intel® McGraw-Hill

REAL-TIME AND SYSTEMS PROGRAMMING FOR PCS

USING THE iRMX® FOR WINDOWS* OPERATING SYSTEM

Christopher Vickery

Real-Time and Systems Programming for PCs

Real-Time and Systems Programming for PCs

Using the iRMX® for Windows® Operating System

Christopher Vickery

TAB Books
Division of McGraw-Hill, Inc.
Blue Ridge Summit, PA 17294-0850

Library of Congress Cataloging-in-Publication Data

Vickery, Christopher.
Real-time systems programming for PCs : using the iRMX for Windows operating system / by Christopher Vickery.
p. cm.
Includes index.
ISBN 0-07-067466-3 (pbk.)
1. Real-time programming. 2. Operating systems (Computers)
3. iRMX for Windows. 4. Microcomputers—programming. I. Title.
QA76.54.V53 1993
005.4'469—dc20 92-42856
CIP

1 2 3 4 5 6 7 8 9 0 DOC/DOC 9 9 8 7 6 5 4 3

ISBN 0-07-067466-3

The editors for this book were Gerald Papke and Marianne Krcma, and the production supervisor was Katherine G. Brown. This book was set in ITS Century Light.

Printed and bound by R.R. Donnelley, Crawfordsville, Va.

Contents

Introduction

Historically, little support and little discipline existed for either real-time or systems programming on the PC platform. With the advent of more powerful PC computers, however, PCs have become the development target for increasingly ambitious applications. With the arrival of iRMX for Windows, large real-time systems integrated with DOS and/or Microsoft Windows are included in the set of possible applications that can be developed for PCs, the most ubiquitous computing platform available.

There is still little support or discipline for DOS systems programmers, but the situation is much better for iRMX. As a real-time operating system, iRMX has traditionally provided its application developers with a set of coordinated resources that systems programmers have traditionally needed, but had to do without. These resources include the management of memory, concurrency, interrupts, and peripheral devices. As an object-based operating system, iRMX provides these resources in an integrated and protected fashion that allows programmers to develop fast, robust systems. The reason a single book can cover both real-time and systems programming is that iRMX provides real-time developers with the same facilities the systems programmers used at Intel to develop iRMX itself.

Although most readers of this book will undoubtedly be real-time developers, specifically those who work with iRMX, this book's origin is actually academic. Computer science curricula at both the undergraduate and graduate levels typically include a course on operating systems principles. Such a course is often a reading course that covers the traditional topics of resource management and concurrency. Actual software development, if any, is often limited to simulations due to the lack of suitable laboratory facilities for true systems programming. For several years, I have taught operating systems laboratory courses that use relatively inexpensive computer systems running iRMX. The courses, which are offered to upper-level undergraduates or graduate-level students, have a traditional operating systems principles course that used texts such as Deitel (1990) or Milenkovic (1992) as a prerequisite. The goal of the laboratory courses has been to use hands-on experience to provide solid mastery of the principles covered in other operating systems courses.

Two other approaches could be taken to obtain this laboratory experience. One would be to develop systems code from scratch. For example, Wirth's first Modula programming language provided all the constructs needed to build a complete operating system with the addition of only a 98-byte runtime kernel. This approach certainly allows the student to deal with all the fundamental issues of systems programming, but it makes it very difficult to deal with the levels of complexity encountered in real systems.

The second approach is to study and modify an existing operating system. The Mt. Xinu (Comer & Fossum, 1988) and Minix (Tannenbaum, 1987) projects take this approach by providing the source code for Unix®-like systems for students to work with. Another way to use this approach is to let the student modify and extend a real operating system. Unfortunately, "real" systems are synonymous with "proprietary" systems, which means that source code and the tools for working at the systems programming level are not normally available to students. Although Unix is a proprietary system, it is often used as a basis for operating system laboratory courses because AT&T has made source code licenses relatively accessible to universities. The list of books that can be used for Unix laboratory courses is extensive, and includes Andleigh (1990), Bach (1986), Kernighan and Pike (1984), Leffler *et al.* (1989), and Rochkind (1985), among others.

Enter real-time systems, the category of applications that must be both logically and temporally correct. Real-time applications are event driven: asynchronous external events trigger computation sequences, which must be completed before temporal deadlines pass in order for the application to operate correctly. Real-time application programs must deal explicitly with exactly the same issues as systems programs, namely concurrency and resource management.

The premise of this book is that a commercially available development environment for real-time applications provides an excellent laboratory vehicle for studying systems programming. Of course, using such an environment also provides the student with a working knowledge of real-time systems in general and with the chosen development environment in particular. In addition, *Real-Time and Systems Programming for PCs* is a practical laboratory guide to systems programming concepts and techniques.

A number of real-time systems are available commercially. These fall into the two broad categories of kernels and operating systems. Kernels provide support for concurrency control and resource management, but do not provide complete operating system functionality such as a full I/O system, networking functions, and the like. To use a real-time system to study systems programming, it is better to choose a real-time operating system over a kernel not only because an operating system provides a complete set

of operating system facilities to investigate, but also because a single platform can be used for both development and testing.

From an academic viewpoint, Intel's iRMX for Windows operating system is a particularly attractive vehicle for an operating system laboratory. One reason is that the operating system runs on relatively inexpensive PC platforms. Another reason is that Intel has historically provided very good support for universities choosing to use iRMX in their courses.

A number of alternative real-time systems can be chosen for an operating systems laboratory. Particularly interesting are several systems that run on different manufacturers' processors (not just Intel's), and some newly emerging systems based on the POSIX 1003.4 real-time standard.

Real Time System Programming for PCs is based on my experience using iRMX to teach semester-long laboratory courses on systems programming at the graduate and advanced undergraduate level at Queens College. The backbone of the course has been a sequence of projects designed to illuminate various features of the operating system (OS). My students are already familiar with the principles of systems programming, so little time is spent explaining concepts of memory management, process scheduling, or concurrency control. Rather, the first project is usually a simple application designed to familiarize students with the iRMX development environment, which is significantly different from the sheltered environments used to provide instruction in computer science principles and applications programming. A second project typically involves developing a utility program that exposes the students to most of the resources and facilities of the OS. A third project concentrates on concurrency by developing either a program to demonstrate or exercise the multitasking features of the system or a device driver, including interrupt handlers and request block management. Other projects have involved networking utilities, library management, and source code preprocessing.

The book is logically divided into two parts. Part I is an overview of real-time and systems programming concepts, the use of an iRMX system and its development tools, and the architecture of Intel microprocessors. Chapter 1 introduces the fundamental concepts of real-time systems: determinacy, speed, and robustness. Chapter 2 is a guide to iRMX from a user's perspective: how to log on, how to use the file system, how to use development tools, and the like. Chapter 3 discusses the development process for iRMX applications. If the reader is familiar only with student compilers or fully integrated development environments, the material in this chapter is particularly important; otherwise you may need only to skim through the chapter to get some iRMX-specific details. The traditional development languages for iRMX have been assembler and PLM, a PL/I-like language developed by Intel specifically for use with its own microprocessors. With the emergence of C as the most commonly used language for both application and systems programming in other environments, C is rapidly replac-

ing PLM as the standard high-level language for iRMX programming. Chapter 4 investigates the language issues in developing iRMX code. An assumption throughout the remainder of the book is that the reader will be able to follow code written in either PLM or C.

The first part of the book ends with a chapter on the architecture of the Intel x86 microprocessor. This chapter on hardware is included in a book on software development simply because of the nature of both real-time and systems programs: their software comes in the most direct contact with the processor itself. Programmers must understand the underlying processor well in order to develop efficient and fast real-time or operating systems. It is not necessary to program in assembly language to do most real-time and systems programming tasks because both C and PLM can be used as effective high-level System Implementation Languages (SILs). This book, however, does include some assembly language code and many references to assembly language concepts. Chapter 5 is designed to prepare the reader to understand that material without actually covering assembly language programming.

The second part of the book covers the iRMX operating system itself. Although the book features the iRMX for Windows operating system, most of the material covered applies to other versions of the operating system as well. Readers interested in iRMX I, however, which operates in the processor's real mode, must remember that the book assumes a protected-mode version of the operating system (iRMX II, iRMX III, or iRMX for Windows) in much of the material presented in the second part. Some of the sample code also assumes a 32-bit version of the OS (iRMX III or iRMX for Windows). Finally, the sample code presented in the book has been tested only on an iRMX for Windows system. It may well work on other versions of the operating system, but is not guaranteed to do so.

Part II begins by introducing some fundamental concepts about iRMX in chapter 6. These concepts include the object-based nature of the system and a description of the three fundamental types of iRMX objects: jobs, tasks, and memory segments. Chapter 7 surveys many basic iRMX system calls, and chapter 8 deals specifically with the system calls used for I/O programming. Chapters 9 and 10 introduce two important facilities that iRMX provides for extending the operating system. Chapter 9 discusses the issue of adding device drivers, and chapter 10 covers the facilities available for adding new object types and system calls to the operating system itself.

Chapter 11 introduces the networking facilities provided with iRMX. The use of a network is integrated with the rest of the book, but this chapter specifically discusses the programming issues involved in interacting with the various parts of an iRMX network.

Finally, chapter 12 is devoted to those aspects of iRMX for Windows that are not present in other versions of the operating system. Some of these features, like console sharing, interrupt management, and file shar-

ing, are necessary to allow iRMX, DOS, and Windows to operate in an integrated, reliable fashion. Other features, such as run-time configuration and loadable operating system layers, are conveniences that were introduced with iRMX for Windows, but which may well be integrated with other versions of the operating system as well. Still other features, such as DDE networking support in particular, combine the individual features of iRMX and Windows in ways that extend well beyond the simple sum of two parts.

Acknowledgments

Intel supported the development of this book in a number of ways. Materially, Intel provided me with current versions of the software and documentation throughout the book's gestation period, as well as with an email account so that I could interact with people within the company. Intel also provided two technical reviewers for the manuscript, Krishnan Rajamani and Steve Snyder. Their comments were extremely valuable and constructive. More than one sentence in the book now says exactly the opposite of what it said before Krishnan or Steve saw it. The errors that remain, of course, are my own. There are others at Intel who provided help to me in various ways. Janet Brownstone coordinated the interactions between Intel and McGraw-Hill for this series of books. Rick Gerber provided answers to many of my questions. Bill Corwin provided me with some of the POSIX material. My thanks to all.

Then there is Paul Cohen. He first suggested this book to me, he brought the idea for the book to Intel, and he explained the merit of the book to McGraw-Hill as well. But Paul's contributions go far beyond the role of facilitator. Paul has been the chief guru to many iRMX users for years: always enthusiastic about iRMX, always patient with our questions (he gets excited sometimes, but he answers even dumb questions anyway), and always generous with first-rate technical information. I wrote the book, but it wouldn't have happened without Paul.

Real-Time and Systems Programming for PCs

Part

1

Basics

Chapter

1

Introduction to Real-Time and Systems Programming

1.1 Overview

iRMX, Intel's real-time operating system, is an excellent vehicle for studying systems programming. In fact, it is virtually impossible to develop a real-time system without doing systems programming. In turn, many crucial parts of a systems programmer's job deal with real-time issues.

This chapter introduces systems programming, real-time systems, and the iRMX operating system (OS) to provide a context for the remainder of the book, as well as to support the argument that real-time systems and systems programming have much in common. The first part of the chapter looks at the conventional view of systems programming, and the second part looks at real-time systems, including some of the features of iRMX that make it a real-time OS. Finally, we look at how iRMX compares with conventional operating systems such as MS-DOS and Unix, as well as alternatives to iRMX for real-time systems.

1.2 Systems Programming

To help put systems programming in perspective, consider the following hierarchy of programming classes: user, application, and systems. Real-time programming is included here as a parallel entity, spanning the range of both application and systems programming. The reason the left side is shown as a hierarchy is that each type of programming builds upon resources provided by the level below.

<table>
<tr><td>User Programming</td><td></td></tr>
<tr><td>Application Programming</td><td rowspan="2">Real-Time Programming</td></tr>
<tr><td>Systems Programming</td></tr>
</table>

User programming. User Programming refers to the types of things an end user might do to customize a particular application. Examples include spreadsheet and word processing macros and simple command files (batch files). The programming language used could be fairly primitive, perhaps just a matter of recording a sequence of keystrokes. Nevertheless, the programs implement some sort of algorithm and, thus, qualify as programming by almost any definition of the term, even if the user does not realize it. This category of programming can require a good amount of sophistication, and there are people who do user programming professionally.

Application programming. Application Programming is what most people think of when the term *programming* is used. It refers to the development of programs used by end users to perform tasks or sets of related tasks. Application programs range from spreadsheet and word processor programs to graphical modeling or scientific data analysis packages. These programs rely on an operating system to perform certain functions, such as controlling input/output (I/O) devices, but high-level programming languages, such as C and FORTRAN, often interpose a layer of software called a *run-time library* between applications and the OS to make applications portable across operating systems. Run-time libraries for two high-level languages commonly used for application programming with the iRMX operating system, PLM and C, are discussed in chapter 3.

Systems programming. There are really two types of programming that qualify as systems programming. One type is the construction of the operating system itself, and the other is the development of systems programs, which provides the tools that application programmers use in their work. In turn, system programs fall into two categories: development tools and utilities.

1.2.1 Constructing an operating system

An operating system serves two major functions. The first function is to provide application programmers with an abstract machine, a computer that is easier to program than the actual processor on which the OS is implemented. This function is normally provided through a set of subroutines referred to as *system calls* that any application program can invoke as needed. Although they are actually software routines, system calls serve conceptually as extensions to the hardware instruction set of the central processing unit (CPU).

The second function of the OS, which is normally closely integrated with the first, is to manage resources in a controlled way for the various applications running on the system. Resources that must be managed include primary memory, use of the CPU, and control of I/O devices. Resource management is integrated with the abstract machine in the sense that

application programs make system calls to access the resources managed by the OS.

The system calls provided by the OS relieve application programmers of the burden of rewriting the code for functions needed by many different applications. The code is written once by the systems programmer and is either always resident in primary memory as part of the OS, or is supplied as part of a library linked with just those applications that need it. Functions performed by system calls include allocating memory segments into which applications can store dynamically created data structures and routines to read and write data between an application's data buffers and peripheral devices. Both of these examples stress the necessity of functions incorporating resource management as an OS system call: two applications running at the same time must not interfere with each other's use of system resources, and the OS must provide the mechanisms for coordinating their activities. Furthermore, when exceptional conditions occur (such as one application attempting to access another application's private data segment), the OS also provides the code that responds to these conditions.

A major feature that distinguishes programming an operating system from most application programming is the need to manage *concurrent threads of execution.* In a single CPU system, the processor can execute only a single instruction at a time, but hardware devices (such as I/O controllers and the device that keeps track of the time) generate interrupt requests that are not generally synchronized with the processor's execution. The OS must manage the switch of CPU control to the routine that services an interrupt and then back to the application that was running when the interrupt occurred. It must also switch among the various applications that are ready to run at any particular moment. As you will see, managing concurrency is also a hallmark of real-time applications. This common feature of the two types of programming is the main reason this book claims to discuss both systems programming and real-time programming as it covers the iRMX real-time OS.

In addition to the issues of developing an abstract machine, managing resources, and dealing with concurrent threads of execution, an OS developer must decide how the code for the OS is to be structured. It is possible to create an OS as a single, monolithic piece of code, but this is not normally done except in the case of very simple systems. More likely, various subsystems, such as the memory manager, I/O system, or user interface, are coded as separate modules and linked together to build the OS itself. Adding parts to the OS or changing existing parts involves developing or altering the code for a module and then rebuilding the OS to include the changes.

