:

- Jobs make up your application.
- Jobs are passive; they cannot make system calls.
- Jobs include one or more tasks.
- Jobs provide resource isolation for their tasks, particularly for dynamically allocated memory. Two tasks of one job compete for the memory associated with their job. Tasks in different jobs typically do not.
- Jobs provide error boundaries. Errors within one job do not corrupt other jobs or the OS because they have separate memory pools.
- When you delete jobs, the objects associated with them also are deleted.

You can create several kinds of jobs:

- *Dependent* or *child* jobs are children of their parent job.
- *Loadable* jobs are children of the HI.
- *First level* jobs (configurable in the ICU) are children of the root job.
- *I/O jobs* are children of the EIOS.

The jobs in a system form a job tree. Each job except the root job obtains its resources from its parent as shown in the next figure.



1. Initially, the root job owns all of free space memory, which is allocated as other jobs are created.
2. Some of the OS layers are first level jobs, or children of the root job. Applications can also be first level jobs.
3. Applications loaded at system initialization are children of the HI. In DOSRMX and iRMX for PCs, this is how you typically load applications.

   Tasks in jobs can create other jobs, enlarging the tree.

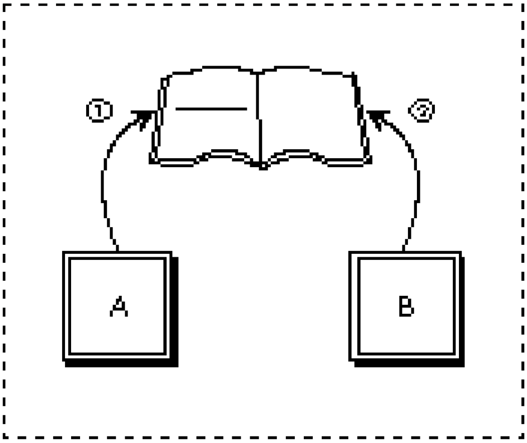
Figure 2-1. Jobs in a Job Tree

There are system calls to create and delete jobs, obtain information about a job's children, and control a job's memory pool.

See also: Jobs, *System Concepts*;
Jobs in Nucleus system call summary, Chapter 1 of *System Call Reference*

Object Directories

Each job has an associated *object directory*. As a task creates an object, the Nucleus creates a token for it. A task can catalog an object, with its token and a corresponding name, in its own job or any other job it knows about. Typically, you catalog objects in the root directory if you want them accessible from several other tasks. Other tasks that know the name can use the object directory to look up and access the object, as shown in this figure.



1. Task A catalogs an object in its own job's object directory.
2. Task B looks up the object, such as a mailbox, in the object directory in order to use it. Now, Task A can send a message to the mailbox and Task B can receive it.

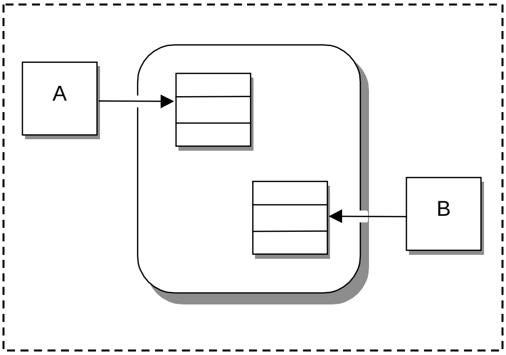
Figure 2-2. Tasks Using the Job's Object Directory

Objects that are cataloged can also be shared across job boundaries.

See also: Object directories, tokens and descriptors in this chapter

Memory Pools

Each job has an associated *memory pool*. This is an amount of memory, with a specified minimum and maximum, that is allocated to the job and its children. The minimum memory is always contiguous. Usually, all memory needed for tasks to create objects in the job comes from the job's memory pool, as shown in this figure.



Tasks A and B obtain memory from the job's memory pool.

Figure 2-3. Tasks Using Their Job's Memory Pool

If there is not enough contiguous memory currently available (up to the maximum size of the job's memory pool), the OS tries to borrow from the job's parent, and on up the job tree if necessary. In general, you should allocate enough memory so that jobs do not need to borrow memory or create additional segments, since both of these require additional time.

In the iRMX III OS, you can also statically allocate memory to jobs. But once the memory is allocated, it cannot be freed for other jobs. The total memory requirement of the system is always the sum of the memory requirements of each job. Static memory allocation uses more memory than dynamic allocation, but may be safer.

See also: Memory pools, *System Concepts*;
Segments and memory pools in Nucleus system call summary, Chapter 1 of *System Call Reference*

Tasks

Tasks are the threads of execution or active, code-executing objects in a system.

Tasks typically respond to external interrupts or internal events. An external interrupt can be a keystroke or a system clock tick; an internal event can be the arrival of a message at a mailbox. Tasks have both a priority and an *execution state* (whether the task is running or not).

There are system calls to create tasks and delete tasks, view and manipulate a task's priority, control task readiness, and obtain task tokens.

See also: Nucleus interrupts, events and exceptions, in this chapter;
tasks, *System Concepts*;
tasks in Nucleus system call summary, Chapter 1 of *System Call Reference*

Memory Segments

Segments provide memory for tasks to use for many purposes, including communicating and storing data. A task requests a segment of whatever size it needs. The segment is usually allocated from the memory pool of the task's job, as shown in Figure 2-3. If there is not enough memory available (up to the maximum size of the job's memory pool), the Nucleus tries to borrow from the job's parent, and on up the job tree if necessary.

There are system calls to request a memory segment of a certain size, delete a segment, and find out the size in bytes of a segment.

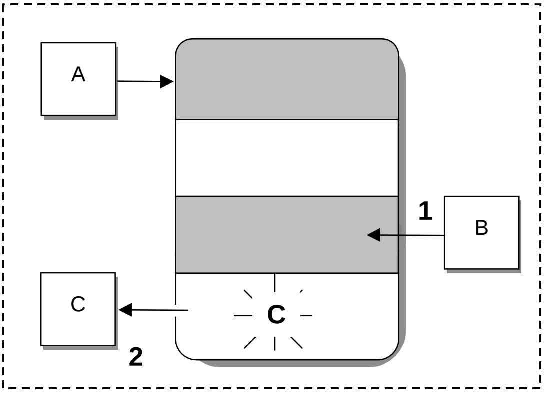
See also: Segments, *System Concepts*;
segments and memory pools in Nucleus system call summary, Chapter 1 of *System Call Reference*

Buffer Pools and Heaps

Buffer pools provide memory for tasks. A buffer pool is a set of existing memory segments that you can dynamically allocate.

Heaps provide memory for tasks. A heap is a single memory segment portion of which you can dynamically allocate as pointers to the beginning of each allocation.

Tasks request buffers, use them, and then release them back to the buffer pool, which manages them, as shown in this figure. Using a buffer pool cuts down on system overhead because allocating existing buffers is faster than creating and deleting memory segments.



1. Tasks A and B are using buffers in the buffer pool.
2. Task C is releasing its buffer. The buffer can be reallocated to another task.

Figure 2-4. Tasks Using a Buffer Pool

There are system calls to create a buffer pool, fill it with buffers, view the attributes of a buffer pool, and delete it.

Tasks request allocations from a Heap, use them, and then release them back to the Heap, which manages them. Using a Heap cuts down on system overhead since there are no segment creations or deletions involved when requesting and/or release Heap allocations.

See also: Buffer pools, *System Concepts*;
Heaps, *System Concepts*;
buffer pools in Nucleus system call summary, Chapter 1 of *System Call,*
Heaps in Nucleus system call summary, Chapter 1 of *System Call Reference*

Exchange Objects

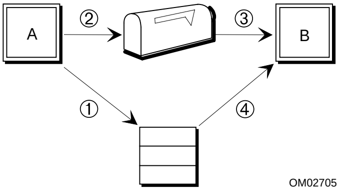
The four Nucleus exchange objects are mailboxes, semaphores, regions, and ports.

Mailboxes

Mailboxes provide intertask communication between tasks in the same job or in different jobs. They can send information and, since a task may have to wait for information before executing, they can synchronize task execution. There are two types of mailboxes:

Message mailboxes	Send and receive object tokens.
Data mailboxes	Send and receive data.

This figure shows how tasks use a message mailbox to send a token for a segment.



1. Task A creates a segment and puts data into the segment.
2. Task A sends the segment token to a mailbox.
3. Task B waits to receive the segment token at the mailbox. You can specify whether or not Task B should wait if the token isn't in the mailbox.
4. Task B obtains the token and receives the data in the segment.

Figure 2-5. Tasks Using a Message Mailbox

Mailboxes have task queues, where tasks wait for messages, and message queues, where messages wait to be given to tasks. The task queue may be FIFO- or priority-based; the message queue is FIFO-based.

You use the same system calls to create and delete message and data mailboxes. However, you use different calls to send and receive messages or data.

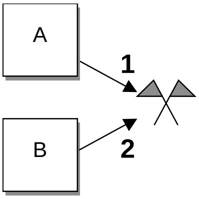
See also: Mailboxes, *System Concepts*;
Mailboxes in Nucleus system call summary, Chapter 1 of *System Call Reference*

Semaphores

Tasks use *semaphores* for synchronization.

A semaphore is a counter that takes positive integer values. Tasks send units to and receive units from the semaphore. When a task sends *n* units to a semaphore, the value of the counter is increased by *n*; when a task receives *n* units from a semaphore, the value of the counter is decreased by *n*.

This illustration shows a typical example of a binary (one-unit) semaphore used for synchronization.



1. Task A needs to do some work before Task B starts running. Task A creates a semaphore with one unit. To enable synchronization, Tasks A and B should request and obtain the unit before running.

   Task A obtains the unit. Because the semaphore has no units, Task B cannot run.
2. When Task A completes, it returns the unit to the semaphore. Task B can now obtain the unit and begin running.

Figure 2-6. Tasks Using a Semaphore for Synchronization

Semaphores enable synchronization; they don't enforce it. If tasks do not request and obtain units from the semaphore before running, synchronization is not achieved. Each task must send a unit back to the semaphore when it is no longer needed. Otherwise, tasks can be permanently prevented from running.

Semaphores can also provide mutual exclusion from data or a resource like this:

1. Task A requests one unit from a binary semaphore, and uses the resource when it receives the unit.
2. Task B requests one unit from the semaphore before using the resource. Task B must wait at the semaphore until Task A returns the unit.

Semaphores enable mutual exclusion; they do not enforce it.

Semaphores have a queue where tasks wait for units. The queue may be FIFO- or priority-based. There are system calls to create and delete semaphores, and to send and receive units.

See also: Semaphores, *System Concepts*;
Semaphores in Nucleus system call summary, Chapter 1 of *System Call Reference*

Regions

A *region* is a binary semaphore with special suspension, deletion, and priority-adjustment features. Regions provide mutual exclusion from resources or data; only one task may control a region at a time; only the task in control of the region can access the resource or data. Once a task gains control of a region, the task cannot be suspended or deleted until it gives up control of the region. When the running task no longer needs access, it exits the region, which enables a waiting task to access the resource or data.

Regions can have a priority queue, which you should use. Then, if a higher-priority task tries to enter a busy region, the priority of the task in the region is raised temporarily so that it equals the waiting task's priority. This helps prevent priority-inversion, as shown in this example:

1. Task A is the running task. It is a low-priority task with control of a region, accessing some data. The region has a priority queue. The only other task that uses the data is Task C, a high-priority task that is not ready to run.
2. Task B, a medium-priority task, becomes ready to run and preempts A.
3. Task C becomes ready to run and preempts B. It runs until it tries to gain control of the region. Task A's priority is raised to equal Task C's priority until Task A releases the region; then its priority returns to its initial level.
4. When Task A releases the region, Task C receives control of the region and uses the data. When Task C completes, Task B runs.

Without a high-priority queue, Task B would have preempted A while A had control of the region; C would have preempted B, but would have been unable to use the data because A had control of the region.

Regions require careful programming to avoid *deadlock*, where two tasks need access to two resources protected by regions at the same time and one task has control of one region while the other task has control of the other region.

Regions have a task queue where tasks wait for control. The queue can be FIFO- or priority-based. There are system calls to create, control, and delete regions.

See also: Regions, *System Concepts*;
Regions in Nucleus system call summary, Chapter 1 of *System Call Reference*

Ports

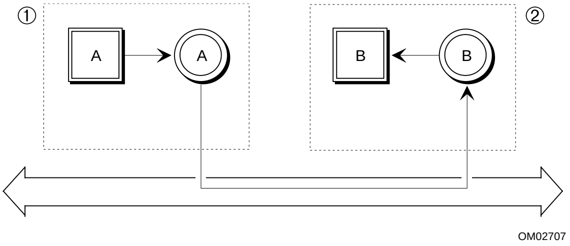
Ports have two uses:

- Message-passing on all platforms
- Signal passing to synchronize Multibus II operations

On all platforms, you can use message ports for short-circuit message passing. Ports enable efficient communication between tasks on the same host, in the same or different jobs: a message is copied from the source buffer in the sending task to the destination buffer in the receiving task. You use buffer pools to provide fast storage allocation for messages received at ports.

On Multibus II systems, you can use ports between host boards in the system, including hosts that are not running an iRMX OS. Each port is an access point into the bus, through which you can send or receive messages or send signals.

This figure shows communication between tasks on different hosts in a Multibus II platform. Tasks on different hosts must each have access to a port. Each port must have a *socket* identifier: a combination of the port and host IDs.



1. Task A is the sending task. It sends the message to a port on its host. The message travels on the bus.
2. Task B is the receiving task. It receives the message at a port on its host.

Figure 2-7. Tasks Using Ports for Communication on a Multibus II Platform

Ports provide more functionality and require more programming effort than mailboxes. There are system calls to create, attach, manipulate, detach, and delete ports. There are several system calls to send, receive, and cancel messages.

See also: Ports, *System Concepts*;
Communication service calls in Nucleus system call summary, Chapter 1 of *System Call Reference*

Extension Objects and Composite Objects

If your system requires an object type that is not supplied by the iRMX OS, you can add a new object type by extending the iRMX OS. A new object type is an extension object. Each new extension object type requires its own OS extension.

Each new extension object type must be manipulated by its own system calls; these calls should not be used for other object types. There can be numerous objects of the given type and they must have the same form and function within the type.

An OS extension that you write for an extension object type is called a type manager. Each extension object type requires its own type manager that must provide system calls to:

- Create objects of the new type
- Manipulate the objects as required
- Delete the objects

You can write OS extensions to add utilities to the OS or to provide some additional functionality. For example, in designing a system to control heat-saving blinds in a greenhouse, you might need a data structure that includes information from a photo-sensitive cell measuring the sunlight falling on the greenhouse, and from a thermostat reading the temperature. The application could use this data to control when to open or close the blinds. This data structure is an extension object. The system calls that read or write its contents are part of the type manager.

There are system calls to create and delete an OS extension.