The iRMX for Windows version of iRMX has the ability to change the OS's configuration while the OS is initializing and, to a lesser extent, while the OS is running. The features of iRMX for Windows that support this configuration process are covered in chapters 9 and 12. Another iRMX fa-

cility for building customized versions of the OS is called the *Interactive Configuration Utility* (ICU), which is also introduced in chapter 9. The ICU is not used with iRMX for Windows.

1.2.2 Developing development tools and utilities

Development tools include the compilers, linkers, loaders, and debugging programs that application programmers use to code and test their programs. Development tools are coded much like application programs themselves. That is, compilers, linkers, loaders, and debugging programs are developed using compilers, linkers, loaders, and debugging programs. What differentiates development tools from application programs is that development tools must be compatible with both each other and the operating system to generate other programs that can be executed. As a result, systems programmers producing development tools must generally know more about the structure of the underlying operating system than application programmers. Also, developers of development tools have historically not been as concerned with portability as application programmers.[1]

Utility programs are routines that make an application programmer's job easier but can also be useful to end users. Examples of utilities include basic file maintenance programs (list, copy, move, and delete files), text editors, and anything else someone deems useful. Examples of utilities for Unix are particularly numerous (*grep, sort, more*, etc.), and versions of many Unix tools have been ported to iRMX, DOS, and other operating systems. As this process suggests, many utilities either are, or can be made to be, portable across operating systems.

Although portability is not a general concern in this book for reasons that should be clear by the end of the chapter, be aware that it is a matter of utmost concern to many software consumers, and thus is extremely important to many software producers. Portable utility programs that fall into the systems program category must provide specific code for the different systems on which they will run. Which code will actually be executed must be selected at compile time, link time, or run time. These terms are discussed in more detail in chapter 3, which reviews the entire software development process.

The programming hierarchy shown at the beginning of this chapter shows real-time programming as a separate entity from user, application, and systems programming, one that parallels both the application and sys-

[1]For perspective, consider the Portable C Compiler available for early Unix systems. This compiler was written mostly in the C language and could be easily ported to different systems, thus providing a convenient tool for porting Unix itself to new systems. The availability of this portable development tool was partly responsible for the early rise in the popularity of Unix. However, the Portable C Compiler could never be as efficient as a compiler built specifically for a particular processor, and it was therefore replaced with more efficient, non-portable versions as soon as practical.

tems levels. Real-time systems are applications in the sense that there are end users for real-time systems just as there are end users for conventional applications like word processing and spreadsheet programs. In addition, developing real-time applications requires the use of systems programming techniques that go beyond those used to develop conventional applications. These techniques include explicitly managing resources such as the I/O system, primary memory, and the use of the CPU, and might go so far as to involve modifying or replacing OS modules or adding new system calls to the OS. The next section describes some of the important features of real-time systems that lead to this state of affairs.

1.3 Real-Time Systems

The defining characteristic of real-time systems is their need to meet deadlines, which are constraints on the amount of time the system is allowed for completing a computation or set of computations. Although real-time systems connote high speed, there is nothing preventing real-time systems from operating with deadlines measured in hours rather than fractions of a second.

To develop the concept of real-time systems more fully, you must look at the environment in which real-time systems normally operate and the structure of many real-time systems that operate in these environments. Section 1.3.4 discusses deadlines specifically in the context of task scheduling algorithms. Before looking at real-time applications, however, you should know that there are three ways in which the software for real-time systems can be structured: *monolithic*, *kernel-based*, and *OS-based*.

***Monolithic* systems.** Monolithic systems include all software for the system as a single block of code. This structure is usually practical only for very simple systems.

***Kernel-based* systems.** Kernel-based systems use a real-time kernel, available from a vendor or developed in-house, to manage such real-time entities as tasks and interrupts. The logic for the real-time application is coded separately from the kernel, and then linked with it to form the complete real-time system.

***OS-based* systems.** OS-based systems differ from kernel-based systems only in the range of functions provided by an OS compared to a kernel. A real-time OS provides normal OS functions (file system, user interface, etc.) in addition to the real-time functions supplied by a kernel. Some versions of iRMX, for example, are based on an internal real-time kernel called iRMK. Some versions of iRMX, including iRMX for Windows, allow real-time applications to access iRMK functions directly. This feature was added to the OS too late to be covered in this volume.

Although this book is concerned with a real-time OS (iRMX), the discussion of real-time concepts in this section generally applies to all three

types of real-time systems and includes some hardware topics that go beyond the scope of software structure as well.

1.3.1 Real-time and embedded systems

Real-time and embedded systems are practically identical. The choice of terms has more to do with what aspect of the system is being stressed than with different classes of systems. Embedded systems abound in everyday life, although the end users who come in contact with them seldom use the term. Just about any modern equipment has some form of automatic control, usually depending on a computer embedded within it to perform its control functions. Robots and sophisticated military weapons are obvious examples of devices with embedded systems, but many microwave ovens, automobiles, manufacturing tools, and laboratory instruments also use embedded microprocessors. Conventional computer systems often include embedded systems in addition to the main CPU to perform high-performance graphics processing or smart disk caching.

What characterizes embedded systems is that the end user does not interact with the system as a computer but as something else. The user interface to the embedded computer is perceived as the interface to the equipment being controlled rather than as a computer itself. Although a conventional keyboard, CRT, and pointing device might be used as the user interface for embedded systems, these interfaces often feature knobs, buttons, lights, and panel displays instead. Further, many embedded systems are self-contained and do not need any user interface other than a switch to turn them on or off.

Another feature of embedded systems is that they are typically dedicated systems. For example, the computer that controls your car's ignition does just that. It does not do word processing, spreadsheets, or games. The processor itself is often a general-purpose CPU, but the only code available to it is for the application at hand. There is no connection between your Nintendo's embedded computer and your microwave oven (yet!).

Embedded systems almost always operate with real-time constraints. They must meet deadlines and, thus, are real-time systems by definition. As in real life, a deadline is simply the time at which a piece of work must be completed. Also as in real life, the contingency for missing a deadline might range from minor inconvenience (for example, stay late at work to finish the job in real life; achieve slightly less than optimal fuel efficiency in an automotive embedded system) to major catastrophe (lose your job for not completing a report; stall the engine in the middle of avoiding a collision).

The term *soft real-time* refers to systems that can operate at a satisfactory level even if some deadlines are missed. The term *hard real-time* refers to systems that are considered to have failed if a deadline is missed. An example of a soft real-time embedded system might be a program that determines the amount of fuel to be delivered each time a cylinder fires in an en-

gine, but will use the value from the previous cycle if it misses its deadline. As long as too many deadlines are not missed, the engine will operate, but at less than its optimum performance. An example of a hard real-time system might be a robot that will walk off a cliff if its control system does not tell it to turn around soon enough.

1.3.2 The structure of real-time and embedded systems

To help you understand the nature of real-time systems, consider Figure 1.1, which represents the general structure of the software for an embedded application. Each block represents a separate thread of execution called a *task*, which are called *processes* in the general OS literature. Each task typically executes code that is structured like Figure 1.2: after some initialization, the task enters an endless loop in which it waits for an event to occur, processes the event when it does occur, and then returns to the top of the loop to await the next event. This type of processing is called an event loop, and is not unique to real-time systems. For example, graphical window systems are typically based on an event loop structure, where the events to be processed include keyboard presses, mouse clicks, and mouse motion reports.

As a task computes its response to an event, it might generate additional events to be processed by other tasks in the system. For example, a mouse motion report might result in a mouse-entered window event in a graphics system. These internal events are shown in Figure 1.1 as lines connecting the input tasks to the processing tasks and connecting the processing tasks to the output tasks. The figure shows the most general case, but a single task might very well combine input, processing, and output functions without using any internal events.

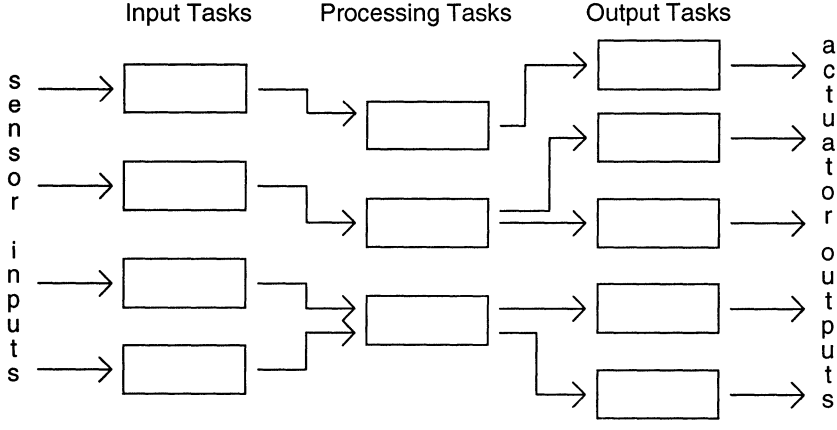


Figure 1.1 Task structure of embedded application.

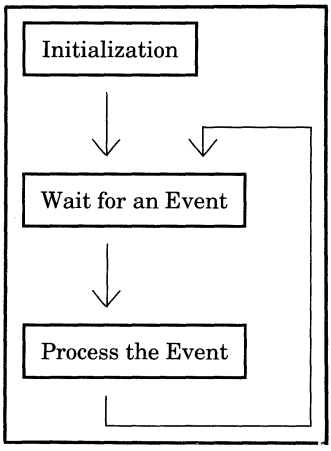


Figure 1.2 Code structure for a task.

The input events to an embedded system might come from conventional input devices but, as Figure 1.1 indicates, they might also come from sensors. Keyboards and mice are sensor input devices (they sense finger and hand movements), but embedded systems often receive their input from other types of sensors, such as a robot's visual sensors, or an automobile's air and engine temperature sensors. Another example of a real-time system that receives input events from a nonstandard input device is a stockbroker's program-trading system, which receives prices directly from the stock exchange and generates buy or sell orders in response.

Figure 1.1 also indicates that embedded systems can generate nontraditional outputs, such as the control signals that operate actuators, the motors to move the parts of a robot, or the valves to control a manufacturing process.

Nonstandard I/O devices are easily interfaced to computers so that they can be sensed and controlled in the same ways as traditional peripherals. On the other hand, real-time operating systems such as iRMX need to incorporate provisions for interfacing application software to these nonstandard devices while maintaining real-time performance. Techniques for doing this with iRMX are covered in chapter 9.

1.3.3 Factors affecting real-time performance

At one level, you can summarize the performance of a real-time system with one Boolean variable: either it meets its deadlines or it doesn't. Other important measures of a real-time system's performance are not discussed here, namely cost, fault-tolerance, and robustness. Rather, let us look at some of the secondary measures that contribute to the ability of a particu-

lar computer to meet real-time deadlines. These include determinism, speed, context switch time, and interrupt response time.

Up to now, I have used the terms *real-time system* and *embedded system* without making much distinction between the hardware and software that compose the system. This lack of distinction is appropriate, because a system as a whole relies on both hardware and software for successful operation. In fact, many functions can be implemented using hardware, firmware (microcode), or software techniques, whatever is most appropriate for the situation.

Hardware and firmware modules are typically faster than equivalent software routines but cost more to produce. Another way of looking at this issue is to remember that one crucial role of an operating system is to implement an abstract machine architecture on top of the real hardware, an abstract architecture that provides functions that match the needs of the system's applications more closely than the actual microprocessor's machine instructions. This abstract machine can conceptually be implemented in software, firmware, or hardware, or any combination.

A controversy in computer architecture exists that is relevant here. The controversy hinges on what level of abstraction is implemented in the processor's hardware or firmware. It is axiomatic that a more complex logic system must take more time to operate than a simpler one that uses the same circuit technology. The controversy is based on the notion that a system as a whole can execute faster by providing a very simple but very fast abstract machine in hardware, with software providing a more powerful abstract machine to the operating system user. Such processors are called *reduced instruction set computers*, or RISC processors. Processors that provide a more powerful abstract machine in hardware and firmware, such as the Intel microprocessors that are used to run iRMX, are called *complex instruction set computers*, or CISC processors. RISC processors presently enjoy a reputation for better performance than CISC processors using comparable fabrication technologies.

The RISC/CISC issue is relevant to the present discussion because a processor's average speed is often considered an important measure of how well-suited it is for real-time applications. As mentioned earlier, however, nothing prevents real-time systems from operating with deadlines measured in hours rather than fractions of a second. It's simply the existence of the deadlines that makes a system real-time. Nonetheless, it would seem likely that one processor that executes instructions faster than another would be more suitable for real-time systems. However, instruction execution rate is not necessarily a good measure of a processor's speed for two reasons.

Comparing the speeds of two processors with different instruction sets is an extremely difficult job to do, despite the variety of standard benchmark programs that claim to do so. The problem is that you must compare both the rate at which instructions are executed and the amount of useful work

done by each instruction. If a RISC processor executes its instructions twice as fast as a CISC processor, it must use no more than twice as many of those instructions to provide users with an abstract machine that is equivalent to the one provided by the CISC processor.

More important than raw computing speed for real-time performance is a computing system's *determinacy*, meaning how much variability exists in the time it takes a given computation to be performed. As an example, consider a real-time application that imposes a 1-millisecond deadline on the time a task is allowed to compute its response to some event. If Computer A can perform the computation in an average of 500 microseconds (half a millisecond) and Computer B requires an average of 550 microseconds to perform the same task, it is tempting to think that Computer A has a better real-time performance. But what if that 500-microsecond average consisted of 10 times that took 50 microseconds (Wow, look at that speed!) and one time that took 5000 microseconds (oops!) because the system's virtual memory manager had to swap in a page from disk for the task to complete? That 1 case in 11 trials is a missed real-time deadline, and Computer A could not be used for the real-time application. As long as Computer B's average is not based on any values greater than the 1-millisecond deadline, one would have to say that it is the better one (indeed, the only one) for the application.

Thus, the number of instructions executed per second and the average time to perform a computation are not the best measures of a processor's suitability for real-time applications. Two other measures of a processor's speed are often crucial in determining real-time performance, however. To complicate matters, these two measures are not purely dependent on the hardware being used but also on policies the OS uses in managing various resources.

The two measures are *context switch time* (CST) and *interrupt response time* (IRT). Context switch time is the time it takes the CPU to stop executing code for one task and start executing code for another task. This interval consists of three phases:

1. Recognizing the need to perform a context switch and selecting the next task to execute. This is the scheduling problem discussed in the next section.
2. Saving the state of the CPU's registers so that the current task can be resumed at a later time.
3. Loading the CPU's registers with the values needed to start execution of the new task.

The first phase is the responsibility of the OS's task scheduling software, and the other two phases depend, in part, on the microprocessor instructions that are available for saving and restoring CPU registers to and from primary memory. It is debatable whether the scheduling phase is really

part of a system's CST or a separate (important) measure of a real-time system's performance. The iRMX techniques for keeping CSTs small are covered in chapters 5 and 6, which discuss hardware and software issues respectively.

Interrupt response time is a measure of how much time elapses from the moment an I/O device indicates that it is ready to generate an event until the processor actually starts executing code in response to that event. At the hardware level, IRTs are limited by the constraint that processors recognize interrupt requests only between the execution of machine instructions. Many CISC processors include complex instructions that can take a long time to execute (a ratio of about 100:1 execution time for the slowest and fastest 8086 instructions exists, for example), which can significantly impact hard real-time designs, which must be based on worst-case values. (A dedicated hard real-time system would be coded to avoid use of the slowest instructions of the processor's repertoire.) However, IRT hardware considerations can easily be outweighed by the interrupt management policy of the operating system, because OS routines can totally disable the CPU's response to interrupts for arbitrarily long periods of time. Real-time operating systems minimize the time that interrupts are disabled as much as possible, even at the expense of a longer IRT average (or other measures of average system performance).

Most operating systems that support multiple threads of execution (not just real-time operating systems) reduce IRT by providing for two levels of software to be invoked by interrupts. For iRMX, these are called *interrupt handlers*, which execute in the same context as the currently running task (no CST), and *interrupt tasks*, which are scheduled for execution in competition with all other tasks in the system. Various interrupt hardware mechanisms are discussed in chapter 5, and iRMX interrupt handlers and interrupt tasks are covered in chapter 9. Rick Gerber of Intel has developed two programs that can be used to determine the IRT (*inttest*) and the CST (*switch*) of an iRMX system. They are available, along with all the code presented in this book, from the author.[2]

1.3.4 The scheduling problem

The *scheduling problem* refers to the issue of which task is selected to use the CPU at a particular moment. The scheduling problem is fundamentally different for real-time systems than for other systems, such as timesharing systems. Real-time systems must schedule tasks so that they all meet their execution deadlines. Generally, the number of context switches should be minimized for real-time systems so that more CPU time is available for tasks working toward their deadlines. Timesharing systems, however, often interrupt a running task (incurring an extra CST) to provide

[2]Anonymous *ftp* to `ipcl.cs.qc.edu`.

other tasks with their fair share of CPU time. This section looks at some of the variables of real-time task scheduling. The iRMX task scheduling algorithm is covered in more detail in chapter 6.

A common feature of real-time task schedulers is their use of an algorithm called *preemptive priority-based scheduling*. This scheduling scheme simply assigns a numerical priority to each task in the system, the system keeps track of the scheduling state for each task, and the highest priority task that is in the "ready" state is always the one selected to run. The running task continues to execute indefinitely. The only way for the running task to stop is for either of the two conditions that caused it to be selected for execution in the first place to become false. Either the task enters a scheduling state other than ready, or another task of higher priority enters the ready state. In the first case, the task relinquishes the CPU, either because it has completed processing an event and met its deadline or because the task was blocked and cannot use the CPU until resources become available. In the second case, the task has been preempted by another task.

This simple scheduling algorithm can lead to very complex sequences of task selection, and by itself, might not provide an optimal solution to the scheduling problem for a particular application. The following examples illustrate these two points.

Suppose, for example, that T_i represents the ith task in the system, and E_i represents the events to be processed by T_i. Assume that events E_i arrive for processing at a rate of λ_i per second, that each E_i requires τ_i seconds of processing by T_i, and that those τ_i seconds of processing time must be completed within δ_i seconds of real time to meet T_i's deadline. Finally, p_i represents the scheduling priority of T_i (0 = highest priority). If two tasks with unequal priorities are ready to run at the same time, the one with the higher priority is the one that executes.

Table 1.1 shows the results of simulating the behavior of three tasks using a preemptive priority-based scheduler. The values chosen for the parameters were the following:

Ti	λi	τi	δi	Pi
1	1.00	0.25	1.00	5
2	0.67	0.30	1.50	10
3	0.50	1.00	1.90	15

These values cause the same sequence of events to repeat every 6 seconds, so the simulation was allowed to run for that amount of simulated time. The priorities of the three tasks were made proportional to the arrival rates of events for each task. That is, the higher the value of λ_i, the lower the numerical value of p_i. This positive relationship between a task's priority and the arrival rate of the task's events (remember, numerically low means

TABLE 1.1 Scheduling Simulation of Three Tasks Running for a Six Second Period Using a Preemptive Priority-based Scheduling Algorithm.

Current Time (seconds)	Running Task (T_i)	Task Balances (seconds)		
		1	2	3
0.00	1	0.25	0.30	1.00
0.25	2	0.00	0.30	1.00
0.55	3	0.00	0.00	1.00
1.00	1	0.25	0.00	0.55
1.25	3	0.00	0.00	0.55
1.50	2	0.00	0.30	0.30
1.80	3	0.00	0.00	0.30
Deadline Missed for Task 3 by 0.20 seconds.				
1.90	3	0.00	0.00	0.20
Error: Missed Event for Task 3 when balance = 0.10 seconds.				
2.00	1	0.25	0.00	0.10
2.25	3	0.00	0.00	0.10
2.35	none	0.00	0.00	0.00
3.00	1	0.25	0.30	0.00
3.25	2	0.00	0.30	0.00
3.55	none	0.00	0.00	0.00
4.00	1	0.25	0.00	1.00
4.25	3	0.00	0.00	1.00
4.50	2	0.00	0.30	0.75
4.80	3	0.00	0.00	0.75
5.00	1	0.25	0.00	0.55
5.25	3	0.00	0.00	0.55
5.80	none	0.00	0.00	0.00

There were 1.30 seconds of idle time, and 17 context switches

high priority) is known as the *rate-monotonic scheduling algorithm*, which is commonly used in real-time systems.

Note that the sequence in which the tasks execute does not follow a simple pattern, despite the small number of scheduling parameters involved. Also note that this set of parameters leads to a missed deadline and a lost event for T_3. The simulation program assumed that an event that arrives for a task while that task is still processing a previous event will be discarded rather than queued for later execution.

If the lost event in Table 1.1 had been queued instead of discarded, T_3 would have executed for one additional second of CPU time (the value of τ_3), reducing the idle time for the simulation from 1.3 to 0.3 seconds. (Another version of this example showed that there would have been no additional missed deadlines in this case.) Because it appears that there would have been 0.3 seconds of idle CPU time even if T_3 had processed all of the

required events, the question arises of whether a different scheduling algorithm could have avoided the missed deadline.

Table 1.2 shows that an *adaptive scheduling algorithm* could, indeed, have achieved the desired result by changing some of the values of p_i dynamically rather than maintaining fixed values at all times. In particular, at time 1.50 seconds, the adaptive scheduler would see that the processing balance for T_3 plus the balance for T_2 is greater than the time until T_3's deadline, and would temporarily raise T_3's priority above T_2's.

It is possible to create a working real-time system without carefully considering the scheduling problem. Just build the system and see if it works. But in cases where the system must not fail, the scheduling issue must be addressed. The actual values of λ_i, τ_i, and δ_i for each task must be either measured or computed and the corresponding scheduling algorithm must be determined, possibly using a simulator such as the one that generated Tables 1.1 and 1.2. Furthermore, if the system requires an adaptive scheduling policy, there must be a means for communicating each task's τ_i and δ_i to the scheduler, which must monitor each task's progress towards its

TABLE 1.2 Scheduling Simulation of Three Tasks Running for a Six-Second Period Using an Adaptive Scheduling Algorithm.

Current Time (seconds)	Running Task (T_i)	Task Balances (seconds)		
		1	2	3
0.00	1	0.25	0.30	1.00
0.25	2	0.00	0.30	1.00
0.55	3	0.00	0.00	1.00
1.00	1	0.25	0.00	0.55
1.25	3	0.00	0.00	0.55
1.50	3	0.00	0.30	0.30
1.80	2	0.00	0.30	0.00
2.00	1	0.25	0.10	1.00
2.25	2	0.00	0.10	1.00
2.35	3	0.00	0.00	1.00
3.00	1	0.25	0.30	0.35
3.25	2	0.00	0.30	0.35
3.55	3	0.00	0.00	0.35
3.90	none	0.00	0.00	0.00
4.00	1	0.25	0.00	1.00
4.25	3	0.00	0.00	1.00
4.50	2	0.00	0.30	0.75
4.80	3	0.00	0.00	0.75
5.00	1	0.25	0.00	0.55
5.25	3	0.00	0.00	0.55
5.80	none	0.00	0.00	0.00

There were 0.30 seconds of idle time and 18 context switches.

deadline. Real-time kernels and operating systems do not generally support adaptive scheduling, so it would have to be implemented by the application itself.

1.4 iRMX in Perspective

iRMX is actually the name of a family of operating systems developed by Intel to run on the microprocessors they manufacture. It is a proprietary OS rather than an open system. iRMX, as most operating systems today, is developed and marketed by the same company that makes the processors that run it. Examples of other proprietary operating systems include VMS for VAX computers and VM for IBM mainframe computers. MS-DOS for PCs and MacOS for Macintosh computers are also proprietary operating systems, even though the companies that make the computer and OS are not the company that makes the microprocessor inside the computer. (MS-DOS runs on the same Intel microprocessors as iRMX, and the Macintosh currently uses Motorola microprocessors.)

One characteristic of proprietary operating systems is that they generally are not portable. That is, they are designed to run on one processor's architecture, or a family of compatible architectures from one company, and cannot be implemented on different processors. In the case of real-time systems where execution speed is usually very important, this means that the OS can be built to take advantage of special features of the processor on which it runs. As a consequence, it executes very efficiently compared to an OS that must be coded to work with some lowest common denominator of many processors' features.

Some of the reasons for using a proprietary OS have more to do with marketing decisions than with system performance. If software is developed to run on a proprietary system, customers are unlikely to switch to another vendor's computer because of the expense of porting existing applications to the new OS and processor. An open operating system, however, can run on a variety of different processor architectures, usually because the companies that make the different processors have underwritten the cost of porting the OS to their machines. Customers are less locked into one vendor's computers.

Unix is the primary example of an open system today. Originally developed at AT&T for internal use, Unix has been licensed to dozens of different companies for use on their computers. There are, however, incompatibilities among the many versions of Unix that exist today. BSD Unix from the University of California Berkeley, Unix System V from an organization called Uniforum that includes AT&T as a member, and OSF-1 from an organization called the Open Software Foundation that includes IBM as a member. See the POSIX section later in the chapter for how all this relates to iRMX.

1.4.1 History and versions of iRMX

The iRMX operating systems for the x86 family of microprocessors date back to 1978 when Intel introduced RMX-86 for use with 8086 and 8088 microprocessors. An earlier operating system from Intel had existed with the RMX name and ran on the company's 8080 and 8085 microprocessors. Intel's goal had been to encourage engineers to develop new products based on the 8080 by providing them with the basic software needed to get various projects started and to market as quickly as possible. RMX-86 was designed in the same tradition as RMX-80, and an early version of RMX-86 even shared some code with its predecessor.

Early microprocessors like the 8080 were not powerful enough to provide users with general-purpose computing, but were typically embedded into other equipment to control it. RMX for the 8080 was originally designed to be embedded into ROM along with the microprocessor, and current versions of the OS still support this important feature. The actual development of a real-time application and the combination of the application with the OS were done on a separate computer system, also available from Intel, called a *microcomputer development system* (MDS).

When developing applications, the MDS is known as the *host*, and the system that actually runs the application is called the *target system*. The MDS and its OS (iSIS) are no longer used. Instead, the host for developing real-time applications is now a PC running DOS, a workstation running Unix, or the target system running iRMX itself. Some of the features of iRMX that make it good for real-time systems, however, make it less desirable as a development system. The iRMX for Windows version of iRMX allows developers to run both iRMX and MS-DOS on the same PC at the same time, thus providing the advantages of both environments, which is the standard configuration for running iRMX used in this book.

As Intel introduced microprocessors with different architectures, it also introduced versions of iRMX tailored to those architectures. Today, three versions of the operating system correspond to the 8086 (iRMX I), 80286 (iRMX II), and 80386 (iRMX III) architectures. Chapter 5 covers the architectures of these microprocessors and how they influence each version of iRMX. In 1991, Intel introduced iRMX for Windows, which is compatible with Microsoft's MS-DOS and Windows products. iRMX for Windows includes all the features of normal iRMX III plus additional features that allow a single application to include real-time components that are managed by iRMX and conventional components that run "on the DOS side." DOS programs can make iRMX system calls, and iRMX programs can make DOS calls. Both sides can communicate with each other directly and with the user through Microsoft Windows. If desired, the user can switch control of the PC's keyboard and monitor from DOS to iRMX or vice versa using the hot-key combination <Alt/SysRq>. Chapter 12 explores in some detail how this version of iRMX works, including the features that this version of iRMX adds to normal iRMX III.

1.4.2 MS-DOS, OS-2, and Unix

This section compares iRMX directly with a few other operating systems. The goal is to make it clear why each serves a different role in computer systems rather than to find one that is "better" or "worse" than another.

MS-DOS. MS-DOS was developed to run on a microprocessor (the Intel 8086[3]) that can address, at most, 1 megabyte (MB) of primary memory and includes no hardware mechanism for controlling memory accesses, such as accidentally attempting to execute data instead of instructions. As an OS for the 8086 architecture, DOS was designed (properly) to support a single user running a single application on one personal computer.

The OS includes no provision for multiple threads of execution, essentially no file system security mechanism (why protect the user from accessing the files on his or her own computer?), and allows programs to freely modify the system's memory, device controllers, and registers. The tremendous popularity of the PC has invited a wealth of creative code to be written for DOS systems, including some real-time applications.

The main disadvantages of DOS for real-time applications are its lack of support for multiple threads of execution; its lack of support for asynchronous I/O; the design of many of its device drivers, which disable interrupts for very long periods of time; and the difficulty of incorporating support for nonstandard I/O devices. Device driver software can be developed and then loaded into the system when it is bootstrap loaded, but the OS itself does not provide support for the development process the way a real-time OS such as iRMX does.

DOS allows a degree of systems programming. The command line processor is a separate piece of code from the rest of the OS, so substitute versions can be developed. Custom device drivers can be loaded when a system is initialized. Utility programs and development tools can be built for DOS because the system's interface to such programs is well documented. But as far as programming the OS itself, DOS is closed to systems programmers.

OS/2. When Intel developed the 80286 microprocessor that overcame the 8086 architecture's memory addressing and protection limits, Microsoft and IBM developed OS/2 to provide an OS that is compatible with DOS, but which takes advantage of the 80286 architecture to add new features that would be competitive with Unix, the preferred OS for workstations. Although no one has really pinned this marketing term down, a *workstation* generally connotes a single-user system that is more powerful than a *PC*.

The three most important features of OS/2 for us are its support for multiple threads of execution, its memory management facilities, and its sup-

[3]The Intel 8088 and 8086 share the same processor architecture except for the number of bits that can be read or written per memory access.

port for interrupt management. Like time-sharing and real-time systems, OS/2 provides users with multiple threads of execution, and, like time-sharing systems, the user's control over these threads is more primitive than in real-time systems.

For example, a primary objective of OS/2 is overall system performance, and to this end, the OS can manipulate the scheduling priority for threads (tasks) without informing the applications being run. Actually, there are three classes of priority (with 32 levels within each class), and tasks with a priority in the class, called *time-critical*, never have their levels changed by the OS. But threads that are designated regular or idle-time are subject to hidden priority changes. These hidden priority changes might seem reminiscent of adaptive real-time scheduling mentioned earlier, but in OS/2 scheduling, adaptations are made to improve overall average performance, not so that threads can meet deadlines.

OS/2 provides a protected memory environment for applications. This feature uses hardware mechanisms in the 80286 and later microprocessors to ensure that different applications do not access each other's memory either inadvertently or maliciously. This feature is critical for the integrity of timesharing systems. A protected memory environment is valuable in single-user systems as well because it guarantees that applications that run concurrently will not interfere with each other. Protected memory is particularly valuable during the development phase of any type of application, conventional or real-time. Without memory protection, a program error that causes information to be stored in the wrong part of memory (perhaps in the resident part of the operating system itself) might not be detected until much later, when the corrupted memory is accessed and causes the system to crash. With memory protection, such errors are detected as soon as they occur (even before the damage is done), and can be localized and debugged relatively easily.