See also: Type managers, extension objects, operating system extensions, *System Concepts*;
extension objects in Nucleus system call summary, Chapter 1 of *System Call Reference*

Individual objects that you create from the extension object type are *composite* objects. Composite objects are collections of existing objects (ports, buffer pools, etc.); the OS treats the result as a single object.

There are system calls to create, manipulate, and delete composite objects.

See also: Composite objects, call gates *System Concepts*;
Composite objects in Nucleus system call summary, Chapter 1 of *System Call Reference*;
vo (view object) command in *System Debugger Reference* for information on composite objects

Object Directories, Tokens and Descriptors

The Nucleus manages objects using:

- The object directory of each job
- The *token* for each object, a 16-bit selector or handle for the object
- A *descriptor* that defines the physical address and attributes of the object

The OS assigns each object a descriptor when it is created. Descriptors contain an object's attributes, such as its size and access type.

All descriptors reside in descriptor tables used by the processor. There are three types of descriptor tables:

Global Descriptor Table (GDT)
Contains up to 8K of descriptors. Each descriptor contains the physical address used by the system to access an area of memory. Every task in the system uses descriptors in the GDT. There is only one GDT for the entire OS.

Local Descriptor Tables (LDTs)
Are reserved for system use.

Interrupt Descriptor Table (IDT)
Contains the addresses of up to 256 handlers to execute when events occur. Addresses are entered automatically into the IDT when the system is created and you can enter them dynamically using a system call.

Call gates are special descriptors in the Global Descriptor Table. They enable entry into the iRMX OS and OS extensions, and are established when the system is configured. They redirect flow within a task. Each system call, including those you write, uses a call gate to transfer control to the iRMX system call routine requested.

See also: Tokens, descriptor tables, *System Concepts*;
Descriptors in Nucleus system call summary, Chapter 1 of *System Call Reference*;
Reference manual for your microprocessor for more details

Nucleus Task Scheduling

The Nucleus switches between tasks and makes sure the processor is always executing the appropriate task. The Nucleus maintains an *execution state* and a *priority* for each task.

Priority

The priority is an integer value from 0 through 255, with 0 being the highest priority.

Range	Usage
0 - 16	Used by the OS for hardware exceptions. Cannot be masked.
56 - 127	Used by the OS for servicing external interrupts. Creating a task that handles internal events here masks external interrupts numerically higher.
128 - 130	Use for tasks that communicate with interrupt tasks.

131-255 Use for tasks that handle internal events, such as message passing. You can usually start using round-robin scheduling at about 200.

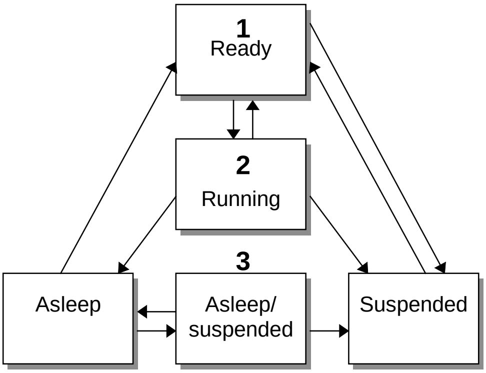
See also: Round-robin scheduling and Nucleus Interrupt and Exception Handling in this chapter

Interrupt tasks mask numerically higher levels. When you assign interrupt levels, give a numerically lower level to interrupts that can't wait, such as serial input, and a higher level to interrupts that can wait, such as cached input.

Execution State

The execution state for each task is, at any given time, either running, ready, asleep, suspended, or asleep-suspended.

Tasks run when they have the highest (numerically lowest) priority of all ready tasks in the system and are ready to run. Tasks can change execution state, as shown in the next figure.



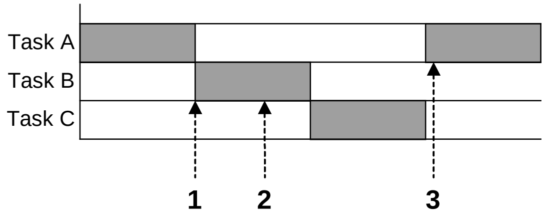
1. Tasks are created in the ready state.
2. The running task, the ready task with the highest priority, does one of these:
   - Runs until preempted by a higher priority task that is ready.
   - Runs until it removes itself from the ready state.
3. A task in any state except ready cannot run, even if it has the highest priority.

Figure 2-8. Execution State Transitions for Tasks

A task can put itself to sleep or suspend itself directly by using system calls for that purpose. A task might indirectly be put to sleep by the Nucleus if it makes a “blocking” call; for example, by waiting at a mailbox until a message arrives. The Nucleus puts the task in the ready state when the message arrives.

Round-robin Scheduling

The iRMX OS also provides *round-robin scheduling*, where equal-priority tasks take turns running. Each task gets a *time slice*, an equal portion of the processor's time. If a task has not finished running when its time slice expires, it goes to the end of a circular queue where it waits until all tasks ahead of it have used up their time slices, as shown in this figure. You adjust the length of the time slice and set the priority level where round-robin scheduling occurs.



Tasks A, B, and C are of equal priority below the round-robin priority threshold.

1. Task A, the running task, stops running when its time slice expires. Task A's state is saved and it moves to the end of the queue.
2. Task B, the ready task, then becomes the running task.
3. Task A runs again when all tasks in the queue finish running expires.

Figure 2-9. Round-robin Scheduling

Of course, a higher priority task will still preempt any running task in the round-robin queue, regardless of the amount of time left in its time slice.

Round-robin scheduling cannot guarantee a predictable worst-case response to events because the number of events in the queue varies.

See also: Task priorities, *System Concepts*

Nucleus Interrupt and Exception Handling

Interrupts and exceptional conditions have special handlers. The Kernel provides additional interrupt handling capability.

See also: Kernel Features in this chapter

Interrupt Handlers

System hardware invokes an *interrupt handler* to respond to an asynchronous interrupt from an external source, based on its entry number in the IDT. The handler takes control immediately and saves the register contents of the running task so it can be restarted later. There are two ways you can service the interrupt:

- Using a handler alone
- Using a handler/task combination

Interrupt Handler Alone

The interrupt handler alone can process only interrupts requiring very little processing and time. Handlers without tasks can do these activities:

- Accumulate data from the device in a buffer. The data must have an associated task to be accessed and used.
- Disable levels. A handler should only disable levels for a very short time and under special circumstances. For example, a device driver procedure may need this call to prevent interrupts when resources needed by the driver are being deleted. Numerically higher levels are disabled anyway.
- Find out what level is currently being serviced. This is useful if one handler services several interrupt levels.
- Send an EOI signal to the hardware.

By itself, an interrupt handler can only do very simple processing, such as sending an output instruction to a hardware port to cause a light to blink, indicating the device is functioning. Handlers can use only a few system calls.

During the time the interrupt handler is executing, all other interrupts are disabled. Since even very high level interrupts are disabled, it is essential that the handler execute quickly and exit.

When the handler finishes servicing the interrupt, it sends an EOI to the PIC, restores the register contents of the interrupted task, and surrenders the processor. The processor returns to the interrupted task.

Interrupt Handler/Task Combination

An interrupt handler/task combination is much more flexible. The handler may do a small amount of processing, but it typically signals its corresponding interrupt task to do most or all of the rest of the interrupt processing. You need to use an interrupt handler/task combination if the processing requires a lot of time or requires system calls that interrupt handlers cannot use.

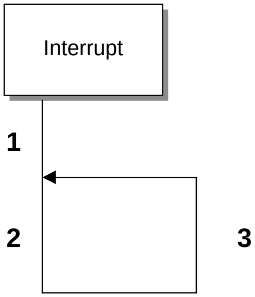
When there is a specified interrupt task, the handler can put the information it accumulates into a segment, if one has been set up by the interrupt task. The interrupt task can access the data in the segment and do whatever is required.

Interrupt tasks have access to the same resources and can use the same system calls as ordinary tasks. The only difference is that interrupt tasks have an interrupt level assigned by the OS, based on the level of the handler. Ordinary tasks have a priority which you assign.

In addition to the usual task activities, an interrupt task can also:

- Cancel an assignment of an interrupt handler to an interrupt level
- Wait for an interrupt to occur
- Enable and disable interrupts

This shows how an interrupt task enters an *event loop* while it waits to service an interrupt.



1. The interrupt task initializes when the system starts and starts waiting for a signal when an interrupt occurs.
2. When signaled, the interrupt task executes the required operations.
3. The interrupt task releases control by waiting for a signal to process the next interrupt.

Figure 2-10. The Task Execution Model

See also: Init_IO procedure and interrupt task, Appendix A in *Driver Programming Concepts* for interrupts in random access devices; *inthand* and *inttask* examples under the */rmx386/demo* subdirectory for a demonstration of an interrupt handler and task

Exceptional Conditions

These are sources of exceptions:

- Environmental errors, such as trying to write to a printer that is offline
- Programmer errors, such as making a mistake in a system call

- Hardware exceptions, such as trying to execute a read/write data segment

Environmental and Programmer Errors

The Nucleus does validity testing and condition checking within system calls. It generates a *condition code* whenever it detects an exceptional condition (an error or something unusual), as well as when a call completes successfully. For successful completion, system calls return 0000H, or the mnemonic E_OK. For a failure, the code indicates what prevented successful completion. For example, 0002H or E_MEM returns if there is not enough memory to complete the call.

There are two ways to handle exceptional conditions:

- Using an exception handler
- Processing exceptions in the task that issues the system call (inline processing)

Using an exception handler simplifies error processing. When an error occurs, control transfers to the task's exception handler. The exception handler can be:

- The exception handler you write and specify for the task
- The exception handler you write and specify for the job
- An default exception handler provided with the OS

Each exception handler has a *mode*, which indicates when it is called:

- Never, meaning tasks handle all exceptions inline
- On programmer errors, but all other exceptions are handled inline
- On environmental errors, but all other exceptions are handled inline
- Always

Exception handlers typically use one of these methods:

- Correct the cause of the problem and try again.
- Log the error and continue.
- Delete or suspend the job that caused the error. (The default Intel handler deletes the job; there are also other supplied handlers.)

The alternative to exception handlers is inline processing. This enables you to provide special processing for unusual circumstances within a task.

Hardware Exceptions

An error that occurs as a result of a hardware exception also causes an exceptional condition. Hardware exceptions result from conditions like dividing by 0, or when a protected mode program tries to access or execute out of a memory segment bounds or tries to execute a read/write data segment.

If you do not designate an exception handler for hardware exceptions, they can cause your application to be caught in an infinite loop or be terminated. You can create exception handlers that process hardware exceptions.

Note
Prior to release 2.2 of the OS, hardware exceptions were not returned to exception handlers. However, exception handlers now can receive any hardware exceptions, along with other exceptions. You must rewrite older exception handlers so that they can deal with the possibility of receiving a hardware exception.

See also: Exception Handling, hardware exceptions, *System Concepts*;
Exception handlers in Nucleus system call summary,
Chapter 1 of *System Call Reference*;
Condition Code list, *System Call Reference*

Nucleus Messaging Service

The iRMX Nucleus Messaging Service is a general purpose message-exchange interface.used to communicate between different processes over many media and between applications and hardware. It enables the abstraction of the low-level hardware interface to support a large variety of I/O device types and also allows for layering of I/O applications based on a common interface model.

The interface model is based on a unit called a **service**. Client tasks communicate with the service using **messages** which are passed to and from the service via **ports**. Messages consist of two parts, a control part and an optional data part, whose format is defined by the service being used. In general, the control part contains information which describes the data part and what to do with it.

Services

A service is the base user of the Nucleus Messaging Service model. A service provides a system-wide interface to a system resource, such as a SCSI interface service, or a network transport service. A client task gains access to the service by creating a port object associated with that service. The client then communicates with the service by sending and receiving messages to and from the port.

Kernel Features

The Kernel provides a set of objects and system calls; some are similar to those in the Nucleus and some are entirely different. Kernel system calls provide functionality beyond that of the Nucleus. In most cases, these calls provide higher performance compared to the equivalent Nucleus calls.

The Kernel provides interface libraries for PL/M, C, and FORTRAN that enable you to use the same data types for any of these languages. The assembly language interface is a register interface that requires additional programming effort.

The Kernel system calls enable you to use these Kernel features:

- Kernel objects
- Kernel task management
- Kernel time management

The Kernel also has more system calls available for interrupt handlers than the Nucleus.

Kernel Objects

The Kernel provides these objects:

- Software alarms (virtual timers) that invoke alarm handlers you write. Alarm handlers operate in similar fashion to iRMX interrupt handlers.
- Semaphores for synchronization and mutual exclusion. There are three kinds of Kernel semaphores: FIFO queue, priority-based queue, and Region semaphores. The region semaphore provides priority adjustment like the Nucleus Region.
- Mailboxes for communication between tasks. Mailboxes have task and message queues. The task queues can be FIFO- or priority-based; the message queue is always FIFO, like the Nucleus.
- Memory pools and areas. A Kernel memory pool always spans a contiguous range of memory. You can create a Kernel memory pool in a specific range. Tasks share a memory pool for dynamic memory allocation by checking out, using, and returning memory areas in the memory pool. The memory manager keeps track of which areas in the pool are currently in use and which are available. It does not protect an area from unauthorized access and deletion.

  Kernel memory pools and areas you allocate are created from iRMX segments; make sure your jobs have enough memory to handle Kernel requirements.

You must allocate memory for Kernel objects and may allocate memory beyond the Kernel's needs. This is different from iRMX objects, where memory is allocated from the job's pool automatically. You can use this additional memory to store application-specific state information associated with the object. When you create an object, the Kernel returns a token that identifies that object.

See also: Kernel objects, *System Concepts*;
Kernel system call summary, Chapter 1 of *System Call Reference*

Kernel Task Management

The Kernel enables you to:

- Control task switching using scheduling locks
- Add task handlers to supply additional OS functions

A running task can use a scheduling lock to protect itself from being preempted in some cases. In those cases, a task switch will not occur until the task releases the lock, even if a higher priority task is ready to run.

Task handlers you write execute when you are creating a task, deleting a task, or switching to another task. Possible functions that task handlers may include are saving and restoring the state of coprocessor registers on a task switch, masking interrupts based on task priority, or implementing statistical and diagnostic monitors.

For example, you can use the task switch handler to determine which tasks in your system execute most frequently.

See also: Kernel task management, *System Concepts*

Kernel Time Management

The Kernel enables tasks to:

- Create single-shot alarms and repetitive alarms for a specified time interval and specify which alarm handler to invoke.
- Specify a clock tick granularity of less than the 10 millisecond granularity provided by the Nucleus.

See also: Kernel time management, *System Concepts*

Kernel Interrupt Handling

Unlike the Nucleus, the Kernel provides many system calls that you can use in interrupt handlers. For example, you can create Kernel semaphores or mailboxes in an ordinary task, then use Kernel calls to send units or messages from the interrupt handler to the ordinary tasks, which execute based on their priority, not on an interrupt level. Ordinary tasks do not mask interrupts; interrupt tasks do.

When To Use the Kernel

Use the Kernel in these situations:

- Only for very well-tested code
- For isolated parts of the application, such as signaling ordinary tasks from an interrupt handler
- When performance is critical, such as high-performance, unvalidated sending and receiving of data mailbox messages and semaphore units

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.

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.

Since the Kernel does not provide a protected and validated environment, it is more difficult to debug.

See also: Kernel data types, Kernel system calls and handlers, *System Call Reference*;
Kernel, assembly language interfaces to the Kernel, *System Concepts*

❑❑❑

BIOS and EIOS Features 3

Several iRMX OS layers provide I/O operations: the Basic I/O System (BIOS), the Extended I/O System (EIOS), the Universal Development Interface (UDI), and the Shared C Library. Each provides a different level of support and unique features. The EIOS, UDI and Shared C Library use the BIOS in most cases.

See also: UDI, Chapter 6 in this manual;
Shared C library, Chapter 8

I/O System Objects, Logical Names and System Calls

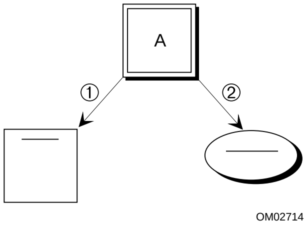
These are the I/O objects that provide access control and I/O capability:

- *User* objects are lists of user IDs.
- *Connection* objects are the bonds between a file or device and a task.
- *I/O Jobs* are similar to Nucleus jobs, but provide the environment for EIOS system calls.

There are BIOS, EIOS, and UDI system calls to create, delete, and manipulate I/O objects.

User Objects, Users and User Access Control

People access files through tasks; user tasks are tasks requesting access to files. The *user object* can prevent an unauthorized task from accessing a file. User objects provide user access control by verifying users, as shown in the next figure.



1. Task A creates a file. Task A is the owner of that file. Task A's user ID is listed first in the file's access list.
2. Task A also creates a user object. Task A's user ID is listed first in the user object. Task A lists user IDs for other tasks that can access the file in the user object.

   Only tasks listed in the user object can access the file.

Figure 3-1. Task Creating a File and a User Object

You specify users when you use the BIOS to create a file. The EIOS, UDI and Shared C Library use the default user object, which applies to all tasks in the job. For DOSRMX and iRMX for PCs, the DOS file system does not support users other than World.

See also: File access rights, later in this chapter

There are BIOS calls to list IDs in user objects and create and delete user objects. There are EIOS calls to list ID's associated with a user and to verify users. A user may have more than one user ID.

See also: User system calls, BIOS, and EIOS system call summary tables in Chapter 1, *System Call Reference*

Example: Multiuser System and the User Object

Suppose that several departments share a computer. An individual in one department may:

- Allow only herself to delete her files. She specifies this when she creates the files.
- Allow people within her department to write and read the files. She specifies this when she creates the user object for her department members.
- Allow people in other departments to only read the files. She specifies this when she creates the user object for other departments.

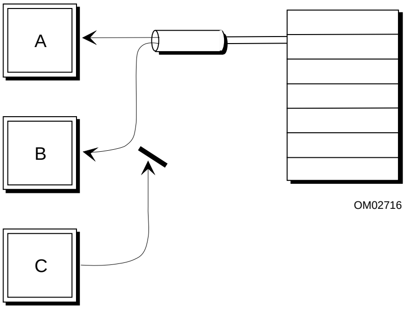
In systems where each user has a password, user access control can also be set up on an individual basis.

Connections and File Access Modes

There are connections for files and connections for devices.

File Connections

A *file connection* object is the bond between a file and a task. Connections provide file access mode control based on the operation performed on the file, as shown in this figure. The connection object can prevent a task from modifying or deleting a file inappropriately.



1. Task A creates a connection to a file and opens it. Task A specifies that this connection is for reading only and this file can be shared by other readers only.
2. Task B wants to read the file so it can share the connection A created.
3. Task C wants to write the file. It cannot use this connection for writing. Since the share mode is share with readers, Task C cannot obtain a connection to the file until Task A deletes its connection.

Figure 3-2. Tasks Sharing a Connection Object

Whenever you create a file, the I/O system returns a connection.

Connections contain the file access modes to the file. You specify file access modes once, when you create the connection, rather than each time you open the file.

The file access modes specified in connections are:

Mode	Meaning
Private use	Share with no one
Open for reads	Share with readers
Open for writes	Share with writers
Open for reads/writes	Share with readers/writers

If the first connection to a file enables sharing, several file connections can simultaneously exist for the same file; several tasks can concurrently access different locations in the file. Each connection maintains a pointer to the location within the file where the task is reading or writing, as shown in this figure.

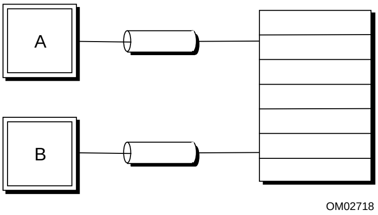


Figure 3-3. Tasks Accessing a File Through Connections

File connections cannot be shared across jobs.

There are BIOS, EIOS, and UDI system calls to create, open, close and delete file connections.

See also: Files and Connections in BIOS, EIOS and UDI system call summaries, Chapter 1 of *System Call Reference*

Device Connections

A *device connection* is the bond between the task and the device. You must attach a device before you can use it for I/O operations.

Device connections can be shared across jobs.

See also: Device connections, *System Concepts*

I/O Jobs

An *I/O job* provides resources for tasks that perform I/O using EIOS system calls. If a task is not in an I/O job, it cannot successfully use all of the EIOS system calls. I/O jobs are very similar to ordinary jobs.

If you use C Library calls, they must be made from I/O jobs.

There are EIOS calls to create, delete, and start I/O jobs.

See also: I/O jobs in BIOS, EIOS and UDI system call summaries, Chapter 1 of *System Call Reference*

Logical Names for Files and Devices (EIOS Only)

The EIOS enables you to use logical names to refer to files and devices. A *logical name* is a string of characters that identifies a file, directory, device, or remote computer system. You can substitute a logical name for a long pathname, and use it in several tasks or jobs.

The iRMX OS uses the logical name *:config:* for the file */rmx386/config*, for example.

If the pathname changes, you redefine the logical name; you don't need to change it everywhere. The EIOS associates each logical name with a particular file connection or device connection.

You can make a logical name available to one job, to a group of jobs, or to all jobs in the system by cataloging the name in the local job's object directory, the global job's object directory, or the root job's object directory, respectively.

See also: Logical names in EIOS system call summary, Chapter 1 of *System Call Reference*;
object directories, *Command Reference*

BIOS and EIOS System Call Differences

There are several differences between system calls in the BIOS and the EIOS. System call prefixes or names may be slightly different, reflecting functional and compatibility differences. The BIOS has two types of system calls: synchronous and asynchronous. The EIOS has only synchronous calls. The BIOS calls generally have more parameters than the EIOS calls, giving greater control and flexibility to the BIOS and more simplicity to the EIOS.

System Call Names

Many BIOS and EIOS system call names are identical except for the prefixes:

rq (16-bit address, 1 Mbyte memory pool)	**rqe** (32-bit address, 4 Gbyte pool)
rq_a (asynchronous)	**rq_s** (synchronous)

For example, the **rq_create_io_job** system call operates on all iRMX OSs and is available for compatibility between iRMX I, II, and III. The **rqe_create_io_job** system call supports the extended features of the Intel386 and later microprocessors, such as memory pools greater than 1 Mbyte. Unless compatibility with iRMX I systems is an issue, use the system calls with the **rqe** prefix instead of the ones with the **rq** prefix.

The BIOS asynchronous **rq_a_create_file** and **rq_a_close** system calls perform analogous functions to the EIOS synchronous **rq_s_create_file** and **rq_s_close** calls.

Synchronous and Asynchronous System Calls (BIOS Only)

These are the two types of BIOS calls:

Synchronous calls — Begin running as soon as your application invokes them, continue running until they finish their tasks or detect an error, then return control to your application. The call names begin with **rq_**.

Asynchronous calls — Run concurrently with your application, which can continue running while the BIOS deals with devices such as disk drives and tape drives. The call names begin with **rq_a**.

This example shows the difference between synchronous and asynchronous operations:

1. Task A and Task B need to read a file. Task A makes an asynchronous call and Task B makes a synchronous call.
2. The parameters for both calls are checked for validity.
3. Task A continues executing application code, doing computations perhaps, until notified that the data has transferred. Task B waits for the data transfer to complete before continuing.

If you make an asynchronous call, you use a mailbox to notify the task when the call has completed. Asynchronous calls do require more programming effort.

Files and Directories

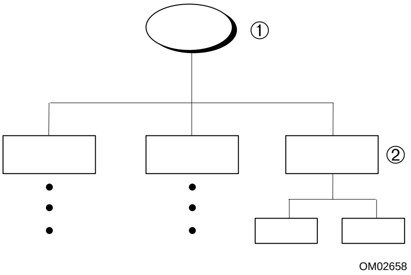
The iRMX I/O Systems provide support for:

- Hierarchical file systems
- File access rights to protect files
- Distinct file types
- System calls to create, delete, read, write, and manipulate files

Hierarchical File System

The iRMX OS uses a hierarchical filenaming system (similar to UNIX and DOS). Your directory and file names can reflect relationships between files and you can assign a unique pathname to each file.

With an unlimited hierarchical file system, you can add directories when you need them, as shown in this figure.



1. You can create a new directory whenever you need to, such as for a new department member.
2. The owner of the new directory can use it to provide unique pathnames to any number of subdirectories and files.

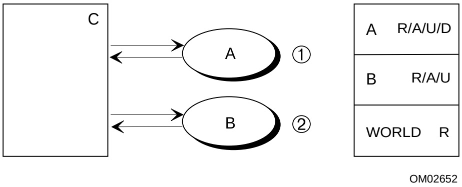
Figure 3-4. Hierarchical File Structure

iRMX filenames can be 14 characters long, and can include more than one . (dot). iRMX is case-insensitive.

The iRMX OS treats directories like files; entries in a directory are just filenames.

File Access Rights

The iRMX access rights for files are read, append, update, and delete; for directories they are list, add, change, and delete. Access rights are maintained on a per-file basis in a file access list. The iRMX OS supports multiple users with different file access rights as shown in this figure.



1. User A is the owner of the file. This user can read, append to, update, or delete the data file C.
2. User B can read, append to, and update the same file.
3. The World user (not shown) can only read the file.

Figure 3-5. Multiple Users with Different Access Rights

The DOS file system does not support users other than World, and file access rights are limited to two options: read-only and read/write/change. For DOSRMX systems, iRMX users and tasks can change their DOS file access to correspond to the DOS read-only and read/write/change attributes. DOS directories cannot be made read-only.

File access rights and owner IDs can undergo translation between remote files on a network that uses the Network File System (NFS). The specific translations used depend on the operating systems and users involved.

See also: File access, *Command Reference*
Accessing NFS Files, *TCP/IP and NFS for the iRMX Operating System*

File Types

These are the types of files: named, remote, DOS, physical, and stream. The same system calls work with any file type, providing *file independence*. For example, you use the same system call to open a named file as you do to open a stream file. This enables you to create tasks and applications that you can readily switch from one file type to another.

Named Files

Named files are for local random-access, secondary-storage devices, such as disk drives and diskette drives. Named files have a hierarchical structure that reflects the relationships between the files and the application; you can store many named files on one device. Named files provide access control because they have associated user and connection objects. The native iRMX file format is maintained by the Named File Driver.

With file independence, you can use named files during development and debugging, even though tasks will ultimately use other file types. For example, your application might need two tasks that communicate by using a stream file. You might implement the writing task before you implement the reading task. For the purpose of debugging the writing task, you could use a named file on a disk in order to examine the information being written. Later, after you implement the reading task, you can route the information to the stream file rather than the disk.

Remote Files

A remote file is a file located on another computer connected by a network. The I/O systems access remote files through networking software, including files on systems running an OS other than the iRMX OS. You can access remote files using the NFS file driver (on networks running TCP/IP and NFS) or using the iRMX-NET Remote File Driver.

See also: Accessing remote files, *Network User's Guide and Reference;*
TCP/IP and NFS, *Network User's Guide and Reference;*
Accessing NFS files, *System Concepts*

DOS Files

A DOS file is a file located on a DOS-formatted mass storage device; the device may be on a network. You access DOS files using the DOS file driver in iRMX III or iRMX for PCs systems or by using the EDOS file driver in DOSRMX systems.

See also: EDOS file driver, *Programming Concepts for DOS and Windows*

CD-ROM Files

A CD-ROM file is a file located on a CD-ROM formatted (ISO9660) mass storage device. You access CD-ROM files using the CDROM file driver in iRMX systems.

See also: CDROM file driver

Physical Files

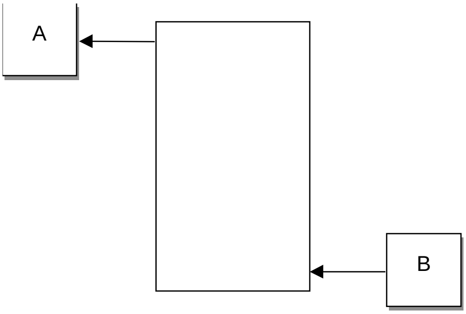
Each physical file occupies an entire device. Applications can deal with a physical file as if it were a string of bytes. Physical files provide these features:

- An application can have direct control over a device. For example, an application can use a physical file to interpret volumes created by other systems.
- Because the application deals with a physical file as a string of bytes, it can conserve memory and still communicate with devices that do not need named files. These devices include line printers, terminals, plotters, and robots.

Physical files do not support hierarchical file systems and file access control.

Stream Files

Stream files provide another means of intertask communication, as shown in this figure:



Task A can read from a stream file while Task B writes to it.

Figure 3-6. Tasks Using a Stream File

Stream files provide no access control. They are implemented in memory; they don't have an attached peripheral device.

See also: File types, *System Concepts*;
Files in BIOS, EIOS and UDI system call summaries, Chapter 1 of *System Call Reference*;
device connections, *System Concepts*

Devices and Device Control

The iRMX I/O Systems provide support for:

- Device independence through distinct file and device drivers
- Device control
- Automatic device reattachment (EIOS only)
- Terminal Support Code (TSC) to control terminal modes and operation

Device Independence

You can use the I/O system calls with a number of devices. This is called *device independence*. Device independence provides flexibility. For example, your application may log events as they occur. You can create an application that logs events on any device, enabling an operator to route logging from a hard disk to a line printer if she needs a printed listing.

The I/O systems manage devices using *file drivers* and *device drivers*; the separation between the BIOS and the device provides device independence.

The EIOS, UDI and Shared C Library all use the BIOS. Ultimately, all I/O requests your application makes pass through the BIOS to the drivers, then to the hardware, as shown in this figure.