The protection features of OS/2 also provide a controlled interface between application programs and the OS itself, including the restriction that application programs cannot perform certain privileged operations, such as I/O transfers. The 80286 protection mechanisms also make it possible for the OS to manage access to hardware interrupts. In OS/2, each interrupt service routine must register itself with the OS to have a chance to respond to the interrupt signals.

A popular DOS programming technique is to load a Terminate and Stay Resident (TSR) program that replaces the normal routine for responding to a particular interrupt. When an interrupt occurs, say from the keyboard, the TSR decides whether it will process the interrupt itself, such as if the interrupt was the user pressing a hot key (a special combination of keys) on the keyboard or not. If not, the new routine simply calls the original interrupt service routine to process the event normally.

The OS/2 technique of registering interrupt handlers provides a more robust and orderly way to manage such chains of handlers than DOS,

which cannot monitor application programs' access to the memory containing the table of interrupt service routine addresses.

Because DOS and OS/2 are based on corresponding processor architectures, they have a similar relationship to each other as iRMX I and iRMX II do. An obvious difference is that iRMX I supports multitasking, while DOS does not. Also, version 1.2 of OS/2 runs on any 80286 microprocessor or better, and version 2.0 runs only on 80386 microprocessors or better. These two versions of OS/2 correspond to the differences between iRMX II and iRMX III, notably the support for very large memory segments with the 80386 architecture. These architectural matters are covered in greater detail in chapter 5.

Unix. I mentioned earlier that Unix is the preferred OS for workstations. In the context mentioned earlier, Unix and OS/2 are competitors for the single-user, high-performance computer system market. Indeed, Unix was first designed as a single-user version of the Multics OS that was running on large mainframe computers at the time Unix was developed. Unix soon became a time-sharing system in its own right, and today, it is implemented on a broad range of processors. This would, however, include processors from single-user workstations to supercomputers and mainframes supporting many users simultaneously. In addition to the wide range of available implementations, Unix is popular because it provides a flexible and powerful environment for the technical user.

Unix is generally perceived as more difficult to use for casual users than DOS or even OS/2. This difference is becoming less of an issue, however, because Microsoft Windows for DOS, Presentation Manager for OS/2, and Motif for Unix's X Window system all provide similar graphical user interfaces. What makes Unix an important consideration is that efforts are being made to develop real-time versions of it, as you will see in the next section. To produce a real-time Unix OS, however, several issues must first be considered:

- Unix processes cannot be preempted while they are in kernel mode.
- Unix processes are expensive.
- Unix use of interprocess communication for real-time applications.

Unix processes cannot be preempted while they are in kernel mode (making system calls). This means that even a high-priority process might have to wait arbitrarily long after becoming ready before being scheduled to use the CPU. The logic for Unix kernel code is thus less prone to error, but it can be intolerable in real-time situations. Because this code is owned by whatever company owns Unix, the solution has been for other vendors to rewrite the kernel themselves to include what are called *preemption points*, places where processes executing kernel code will relinquish control of the CPU to higher-priority processes.

Unix processes are very expensive compared to tasks in a typical real-time system. Processes take a long time to create and context switches are slow because different processes are generally associated with different users. In a time-sharing environment, this means that each process must be protected from other users' processes that might be running at the same time. Process scheduling must be more complex and thus slower than task scheduling.

Another way to look at this issue is to say that Unix processes *compete* with each other for the use of the CPU, whereas real-time tasks typically *cooperate* with each other to meet deadlines. One way that some real-time Unix systems deal with this problem is to introduce lightweight processes. Lightweight processes are similar to real-time tasks in that they can be created quickly and, because they execute in the context of a single process, can be scheduled quickly without the security overhead associated with regular process scheduling.

Interprocess communication. Another issue for developers of a real-time version of Unix is interprocess communication, or IPC. Unix provides a rich and flexible set of IPC mechanisms, including shared memory, kernel-mediated signals, pipes that carry the output of one process through a disk file to the input of another process, and sockets that allow processes to communicate with each other across networks using the same syntax as reading and writing disk files. The problem is that these mechanisms, in order to provide their rich functionality and flexibility, are much too slow to be used for intertask communication in many real-time applications. Even where attempts have been made to provide IPC functions typical of real-time systems, the Unix versions generally involve too much overhead for real-time use. The Unix System V semaphore, for example, is very complex and less efficient to use for synchronizing tasks compared to the equivalent mechanisms for iRMX or other real-time operating systems.

1.4.3 POSIX

Unix is the prototypical open system, but various incompatible versions of Unix are common on different computing platforms. To promote Unix as a portable OS, the IEEE Computer Society is developing a portable version of Unix, called *Portable Operating System Interface for Computer Environments* (POSIX). The idea is that an application coded to meet the POSIX standard can be compiled and run without change on any system that is POSIX compliant. Vendors are free to add their own features of Unix and still claim POSIX compliance, provided their added features do not interfere with the POSIX functions. Thus, System V Release 4 (SVR4) and OSF-1 might be POSIX compliant, but incompatible with each other in various ways.

Standards take several different forms, such as industry standards, national standards, and international standards. An industry standard is simply something that almost everyone in a particular industry does the same way. For example, the PC bus was developed by IBM, but IBM made its specifications public and encouraged other companies to build compatible products, thus making the bus an industry standard. National standards such as those of the American National Standards Institute (ANSI) and International Standards Organization (ISO) are developed by committees that include representatives of the companies or countries interested in the standard. Formal standards are based on current common practice rather than creating new rules for doing things. There is a rich political process involved in developing and approving new formal standards.

The Institute of Electrical and Electronic Engineers (IEEE) is a professional organization that is developing POSIX under its own initiative. One might expect IEEE to submit POSIX to ANSI or ISO for adoption but not necessarily.[4] The only reason this would be of concern is if some other organization produced a competing standard and submitted it to ANSI or ISO.

The IEEE formed several subcommittees to develop different parts of the POSIX standard, and each has a name in the form P1003.*x*, where *x* indicates the area of concern. The standards developed by these subcommittees are often referred to with names like POSIX.1 for the standard developed by subcommittee P1003.1.

POSIX is an important consideration because it is a potential alternative to iRMX as a target for real-time systems. Which system should be used depends on the proper trade-off level between portability and performance for a particular application. A real-time application that is developed for iRMX can only be run on systems based on Intel's x86 family of microprocessors and cannot be expected to be portable to other processors; iRMX is not available for other types of CPUs. The huge number of systems that run Intel x86 microprocessors may or may not be relevant for a particular application. On the other hand, Unix is a very large OS compared to iRMX, with process, memory, and security management features that go far beyond those needed for most real-time applications.

POSIX real-time and threads standards are added to basic Unix functionality. The result is, almost inevitably, a system with more overhead and poorer real-time performance than an iRMX system. There are four POSIX standards potentially related to iRMX. Currently only POSIX.1 has been formally adopted by the IEEE. POSIX.4 should be approved in the near future, and other parts of the POSIX standard are still being developed.

---

[4]For example, the *System Application Program Interface (API) [C Language]* part of the standard (IEEE Standard 1003.1) has been adopted as ISO standard ISO/IEC 9945-1.

POSIX.1. System application program interface [C language]. This standard specifies a standard application programming interface (API) to the OS. The Intel C compilers provide a POSIX.1 interface to iRMX. Chapter 4 explains in more detail how this is accomplished and the implications of how it is implemented. As a practical matter, POSIX.1 compliance means that many utility programs for Unix for which there is publicly available source code run on iRMX systems.[5]

POSIX.4. IEEE Realtime Extension for Portable Operating Systems. The following material stating the scope of this standard is taken from a draft version of the standards document:

> The key elements of defining the scope are a) defining a sufficient set of functionality to cover a significant part of the realtime application program domain, and b) defining sufficient performance constraints and performance related functions to allow a realtime application to achieve deterministic response from the system. . . . The specific functional areas included in this standard and their scope includes:
>
> - Binary semaphores: the minimum synchronism primitive to serve as the basis for more complex synchronization mechanisms to be defined by the application program.
> - Process memory locking: a performance improvement facility to bind application programs into a computer system's high performance random access memory to avoid potential latencies introduced by operating system storage of not recently referenced parts of a program on secondary memory devices.
> - Shared memory: a performance improvement facility to allow separate application programs to have portions of their program image comonly accessible to them.
> - Priority scheduling: a performance and determinism improvement facility to allow applications to determine the order in which processes that are ready to run are granted access to CPU resources.
> - Real-time signal extension: a determinism improvement facility, augmenting the signals mechanism of POSIX.1 to enable asynchronous signal notifications to an application to be queued without impacting compatibility with the existing signals interface.
> - Timers: a functionality and determinism improvement facility to increase the resolution and capabilities of the time-base interface.
> - Interprocess communication: a functionality enhancement to add a high performance, deterministic interprocess communication facility for local communication. Network transparency is beyond the scope of this interface.
> - Synchronized input and output: a determinism and robustness improvement mechanism to enhance the data input and output mechanisms so that

[5]A rich source of such utilities is the Free Software Foundation of Cambridge, Massachusetts.

an application can insure that the data being manipulated is physically present on secondary mass storage devices.

- Asynchronous input and output: a functionality enhancement to allow an application process to queue data input and output commands with asynchronous notification of completion. This facility includes in its scope the requirements of supercomputer applications.
- Real-time files: a performance and determinism improvement facility to allow an application program to pre-allocate mass storage resources and determine characteristics that will enhance the performance of data transfer to and from mass storage.
- Extensions to POSIX.1: those changes needed to complete the definition of the facilities defined by this standard.
- Performance metrics: each facility includes a set of performance metrics to allow a uniform treatment of the measurement of performance between different conforming implementations.[6]

iRMX either already conforms to many of these functional areas of POSIX.4 or can easily be made to do so. If you are interested, you can refer back to the preceding list as various iRMX topics are covered in the chapters ahead and consider what the impact of making iRMX POSIX.4 compliant would be on the OS's design and efficiency of implementation. For example, the complex semaphore mechanism of Unix System V is the basis for the POSIX.4 binary semaphore mechanism, and you might want to consider the issue of providing this function in iRMX after reading the discussion of the iRMX semaphore mechanism in chapter 7.

POSIX.4a. Threads Extension to POSIX. The focus of POSIX.4a is to add lightweight processes, called threads, to POSIX.4, which is based on the standard (POSIX.1) process model. The relationship between POSIX processes and threads is approximately analogous to the relationship between iRMX jobs and tasks, which is discussed in chapter 6.

In addition to introducing threads themselves, POSIX.4a also introduces features for synchronization between threads, control over thread scheduling, and extension of the POSIX.1 signal mechanism to cover threads. The two synchronization primitives introduced by POSIX.4a are *mutexes* and *conditions*. The mutex mechanism is closely related to the iRMX *region* introduced in chapter 7, but conditions have no direct analog in iRMX. The iRMX facility for creating composite objects, also introduced in chapter 7, could be used to create the equivalent of conditions.

---

[6]The indented information contained on pages 24–25 is copyrighted information of the IEEE, extracted from IEEE Std P1003.4/D10-1991, copyright ©1991 by the Institute of Electrical and Electronics Engineers, Inc. This information was written within the context of the IEEE Std P1003.4/D10-1991. The IEEE takes no responsibility or liability for and will assume no liability for any damages resulting from reader's misinterpretation of said information resulting from the placement and context in this publication. Information is reproduced with the permission of the IEEE.

POSIX.4a dictates that compliant implementations are to support at least two scheduling algorithms, priority based and round-robin, and provides functions to assign these algorithms to individual threads. iRMX supports both priority and round-robin scheduling algorithms, although not with as rich a function set as POSIX.4a. iRMX task scheduling is described in chapter 6.

POSIX signals are closely related to a process, which is only approximately the same as an iRMX job. As a result, the signal mechanisms of POSIX.1 and their extensions in POSIX.4a map only roughly onto iRMX systems. Actually, the proposed POSIX.4a extensions to signals were not well enough developed at the time of this writing to say much about them. Two iRMX features provide the functions for the situations that Unix signals are meant to deal with: exception handlers, described in chapter 6, and signal characters, described in chapter 8.

POSIX.16. Multiprocessing. P1003.16 is the name of the group working on multiprocessing extensions to POSIX. Some of the issues involved in developing POSIX.4a were purposely deferred until POSIX.16 becomes established because of an overlap between certain multitasking and multiprocessing concepts.

All the computer systems considered in this book are based on computers with only one CPU, or CPUs operating independently of each other except for passing messages to one another. That is, one CPU runs all the code executed by all the tasks of all the applications that might be in primary memory at one time, as well as all code executed on behalf of the OS itself. Conceptually, several tasks could be executing at the same time, but this is a case of virtual concurrency because a single CPU can actually execute an instruction for only one task at any particular moment. The situation is radically different if there are multiple CPUs to which different threads of execution can be assigned, because the concurrency between tasks becomes real rather than virtual.

From a real-time application programmer's point of view, an application runs correctly on one CPU because the program manages task priorities and intertask synchronization to guarantee logical correctness without regard to the actual rate at which the processor executes the code for a particular task. If two tasks of the same priority are ready to run at the same time, it should not matter whether they run one after the other or, by using multiple CPUs, at the same time. You can expect some interesting issues to be raised as the POSIX.4a and POSIX.16 committees interact. Meanwhile, we can examine the iRMX operating system, confident that the multitasking features it supplies will bear at least conceptually on multiprocessing systems as well.

Chapter

2

Using an iRMX System

2.1 Overview

Most programs developed for iRMX are real-time applications, and many of those applications interact with human users through nonstandard I/O devices — if at all. Such applications are often programmed on a non-iRMX computer, called a *development system*, which can also be used to help debug and integrate the application on an iRMX computer, called the *target system.*

iRMX does, however, include a software layer called the *Human Interface* (HI) that allows you to use the OS as a conventional time-sharing system, one that can be used as a development system in its own right. This chapter introduces you to the features of the HI that a user encounters while using iRMX as a time-sharing system, and the next chapter covers using an iRMX system as a development system.

In addition to the HI layer, other layers of the OS are referenced. These are the Nucleus, the Basic I/O System (BIOS), the Extended I/O System (EIOS), the Application Loader (AL), the Universal Development Interface (UDI), and the C run-time library.[1] These other layers are covered in more detail in chapter 6 and beyond. Even if your iRMX applications do not use the iRMX HI, and you do your coding on a separate development system, you should become familiar with the topics covered in this chapter because they include concepts about the iRMX I/O system that will be important later on.

iRMX is not the first computer system most people work with, so brief references to DOS and Unix are included in the material that follows. These references serve two purposes: (1) They might clarify an iRMX

[1] Versions of iRMX can be configured that omit some of these layers. Such a configuration would be built for a system that has memory constraints or that does not need the functions supplied by certain layers. All configurations of iRMX include the Nucleus layer, however.

function to DOS and Unix users, and (2) they might warn DOS or Unix users that something that seems to be the same in iRMX is actually different. If any particular reference to DOS or Unix does not help you, simply ignore it.