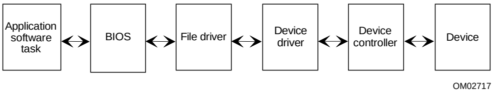


Figure 3-7. I/O Requests from the Application Go Through BIOS and Device Drivers

File Drivers

A *file driver* is a software interface between a device driver and the BIOS. These are the file drivers: named, remote, NFS, DOS, EDOS, physical, and stream. When you first attach a device, you tell the BIOS which file driver to use for that device. Then, the BIOS automatically uses that file driver for the device.

See also: File drivers and device independence, *System Concepts*;
attaching devices, *Command Reference*

Device Drivers

A *device driver* is a software interface between a device controller (the hardware and firmware) and the file driver. A device driver hides the idiosyncrasies of the device from the BIOS. The iRMX OS provides device drivers for many devices.

Loading and Configuring Drivers

You can load file and device drivers dynamically at run-time or at initialization. If you have an ICU-configurable system, you can select drivers during configuration. These drivers become part of the BIOS.

See also: Supplied device drivers and physical device names in Appendix E, *Command Reference*;
Loadable device drivers, in *System Configuration and Administration* and *Driver Programming Concepts*;
File and device driver screens, *ICU User's Guide and Quick Reference*

You can also write custom file and device drivers.

See also: Writing your own drivers, *Driver Programming Concepts*

Device Control

The iRMX OS lets you control:

- Updating files
- Disk integrity
- File fragmentation
- Buffering with overlapped I/O

Fixed and Timeout Updating

Fixed updating and timeout updating are two ways to update devices. They are triggered by the passing of set amounts of time.

You use updating to write all data in a buffer to a designated device, such as a disk, at set time intervals rather than just when the buffer is full. Updating can prevent loss of data in the event of power failure or other problems.

Fixed updating occurs when an amount of time, which is specified for an entire system, passes. At that time, all devices to which updating applies are updated. Fixed updating is independent of I/O activity.

Timeout updating is defined separately for each device, rather than applying to the system as a whole. The timeout period starts at the end of each I/O operation.

In I/O-intensive systems, you can delay updating by setting the fixed update period to longer than the average time between I/O functions.

Fixed updating is a BIOS configuration value. Stream files and physical files do not support updating.

See also: BIOS screens, *ICU User's Guide and Quick Reference*;
IORS, *Driver Programming Concepts*

Disk Integrity

In any computer system, there are many occurrences beyond your control that can cause damage to files or disk volumes. For example, power outages can occur just as a file is being written, or disk sectors can suddenly become unreliable. The I/O systems enable you to maintain disk integrity and determine whether files or volumes have been corrupted. The main features are:

- For hard disks, using system calls to get and set bad track and sector information
- Attach flags and fnode checksum field, which you can check to determine the integrity of named volumes and files
- Disk Mirroring, a hard disk configuration that maintains identical copies (mirrors) of data on two hard disks for increased reliability

See also: Disk integrity, *System Concepts*

Internal File Fragmentation

When information is stored on a mass storage device, space is allocated in blocks called granules. The block size is called granularity. Three kinds of granularity are important:

Device granularity — Is hardware dependent, varies among individual mass storage devices, and is the minimum amount of data that the device can read or write during one I/O operation. For disks, a device granule is called a sector; the device granularity is the sector size. Each buffer that the I/O systems use when reading and writing data is equal to the device granularity.

Volume granularity — Is a multiple of the device granularity and is the minimum amount of space that can be allocated to a file at one time. You specify the volume granularity when you format the volume. The I/O systems use volume granularity when deciding where on the volume to allocate this space.

File granularity — Is a multiple of volume granularity. You assign the file granularity on a per-file basis when you create a file; the granularity applies if the file needs to be extended.

By selecting the proper granularity values, you can minimize fragmentation of your files and balance I/O speed with efficient use of space on the mass storage device.

See also: Granularity, *System Concepts*

Buffering with Overlapped I/O (EIOS Only)

The EIOS provides the additional feature of buffering and overlapping of I/O operations. The EIOS uses the BIOS, however. Blocking and overlapping are more valuable in sequential I/O than in random-access I/O.

Whenever you open a connection, you specify the number of buffers the EIOS uses. This affects how the EIOS reads and writes information through the connection:

Zero buffers — This turns off EIOS buffering. The file is accessed each time you invoke a system call that reads or writes the file. For example, if you ask the EIOS to read 30 bytes, the EIOS accesses the file and reads exactly 30 bytes.

One buffer (Blocking) — The EIOS reads and writes information by blocking (transferring one buffer at a time), even though you may have specified transferring less. Blocking can improve the performance of an application because the EIOS might be able to satisfy several additional requests without reading the file again.

Two or more buffers (Overlapping I/O)

If you request two or more buffers, the EIOS can overlap I/O operations by using read-ahead and write-behind algorithms.

Read-ahead and *write-behind* enable tasks to continue running while the EIOS is transferring information to or from devices. This is because the EIOS can accurately determine, during sequential reading or writing, the location of the next data required by the application.

You can configure the maximum number of buffers that the EIOS can use for files on a particular device.

Automatic Device Reattachment (EIOS Only)

The EIOS constantly monitors the status of devices. When an operator removes storage media from a drive that is capable of detecting a volume being removed, the EIOS detaches the device and deletes all connections to files on the device. When the operator replaces the media, the EIOS automatically reattaches the device as soon as it is accessed, making it available to the tasks in your system. The same principle applies to remote device connections.

Some devices, such as some 3.5 and 5.25-inch diskette drives, cannot detect a volume being removed from the drive. For these devices, the EIOS cannot perform automatic reattachment.

Terminal Support Code

The Terminal Support Code (TSC) is a programmable interface between a terminal driver, the BIOS, and a user application. This support code provides a variety of special terminal modes and operations. The major capabilities of the TSC include:

Editing and controlling terminal input

There are a variety of characters that control and edit terminal input. You can replace default control characters with different characters. You can also switch a terminal to transparent mode, so that editing and control characters have no effect on the input line.

Type-ahead buffer

If you type faster than the OS can read, interpret, and respond, the TSC stores the data you type in a type-ahead buffer. The OS uses the data from this buffer when it is ready for it.

Controlling terminal output

You can set the TSC so that output sent to the terminal displays continuously, scrolls a few lines at a time, stops, or is completely discarded.

Escape sequences (translation) The TSC accepts escape sequences (characters preceded by an ESC character) to define the characteristics of a terminal. This feature enables you to characterize terminals so that the I/O system can use standard control codes and sequences of codes for all terminals. This is called *translation*. You can use escape sequences to set terminal variables, such as the number of lines displayed when in scrolling mode. You can change terminal behavior by entering in escape sequences or by running a program that sends the escape sequences.

See also: Terminal support code, *System Configuration and Administration* and *Driver Programming Concepts*

System Clock

Most boards supported by the iRMX OS have an on-board, battery backed-up time-of-day clock. The I/O systems use this clock in performing reads and writes. The global time-of-day clock is the timekeeper for the entire system. It is accessed only during system initialization, by a running application, or when requested by the operator.

The iRMX OS also maintains a local time-of-day clock in memory. The local clock is a copy of the global clock but has faster access time, for date and time needs.

The clocks keep track of two items:

- The current date (day, month, and year)
- The current time (hours, minutes, and seconds)

Nucleus, Kernel, and UDI system calls enable your applications to get and set the date and time for the local and global clocks.

See also: Time calls in Nucleus, Kernel, and UDI system call summaries, Chapter 1 of *System Call Reference*

Note
The BIOS layer previously provided the system calls to get and set the clock time. These system calls are now part of the Nucleus.

Choosing Between I/O Systems

This section describes the performance differences between the BIOS and the EIOS. It will help you decide whether to use system calls from the BIOS, the EIOS, or from both systems.

Each of the I/O systems satisfies different requirements. The BIOS offers more flexibility and control, while the EIOS offers ease of use. If both systems would be useful in one application, you can use both.

In the iRMX III OS, you can use the ICU to include the BIOS, the EIOS, or both systems. In iRMX for PCs and DOSRMX, both are included.

BIOS

The BIOS provides very powerful capabilities and makes few assumptions about the your requirements. The BIOS provides I/O features that are useful in a wide range of applications. These features illustrate the flexibility of the BIOS:

Custom buffering algorithm

You can design and implement your own buffering technique and control the synchronization between I/O and processing.

Asynchronous system calls

You can explicitly control system call synchronization.

Control of details

The BIOS system calls have many parameters, which enable your tasks to enhance the performance of your application system. This is useful in time-critical or memory-critical applications and for random-access I/O.

EIOS

The EIOS is easier to use than the BIOS, and has these features:

Automatic buffering of I/O operations

You need not become involved with buffering, aside from specifying how many buffers the EIOS uses. If your application system does not require buffering, you can tell the EIOS to use no buffers.

Synchronous system calls

You do not need to explicitly synchronize system calls. You can still use overlapped I/O operations using buffers.

Fewer parameters

EIOS system calls require fewer parameters than BIOS calls. This simplifies and reduces development time.

Making the Decision

Determine whether your application system requires the flexibility and fine tuning capability of the BIOS, the ease of use of the EIOS, or a combination. Before you make the final decision, consider these factors.

Control You may not need the control provided by the BIOS; the time required to develop the application system may be more critical than fine tuning its performance.

Memory The EIOS software requires the BIOS, so using both the BIOS and the EIOS requires no more memory than using the EIOS alone.

Implementing some EIOS features yourself (such as buffering) may use as much memory as including the whole EIOS.

Performance Because the BIOS gives your application system control of many details, you can probably design your application system to run faster with the BIOS than with the EIOS. If you decide to use the EIOS anyway, you can improve performance by optimizing the buffer sizes and the number of buffers.

I/O type Choose the BIOS for applications that require very little I/O or use random-access I/O. Choose the EIOS when development costs are critical, especially in applications that use sequential I/O.

Prototypes Use the EIOS to create a prototype application system, and then later replace it with your custom I/O system.

C Library functions Require both the BIOS and the EIOS.

Use both layers when your application system uses I/O for several purposes, some of which are best accomplished by the BIOS, and others by the EIOS.

Examples

These examples illustrate the advantages of each of the I/O systems. The examples assume that you will produce many copies of the application system.

Application Systems Using Little I/O. If your application system requires very little I/O, such as only occasionally logging information to a diskette, use the BIOS. The ease of use provided by the EIOS can save you very little time during development because the I/O-related part of your system requires so little time to develop. Using the BIOS will also save memory.

Application Systems Using Only Sequential I/O. If your application system requires a substantial amount of sequential I/O, a large amount of your development resources will be expended in support of I/O. Use the EIOS to save time, and because the EIOS provides overlapping I/O. It incorporates read-ahead and write-behind algorithms that operate sequentially, and overlaps I/O operations and processing.

High Performance Applications Using Random I/O. If your system performs a large amount of random-access I/O, the BIOS is the appropriate choice. Performance tuning is also provided by the BIOS. Although such a system might require more development time to implement, it should run faster than the EIOS.

❑❑❑

Human Interface Features 4

The HI provides several features for both you and the users of your application:

- Enables loading file and device drivers, system jobs and your application at initialization time or run-time
- Provides HI commands that perform simple programming functions
- Provides system calls that help you write commands for your application
- Provides multiuser environment support
  - — For the development environment
  - — For the application
- Provides the Command Line Interpreter (CLI) with its own set of commands

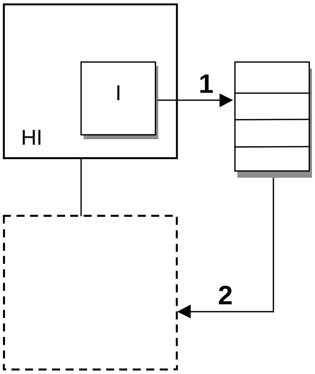
Run-time Loading of Jobs

The HI enables you to load system jobs, networking jobs, file and device drivers, and your application when the system boots or dynamically when the system is running. Loaded jobs become a part of the iRMX OS until the system is shut down or the job is unloaded. Loaded jobs have access to all features of the iRMX OS.

For iRMX for PCs and DOSRMX, this is the only way to load your application and the supplied loadable OS jobs. For iRMX III users, the HI enables you to change the configuration without using the ICU, then rebuilding (linking) the existing system.

A *loadinfo* file is executed during HI initialization. You edit this file to load the jobs and drivers needed for your application. Jobs and drivers loaded this way are child jobs of the HI, as shown in this illustration.

See also: Descriptions of loadable jobs, *System Configuration and Administration*



1. The HI initialization Task I executes the *loadinfo* file to load a job into memory from secondary storage.
2. The loaded job is a child job of the HI.

Figure 4-1. Loading a Job at HI Initialization

HI Commands

The HI commands are small system programs that manage users, files, and devices and provide general utilities during development. You can enter commands interactively from your keyboard or write them into a file. Some example commands supplied with the OS are:

copy	Copies or creates files
copydir	Duplicates a directory and its files and subdirectories
deletedir	Removes a directory, including all its subdirectories and files
format	Formats a disk
password	Adds or deletes users, or changes a logon password
permit	Changes a file's User IDs and access
rdisk	Partitions a hard disk
shutdown	Provides an orderly shutdown procedure for the OS
sysload	Loads a job or driver into memory from secondary storage

You can include HI commands as part of an application system if you need them.

See also: *Quick Reference to Commands* for summaries of commands and equivalent commands in the DOS and iRMX OSs;
Command Reference for complete command descriptions

Human Interface System Calls

The HI provides system calls that enable you to create commands that are appropriate to your application and meaningful to your operator.

By designing commands appropriate to your operators, you can create a user-friendly system and reduce operator errors.

You have great flexibility in creating new commands. The main requirement is that the first word in a command must be the name of an executable file on a secondary storage device such as a disk. When an operator enters a command, the OS loads the named file from secondary storage and runs it. This gives you these advantages:

- You add or modify commands simply by writing new ones.
- The number of custom commands for a system is not limited by the amount of dynamic memory.
- You do not have to rebuild the system to change commands.
- Commands used infrequently do not take up RAM space when they are not being run.

See also: Customizing commands, *System Concepts*

The categories of system calls for creating commands are:

- Command-parsing system calls
- I/O and message-processing system calls
- Command-processing system calls for invoking interactive HI commands programmatically
- Program control system calls to override the default <Ctrl-C> handling task provided by the HI

You can also add commands you need to the development environment if you wish.

See also: Human Interface system call summary, Chapter 1 of *System Call Reference*

Custom Command Line Parsing

The HI system calls for *parsing a command line* retrieve and interpret parameters of a command.

For example, in an application that monitors toxins in the blood of hospital patients, an operator might run a task that displays the toxin level of an individual patient or of all patients being monitored.

You might design a user-friendly approach, with commands oriented to the application and operator, rather than computer-oriented commands. For example, a command might be:

```
    toxin of John Doe
```

The program `toxin` issues a system call to receive the parameters `John Doe`. Because filenames are frequently parameters for commands, there are specialized system calls to interpret filename parameters.

Multiuser Support

You may need multiuser support in your development environment or for your application and operators. In either case, you can use the HI.

The HI enables:

- Adding users to the User Definition File (UDF), which defines user attributes such as user job memory pool size and user job priority
- Identifying each user's initial program or CLI (command line interpreter) (the program that runs when the user logs on)

With multiuser support in your development environment, programmers can execute commands, run development programs (such as editors and compilers), and run applications in a common environment. With multiuser support in applications, multiple operators can communicate with your application simultaneously.

You can implement multiuser support another way using simultaneous multiple-terminal support with I/O system calls. You might do this if you need to implement functions not available with the HI multiuser feature, or if (in an ICU-configurable system) you want to exclude the HI layer from the application.

HI Initialization

When the HI begins running, it does these things:

1. Executes a system-wide setup.
2. Initiates user logon.
3. Creates a user job for each operator or programmer logged on. This job provides the environment where programmers develop applications or operators use applications.
4. Starts an initial program or CLI that is the programmer's interface to the OS or the operator's interface to the application.

See also: Logging on, *Installation and Startup*;
HI initialization, *System Configuration and Administration*;
Multiuser support, *System Concepts*

System-wide Setup

When a multiuser system boots, the HI initializes each terminal in the system as either a *static logon terminal* (a specific operator is always associated with that physical terminal) or a *dynamic logon terminal* (any valid operator can log on and use the terminal). You specify the number and types of terminals.

Multiuser support includes device drivers that communicate with multiple-terminal hardware.

See also: Terminals, *System Configuration and Administration*

Logon

The HI validates terminal users at logon. Terminal operators can:

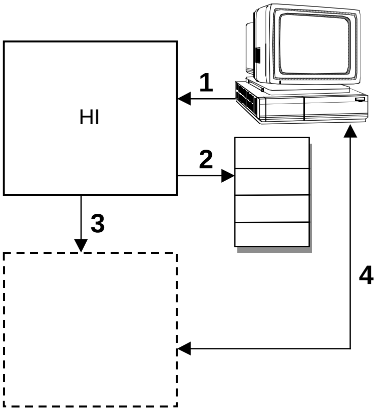
- Share a terminal with no logon; all operators share a single user ID
- Have exclusive use of one terminal
- Share a terminal; each operator has a user ID and uses a password to log on to the OS

If the terminal is on an iRMX-NET communications network, the operator or programmer can use the network to access remote files.

See also: Accessing remote files, *Network User's Guide and Reference*

Operator Job Creation

At logon, the HI associates each operator or programmer with a User ID and creates for each operator a separate job. These jobs are child jobs of the HI, as shown in the next figure.



1. After the HI initiates user logon, the operator enters a user name and password.
2. The HI validates the user by checking the User Definition File (UDF).
3. If the user is valid, the HI creates a user job that is a child job of the HI. The user's initial program or CLI runs in the user job.
4. The user can access and run the application system.

Figure 4-2. Validating Users With the HI

When an operator or a programmer creates files or attaches devices, she is the owner of those files or devices. Access to the files by other operators or programmers depends on the user object created by the owner.

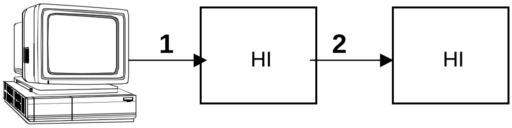
Command Line Interpreter (CLI)

The HI supplies a standard initial program called the Command Line Interpreter (CLI). The CLI is the part of the OS you interact with from the command line after you install the iRMX OS. The CLI has its own set of commands.

See also: Getting acquainted with the operating system in *Installation and Startup*, for a brief tutorial on logging on and using commands

You can use the CLI in the development environment and you can include it for your operators in the application if you wish.

With the CLI, the operator or programmer invokes a command from a terminal command line by entering the command name and specifying parameters if required. The CLI reads the information from the terminal and executes the command as shown in this figure.



1. An operator enters a command at the terminal. The CLI accepts the terminal input and parses the command. If the command is a CLI command or a command you have written, the CLI executes it after parsing it.
2. If the command is an HI command, the CLI passes the command to the HI. The HI loads the command into memory and executes it.

Figure 4-3. User Interacts with the CLI

The CLI provides a number of features:

Support for different kinds of terminals

The attributes of any operator's terminal are stored in the *termcap* file. You can edit this file to change the characteristics or to add support for new terminals and you can dynamically switch terminal types.

See also: *termcap* file, *System Configuration and Administration*

Editing and controlling terminal input

You can input commands at any time, and then press the <CR> or <Enter> key (as defined in the *termcap* file) to send the input to the CLI.

The CLI also contains special function keys, which move the cursor, replace the current command line with a previous command line, execute a command line, delete characters, abort the current command, or continue a command onto the next line.

Type-ahead

You can continuously enter command lines. The CLI sends the first line to the OS for processing and saves additional data in a type-ahead buffer. After the OS finishes with a line, the CLI fetches and processes the next line.

Recalling commands

You can retrieve the last 40 command lines entered, then edit a line and execute the edited command.

Background mode

You can run commands in background mode, display a list of background jobs, and cancel background jobs.

I/O redirection — With I/O redirection, tasks that normally use keyboard input and screen output can receive data from and send data to files or other I/O devices. This permits tasks to run without operator intervention, which is especially useful when running in background mode.

Aliases — You can assign and cancel abbreviations for commands.

You can use the ICU to include your own extensions in the CLI. This enables you to add your own features and still retain the capabilities of the CLI.

See also: Human Interface screens, *ICU User's Guide and Quick Reference*

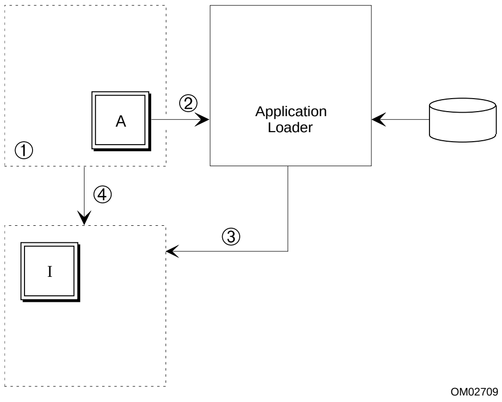
Alternatively, you can supply or create your own initial program that you load at HI initialization or load dynamically at run-time. There can be a separate initial program for each operator.

See also: User Attributes File, *System Configuration and Administration*

❑❑❑

Application Loader Features 5

The AL enables tasks to load programs from secondary storage into memory at run-time. The loaded program can run in the calling task's job or it can run as an I/O job, as shown in this figure.



1. Task A is part of an I/O job.
2. Task A calls the AL to load a program into memory from secondary storage.
3. The AL creates an I/O job for the program to be loaded.
4. The AL creates the initial Task I and loads the program into memory.

Figure 5-1. Using the Application Loader To Load a Program Dynamically

Because the loaded job is the child of a job you have created, you can receive notification when the loaded job is deleted if you wish.

Dynamic Loading

The AL performs dynamic loading: it modifies appropriate addresses in the program at the time it loads the program. Dynamic loading offers flexibility in designing and maintaining application systems:

- The AL loads the programs anywhere in available memory. If you add more memory to the system, the AL will use it.
- You can change programs without rebuilding (linking) the existing system.
- If you have memory restrictions, you can store seldom-used programs on secondary storage until you need to run them.
- You can use overlay modules to execute programs that are actually larger than the memory available.

The AL can load object code from any device if the device supports iRMX named files and you have the appropriate device driver. The AL requires programs to be object code and meet certain other requirements.

See also: RCONFIGURE control, STL format, SEGSIZE control,
DYNAMICMEM, *System Concepts*;
object code, object file, and object module, Glossary in this manual

Loading Flat Model Applications

The Application Loader can recognize and load an application that you write using a flat-model (non-Intel) compiler. Flat model applications require that you use the paging subsystem provided with the OS.

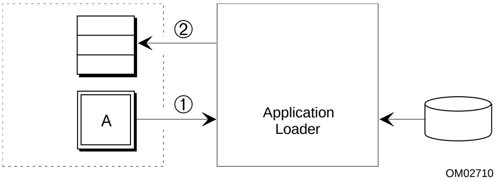
See also: C Compilers, Flat Model, *Programming Techniques*;
flat.job, *paging.job*, *System Configuration and Administration*
Paging System Calls, *System Call Reference*

Objects and System Calls

Most AL system calls require the EIOS because they use connection objects and I/O jobs.

The AL provides synchronous and asynchronous system calls. To overlap processing with loading operations, use asynchronous system calls. If the calling task can wait until the new program is loaded, use synchronous system calls, which are easier.

You can also use the AL to load a program into the calling task's job, as shown in this illustration, but this requires additional programming effort.



1. Task A calls the AL to load a program into memory from secondary storage.
2. The AL loads the program into a memory segment in Task A's job.

Figure 5-2. Using the Application Loader To Load a Program into a Task's Memory

See also: Application Loader system call summary, Chapter 1 of *System Call Reference*;
Application Loader, *System Concepts*

You can use the ICU to include or remove the AL, or you can select the features of the AL to meet your exact needs.

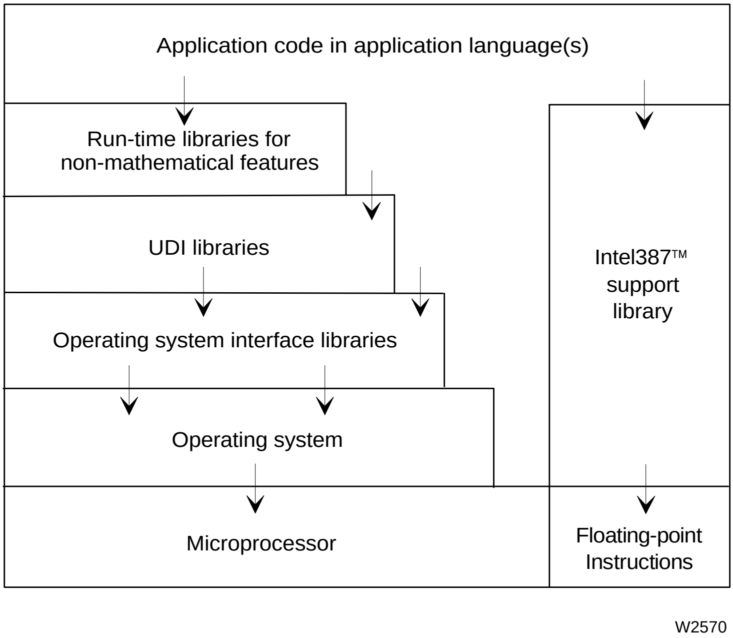
See also: Application Loader screens, *ICU User's Guide and Quick Reference*

❑ ❑ ❑

UDI Features 6

The UDI is a high-level interface to the iRMX OS: a set of system calls enabling language software (such as compilers, interpreters, assemblers, or run-time systems) to use the OS.

If an application makes only UDI system calls with no explicit calls to an iRMX OS, you can transport the application between other OSs that also support the UDI. If you want portability, don't mix UDI calls with BIOS and EIOS calls. This figure illustrates the relationship between the application code, the layers of software and the processing hardware.


Application code in application language(s)
Run-time libraries for non-mathematical features
UDI libraries
Operating system interface libraries
Operating system
Microprocessor
Intel387™ support library
Floating-point Instructions
W2570

The downward arrows represent system call flow and data flow from the application down to the hardware, where the calls are ultimately executed. In this case, the application does not make direct calls to the OS, but interacts through the UDI software. The figure does not show the upward flow of data from the hardware to the application code.

Figure 6-1. UDI Interface Between the Application and the Hardware

When you use the UDI, you can switch OSs by changing the UDI library. The UDI libraries always present the same interface to the application, but the interface with the OS is designed specifically and exclusively for that OS. There are UDI libraries for the iRMX, iNDX, UNIX, and XENIX OSs.

The UDI system calls behave somewhat differently when used in different OSs. This is because each OS has many unique characteristics, and some of them are reflected in the results of the UDI calls.

You can run any language on the iRMX OS as long as the language processor uses the UDI standard system calls and the Object Module Format (OMF) is compatible. The UDI software interface provides two major advantages:

- A language processor can use well-defined, appropriate, standard calls to communicate with the iRMX OS. You can easily adapt existing languages to run on the OS.
- Any language processor or software tool using UDI system calls, including user-written programs, is portable.

See also: UDI system call summary, Chapter 1 of *System Call Reference*;
UDI, *System Concepts*

For the UDI, the only ICU-configurable option is whether to include the UDI in your system.

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

❑❑❑

Networking Features 7

A network is a group of independent computers connected together to exchange information. This chapter describes the software and hardware that Intel provides for this purpose, and provides some basic networking concepts and structure.

See also: *Network User's Guide and Reference, Programming Concepts for DOS and Windows,* and *TCP/IP and NFS for the iRMX Operating System*

Network Concepts and Terminology

An individual computer system is a *node* in the network. The node you are logged into is the *local node*; any other one is a *remote node*. The nodes are connected into a *Local Area Network* (*LAN*), usually by a physical connection such as Ethernet. Systems on a network can share resources such as files, printers, diskette drives, tape drives, and modems. Nodes on an iRMX network can exchange information with computers using other OSs, such as UNIX or DOS. This is called *interoperability*.

iNA 960 provides programmatic access to transport services and iRMX-NET provides transparent file access. An ICU-configurable system running iRMX-NET can be configured as a *server* (a computer that provides resources), a *client* (a computer that requests resources), or both. Typically, iRMX systems run both the iRMX-NET client and server jobs.

TCP/IP for iRMX OSs includes Telnet and FTP for remote login and file access. NFS provides transparent file access. Any iRMX system can be both a client and a server for TCP/IP services and for NFS.

Each network can be divided into smaller units. These are called *Administrative Units* (AUs) in iRMX-NET. An AU is a group of systems that has the same set of users.

You can set up multiple *subnets* in both iNA 960 and TCP/IP. A subnet is used for dividing a network into reasonable sizes or logical groups; systems in the subnet do not necessarily have the same users. To communicate between subnets you implement routing on a node connected to two or more subnets.

A special feature of iNA 960 allows boards in a Multibus II system to treat the backplane as a virtual Ethernet connection without any Ethernet hardware. This

Multibus II subnet lets more than one board in the system run TCP/IP software. With a router in the system, boards that do not have a *network interface card* (NIC) can access the LAN over the Multibus II subnet.

Network Software

You can use iNA960 by itself to provide programmatic access to the network. Along with iNA 960 you can run iRMX-NET to provide transparent file access. You can run TCP/IP along with iNA 960. To use NFS for transparent file access, you must run TCP/IP.