Like the rest of iRMX, the HI is very well documented in the manuals that accompany the system. The manuals, however, serve as reference documents rather than tutorials; thus, the manuals often include references to somewhat obscure features of the system to be completely accurate. This book, on the other hand, tries to guide you in mastering iRMX. Keeping a topic clear while you are learning iRMX means being selective about what is included about that topic at any particular point in this book. If the system does not do what you expect it to, it may very well be that you have stumbled onto something that was only glossed over or is covered later in the book. So, when in doubt, RTM! (Read The Manual!). Of course, RTM only works if you know which manual to read, so do look through the complete documentation set for your version of the OS to find out where to look things up later on. In this book, various volumes in the iRMX for Windows documentation set are referenced. These references are correct for iRMX for Windows version 2.0, but might be different for other releases of the operating system.

2.2 iRMX Platforms

Before using iRMX, you need to understand some background about the different platforms available for running iRMX. A *platform* is a type of computer system that can run iRMX software. Different platforms require different steps in the procedures. Conceptually, the various versions of iRMX (I, II, and III) can run on any computer that uses the appropriate Intel microprocessor. In practice, iRMX has built-in support for application development on just three platforms: the AT Bus, Multibus I, and Multibus II.

The *AT Bus* platform refers to any industry-standard PC compatible with the IBM AT or later computer. Although the term *AT Bus* is used, the computer can use just about any bus at all, including the following:

- The AT bus itself, which is also called the ISA bus.
- The EISA bus, an extended version of the AT bus.
- IBM's Micro Channel Adapter bus (MCA).
- One of the buses used in PCs outside of the United States, such as those used by NEC and Fujitsu in Japan.

The important feature of the platform for iRMX for Windows is simply that the computer contain code in an IBM or compatible ROM-BIOS for performing standard I/O operations. ISA, EISA, and MCA circuit boards cannot be intermixed within one computer system, but they are all programmed the same way using subroutines supplied in the ROM BIOS.

Two versions of iRMX III were available for the AT platform: one version ran as a typical PC operating system, requiring its own disk partition from which the OS had to be bootstrap loaded and providing no interaction with the DOS operating system. This version of iRMX III for the AT platform was sometimes called the *System 120* configuration. iRMX for Windows is the other version of iRMX III, and is the only version that Intel now supports for the AT platform.[2a] iRMX for Windows runs at the same time as DOS (a DOS command is used to bootstrap load iRMX for Windows), with a hot key, <alt/SysRq>, used to switch between the two operating systems. In addition to running concurrently with DOS, iRMX for Windows can use both the DOS disk partition for its files as well as an optional, separate iRMX partition for users who prefer the advantages of multi-user protection and longer file names offered by the iRMX file system.

Note that you can edit and compile iRMX code on any PC platform that runs DOS because the iRMX software development tools (described in chapter 3) run under DOS as well as under iRMX. Normally, a program that is built to run under one OS will not run under a different OS. The development tools for iRMX, however, are an exception; an iRMX program called *run86,* coupled with their internal use of a software layer called the *Universal Development Interface* (UDI) allow Intel's DOS-hosted development tools to run under iRMX. You must, however, be running iRMX itself on a PC to actually run and test an iRMX application, but the rest of the development cycle can be conducted under DOS without running iRMX.

Multibus I and Multibus II were initially developed by Intel as designs for system buses. The designs have been adopted as open standards by the IEEE and are used by a number of vendors in the design of computer systems. Like the various PC buses, the designs of these buses specify both the physical dimensions of the circuit boards that can be used with them and the mechanical and electrical parameters that must be matched for different circuit boards to interact properly in an integrated computer system. There is no standard ROM-BIOS for these two platforms, but to use iRMX with them as a development system, the circuit board containing the CPU must include code in ROM for bootstrap loading the OS from disk or a network.[2]

---

[2]ISA, EISA, and MCA bus systems have the CPU, some memory, and various other circuits on a motherboard, which also holds bus connectors for the other circuit boards that can be installed in the system. Multibus I and Multibus II systems use a passive backplane to hold the connectors for all the circuit boards, including the one that holds the equivalent of a motherboard, which is called a Single Board Computer (SBC). With a passive backplane you can change CPUs by exchanging SBCs, and you can even have more than one SBC in the system. Processors in a multi-SBC system can share memory, and with Mulltibus II they can pass messages among themselves efficiently. Each processor in a multi-SBC system runs its own OS.

[2a]Intel introduced a version of iRMX too late to be included in this book that runs on an AT platform without requiring DOS to be present.

2.3 Logging on to an iRMX System

Actually logging on to an iRMX systems is easy: just enter your user name at the `login:` prompt and your password at the `password:` prompt. Getting those prompts to appear, however, might not be as simple as on other time-sharing systems with which you are familiar!

If you are running iRMX for Windows on a PC, you must run *rmxtsr* and *loadrmx,* either from the DOS prompt or from a batch file such as `autoexec.bat`. Then, press <alt/SysRq> to bring up the login screen from the DOS prompt. This hot-key technique for accessing iRMX works only from text screens on the DOS side. To access iRMX from Windows, which is a graphics application, start the *wterm* application that comes with iRMX for Windows (by double-clicking the icon), and select the [Wterm, Connect] menu options. Note that some early versions of *wterm* require you to click a few "OK" buttons to get to the iRMX login screen. *Winterm* a commercially available terminal emulator from Marketfield Software, Oyster Bay, N.Y., can be used in place of *wterm. Winterm* offers more features and generally performs better than *wterm.*

If you are working with a Multibus I or II platform rather than Windows, just powering up the system should produce the login screen on all the terminals attached to the system.

If a terminal attached to the computer doesn't invite you to log in, there are two possibilities:

First, the system might not be configured to recognize the terminal as a login terminal. For example, it is often convenient to have a terminal attached to a system reserved specifically for debugging programs interactively. SoftScope, the interactive debugger, allows a developer to use such a terminal for its own interactions with the user so that debugging commands and responses do not interfere with the appearance of the program being debugged, which continues to interact with the user's login terminal.

Second, the system might be configured to recognize the terminal as a static login device, which means that a user gets automatically logged in to the system on that terminal at power up. For iRMX for Windows, a "terminal" might be either a separate terminal attached to the system through a serial port or the PC's own keyboard and monitor, referred to collectively as the *system console.*

If you do not have an account on the system yet, you can probably log in with the user name "world," which is normally valid on all iRMX systems. There is usually no password for "world," so just press <Enter> when prompted for the password.[3]

---

[3] iRMX does not distinguish between uppercase and lowercase letters in commands and file names, similar to DOS, but unlike Unix. You can log in as "WORLD," "world," or "World," and it's all the same on iRMX. *The one exception to this rule is your password, which must be entered using exactly the same alphabetic case(s) as when you set it up.*

2.4 Entering Commands

When you first log in, some commands will probably be executed automatically by a mechanism like the DOS `autoexec.bat` file or the Unix `.login` file. You will then interact with a program called the Command Line Interpreter (CLI, pronounced "klee"), which reads your commands from the keyboard and runs them. The CLI is equivalent to the DOS and Unix shell programs, *command.com* and */bin/sh*, respectively. The default CLI prompt string is a hyphen, but `iRMX>` is used in the examples throughout the book. A reference number that would not be typed by the user is also used at the end of command lines in the examples. For example, command line [1] represents a simple *dir* command entered by a user in response to a CLI prompt. (The *dir* command lists the names of the files and directories in the current directory.)

```
iRMX> dir                                                    [1]
```

You can correct typing mistakes on a command line before you press <Enter> by using the typical editing keys:

- <backspace>, <delete>, or <rubout> (depending on your keyboard), which erases the character to the left of the cursor.
- The left and right arrow keys, which move the cursor within the command line so you can edit it.
- <^F>, which erases the character under the cursor.
- <^A>, which erases all characters from the cursor to the end of the line.
- <esc>, which enters the command exactly as it appears on the screen.
- <Return>, which erases from the cursor to the end of the line and then enters the command.
- <&> at the end of the line, which continues long commands on more than one line. If you use <&>, you will see two asterisks as the prompt for continuation lines.

The CLI command history mechanism allows you to recall previous commands for editing and entering. Press the up and down arrow keys to move up and down through the list of previous commands. Alternatively, you can type <!> followed by the first few letters of the command you want to recall, and the CLI will search back for the last command that started with those letters. This mechanism is very similar to the Unix *tcsh* command history mechanism, and similar in concept to the DOS 5.0 *doskey* facility.

There are several other special keys, some of which can cause problems if pressed inadvertently. For example, <^S> is used to stop all output (so it doesn't scroll off the screen), and you have to type <^Q> to allow output to

resume. <^W> causes console output to stop every 20 lines or so. Press <^W> to display the next 20 lines. If your terminal seems frozen, it's possible you pressed <^S> or <^W>. Pressing <^Q> twice will clear most such problems. The *iRMX Command Reference*, volume 10 of the iRMX for Windows documentation set, includes a list of all the special key combinations you can use in its first chapter. The corresponding manual for other versions of iRMX is called the *iRMX Operator's Guide.*

The CLI can process two types of commands: CLI commands and HI commands. CLI commands are those recognized and processed by the CLI itself. HI commands are loaded into memory for execution from files.

2.4.1 HI commands

HI commands are the more commonly used commands. These commands come from a variety of sources. Many are supplied with the system and are known as system commands. A set of commands known as utilities, or Commonly Used System Programs (CUSPs) comes from Intel, the iRMX user's group called iRUG, and others.[4] The distinction between system commands and Intel-supplied utilities can sometimes be obscure, with a command distributed as a utility at one time being promoted to system command status in a later release. A third set of commands is placed in the category of development tools, which includes compilers, linkers, debuggers, and the like. Finally, HI commands also include those programs that you have developed.

The HI commands are fully documented in the *iRMX Command Summary* or *iRMX User's Guide* manual depending on the iRMX version. In addition, an iRMX *help* command displays information about most HI commands. (The DOS *rmxhelp* command can help you when you are using iRMX system calls, which is a different matter.)

For a quick look at most of the names of the HI commands available on your system, type the following commands:

```
iRMX> dir :system:                                    [2]
iRMX> dir :utils:                                     [3]
iRMX> dir :lang:                                      [4]
```

Command line [2] lists the names of the system commands, line [3] lists the names of the utilities, and line [4] lists the names of the development tools. Note that since *dir* is itself a system command, you should see its name in the first list of files.

---

[4] iRUG originally began as the iRMX user's group but has expanded its purview to "all real-time systems based on Intel microprocessors." iRUG can be contacted by calling (800) 255-IRUG (255-4784).

When you enter an HI command line, the CLI passes it on to the part of the OS called the Human Interface (that's why they are called "HI Commands"), which, in turn, searches standard parts of the disk until it finds the file that contains the program, loads the program into RAM (using a part of the OS called the Application Loader), and causes it to start executing.

iRMX command lines have a standard format, consisting, from left to right, of the command name (the name of the file containing the program to run), an *input path list*,[5] a *preposition*, an *output path list*, and a set of *parameters*. If parts of a command line are omitted, default values are usually assumed, which often provide a single command with a number of different, but related, functions. Consider the system command, *copy*, for example.

```
iRMX> copy file1 to file2                                    [5]
```

Here, the input path list is the name of one file, `file1`, the preposition is `to`, and the output path list is the single file named `file2`. No parameters are specified in this example. As you would expect, the command will create a new file, named `file2`, by copying `file1`. If `file2` already exists, the user will be prompted whether to replace it or not. Changing the preposition from `to` to `over` suppresses that prompt, and any existing file named `file2` is replaced automatically. The only other prepositions used by iRMX commands besides `to` and `over` are `after` and `as`. Changing `to` to `after` in the example would cause `file1` to be appended to the end of `file2`. The `as` preposition cannot be used with the copy command. (In fact, purists claim that `as` is not a true iRMX preposition because it is not recognized automatically by the normal iRMX command line parsing routines.)

Now let's experiment with input and output path lists. The items in a path list are separated by commas, so the input and output path lists in the following example consist of three file names each:

```
iRMX> copy file1, file2, file3 to filea, fileb, filec        [6]
```

For this example, `file1` is copied to `filea`, `file2` is copied to `fileb`, and `file3` is copied to `filec`. It is possible to not match the number of input and output lists evenly, provided the command makes sense. For example, the command

```
iRMX> copy file1, file2, file3 to filea                      [7]
```

---

[5] A distinction exists between file names and path names. See the section on file management later in this chapter for more information.

would copy `file1` *to* `filea`, then copy `file2` *after* `filea`, and `file3` *after* that, resulting in the concatenation of the three input files in the single output file. The HI shifts the preposition from `to` to `after` automatically.

If you omit the preposition and output file list, when using the *copy* command the input files are copied to the terminal screen. (You cannot omit the input file list from a *copy* command, but you can for many other commands.)

For an example of a command line parameter, consider the following:

```
iRMX> copy file1, file2 to filea, fileb query                    [8]
```

You can tell that `query` is a parameter because there is no comma between it and `fileb`. Commas separate items in the input and output path lists, while spaces separate path lists from the preposition, the output path list from the parameters, and the parameters from each other. The *query* parameter causes *copy* to prompt you for permission before copying each file.

Wildcards are supported for input and output path lists. An <*> substitutes for any zero or more characters in a file name, and a <?> substitutes for any single character. Examples of wildcards are shown in Section 2.5, File Management, below.

The CLI also supports redirection of console input and output using the '<' and '>' characters, respectively. iRMX uses the terms *console input* and *console output* as well as these redirection characters the same way that DOS and Unix manipulate what they call *standard input* and *standard output*. Commands that normally accept input from the keyboard and produce output on the screen can have files substituted for these devices using '<' and '>'. Console input redirection cannot be illustrated using *copy* because it does not take input from the keyboard, but you have already seen that *copy* uses the screen as the default output device if no preposition and output path list are specified. Thus,

```
iRMX> copy file1 > file2                                         [9]
```

has the same effect as line [5]. If you understand that the input path list and console input are different, and that the output path list and console output are also different, you should see that lines [5] and [9] are accomplishing the same command two totally different ways. For instance, you could not substitute '>' for *to* in line [6] (copying three files to three other files) because it makes sense only for a single file name to follow the '>' character.

Although a standard format exists for iRMX command lines, this format is not always followed. Processing the command line according to the standard format is something that must be done by the program itself, and not all commands need all parts of the standard line. The HI provides system

calls to do most of the parsing for a program, but some programs prefer not to use the HI routines but rather work with another syntax for the command line for one reason or another. In particular, programs written in the C language normally treat the command line as a list of words separated by spaces rather than commas, and no notion of input list, preposition, output list, or wildcards exists. For example, a standard C program would read the command line

```
iRMX> mycommand file1, file2, file3                    [10]
```

as a line with three command line arguments after the command name (each one a character string that has a comma at the end), but it would read

```
iRMX> mycommand file1,file2,file3                      [11]
```

as a command name followed by a single argument, because there are no spaces following the commas. A standard iRMX program using the HI parsing routines would read both as lists of three file names because the HI parser ignores the spaces after the commas in line [10].

2.4.2 CLI commands

CLI commands are executed directly by the command line interpreter. These commands support eight different features: history, CLI parameters, aliases, background processing, command files, super, and log off. This section describes these classes of CLI commands. They are fully documented in volume 10 of the iRMX for Windows documentation set, the *iRMX Command Reference* (called the *iRMX Operator's Guide* for some other versions of iRMX).

History commands. As you type in commands at the `iRMX>` prompt, the CLI stores them in an internal list so that you can reuse them later. You can see this list by typing the *history* command, and you can recall previous commands by either using the up and down arrows to scroll through the history list, or using the <!> character to recall previous commands. A command line that starts with <!> followed by either a number or a few characters causes the CLI to recall either the command with the matching number (the history list provides the numbers) or the most recently entered command that started with the same few characters.

CLI parameters. The CLI maintains a number of parameters about the user's session. These parameters include the user's prompt string, how much space to reserve for the alias table (described next), and what type of terminal is being used. You can add special features to the CLI, such as

automatically timing the execution of all HI commands, and the CLI will display parameters for these added features, if they are present, as well. The *set* command is used to modify the CLI's parameters, and if no arguments are used, *set* displays the current values of all its parameters.

Aliases. Aliases let you define abbreviations for commands. This can be useful for either reducing typing time (some of the iRMX command names are very long), or for customizing the iRMX system to recognize the commands you are accustomed to using on another system. For a combined example, consider using the alias *cd* for the name of the iRMX *attachfile* command, which is roughly the same as *cd* in both Unix and DOS. iRMX aliases can take arguments, as this example illustrates:

```
iRMX> alias cp = copy #0 to #1 query                    [12]
```

After entering this alias, the following command could be used to copy two files, with a prompt for confirmation before each one is copied:

```
iRMX> cp file1,file2 filea,fileb                        [13]
```

As you might infer, `#0` and `#1` in the alias are place holders for the arguments that are specified on the command line when the alias is actually used. Note that there are no spaces between the file names on the cp command line. The significance of this is that *alias* substitution uses spaces to separate the command line into the parameters, `#0` and `#1`, so the entire string, `file1,file2` is substituted for `#0`, and the string `filea,fileb` is substituted for `#1`. The embedded commas in these strings then signify lists of file names for the *copy* command.

Background processing. The CLI lets you run more than one program at a time. Any HI command line can be preceded by the CLI command *background*, and the CLI will start the command running and return with a prompt for another command to be run at the same time as the first. The *jobs* command lists all background commands currently running, and the *kill* command aborts background commands. There is a standard alias for *background*, which is *bk*.

The subject of background processing raises the issue of iRMX's memory management policies. Like DOS but unlike Unix, iRMX does not include support for virtual memory. Therefore, all programs you want to run simultaneously must be loaded into memory in their entirety at the same time. This strategy is good for real-time systems that cannot afford the uncertainty in execution time associated with demand-paging algorithms. However, this strategy can become a problem when you run certain programs in the background, such as compilers, that ask the OS to allocate as much memory as possible to themselves when they first start running. The

more memory a compiler can get, the faster it can run, but the less memory is then left for running other commands at the same time.

The *background* command lets you specify both the minimum and maximum amount of memory a program can have available as it runs in the background. You might set the minimum high enough to get just acceptable performance from a program, and set the maximum low enough to ensure the program does not occupy too much memory and leave you without enough memory to do anything else. For example, the command

```
iRMX> background(450,1024) plm386 myprog.plm                    [14]
```

specifies that the PLM compiler is to run with a minimum of 450 kilobytes (KB) of memory and a maximum of 1 MB (1024 KB). Because the compiler can run with as little as 380 KB, the command guarantees that the compiler will not start unless enough memory exists to give the compiler what the user considers to be acceptable performance. At the same time, the compiler is not allowed to use more than 1 MB, presumably because the user knows that enough memory is available beyond 1 MB to allow other commands to be run at the same time.

If you entered line [14] as shown, the system would ask you for the name of a log file. Background commands cannot read or write from or to the operator's console because doing so would interfere with the use of the console for normal commands read by the CLI. The CLI thus asks for the name of a log file to which all console output will automatically be sent as the program runs in the background. You can view the contents of the log file as the background command is running if you want to track the progress of the command. The *skim* utility is a convenient way to display text files on the screen.

If a background command tries to read from the console input device (the keyboard), the command will be aborted. You can use '<' and '>' on the command line to redirect console input and output from and to files. If you redirect console output, you will not be prompted for the name of the log file. Below is an example of an interactive program named *interact* that runs in the background. The input the program reads comes from a file named `input.data`, and the output is redirected to `interact.log`:

```
iRMX> bk interact < input.data > interact.log                   [15]
```

Command files. Command aliases can reduce the typing needed to enter a single command; command files can extend this concept to sequences of commands. Use an editor to put the commands to be executed into a text file, and issue the CLI's *submit* command with the file name as an argument.[6] If the command file name ends in `.CSD` you can omit that

[6]Command files are often called submit files because they use the *submit* command.

part of the file name. You can supply arguments to the command file by enclosing a parameter list in parentheses on the command line and referring to them as %0, %1, . . . %9 inside the file. For example, suppose the file `doit.csd` contains the following text:

```
copy %0 to %1
delete %0 query
```

This command file might be invoked using the following command line:

```
iRMX> submit doit(file1, file2)                                    [16]
```

In this case, `file1` would be copied to `file2` and then would be deleted after the user confirms the file deletion.

Command files can contain CLI commands, including other *submit* commands, *alias* commands, *background* commands, and the like. A useful strategy is to make *submit* the object of a *background* command. Chapter 3 provides such an example after some of the program development tools have been introduced.

iRMX command files are different from DOS batch files and Unix shell scripts primarily in the way they are invoked. DOS knows that a file is a command file by its `.BAT` extension, and Unix knows the same thing by looking at the state of the file's execute permission bit. iRMX lets you use any file as a command file, but you need to type *submit* (or a brief alias for *submit*, such as *s*) on the command line.

Below is an *alias* for a command called *do* that will submit the command file named `makeit.csd` and pass three arguments to it:

```
iRMX> alias do = submit makeit (#0, #1, #3)                        [17]
```

With this alias in place, you can save typing by entering the command

```
iRMX> do myprog compact debug                                      [18]
```

which would be equivalent to the command

```
iRMX> submit makeit (myprog, compact, debug)                       [19]
```

Although *submit* is a CLI command, there is also an HI command by the same name that comes with the system. The HI *submit* command operates the same as the CLI version, except that it does not recognize CLI commands within the command file. The HI command version of *submit* is useful to use from within the editor or debugger. These programs allow you to run HI commands without exiting the program; however, they use their

own command line interpreters, which can run only HI commands, and not CLI commands.

Another HI command, called *esubmit*, is also supplied with iRMX. It not only supports the standard CLI commands within command files, but also supports conditional execution of commands based on the results of earlier commands. An example of an *esubmit* command file is shown in chapter 3.

When you first log on to an iRMX system, a command file named `:prog:r?logon` is automatically submitted. (File names, and the file system in general, are described in the next section.) You can edit this file to contain any commands you want. Most users' `r?logon` files contain *submit* commands for files consisting of *alias* commands that set up shorthand command names. There are usually two of these alias files submitted, one that is the same for all users on a system, and one that is unique to each user.

In addition, whenever an iRMX system is first started, the file `:config:r?init` is automatically submitted as a command file. The commands in this file usually establish system-wide values, such as the system's network node name. A particularly important file submitted from `:config:r?init` is `:config:loadinfo`, which loads programs that run while the operating system is running, including layers of the operating system itself.[7]

Super. Every user of an iRMX system is assigned a unique ID number between 0 and 65,535. The file system uses these ID numbers to provide a basic protection mechanism for controlling one user's access to other users' files (discussed in the next subsection). The ID number 0 belongs to the Super user, who can read or change the access rights for any file on the system. The CLI's *super* command permits a user to gain Super user status. There is also a *super* HI command for use when commands are processed by a nonstandard command line interpreter, such as from within the editor or debugger. Both the CLI and the HI *super* commands use the command *exit* to leave Super-user mode.

Logging off. Use the CLI's *logoff* command to end a time-sharing session. The file `:prog:r?logoff`, if it exists, will be submitted as a command file, and the `logon:` prompt for the next user will appear. If the terminal is configured for static logon rather than the usual dynamic logon, the static user will automatically be logged back on after submitting `r?logoff`.

Table 2.1 lists all the iRMX files accessed when the system starts running and when users log on and off. The files that have names beginning

[7]At the time of publication, `loadinfo` is used only with iRMX for Windows and iRMX III systems.

TABLE 2.1 Text Files Accessed When an iRMX System Starts Running, When Individual Users Log On or Off, and When the C Language *getenv*() Function is Called.

File name	Purpose
`:config:rmx.ini`	iRMX for Windows systems only. Contains operating system configuration parameters.
`:config:r?init`	Contains HI commands that are automatically executed when the system starts running.
`:config:loadinfo`	iRMX for Windows systems only. Invoked by a command in `:config:r?init`. Contains commands to install loadable parts of the operating system, such as device drivers.
`:config:terminals`	A list of devices to which terminals are connected for accessing the system. The first line lists how many terminals I/O lines exist, and the succeeding lines list, for each line, the device name of the terminal, an optional user name who is to be logged onto the terminal automatically (static logon user), a reserved field, and the type of terminal connected to the I/O line. If no static logon user exists, the line is used for dynamic logins in which users must supply their name and password to access the system. If no initial program exists, the CLI is used.
`:config:logon.msg`	Contains the text of the message displayed on a terminal while the system waits for a dynamic logon user to log on.
`:config:udf`	User Definition File. User names, their encrypted passwords, and their user ID numbers are contained here. Used to validate login attempts. This file uses exactly the same format as the Unix `/etc/passwd` file, so it can be shared among iRMX and Unix systems in a networked environment.
`:config:user/*`	The * represents a set of files, one per authorized user. The file names match the user names in `:config:udf`. For each user, the corresponding file in this directory tells the minimum and maximum amount of memory the user is allowed to use, the maximum priority for user programs, the pathname of the user's home directory, and an optional initial program name.
`:config:termcap`	Contains information used by the CLI, the editor, the SoftScope debugger (version III) and other programs to determine how a particular user's terminal handles moving the cursor, clearing the screen, and other such control operations. The CLI's `set terminal` command is used to select an entry from this file that can then be accessed by all programs.
`:config:alias.csd`	A set of CLI *alias* commands that are to be established for all users who log onto the system. See `:prog:r?logon` below.
`:configilang386.als`	A set of aliases for running development tools. Submitted by `:config:alias.csd`.
`:config:r?env`	One of two files that are accessed by C programs to determine the value of environment variables using that language's *getenv()* function.
`:sd:user/*`	The home directories for users. The * normally matches the user's login name, but the actual path is determined by the contents of the user's `:config:user/*` file. The home directory contains another directory named `prog` that is referenced using the logical name `:prog:` in the following entries.

TABLE 2.1 Text Files Accessed When an iRMX System Starts Running, When Individual Users Log On or Off, and When the C Language *getenv*() Function is Called. (Continued)

File name	Purpose
`:prog:r?logon`	Contains commands that are automatically executed when a user first logs on to the system. This file normally contains commands to submit `:config:alias.csd` and `:prog:alias.csd` as well as other commands to tailor the environment to the user's preferences.
`:prog:alias.csd`	The other set of alias commands in addition to `:config:alias.csd` that are usually set up by a submit command in `prog:r?logon`.
`:prog:r?logoff`	Contains whatever commands a user wants executed automatically each time he or she logs off the system.
`:prog:r?env`	The second file that is used to resolve references to environment variables using the C language *getenv()* function.

with `:config:` are normally managed by a system administrator, while the files with names that begin with `:prog:` can be edited by individual users to tailor the system to their own needs. Note that file names that begin with `r?` (or `R?`) are invisible for normal directory listings. Use the `invisible` (abbreviated `i`) parameter on the *dir* command line to see these files. For example, the command line

```
iRMX> dir :prog: i                                    [20]
```

lists the names of all files in the user's `:prog:` directory, including invisible ones.

2.5 File Management

All I/O facilities of iRMX are provided by a layer of OS software called the *Basic I/O System*, or BIOS. This BIOS is not the same as the ROM-BIOS in a PC, although the iRMX for Windows BIOS makes some use of a PC's ROM-BIOS when accessing standard PC peripherals. More information about the iRMX BIOS, and the related EIOS, is provided in chapter 8. For now, just remember that BIOS is a software layer of the iRMX OS.

2.5.1 File protection

Each iRMX file or directory has four protection attributes associated with it. They are called *delete*, *read*, *append*, and *update* for files. For directories, *read* is called *list* and *update* is called *change* to reflect the semantics of directories more accurately. The BIOS keeps the protection attributes for

three different users of each file in a data structure called the file's *accessor list*. The three users are the file's owner (the user who created the file), and up to two others. User 0 (Super) has read access to all files, but to perform other operations on a file, even Super must be either the owner or one of the other two users identified in the file's accessor list.

Your access rights to files can be displayed using the `L` (Long) parameter of the *dir* command, or the `E` (Extended) parameter, which allows you to look at the entire accessor list for the files. For example, to see the accessor lists for all files in the `:system:` directory, type:

```
iRMX> dir :system: e                                    [21]
```

A lot of output is generated from this command, including four or five lines of output for each file. The first line for each file includes the name of the file and the name of the owner of the file. The accessor list will appear toward the right side of the output, in a section that might look like this:

```
ACCESSORS    ACC
0            DRAU
65535        -R--
```

This list is from a file owned by the Super user, so the first accessor in the list is 0, with ACCess rights of DRAU, which means delete, read, append, and update. The second user in this list is 65,535 (0xFFFF), which is the World user, who has only read access to this file. There is no third user.

Everyone who logs on to an iRMX system is given at least *two* user IDs, one of which is unique to the individual and the other of which is always 0xFFFF. By giving read access rights for this file to user 0xFFFF, everyone who logs on to the system can read the file without being added to its accessor list. iRMX, by the way, does not distinguish between reading a file to copy it somewhere and reading it into memory for execution. Since the sample accessor list is for a file that contains a system command, all users must have read access to the file to be able to run it.

The reason accessor lists allow three user IDs is based on the Unix file protection mechanism that provides independent access control for the owner of a file, a named group of users, and all other users. iRMX does not implement the notion of a named group of users, but users are assigned a group ID number when they log in if one has been established for them in the `:config:udf` file (see Table 2.1). The accessor list entries other than the first can contain any user IDs the file owner desires, or can be unused. There is no iRMX equivalent to the Unix *chown* command for changing the first ID in the accessor list for a file or directory.

The entire notion of an accessor list applies only to native-mode iRMX file systems. When iRMX is used to access the files on a DOS disk, it must work within the constraints imposed by DOS itself, which does not support

a file protection mechanism based on user IDs. Instet ad, iRMX for Windows just accepts all iRMX commands concerning the iRMX protection mechanism, such as *permit*, but treats all user IDs as iRMX's World user. The issue arises again in the context of network access to file systems managed by another operating system. For example, a Unix file system does not differentiate between the iRMX *append* and *update* privileges (Unix has a single privilege called *write*), and iRMX does not differentiate between the Unix *read* and *execute* privileges (iRMX has a single *read* privilege). These disparities are handled as transparently as possible, but the result is not always exactly what the user expected.

2.5.2 The file driver concept

A design feature of the iRMX BIOS is that all I/O devices look like files to application programs. This is true whether the I/O device is a disk containing real files, non-file devices such as terminals and printers, or devices and files accessed over a network. We have already seen how this feature provides good flexibility for application programs in the example that showed input and output redirection using the CLI '<' and '>' characters. A program that normally reads input from the console keyboard and writes its output to the console screen does so in the same way it reads and writes files, so that substituting disk files for both the keyboard and the screen devices is easy for the CLI to do without changing the program that does the I/O itself.

To accomplish this device independence, the BIOS uses a mechanism known as a *file driver*. The iRMX file drivers are called *physical*, *named*, *stream*, *remote*, and (for iRMX for Windows) *EDOS*. When the BIOS is first informed that a particular device is to be used, the BIOS is also told which file driver to use for that device. After that, all operations involving that device are automatically filtered by the appropriate file driver. If you attempt something that does not make sense for a particular file driver, such as accessing a named file on a printer, the file driver will reject the request, after examining it, with an error message, `E$IFDR`, which stands for "Illegal file driver function." (iRMX error-handling is introduced later in this chapter.)