TCP/IP for iRMX OSs

TCP/IP network software enables users to access other computers on the network. You can configure TCP/IP as both a first-level job by using the ICU, or as a job loaded through the **sysload** command. TCP/IP supports these features:

- Telnet client and server software provides virtual terminal access to and from non-iRMX computers.
- File Transfer Protocol (FTP) client and server software enables file transfers to and from other computers, as well as basic directory management.
- Network File System (NFS) client and server software enables transparent access of remote files and directories using TCP/IP protocols.

TCP/IP software provides industry standard networking protocols. This enables interoperability with most other OSs. Administrators of multiple OS networks, as well as many users, are likely to be familiar with TCP/IP networks.

See also: *TCP/IP and NFS for the iRMX Operating System* for more details

iNA 960 and iRMX-NET

iNA 960 provides general-purpose network communication services, including the Data Link, Network, and Transport layers defined in the Open Systems Interconnection (OSI) model. By itself, iNA 960 provides a programmatic interface to the International Standards Organization (ISO) OSI protocol.

iRMX-NET is part of Intel's family of OpenNET Local Area Network (LAN) products. The iRMX-NET software requires the Nucleus and BIOS layers and an underlying iNA 960 job. iRMX-NET is compatible with Microsoft MS-NET for DOS platforms. iRMX-NET provides transparent file access and user-interface commands.

iNA 960 includes: ISO transport software that provides general-purpose services, including the Data Link, Network, and Transport Layers as defined in the ISO OSI model. iNA 960 is available as customized jobs suitable for particular NICs.

You can use iNA 960 with or without iRMX-NET. When used by itself, iNA 960 provides only a programmatic interface to network services through layer 4 in the OSI Reference Model.

iRMX-NET includes:

Server job: Allows remote systems to access public files on the local system.

Client job, including File Consumer and Remote File Driver (RFD): Lets you access public files on any system that runs the server job, as if the files were local

iNA 960 and iRMX-NET operate within the Open Systems Interconnection (OSI) Reference Model, a seven-layer reference model defining network architecture.

The iNA 960 software operates in either of two hardware environments: COMMengine (offboard NIC) or COMMputer (onboard NIC).

iNA 960 supports Multibus I, Multibus II, and PC bus architectures and is provided as a set of ICU-configurable and loadable jobs, each specific to the particular bus and LAN hardware.

See also: iNA 960, iRMX-NET, COMMengine and COMMputer, *Network User's Guide and Reference*;
Network jobs, *System Configuration and Administration*

Network Security

Any iRMX-NET network must have at least one AU; an AU can be as small as a single system. AUs provide easy maintenance and security.

iRMX-NET uses two files for network definition and security:

User Definition File (UDF): Defines users. The same user can be in multiple AUs, but must have a unique ID in each one. A client uses the UDF to validate a user when the user logs on.

Client Definition File (CDF): Defines clients with names and passwords. A server uses the CDF to validate a client when the client establishes a connection with it.

The system administrator sets up and maintains the AUs, UDF, and CDF.

See also: Network Administration, AUs, UDF, CDF, in *Network User's Guide and Reference*;
UDF, CDF, in *System Configuration and Administration*

For TCP/IP users, the Telnet and FTP applications each provide some level of security when accessing file-based data. NFS users are provided with Unix and Short style user authentication, but not with DES encryption as described by Request For Comment (RFC) 1057.

Networking Between Operating Systems

Using the appropriate networking software, you can exchange information with computers using other OSs, such as UNIX and DOS. If you use iRMX-NET, these different systems interoperate using the Network File Access (NFA) protocols. Each OS must run with a corresponding OpenNET product that uses NFA protocols.

This figure illustrates the interoperations of an iRMX OS using the iRMX-NET Software with other OpenNET systems, and the relationship between a server and a client. The direction of the arrows indicates the flow of resource requests.

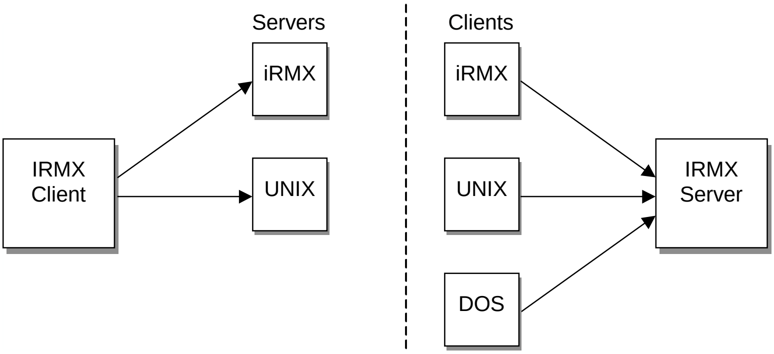


Figure 7-1. iRMX-NET Interoperability with other OpenNet Systems

The left side of the figure shows an iRMX system configured as a client with iRMX and UNIX systems operating as servers. The right side of the figure shows an iRMX system that is configured as a server for iRMX, UNIX, and DOS clients.

TCP/IP for the iRMX OSs provides interoperability with OSs running TCP/IP and Telnet, FTP, or NFS servers.

See also: Interoperability, *Network User's Guide and Reference*

❑❑❑

System Development 8

The iRMX OS helps you develop real-time application systems quickly and enables you to concentrate on the software that relates specifically to the application. The OS includes:

- industry-standard languages:
  - PL/M
  - C
  - Assembler

  You can also use non-Intel tools, such as Microsoft, Borland and Watcom C.

  See also: C compilers, *Programming Techniques*

- Shared C library
- These software tools for editing and building ICU-configured systems:
  - Aedit text editor
  - BLD386
  - BND386 for linking your code
  - OH386
  - Mapper
  - Librarian
- These debuggers:
  - Soft-Scope for Windows and Soft-Scope III for the iRMX command line
  - iRMX static System Debugger (SDB)
  - System Debug Monitor (SDM)

  See also: Application debugging, in this manual;
  Programming Techniques and Aedit Text Editor,
  System Debugger Reference
  Soft-Scope Debugger User's Guide

- Online help

You can develop your code on the same computer that will run your application (on-target development) or develop on one for installation later on a another platform.

See also: Installation options, *Installation and Startup*;
Development environment, *ICU User's Guide and Quick Reference*

Shared C Library

The C library supports hundreds of C functions and macros for applications that run in the multi-tasking iRMX OS environment. This includes many standard C functions that enable applications to perform common I/O operations without making direct iRMX system calls (OS-independent). There is also support for iRMX OS-dependent operations such as multitasking, time-of-day, signal management, and environment management; this enables you to create portable code using standard ANSI and POSIX programming practices. You can mix C library calls with direct iRMX system calls.

The Shared C library, sharable by multiple tasks and jobs, is available as an iRMX OS extension job in two ways:

- Run-time loadable job.
- Resident first level job in ICU-configurable systems. The C Libraries require both the BIOS and EIOS.

Any number of tasks and jobs (up to the maximum that the OS allows) may share the C library concurrently, each with its own independent C environment. The C library automatically manages common system resources such as I/O interfaces and memory when your code makes C library calls that use these resources.

See also: *C Library Reference* for information on supported functions

Online Help Systems

These online help systems are included:

- Windows Help for iRMX system calls and condition codes (Windows systems only)
- Manuals viewable in Adobe Acrobat (.PDF) format (Windows systems with CD-ROM drive only)
- Help for iRMX commands (at any iRMX command line prompt)
- The Interactive Configuration Utility (at the ICU prompt)

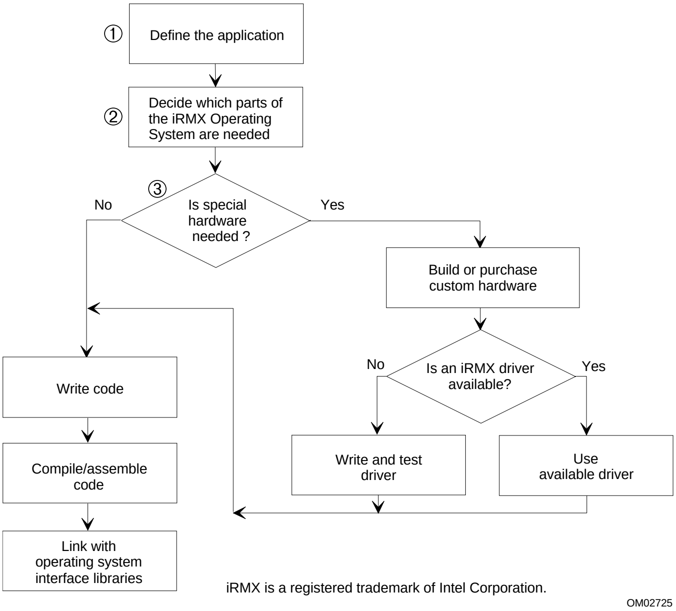
See also: Using online help, *Installation and Startup*;
help command, *Command Reference*;
ICU help screens, *ICU User's Guide and Quick Reference*

System Design

There are some general guidelines for designing and developing real-time systems. The first step is to define the application. This step should include:

- Listing all the various inputs and outputs in the application. Decide which objects to use for intertask coordination and communication.
- Listing all the tasks that need to be done to produce the input and output. Define interrupts and decide which ones require determinism. Assign interrupt levels and priorities to take advantage of multitasking and preemptive, priority-based scheduling.
- Develop the detail for each task in a block diagram.
- Decide if the application requires multiple jobs, and if so, how they will use shared memory and dynamic memory allocation.
- Design your user interface.
- Determine if you require custom devices. Decide whether to use loadable or resident file and device drivers. Decide whether to use custom or Intel-supplied drivers.

This flowchart shows steps typically taken by iRMX designers.


①
Define the application
②
Decide which parts of the iRMX Operating System are needed
③
No
Is special hardware needed ?
Yes
Build or purchase custom hardware
No
Is an iRMX driver available?
Yes
Write code
Compile/assemble code
Link with operating system interface libraries
Write and test driver
Use available driver
iRMX is a registered trademark of Intel Corporation.
OM02725

1. Define jobs. Define tasks, interrupt levels, and priorities. Decide which objects to use. Define interrupts, handlers, and levels.

2. Decide whether to use loadable or resident file and device drivers. Decide how to implement a multiuser environment and/or an operator interface.

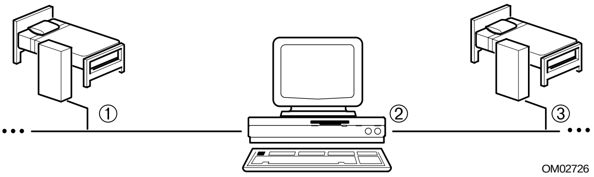
3. Decide if you need custom hardware that solves some unique problem or gathers data in a unique way.

Figure 8-1. Typical Development Cycle for iRMX Applications

At the completion of these steps, the prototype system is ready to be tested, debugged, and fine-tuned.

A Hypothetical System

This hypothetical application system monitors and controls dialysis. The system consists of three main hardware components, as shown in this figure.



1. A bedside unit is located by each bed. Each unit contains a processor board with the iRMX OS, which performs these functions:
   - Measures the toxins in the blood as it enters the unit
   - Adjusts the rate of dialysis
   - Removes toxins from the blood
   - Generates the bedside display for bedside personnel
   - Accepts commands from the bedside personnel
   - Sends information to the MCU

2. The master control unit (MCU) is a PC with a screen and a keyboard. This system also runs a version of the iRMX OS. The MCU enables one person to monitor and control the entire system. It performs these functions:
   - Accepts commands from the MCU keyboard
   - Accepts messages from the bedside units (toxicity levels, bedside commands, emergency signals)
   - Creates the display for the MCU screen

3. iRMX-NET connects the bedside units to the MCU.

Figure 8-2. The Hardware of the Dialysis Application System

The next sections describe how various iRMX features are used in the hypothetical system.

Interrupt and Event Processing

Interrupts and internal events occur at the bedside units: bedside personnel enter commands asynchronously and the system computes toxicity levels at regular intervals.

Toxicity levels, measured as the blood enters the bedside unit, are not subject to abrupt change. The machine slowly removes toxins while the patient's body, more slowly, puts toxins back in. The result is a steadily declining toxicity level. The bedside units must monitor toxicity levels regularly, but not too frequently. For instance, the bedside units could compute the toxicity levels once every 10 seconds, using a clock for timing. The measurement task would measure and compute the toxicity, put the information in a mailbox for the MCU, and suspend itself for 10 seconds.

Command interrupts from the bedside unit occur when a medical operator types a command and presses <Enter>. Interrupts from command entries occur at random times. The interrupt handler signals the interrupt task. The interrupt task performs any required processing and waits for the next interrupt.

Processing Commands From the Bedside Units

Each time a medical operator types a command and presses <Enter>, the bedside unit receives an interrupt signal from the terminal. The bedside unit stops executing the current instruction and begins to execute an interrupt handler.

1. The interrupt handler accumulates the characters in a buffer and puts them in a segment. The interrupt handler signals the interrupt task for bedside commands.
2. The interrupt task gets the contents of the segment where the handler put the command. It parses the command and does the required processing.
3. It puts the command information, along with the number of the bedside unit, into a message.
4. It sends the message to the predetermined mailbox for the MCU.
5. The interrupt task begins waiting for the next interrupt. The system returns to its normal priority-based, preemptive scheduling.

Multitasking

Tasks in the application run using preemptive, priority-based scheduling. This allows the more important tasks, such as those controlling the rate of dialysis, to preempt lower-priority tasks, such as updating displays. New capabilities could be added to the system by simply adding new tasks.

Intertask Coordination

A number of mailboxes used to send information from one task to another are the only form of intertask communication.

Enhancing the System

Multi-programming

The application can perform statistical analysis in a different job. The statistical application and dialysis application don't need to share any objects. Using two different jobs minimizes the chance that one application can affect the other.

If the two applications need to share a little information, the shared data can be passed from one job to the other without losing the benefits of isolation.

Mass storage files

The application can include recording information about patients in files on tape, diskettes and hard disks.

Device independence

If the application is extended to allow the MCU operator to send recorded data to several devices (such as a printer, magnetic tape, or disk), the device-independent I/O system enables recording the data without adding code specific to each possible device.

Bootstrap Loading (iRMX III OS and iRMX for PCs)

The iRMX III OS and iRMX for PCs provide a Bootstrap Loader that enables your application system to reside on disk and be loaded into RAM (random access memory) when the system starts.

The Bootstrap Loader resides partly in ROM (read-only memory) and partly on disk on your application hardware. When your system is reset, the Bootstrap Loader receives control, and loads the rest of the software, including the iRMX OS and the application software, into RAM. The Bootstrap Loader provides these advantages:

- By placing the Bootstrap Loader in ROM, you can shift the rest of your application system to RAM. This decreases the amount of ROM required. Since ROM requires that information be burned or masked into memory, the Bootstrap Loader reduces your masking or burning expenses and manufacturing costs.
- The Bootstrap Loader simplifies providing updated software to your customers. You can ship diskettes containing the updated software, reducing the cost of updating your software.