On the other hand, if you try to access a file located on another computer system, the remote file driver will recognize the situation and automatically negotiate with the remote system to read and write the file over the network for you. More discussion on file drivers is provided in chapter 8.

2.5.3 Named files

The BIOS supports access to disk files by name using the named, remote, and EDOS file drivers. The *named file driver* supports the native-mode iRMX file system, the *remote file driver* supports any type of file system that can be accessed over the network (iRMX, DOS, VAX/VMS, or Unix

file systems), and the *EDOS file driver* supports MS-DOS file systems for iRMX for Windows. In all three cases, the file system is a way of organizing a disk volume to support a tree-structured hierarchy of directories and files. The root of the tree is the root directory, which contains the names of other directories and files. A particular file in the tree is uniquely identified by its full path name, which consists of the names of all directories, starting at the root, that must be accessed to locate the file. The </> character separates each directory level in a path name, so the path name `/dir1/dir2/file1` refers to a file named `file1` in directory `dir2`, which is listed in directory `dir1`, which is in the root directory. The character <^> can be used in a path name to indicate going up in the tree rather than down, as is indicated with </>. For the tree structure shown in Figure 2.1, the file `file2` could be accessed using either the path name `/d1/d2/file2` or `/d3^d1/d2/file2` (among others). Both DOS and Unix use <..> to represent the same thing as iRMX's <^>.

An iRMX directory is simply a file that contains a list of file and directory names along with an internal pointer to all the information known about the file for each file name. The actual information about the file, such as its size, its location on disk, and its accessor list, is kept in a separate file (called the *fnode* file) rather the directory itself.

File names. iRMX file names and directory names can consist of up to 14 characters. No distinction is made between upper- and lowercase letters. You can use just about any characters you want in file and directory names, such as multiple dots, spaces, and the like. If you want to put wildcard characters or special symbols inside a name, enclose the name, or the part containing the special characters, in quotation marks.

For example, <?> is a special character in a file name because it normally acts as a wildcard substituting for any single character. To copy a file named `:prog:r?logon` (the <?> is part of the file name) to another file

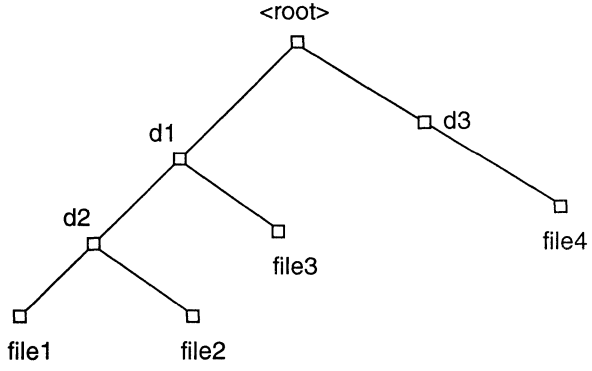


Figure 2.1 Sample file system tree structure.

named my logon file (note the two embedded spaces), you could enter one of the following commands:

```
iRMX> copy :prog:r''?''logon to ''my logon file''                    [22]
iRMX> copy '':prog:r?''logon to my'' ''logon'' ''file                [23]
```

iRMX file names might or might not include an extension, which is a set of characters following the last dot in the file name. However, several development tools do create names of new files by changing the extension part of a file name. For example, the editor normally saves the original contents of an edited file in a file with the extension part of the name changed to .bak. Editing a file named myfile.text results in the original file being preserved as myfile.bak and the edited version of the file being saved in myfile.text.

Because dots are part of the file name, *.* is different from * as a wildcard specification. The former consists of all files that have names with zero or more characters, followed by a dot, followed by zero or more characters. The latter refers to all files regardless of their names.

Hidden files are those that do not appear in normal directory listings. This feature is normally used to reduce the clutter of directory listings rather than for any particular security reasons. iRMX hidden files have names that start with r? or, equivalently, R?. You can always view hidden file names by using the *i* or *invisible* parameter to the *dir* command, as mentioned earlier. The file named :prog:r?logon is an example of a hidden file because :prog: is actually the path name for a directory (discussed in the next section), and the name of the file itself starts with r?. DOS supports hidden files with its hidden file attribute, and Unix hides file names that start with a dot.

Similar to file attributes, iRMX file naming rules only apply to iRMX file systems. If you access files from another OS over a network or use the iRMX for Windows EDOS file driver to access MS-DOS files on a PC, some sort of mapping must exist between the file names of the two systems, which is never perfect. For example, MS-DOS file names cannot have <?> in them, so the EDOS file driver drops the r? from hidden file names and sets the MS-DOS hidden attribute for them. The EDOS file driver also forces file names to conform to the DOS 8.3 rule (a maximum of 8 characters in the base plus a maximum of 3 characters in the extension) by shortening the base and extension parts of the file name as necessary. The EDOS file driver totally rejects iRMX file names that contain illegal DOS characters, such as multiple dots.

Logical names. A *logical name* is an identifier for a device, remote computer system, file, or directory. For now, our focus is on logical names for directories. Logical names are normally written with colons around them to distinguish them from regular file, directory, or device names, but the

colons can sometimes be omitted when no ambiguity exists, for example, when the only type of file that would appear somewhere is a logical name. Logical names follow the same rules as file and directory names except that they are limited to 12 characters between the colons, so that even with the two colons they are never longer than file or directory names.

Two major reasons exist for having logical names. First they can save a lot of typing. If you want to reference the files in the directory `/user/jones/project1/source` many times, for example, you could assign the logical name `:s:` to the directory and refer to a file in it as `:s:mainprog.c` instead of `/user/jones/project1/source/mainprog.c`. The command to set up logical names for files and directories is *attachfile*, so the command to create `:s:` would be:

```
iRMX> attachfile /user/jones/project1/source as :s:                [24]
```

This is a case, by the way, where the colons around the logical name can be left off. The identifier after the `as` in an *attachfile* command is always a logical name, so the colons around `s` are optional.

Logical names are much more than just a convenience to save users typing time. They also contribute to increasing the speed at which disk files are accessed. To locate a file with a long pathname, the BIOS must read each directory in the path from disk into memory to find the location on the disk of the next directory in the path. The BIOS then repeats the process for each directory in the path until it locates where the file is stored. Each disk access involved in processing a path name requires time, but the BIOS saves the information about the disk location of the last item in the path (either a directory or a file) when you set up a logical name, so the BIOS does not need to repeat the search process again when the logical name is used instead of the full path name.

Logical names for directories can always be used as the first part of a path name. For example, the logical name `:s:` defined above could be used to access a file named `/users/jones/project1/source/old/first_try.c` by using the path name `:s:old/first_try.c`. The second form eliminates the time needed for disk accesses to the root directory, to the `users` directory, to the `jones` directory, to the `project1` directory, and to the source directory when the file is first accessed. Furthermore, the same overhead would be eliminated for accesses to all other files in `source`.

Logical names for devices look just like logical names for files or directories, and the two are generally interchangeable. In fact, the logical name for a disk device with a named file system on it can be used as a logical name for the root directory of the file system. For example, `:sd:` is the logical name for the system disk, the disk from which the OS was bootstrap loaded, so `:sd:d1/d2/file1` is generally the same as `/d1/d2/file1`. The two

forms do not always have identical meanings, however, which brings us to the next topic.

Current and home directories. The current directory is that directory which serves as the default when you specify a file name only without an explicit path name before it. The logical name `:$:` always refers to the current directory, and the current directory is changed with the HI command, *attachfile*. For example, to change the current directory to `/user/jones/projectl`, enter the command:

```
iRMX> attachfile /user/jones/projectl as :$:                    [25]
```

This form of the *attachfile* command is so commonly used that `as :$:` is the default for the preposition and output file list if nothing else is specified on the *attachfile* command line. Most iRMX users set up the alias *cd* for *attachfile* so that the same effect as [25] is obtained by:

```
iRMX> cd /user/jones/projectl                                   [26]
```

The colons around <$> are almost always optional, since it is impossible to start a file name with the <$> character. As a result, the names `:$:filel`, `$filel`, `filel`, and even `''$filel''` all refer to exactly the same file. The HI *path* command displays the current path name of `:$:`.[8]

When you first log on to an iRMX system, the current directory is the home directory that was assigned to you when your account was first set up. The logical name for your home directory is `:home:`, and its position in the file tree can never be changed. Thus, `:$:` and `:home:` are both logical names for the same node in the file tree when you first log on. No matter where you go with your current directory, you can always return it to your home directory with the command:

```
iRMX> attachfile :home: as :$:                                  [27]
```

This operation is so commonly done that the *attachfile* defaults to this command when it is entered without command line arguments. Command [27] is similar to the *cd* command in Unix, but not to the *cd* command for DOS (which does what the iRMX *path* command does!).

The difference between the two path names used previously, `:sd:dl/d2/first` and `/dl/d2/first`, is based on the meaning of the </> at the beginning of a path name, which always refers to the root directory of the

[8]DOS and Unix users should note that the iRMX *path* command is not the same as the DOS and Unix *path* commands, and no corresponding iRMX command exists for the Unix and DOS *path* commands.

file system that holds :$:. If you use the *attachfile* command to move :$: to another disk, </> becomes the root directory of that other disk, but :sd: remains the root directory of the system disk, normally the one from which the OS was bootstrap loaded.

Other standard logical names. So far, I have mentioned the logical names :sd:, :home:, and :$:. An essential difference between :sd: and the other two is that every user who is logged on to a particular system is referring to the same location with :sd:, but each user has separate copies of the logical names :home: and :$:. Logical names that are the same for everybody, like :sd:, are called *system logical names*, and logical names that are different for each user on the system, like :home: and :$:, are called *user logical names*. Examples of *system* logical names include the following:

sd:	The system device or the root directory of the system device.
:system:	The directory that contains the HI system commands.
:utils:	The directory that contains utility programs.
:util286:	On iRMX for Windows and iRMX III systems, the directory that contains utility programs that run under iRMX II, iRMX III, or iRMX for Windows.
:lang:	The directory that contains the development tools.
:config:	The directory that contains certain configuration information needed by the HI, including the User Definition File (UDF) that contains users' passwords, the terminals file that identifies static and dynamic logon terminals, and the r?init file that is *submitted* when the system is initialized. This directory also contains the loadinfo and rmx.ini system initialization files on iRMX for Windows and iRMX III systems.
:rmx:	A directory that contains operating system-dependent files. For iRMX III and iRMX for Windows, this directory is :sd:rmx386.
:bb:	A pseudo-device that discards all information written to it and that always returns an end-of-file indication when read from. The name is an abbreviation for byte bucket. The device is similar to Unix /dev/null and DOS nul devices.

The standard *user* logical names are the following:

:home:	The user's home directory, which can never change.
:$:	The current working directory. It can be changed with the *attachfile* command.
:prog:	The same as :home:prog. The directory containing the user's r?logon and r?logoff command files. The name is meant to suggest that this is the proper directory for keeping executable

copies of utility or application programs that the user has developed.

`:ci:` The user's console input device. Normally the terminal or computer keyboard, but can be redirected to a file or other device using '<'.

`:co:` The user's console output device. Normally, this is the terminal or computer screen but can be redirected to a file or other device using '>'.

`:term:` The user's error output device. Same as `:co:`, but cannot be redirected.

Use the *logicalnames* command to display all the logical names defined on a system. Include the *long* option on the command to see what each logical name represents. The alias *logs* is normally defined for the *logicalnames* command to save typing.

Search path list. The *search path list* is the list of directories that the HI searches to find a file containing a command to be executed. This list is established when the system is set up, and it cannot be changed by users. Thus, all users on an iRMX system must use the same search path. The normal search path list for iRMX for Windows is the sequence of logical names:

```
:prog:
:utils:
:util286:
:system:
:lang:
:$:
:rmx:
```

The search path list mechanism can be both convenient and disconcerting. It is convenient, for example, if you develop a program named *copy* and place it in your `:prog:` directory. You can run it by typing

```
iRMX> copy a to b                                    [28]
```

The HI will find your version of *copy* to execute before it finds the normal iRMX *copy* command because `:prog:` comes earlier than `:system:` in the search path list. A system administrator could achieve the same result for all the users on a system by putting a local version of a system command in the `:utils:` or `:util286:` directory.

The HI uses the search path list only for command lines that do not include an explicit path name for the file containing the command to be run.

For example, if you have your own version of *copy* in `:prog:` but want to use the standard version that is in `:system:`, use a command line like the following:

```
iRMX> :system:copy a to b                                    [29]
```

If the first character of a command line is <:>, </>, <^>, or <$>, the HI recognizes that an explicit path name is being used and ignores its search path list and goes immediately to the specified file to get the command to be executed. You can use an alias to make an HI command start a little faster by specifying an explicit path name in the alias. For example,

```
iRMX> alias cd = :system:attachfile                          [30]
```

The search path list mechanism can be disconcerting, however, if a file with the same name as a command that you want to execute is in one of the directories in the HI search path. For example, if you create a text file named `copy` in your `:prog:` directory and then issue a *copy* command, the HI would find your text file in `:prog:copy` and try to execute it. The command would "mysteriously" fail because the text file named `copy` is not an executable program. The error message you would get would be: `E$BAD_HEADER, while loading command`, which means that the first part of the file (the header) was not in the proper format to be treated as an executable file.

Another disconcerting phenomenon occurs when an installation uses a different search path from the one previously given.[9] For example, consider what happens if `:utils:` comes before `:prog:` in your system's search path list and you develop a program that happens to have the same name as a utility command that you didn't know about. The first time you test your program you will actually run the utility with the same name. You can lose a lot of time trying to understand why your program is doing things you never coded into it!

One last point while we are discussing the HI command search mechanism: there is no rule about naming executable files differently from other files. If you put an executable program in a file named `myprog.exe`, then the name of the command is *myprog.exe*, not *myprog*, or anything else. The HI always looks for an exact match between the name of a command and a file name.

---

[9] iRMX for Windows does not support a search path other than the one given above. A different search path can only be set up for systems that can be reconfigured using the ICU.

2.5.4 Using floppy disks

Floppy disks are good for saving copies of your work and distributing files to other people, so learning how to use them with iRMX has practical advantages. Using floppy disks also introduces you to some fundamental I/O concepts that are developed further in chapter 8.

Before any device can be used with an iRMX system, someone must create a device connection that gives a logical name to the device. Many device connections are automatically created when a system is first started, such as `:sd:` mentioned above. The *attachdevice* and *detachdevice* commands can be used to create and delete device connections and are the usual technique for handling the device connections for floppy disks.

The *attachdevice* command requires you to specify the name of the device, the associated logical name, and the name of the file driver to be used with the device. Device names are sometimes called *physical names*, or DUIBs (pronounced "doo-ib"), because they are the names of BIOS structures called Device Unit Information Blocks. iRMX for Windows and iRMX III support a command called *physnames* that can be used to obtain information about the DUIBs available on a system.

To use DOS diskettes from iRMX for Windows, you must use the EDOS file driver and the device names `a_dos` and `b_dos` to refer to the PC's `A:` and `B:` drives. If you booted DOS from the `C:` drive, `c_dos` would have been attached as `:sd:` when iRMX started running. A typical *attachdevice* command would be

```
iRMX> attachdevice a_dos as :a: edos                    [31]
```

If you are running iRMX for Windows and only using DOS-formatted diskettes in your system, this command needs to be issued just once, from either a user's `r?logon` file or from the system's `:config:r?init` file. A diskette must be in the drive when this command is issued for the command to work.

iRMX has its own way of formatting diskettes, however, that are advantageous for users who want longer file names and the security of the file protection mechanism that DOS cannot provide. These are the same advantages that might prompt you to install an iRMX-formatted partition on the hard drive of an iRMX for Windows system, by the way. Of course, pure iRMX systems must rely on the native iRMX file system, although that system is flexible enough to allow iRMX users to import and export DOS diskettes by using some utility programs.