❑❑❑

Application Debugging 9

The iRMX OS provides several levels of debugging support. Sometimes you will use the features listed in this section as part of your system, and sometimes you will use them only during development.

System Debug Monitor

The System Debug Monitor (SDM) is an assembly-level debugger that enables you to load and run code, examine registers and memory, and disassemble code.

This monitor is soft-loaded into RAM. The monitor can run on target on a stand-alone development system, or in a development environment with a separate iRMX host and target.

The SDM monitor provides commands that perform these functions:

- Load and execute the code module
- Examine and modify memory and CPU registers
- Display the contents of descriptor tables (protected mode only)
- Move, compare, and search blocks of memory
- Read and write to an I/O port
- Disassemble code and execute one instruction at a time
- Disassemble code and sequentially execute instructions until encountering a call instruction

See also: SDM, *System Debugger Reference*

System Debugger

The iRMX System Debugger (SDB) extends the capabilities of SDM. It provides static debugging for when the system hangs or crashes, when you wish to freeze the system and examine it, or when synchronization requirements preclude debugging selected tasks. By stopping the system, the SDB provides a global view of the system. The SDB requires only the Nucleus to run.

The SDB enables you to:

- Identify and interpret iRMX system calls
- Examine a task's stack to determine system call history
- Display information about iRMX objects
- Display information about the job hierarchy
- Display the register contents
- Single step

See also: *System Debugger Reference* for details on debugging

Soft-Scope Debugger

The Soft-Scope debugger is a tasking debugger, which enables multiple tasks to be debugged simultaneously while the rest of the system continues to run. It is an interactive, source-level, symbolic debugging tool that enables debugging code modules at the source level, such as C or PL/M, rather than at the assembly level. There are two versions of this debugger:

- Soft-Scope III: Installed with the OS, this version operates from the iRMX command line. You invoke it with the **ss** command.
- Soft-Scope for Windows NT: Installed from CD on a Windows NT Host system, this version downloads and debugs iRMX applications remotely over either a TCP/IP or serial connection. You invoke it using its icon (which points to sswin32.exe).

You do not have to deal with the details of the CPU’s machine code, or with the inner workings of the iRMX OS. Features of the Soft-Scope debugger include:

- All the features of SDB and SDM
- Full-screen windowed display, mouse- or keyboard-activated menu and dialog boxes (Windows version)
- Source code interface and online listings
- Access to program variables by source-code name, including arrays, structures, and bit fields
- High-level breakpoints, execution breakpoints in ROM, and access to data breakpoints
- Disassembly of instructions
- Second terminal option for remote debugging

- Unlimited size of source files and number of symbols
- Ability to create C-like macros and C-like expression syntax within commands
- Run-time exception handling
- Ability to suspend and resume tasks
- Full support for the protection features of the microprocessor, including automatic trapping of protection exceptions

See also: *Soft-Scope Debugger User's Guide*: the first manual bound into this volume describes Soft-scope for Windows; the second manual describes Soft-Scope III for the iRMX command line

❑❑❑

System Configuration 10

Configuration really means two things:

- Setting up your development system to include the programmers and terminals in your work environment
- Modifying the iRMX OS as your application requires, before installing it on your application systems

See also: Configuring users and terminals in *System Configuration and Administration* for information about configuring the development environment

You can use the Interactive Configuration Utility (ICU) and/or loadable jobs and drivers to modify your OS configuration.

ICU Configuration

You can configure any iRMX OS with the ICU. In an DOSRMX or iRMX for PCs installation you must specify a generation environment if you want to include the ICU. The iRMX III OS always includes an ICU and the underlying generation tools. The ICU creates a custom system for your application. It enables you to choose and modify the parts of the OS you need.

The OS installation contains preconfigured bootable images of the iRMX OS that you can use. The ICU definition files used to create these images are also included. Using the definition files provides these advantages:

- You don't have to make hardware or software changes to install the OS on Intel's System 310, 320, and 520 microcomputers.
- The start-up systems use the most current, complete, and accurate version of the OS.
- You can start using the OS immediately, and perhaps even run your application software, without redefining or reconfiguring the OS.

The advantages of using the ICU include:

- You can configure application systems, even complex systems, relatively easily. The ICU displays a series of menus, each describing a number of features. You can accept the default or change the value for each feature.
- The choices you make during configuration are saved in a *definition* file. You can use this file later as a base when you need to change your configuration.

See also: *ICU User's Guide and Quick Reference* for details on using the ICU screens and for a list of definition files

iRMX for PCs and DOSRMX Configuration

At installation, the system asks questions about the system bus type, networking, etc. Then it sets up an initial configuration.

You can further configure iRMX for PCs and DOSRMX by modifying an initialization file, *rmx.ini*. With this file, you can modify parts of the OS, although you cannot exclude layers.

During initialization, each layer reads a block of entries from this file. The values you enter here override the default values shipped with the software.

See also: *rmx.ini* in *System Configuration and Administration*

You can also use loadable jobs and drivers to modify your configuration.

Loadable Jobs

Loadable jobs enable you to add drivers, networking, the C library, and your application to the OS either during initialization or dynamically while the system is running. Loading jobs dynamically reduces the size of the boot image and can help conserve memory if you remove jobs when you no longer need them. You can also load custom device and file drivers you have written.

Typically you load jobs and drivers with the HI **sysload** command in the *loadinfo* file when the system initializes. Loaded jobs become a part of the OS and remain a part of the system until you explicitly delete them or reboot the system. The *loadinfo* file is one of the files in the *:config:* directory.

See also: Loadable jobs, **sysload** command, *Command Reference*;
Loadable jobs and drivers, *System Configuration and Administration*;
Writing drivers, *Driver Programming Concepts*

❑❑❑

Related Publications A

iRMX Manual Set

Startup Manuals

- *Installation and Startup*
- *Introducing the iRMX Operating Systems*

Programming Concepts Manuals

- *iRMX System Concepts*
- *iRMX Driver Programming Concepts*
- *iRMX Network User's Guide and Reference*
- *TCP/IP and NFS for the iRMX Operating System*
- *Programming Concepts for DOS*
- *iRMX Programming Techniques and Aedit Text Editor*
- *Peripheral Controller Interface (PCI) Server*
- *Real-Time and Systems Programming for PCs,* by Christopher Vickery

Reference Manuals

- *iRMX Command Reference* and *iRMX Quick Reference to Commands*
- *iRMX System Call Reference*
- *iRMX C Library Reference*
- *iRMX System Debugger Reference*
- *iRMX Master Index*

Configuration Manuals

- *iRMX System Configuration and Administration*
- *MSA for the iRMX Operating System*
- *ICU User's Guide and Quick Reference*

Tools Manuals

- *ASM386 Macro Assembler Operating Instructions* and *ASM386 Assembly Language Reference Manual*
- *iC-386 Compiler User's Guide*
- *Intel386 Family Utilities User's Guide*
- *PL/M-386 Programmer's Guide*
- *Soft-Scope Debugger User’s Guide*

❑❑❑

Glossary

absolute address

The physical address that is permanently assigned to a storage location in memory.

access control

Controlling a user's access to perform selected operations, such as reading or changing, on an object, file, or directory.

access rights

The bit settings that determine a user's permission to perform operations on an object, file, or directory.

AL

Application Loader loads programs into memory and executes them from an application program.

alias

A symbolic name for an object, file, directory, command, etc.

ANSI

American National Standards Institute.

application system

The set of components needed to solve an application problem: your program, other software, and hardware.

asynchronous

Non-synchronous timing. A method in which signals between networked systems are not timed; sending data a character at a time without prior arrangement. An event or device that is not synchronous with CPU timing or another device's timing. See synchronous.

asynchronous system call

Can run concurrently with the calling task. See synchronous system call.

attributes

The set of characteristics and properties that define a given object type.

AU

Administrative Unit. An iRMX-NET concept that defines a logical grouping of systems in a network. The systems that share a common set of users.

background process

A command or program that runs without interaction with the operator, and allows the operator to enter other commands while it is running.

BIND

Linking object modules using the BND386 utility.

binding

Letting each task know the locations of the variables and procedures that it uses.

BIOS

The Basic I/O System layer of the iRMX OS. This is different from the ROM BIOS stored in ROM on a DOS system.

blocking

Two meanings: reading or writing a file in sector-size blocks; a system call waiting at an exchange until a necessary resource or object is available.

boot

To use a bootstrap loader. This term is generally used to describe starting a computer system.

boot client

The system that requests a remote boot from the remote boot server. Typically, this system does not have mass storage provided by a hard disk or diskette drive. A diskless system or workstation.

bootloadable

A program with absolute addresses instead of relocatable addresses.

bootstrap

Starting a computer, which usually clears memory, sets up I/O devices, and loads the OS.

bootstrap loader

A program that resides in ROM. When the system is reset, the bootstrap loader receives control, and loads the OS and application software into RAM.

buffer

A temporary holding area for memory segments; used for reading and writing data.

buffer pool

A collection of preallocated buffers that provides quick access to reusable memory.

buffered device

An intelligent communications device that has its own CPU. It manages its own character buffers separately from those managed by the Terminal Support Code or Random Access Support Code. See non-buffered device.

cache

A high-speed buffer memory used between the CPU and main memory. Instructions and programs can operate at higher speed if they are in the cache.

call gates

Redirect flow within a task from one code segment to another. Used to enter the iRMX OS and OS extensions.

CDF

Client Definition File. Contains the names and passwords of client systems in a network's Administrative Unit.

channel

A data path.

checksum field

A field in the fnode file used to verify disk integrity.

child job

A job created by another job, called the parent job. Child jobs obtain their resources, such as memory, from their parent job.

:ci:

Standard logical name for the terminal keyboard, or console input. Each user's *:ci:* refers to the terminal associated with that user.

CLI

Command Line Interpreter, the default initial program; includes commands with optional parameters.

client, network

A network system that requests and uses resources located at another (remote) network system. Intel's transport protocol is based on the client-server model, in which client jobs request data from server jobs, and server jobs respond to the requests. Sometimes called consumer.

:co:

Standard logical name for the terminal screen, or console output. Each user's *:co:* refers to the terminal associated with that user.

command line parsing

Retrieving and interpreting the parameters of a command.

COMMengine

A networking hardware environment that uses separate boards for the host CPU and LAN controller. The iNA Transport Software runs on the LAN controller and the iRMX-NET runs on the host CPU with iRMX OS.

COMMputer

A single-board computer with on-board integrated networking hardware. The COMMputer hosts iRMX-NET, iNA 960, and the iRMX OS.

composite object

An object of a new type designated by an extension object.

concurrent condition code

A condition code that is returned as a result of asynchronous processing.

condition code

A message returned when an error occurs during execution of a program or system. Same as exception or error code.

configuration

Using the ICU to change attributes about the iRMX III OS. Also loading and modifying *:config:*, *rmx.ini*, and *loadinfo* files in iRMX for PCs and DOSRMX.

connection

An object, returned by the I/O system whenever a file is created, that represents the bond between a device or file and a program.

console

A specific terminal attached to a system that is used to invoke the Bootstrap Loader.

CPU

The central processing unit of a computer: the module in charge of receiving, decoding, and executing instructions.

CPU trap
See hardware exception.

current directory
The iRMX directory that acts as the default when you specify a filename without a preceding pathname. The current directory always has the logical name *:$:*.

datagram
A connectionless message-delivery mechanism that does not guarantee delivery or the order of delivery.

data file
A file containing programs and data. See directory file.

deadlock
The impasse resulting when two or more tasks each hold exclusive access to resources needed by the other task(s).

dedicated server
A network system used exclusively to provide resources to client systems.

default prefix
A prefix ID used by default whenever a null prefix is presented to the I/O system.

descriptor
An entry in a descriptor table that contains the physical address, length, and other information about an object, which is viewed by the Nucleus as the token for the object. It is assigned by the iRMX OS when an object is created.

descriptor table
A hardware-defined table that contains descriptors, which point to memory. There are three kinds: a global descriptor table (GDT), one or more local descriptor tables (LDT), and an interrupt descriptor table (IDT).

device
Hardware connected to the computer system that is used for reading and writing data, such as terminals, printers, plotters, display tubes, and robots.

device driver
Software that controls device operation, and provides a system-defined, device-independent interface to the device. Device drivers are implemented at the BIOS level.

directory file
The type of file that contains the disk addresses of associated data files and other directory files. See data file.

download
Sending data from a server system to a workstation or end system.

DUIB
Device Unit Information Block, a collection of information about a device unit (a device and a controller) that includes its name, granularity, and addresses of device driver routines.

dynamic logon terminal
A terminal configured to service many different operators on a request-by-request basis. Users log on to the system with a name and password. See static logon terminal.

dynamic memory allocation
Memory that is allocated to jobs only when tasks request it. This enables jobs to share memory and change the amount of memory they use as their needs change, using less memory overall. See static memory allocation.

dynamic user
A Human Interface user created by entering a name and password on a dynamic logon system. The user must be defined in the UDF prior to being created at logon.

EIOS
The Extended I/O System.

end point
The system at either end of a network communication connection. A network system or node. Also called an end system.

end system
See end point.

environment
The general operating characteristics that are imposed on a computer system.

error code
Same as a condition code.

escape sequence
Characters preceded by an ESC character.

event
A system state change.

exception code
Same as a condition code.

exception handler
A procedure that corrects certain exceptional conditions, or deletes or suspends the job that caused the error.

exchange object
A class of objects used to aid communication, synchronization, and mutual exclusion between tasks. See mailboxes, ports, regions, and semaphores.

extension object
Designates a new type of object. See composite object.

FIFO
First In, First Out order of operation, meaning that the least recent item added is the first one removed.

file consumer
The iRMX-NET software module that enables a local user to transparently access remote files.

file driver
Software that the BIOS uses to control file operation. There is a driver for each of the types of files: named, physical, stream, remote, DOS, and EDOS. There are also loadable file drivers, such as NFS.

file pointer
An indicator that marks a connection's current position in a file. The next sequential read or write starts at the pointer. When a file is opened, the pointer is at the beginning of the file.

file server
The iRMX-NET software module that receives requests from remote users.

file system
A complete hierarchy of logically related files, including a root directory.

file tree
A hierarchical file structure that reflects the relationships between files.

file types
There are these types of files in the iRMX OS: named, physical, stream, remote, and DOS.

firmware
Software that is permanently fixed onto a memory chip (ROM).

first level jobs
Children of the root job: some of the OS layers are first level jobs, and applications can also be first level jobs.

fixed updating
Updating done at the same time interval for all devices, with the interval being independent of I/O activity. See timeout updating.

fnode file
The file descriptor node file, a file in the BIOS that stores information about named files, such as the file name, location, creation and last modification dates.

Gbyte
Gigabyte

GDT
Global Descriptor Table, the system-wide table containing descriptors that are shared among all jobs in the system.

generation output file
A file containing the results of the ICU generate command, which generates an iRMX system. The file has an *.out* extension.

global
A programming reference that means the same thing throughout the entire program.

global job
An interactive job or user session.

global object directory
An object directory, found in each global job, that stores objects (including logical names) for that job. These objects remain valid for the life of the user job or until they are detached, and other users do not have access to them.

granule
A block of allocated space on a mass storage device.

granularity
The size of a block of allocated space on a mass storage device.

handshaking
Signals that are used to synchronize communications equipment during the set-up period.

hardware
The physical equipment of a computer system.

hardware exception
An error that occurs as the result of a hardware protection feature.

hardware interrupt
The point at which external processes enter the computer. In the iRMX OS, the device that handles hardware interrupts is the 8259A Programmable Interrupt Controller (PIC).

HI
Human Interface, which performs logon and logoff functions, creates jobs, assigns memory, and starts initial programs.

home directory
The directory you automatically enter when you log on to the iRMX OS. The system manager assigns your home directory when initially setting up your account. It has the logical name *:home:*.

host
The CPU board in a computer.

ICU
Interactive Configuration Utility. A screen-oriented utility to help build the OS configuration you want.

IDT
Interrupt Descriptor Table, the system-wide table that contains descriptors for the system's interrupt handlers.

initial task
The first task to execute after creation of a job. Its sole purpose is to initialize the environment for the new job.

interoperability
The ability of iRMX-NET to share files with other systems besides the iRMX OS.

interrupt handler
A procedure that is invoked by hardware to respond to an external asynchronous event (an interrupt). The handler decides how important the interrupt is and either returns to the original task or invokes an interrupt task.

interrupt task
A task that runs when a specific interrupt occurs.

I/O job
A job that is a child of the EIOS rather than the Nucleus. I/O jobs can use EIOS system calls.

IORS
I/O Request/Result Segment, a device driver data structure created to record and control the action taken for each I/O request.

I/O system
The layer of an OS that provides input and output functions. The iRMX OS has two: the BIOS and the EIOS.

iRMX
Intel's Real-time Multitasking Executive, the OS.

iRMX-NET
Intel networking software that provides transparent file access between systems.

ISO
The International Organization for Standardization.

job
One or more tasks and the resources they need (objects, an object directory, and a memory pool).

Kbyte

Kilobyte.

LAN

Local Area Network. An in-house data communications system that connects a number of independent devices.

LDT

Local Descriptor Table, a table that stores descriptors and is managed by the iRMX OS.

LIFO

Last In, First Out order of operation, meaning that the last item added is the first one removed.

local

In networking, the specific environment that is directly controlled by a given computer, such as disk drives and printers attached to a system. Local also refers to a user's own system and files, as opposed to those available across a network.

local environment

The execution environment for a set of tasks. Same as job.

local object directory

When you invoke a command, the OS creates a job and a local object directory for that command. The objects cataloged in this directory can only be used in the context of this job, and they remain valid only until the job exits or is deleted.

local node

The node a user is logged into is the local node. All other nodes are remote nodes.

logical name

An identifier (a string of characters, usually bounded on both ends by colons) for a file, directory, device, or remote computer system that the EIOS associates with a particular file connection or device connection. May be either local to a job or global across all jobs.

LRS

Loader Result Segment. Records the action taken by Application Loader system calls.

mailbox

An object that a task uses to exchange objects, tokens, or information with other tasks. There are two kinds: data mailboxes and message mailboxes.

Mbyte

Megabyte.

media

The physical parts that store or transport data, such as a CD-ROM, or the interconnection between devices attached to a LAN (broadband coax, twisted pair, and fiber optics, etc.).

memory pool

A configurable amount of memory allocated to a job and its children.

MIP

The software module that provides an interface between the local CPU board and iNA 960 Transport Software operating on a separate network interface controller (NIC).

Multibus
Two bus standards (Multibus I and Multibus II) designed and supported by Intel for multiprocessor systems.

multiplex
To use one structure for more than one function.

multiprogramming
A technique used to independently run several unrelated applications on a single application system.

multitasking
A type of system that supports the execution of multiple tasks, each of which needs control of the processor to run.

multiuser
A type of system that enables multiple users to log in and perform work as if each were the only user on the system.

mutual exclusion
A means of allowing only one task to have access to a shared resource at any given time.

Name Server
The iRMX-NET software module that provides the network directory service for local and remote users.

network
A group of independent computer systems that are interconnected for communication.

network object
A resource that can be accessed over a network by clients, such as file servers, print servers, or virtual terminal servers.

NFS
Network File Support. NFS enables hosts to share their local resources with remote hosts (clients) in a manner that hides the heterogeneous nature of a network. For example, a server running the iRMX OS may share a specific directory with a client machine running the Unix OS. The client can access the directory using commands and calls that appear to be directed at local resources.

NIC
Network Interface Controller.

node
A computer system functioning as the end point in a network. Each node is identified by a network address.

non-buffered device
A communications device that must be managed by the host processor board. The Terminal Support Code on the host processor board must manage the character buffers of the non-buffered device. See buffered device.

nonresident user
A user defined either in the system configuration files or with the **password** command. See resident user.

Nucleus
The basic layer and computational heart of the iRMX OS.

object code
Output of a BIND (linker) command or of a compiler such as C, PL/M or ASM.

object directory

A place in memory where a task can catalog an object under an ASCII name, which can then be used for access instead of the object's token.

object file

A file that contains the binary object code that results from the compilation of a program or procedure.

object module

The output of a single compilation, a single assembly, or a single invocation of a BIND command.

objects

Data structures and the operations performed on them. Objects are system building blocks, and include:

- Segments
- Semaphores
- Jobs, I/O jobs
- Tasks
- Buffer pools
- Connections
- Mailboxes
- Regions
- Extension objects
- Composite objects
- Ports
- Users

object-based

A concept that focuses on data structures and the actions performed on them.

OMF

Object Module Format, the format of linkable modules.

operating system

The software that manages the hardware and logical resources of a system, including device handling, scheduling, and file management.

OS extension

Operating system extension, a way to add custom functions to a system to meet the needs of the design.

OSC

Operating System Command: in the context of a terminal driver, a sequence of characters used by an application task or the operator to communicate with the Terminal Support Code.

OSI Reference Model

Open Systems Interconnection Reference Model, a model that defines network architecture with seven layers:

1) Physical
2) Data Link
3) Network
4) Transport
5) Session
6) Presentation
7) Application

overlays

Logically independent subsections of a program, which can be loaded one at a time in the same block of memory by the AL.

owner

The user ID associated with a file.

packet

A group of data bits and control elements that are transmitted across a network as a composite whole.

parameter

A variable that can be assigned a constant value for a specific function.

parent directory

The file directory immediately above the current directory.

parsing a command

Retrieving and interpreting the parameters of a command.

partition

An area on a block device such as a disk or tape.

pathname

The designation used by the OS to find or specify a file or directory.

peer

Equivalent computer systems, software modules, or protocol layers.

peer-to-peer resource sharing

In networking, an organization where all the nodes can provide resources for each other while also running local applications, so each one is both a server and a client.

portability

Used to describe language processors and software tools that can run on several OSs, often because they use UDI system calls.

port

An object that can send messages to or receive messages from other processors on the same bus. Ports enable message addressing to a given task, and are usually used for synchronization.

private

Files, accessed locally, that are not available to remote users across a network.

priority

A number used for scheduling a task relative to other tasks. Priorities range from 0 through 255, with 0 being the highest priority and 255 the lowest.

program state

The registers and data used by a program. If the program state is saved during task switching, the OS can restart the program later.

PROM

Programmable Read-Only Memory. A memory device in which information can be changed after manufacture, but is then permanent.

protected mode

See PVAM.

protocol

Rules for network communications between equivalent (peer) layers in regard to the format and content of the messages exchanged.

public

Files that are available for access by remote users on a network.

PVAM

Protected virtual address mode: microprocessor memory management feature that translates virtual addresses to physical memory addresses. It supports 4 Gbytes of physical memory. It also protects the OS from unauthorized modification by application programs, and isolates each user from other users. See real address mode.

read-ahead

A method of overlapping I/O operations so that tasks can continue running while the EIOS is transferring information to or from devices. See write-behind.

real address mode

The method of execution on an Intel386 or later microprocessor that supports 1 Mbyte of physical memory in RAM or ROM. The iRMX OS does not run in real address mode except for a short time when the system boots up. DOSRMX switches to Virtual 86 mode, a form of real mode, to run DOS and its applications. See PVAM.

rebooting

Resetting the processor without cutting power. Only the contents of static memory remain valid after rebooting.

recovery resident user

See resident user.

region

An object that controls access to critical areas, such as a collection of shared data. A region has special deletion and suspension features. Use regions in cases where a section of data must be read and written completely by one task before another can access it.

remote

In networking, an environment that is not local, or directly controlled by a given computer. For example, disk drives and printers that are attached to another network system are remote disk drives and printers. See local.

remote node

In a network, a node other than the local node.

resident user

A user defined in the Human Interface configuration files, which gains control only if an initialization error occurs in the configuration files. Same as recovery resident user. See nonresident user.

resources

The data and devices on a server that may be accessed by a client.

response time

The time it takes between the occurrence of an event or interrupt and the system's response to it.

RFD

Remote File Driver is part of the BIOS subsystem and closely parallels the Named File Driver of the local OS.

root directory

The topmost directory in a hierarchical file system.

root module

An object module that controls the loading of overlays.

root object directory

An object directory for the root job, containing logical names for devices. These objects remain valid until they are detached or the system is reinitialized, and every user has access.

round-robin scheduling

A priority-based time-slicing system of scheduling tasks for processing. Multiple tasks of equal priority are each allotted the same amount of execution time, and alternate running until finished.

run-time linking
Letting each program know the locations of the variables and procedures that it uses while the system is actually running.

SBC
Single Board Computer.

SBX
Single Board Expansion Module.

SDM
System Debug Monitor, a software debugging tool.

sector
A device granule (block of allocated space) for disk media.

segment
A contiguous unit of memory addressed by a descriptor. Tasks use segments for purposes such as stacks, data storage, and buffers.

semaphore
An object that is a counter, and provides a very fast method of task synchronization, or can be used for mutual exclusion.

server, network
A network system that responds to and provides the resources that are requested and accessed by a client system.

session
In networking, a point-to-point (virtual-circuit) connection between peer systems.

spokesman
In networking, the system that contains the names and addresses of other systems.

stack
An ordered collection of items in memory, into which new items may be inserted or removed. When a program makes a call, data is passed or stored on the program’s stack. Stacks are LIFO (last in, first out) meaning that the most recent item added is the first one removed.

start-up systems
Bootable images of the iRMX OS that are ready to run.

static logon terminal
A terminal configured to service one specific user. The logon is invisible to the user. See dynamic logon terminal.

static memory allocation
A memory-allocation method in which memory is allocated to jobs when the system is started, and cannot be freed for other jobs. Thus, the total memory requirement of the system is always the sum of the memory requirements of all jobs. See dynamic memory allocation.

static user
A user that comes up automatically when a static logon system is booted.

stream file
A temporary stream of bytes in memory, which is read on a FIFO basis and destroyed after reading.

string
An iRMX string is a character string consisting of 1+*n* consecutive bytes. The first byte contains the character count. The following *n* bytes contain the ASCII codes for the characters.
This is different from a C string, which is null terminated: there is no count byte and the string ends at the first byte containing zero.

subnetwork
Synonym for Administrative Unit (AU) used by UNIX and other OSs.

subtree
All the data files and nested directories contained in a directory.

Super user
Usually the system administrator; has a user ID of 0 and access to all files and devices on the system.

SXM
System Extension Module.

synchronization
A programming technique that enables dependent tasks to take turns in their use of shared data or system resources.

synchronous
At precisely the same time. An event or device that is in time with the CPU or another device. In networking, a method by which signals between systems are timed, with a pre-arranged number of bits per second being sent across the communications line. See asynchronous.

synchronous system call
A call that must complete before control of the computer is returned to the calling task. See asynchronous system call.

system
All the hardware and software components of a given computer.

system call
A programmatic interface used to manipulate objects or control the computer's actions.

system manager
The Super user, with an ID of 0, who defines users and systems within an Administrative Unit.

task
System activities that execute instructions and manipulate data. Can be viewed as a simple program that appears to be running on a computer by itself.

task priority
A value from 0 through 255, with 0 being the highest priority. Used in task scheduling along with the task state.

task switching
Temporarily stopping one task and running another. The registers and data (called the program state) of the first task are saved, and it can be resumed later at the same point at which it was interrupted.

TCP
Transmission Control Protocol. A transport layer protocol for the Internet. It is a connection-oriented, stream protocol defined by RFC 793.

TCP/IP

Transmission Control Protocol/Internet Protocol. A set of computer networking protocols and applications that enables two or more hosts to communicate. TCP/IP includes a suite of protocols besides TCP and IP; it has been widely adopted as a networking standard.

Telnet

A TCP/IP protocol used for remote login between hosts.

Terminal Support Code (TSC)

A set of commands that control terminal modes and operation.

timeout updating

Updating done at intervals set individually for each device, with the interval beginning at the end of each I/O operation. See fixed updating.

time-slicing

A non-priority-based system of scheduling tasks for processing. Multiple tasks are each allotted the same amount of execution time, and alternate running until finished.

token

A value representing the logical address and the characteristics of an object. See descriptor.

top-down programming

A programming concept that focuses on control flow, in contrast to object-based programming, which focuses on data structures and the processes performed on them. See object-based programming.

transparency

Remote file access that enables the user application to manipulate remote files as if they were local.

trap

See hardware exception.

TSC

Terminal Support Code, a set of commands that control terminal modes and operation.

UA

User Administration. The iRMX-NET software module that maintains the files used by a system manager when making additions and deletions of users and systems in an iRMX-NET environment.

UDF

User Definition File. An iRMX OS file that contains information about valid users of a system. Used by the server system in maintaining the server-based protection scheme.

UDI

Universal Development Interface acts as an interface between the OS and the application program.

user

Used by iRMX to determine access rights to files and systems. A user job is created when an operator logs onto a system to obtain access to the system.

user job

A child job of the HI.

verified client

A client system verified by a server as a node entitled to access the server.

verified user

A dynamically logged-on user. Verification only has meaning when the user is attempting to access remote files using a network.

Virtual 86 mode

A form of real mode used by DOSRMX to run DOS and its applications.

virtual circuit

A reliable, connection-oriented message delivery service. A connection through the Transport Layer of the OSI Reference Model that delivers error-free, point-to-point messages in the same sequence as the messages are sent.

virtual root directory

The root directory of a remote server system, as seen from the client system.

VM86 Dispatcher

Allows DOS to run as a task under DOSRMX.

volume

The medium used to store information on a device, such as a diskette, tape reel, or hard disk.

wildcard character

A character that can substitute for any single character (typically ?) or any sequence of characters (typically *), providing the ability to specify several files in a single reference.

write-behind

A method of overlapping I/O operations that enables tasks to continue running while the EIOS is transferring information to or from devices. See read-ahead.

❑❑❑

Index