Using iRMX diskettes is a more involved process than using DOS diskettes under iRMX for Windows, but understanding the process helps lay the groundwork for a deeper understanding of how the entire I/O system works.

First, there are dozens of floppy disk DUIB names to choose from when doing an *attachdevice* for an iRMX floppy diskette, depending on the size, density, and format of the diskette, as well as on which particular hardware controller (and thus, which computer platform) is being used for the drive itself. Knowing which device name to use can be a bit of a challenge. You can use the *physname* command to list the DUIBs on a system, but you have to know the meanings of the fields in a DUIB to decide which one is correct for your needs. For iRMX for Windows, the list is in Appendix A of the iRMX for Windows *iRMX Command Reference.* For other versions of iRMX, the list is in the documentation for the *attachdevice* command itself, which is in the *iRMX Operator's Guide.* Below is an example of an *attachdevice* command for a 3.5″ 1.44 MB floppy disk located in the `A:` drive of a PC:

```
iRMX> attachdevice amh as f named                    [32]
```

After line [32] has been entered, both the device connection and the root directory of the iRMX file system on the floppy disk will be known as `:f:`. You could type

```
iRMX> dir :f:                                        [33]
```

to list the names (files or directories) in the root directory of the floppy, or

```
iRMX> copy * to :f:                                  [34]
```

to copy all the files in the current directory to the root directory of the floppy. Line [34] might not do exactly what you want, though, because *copy* will treat any subdirectories in the current directory as files. You will end up with a file rather than a subdirectory on the floppy with the same name as the subdirectory, but without the contents of the subdirectory on the floppy. The file with the name of your subdirectory will be a data file that contains the names and fnode pointers for the files in your original subdirectory. If you want to copy directories and subdirectories, you must use the *copydir* command instead of *copy.*

To delete a device connection, use the *detachdevice* command, which simply needs the logical name, with or without colons, as an argument. For example:

```
iRMX> detachdevice f                                 [35]
```

Now for the messy part. Every time you remove an iRMX-formatted diskette from a drive and insert another one, you must do a *detachdevice* command and another *attachdevice* command, even if the two diskettes are formatted exactly alike and even if all the files on the first diskette were

properly closed before removing it. *If you don't do this, all the information on the second diskette will be destroyed the first time you write to it.* The explanation for this unusual (to put it mildly) behavior illuminates a design philosophy of iRMX nicely.

The iRMX BIOS improves diskette performance by keeping certain housekeeping information about the file system in RAM rather than going to the disk to get the information each time the diskette is read or written. It continues to use this information from memory until the device is detached. Thus, accessing the second diskette without detaching the first causes the BIOS to update basic file system information for the second one incorrectly. If the second diskette is write-protected, it will not be damaged, but you will not be able to read from it correctly until you detach and reattach the device.

You do not have to do this procedure with DOS diskettes, even if they are accessed from the iRMX side of iRMX for Windows, because DOS always assumes a diskette has been changed when you access it and re-reads its housekeeping information from the diskette every time it is accessed. iRMX for Windows users will encounter the same overhead when accessing DOS disks because the EDOS file driver uses DOS I/O routines to do the actual disk I/O.

It is not always this way for iRMX. All 8-inch diskette drives and most early 5.25-inch diskette drives had a contact switch in the door that sent a signal to the processor whenever someone opened the drive door to change diskettes. When this switch is present, iRMX detaches and attaches the device automatically. Most 5.25-inch and 3.5-inch drives today, however, don't generate this signal, so the process must be done manually when using the drives. Rather than degrade the speed of the system's floppy disk system, the designers of the iRMX I/O system placed the burden on the operator to use the system correctly. Other systems are willing to sacrifice performance to provide a more user-friendly environment.

The flexibility of the iRMX I/O system should not be overlooked in this context. One reason there are so many DUIBs for floppy disks, for example, is that they are easy to create. Adding a new DUIB to a system, such as to support a different number of sectors per track, simply involves loading a device driver with the correct parameter values when an iRMX for Windows or iRMX III system initializes. For systems that support configuration using the ICU, the process consists of filling in a few menu screens and then building a new copy of the OS. The entire ICU configuration process can be completed in 15 minutes.

2.5.5 Accessing network files

This section assumes you have access to a local area network that supports ISO transport layer connectivity. Examples include Intel's OpenNet for iRMX, Xenix, Unix System V, VAX/VMS, and DOS. OpenNet uses ISO

standard protocols to pass messages among file servers and consumers. The messages themselves follow Microsoft's Server Message Block (SMB) format, so the list of compatible systems might expand if other vendors choose to follow these standards for their networking products. Internetworking with Transmission Control Protocol/Internet Protocol (TCP/IP) can be accomplished by connecting through Unix systems, and direct support for TCP/IP under iRMX is being developed by Intel at the time of this writing.

Because the iRMX for Windows EDOS file driver allows iRMX users and programs to access DOS peripherals, users running a Novell network on a PC with iRMX for Windows can access network drives from the iRMX side the same as they access local DOS disks. Chapters 11 and 12 provide information about network programming, as well as more information about internetworking with TCP/IP and Novell networks from iRMX systems. The following material applies to the use of the OpenNet networking facilities for iRMX, called iRMX-Net.

The remote file driver makes it as easy to access files across a network as on the system you are logged in to. It may be a bit slower, but it is just as easy. The process is just like using a floppy disk: issue an *attachdevice* command to create a device connection to a remote system and to give that connection a logical name. Then, use that logical name just like any other logical name as the first part of a path name.

To issue the *attachdevice* command, you need to know the name of the remote system to which you want to connect, just as you had to know the device name of the floppy disk in the previous section. The difference between DUIB names and network names is that you can see a list of available network names by using the *netstat* utility program, available from the user's group iRUG. You use the remote file driver for attaching over the network:

```
iRMX> attachdevice system1 as 1 remote                    [36]
```

Assuming there is a computer currently up on the network that has set its network name to `system1`, line [36] will create the logical name `:1:`. You can see what public directories that system has offered to the network by giving a *dir* command

```
iRMX> dir :1:                                             [37]
```

The directory you will see is analogous to the root directory of a disk volume. This directory is called a *virtual root*, and important differences exist between a virtual root and the root of a disk volume. The first difference is that you can never write anything to a virtual root directory over the network. Only users logged on to the local system can change the contents of that system's virtual root directory, which is done by using the *offer* and

remove commands. You can use the *publicdir* command to see your local system's virtual root directory.

The other difference between the two directories is that a virtual root directory can contain names for both devices and disk directories, and the disk directories can come from different disk volumes or from different directory levels in a single volume. Assume that the following *offer* commands have been executed on `systeml:`

```
iRMX> offer :f: as floppy                                    [38]
iRMX>offer :sd:user/jones/projectl as projl                  [39]
iRMX>offer :sd:user as usr                                   [40]
```

Assuming that a floppy disk has been attached with the logical name `:f:`, line [38] lets remote users access the root directory of the floppy by using the public directory name `floppy`. For example, the user who accessed `systeml` with the logical name `:l:` (line [37]) could copy a file to the floppy with the command:

```
iRMX> copy myfile to :l:floppy/myfile.backup                 [41][10]
```

Line [39] demonstrates the use of a subdirectory as a public directory. Creating a public directory can be done either to save remote users the trouble of typing long pathnames, or to restrict remote users to accessing only parts of a disk's file system. A remote user would not be able to access the `jones` directory, for example, by referring to `:l:projl^`. That is, you can't go up from an entry in a remote system's virtual root directory. Line [40] illustrates that there might be different directories from the same disk in a virtual root, and, in this case, the path name `:l:usr/jones` would refer to the `jones` directory.

OpenNet distinguishes between file consumers and file servers. In the above examples, `systeml` was a file server, and the computer from which the user issued the *attachdevice* command was a file consumer. In practice, most OpenNet systems are configured as both servers and consumers simultaneously, with the exception of pure DOS systems, which cannot be servers without shutting out command processing for the local user. Networks like OpenNet that are based on systems acting as both file servers and file consumers are called *peer-based systems.*

The flexibility of peer-based systems can be seen using an example. Assume that `systeml` and `systemu` are remote iRMX and Unix computers that have been attached with the logical names `:l:` and `:u:` by a user on a local system running iRMX for Windows. The following command could

[10] Assuming that `myfile.backup` is not the name of a directory, this is the first example of a *copy* command that creates a file with a different name from the original file.

be issued by the iRMX for Windows user to copy files from the iRMX system to the Unix system:

```
iRMX> copy :1:user/smith/*.txt to :u:usr/jones/*.bak          [42][11]
```

Of course, this example works only if the iRMX for Windows user has proper access rights to the directories and files on the two systems.

2.6 Printing Files

Printer support for iRMX ranges from rudimentary to modestly rich. On a single-user system with a local printer, you can print files by using *copy* with the printer device as the destination. Of course, the device must be attached first. On a network or multiuser system the situation must be more complex to prevent multiple users from writing to the same printer at the same time. Unix systems provide good print spooling facilities for controlling this situation. Thus, a Unix system on the network can be a good resource for managing shared printers. Some DOS systems support print spooling as well. A print spooler for iRMX is also available from iRUG, along with an `rprint` command used to send files to that spooler or to Unix or DOS systems.

2.7 Remote Login

OpenNet supports logging on to a remote system through a mechanism called *Virtual Terminals* (VT) available from Intersoft, Inc. in Lake Oswego, Oregon. An iRMX system must be explicitly configured as a VT server to allow remote users to log on. Unix OpenNet systems, however, normally act as virtual terminal servers by default. iRMX for Windows systems are not configured to be VT servers, but a VT server can be started after the system is loaded.

The *vt* utility command is used to gain login access to a remote virtual terminal server. The only argument to the command is the network name of the remote computer. When the access is successful, you will see the `logon:` prompt from the remote computer, and you can log on as usual. Logging off returns you to the remote system's `logon:` prompt. To break the connection and return to your local system, type the sequence of characters, <cr><~><.><cr>. (That's "tilde-dot" at the beginning of a line.) Breaking the connection also logs you off the remote system if you have not done so already.

The difference between working by remote login and using remote file access is a matter of which computer runs your commands. Using remote

[11] This is the first example of a *copy* command that shows the input path list and output path list matching through wildcards. The example copies all of smith's `.txt` files to files with the same base name in jones directory, but with the extensions changed to `.bak`.

file access, you could change your current directory to a directory on another computer's disk using the *attachfile* command, and copy, delete, and otherwise manipulate files just as if they were on your own computer disk. A *copy* command, however, is run on the local computer, and the files being copied pass over the network to your computer on the way to their destination. That is, to copy a file from one remote computer to another remote computer, as in the iRMX to Unix example in line [42], you must copy the file over the network twice, once from the remote computer to the local computer's memory and again from the local computer to the remote computer's file system.

With remote login, commands are executed by the remote system. If you do a remote login to a Unix system, for example, you must use *cp*, the name of the Unix copy command, to copy files. If you were to copy files across directories on the Unix system itself, the only information to travel over the network would be the characters you type and the messages that appear on your screen, not the actual files.

2.8 Error Conditions

Many things can go wrong when you run a command. iRMX provides a mechanism called an *exception handler* to deal with these situations. An exception handler is a subroutine that is called automatically whenever an error is detected by either the hardware or the operating system. Errors the hardware detects include arithmetic faults, such as division by zero, and general protection (GP) faults caused by illegal memory accesses on the 80286 and above processors. Errors the OS detects are always associated with system calls (subroutines in the OS called by application programs to perform OS functions).

Whenever a system call detects an error, it generates a numeric condition code, also called an exception code, to identify the cause of the error. The operating system passes that code to the subroutine set up as the exception handler. Application programs have complete control over which routine handles exceptions, but the OS supplies a default handler, the system exception handler, in case no other routine has been specified. For example, the HI sets up its own exception handler for the commands it runs, which is in effect unless the application overrides it with its own exception handler. The HI exception handler always aborts any command that causes an error after displaying a message on the console output device.

The term *exception* is used instead of *error* because it more accurately reflects what is being detected rather than out of some need to use delicate terminology. Some exceptions really are errors, such as passing an illegal value as a parameter to a system call. But some exceptions are beyond the control of the programmer, such as trying to write to a printer that is out of paper. The exception handler mechanism lets programs deal with these two classes of exceptions differently.

Some configurations of iRMX are set up to use a monitor program as the exception handler for errors detected by the hardware. A *monitor* is a debugging facility that allows you to examine and modify registers and memory and to execute individual machine instructions. Some monitors are implemented as software loaded with the OS or shortly afterward, while other monitors are stored in ROM so that they are available as soon as power is supplied to the computer. Using a monitor program as an exception handler can be valuable when you are debugging an application, but for now it's more likely that you will simply want to abandon the program that caused the error and return to the CLI's `iRMX>` prompt. You will know when you are in the monitor program because it prompts for commands with two dot characters (..), usually after displaying a message about what the error was and the memory address of the machine instruction that caused the problem.

The command to exit the monitor is *g*, followed by an address. For iRMX III and iRMX for Windows, use `g284:1c`, while for iRMX II, `g284:14` should work. iRMX I is a real-mode operating system (see chapter 5 for an explanation or real and protected modes), so there are no hardware traps that will take you into the monitor. iRMX for Windows can be set up to either break to a monitor or abort a program that encounters hardware faults. The choice is made by setting a parameter called `DEH` in the system's `:config:rmx.ini` file to true (`0FFh`) if faulting programs are to be aborted, or to false (`000h`) if faulting programs are to cause a break to the monitor.

A third type of error should be included here: those errors detected by an HI command because of invalid input data. For example, if you provide a compiler with a source program that contains syntax errors, the compiler (which is an HI command) will detect the problem itself and issue a diagnostic message.

The question remains what an iRMX user can do when faced with an error message from an exception handler, such as the HI's, that contains some cryptic string, such as `0021: E$FNEXIST`. The number is the condition code in hexadecimal generated by a system call, and the `E$` string is a mnemonic name for that code. In this case, a *F*ile does *N*ot *EXIST*. A complete list of exception codes, their `E$` names, and a brief statement of what each one generally means is available to iRMX for Windows users through the DOS *rmxhelp* command supplied with that system. All the information available with *rmxhelp* is also contained in the *iRMX System Call Reference* manual, Volume 9 of the iRMX for Windows documentation set, if you prefer to work from hard copy documentation.

If you get an error message when you run a command that someone else programmed, it usually means you did something wrong. If you do not know what the problem is from the exception code (and by reading the doc-

umentation for the command!), you can only write down the message and find someone who can help you. When running your own code during iRMX programming, such messages will tell you that your application is not yet fully debugged, and you will need to determine which system call in your program generated the exception and then fix the problem.

Chapter

3

Developing an Application

3.1 Overview

Developing a real-time application involves two processes. The first is to design the application to match the requirements of the project using the resources available to implement it. The second is to construct the executable code.

For an iRMX application, the design process includes deciding what tasks are needed for the application and how the tasks will synchronize and communicate with each other and the external environment. Later chapters in this book present the resources that are available for implementing real-time applications on iRMX systems. Formal or structured methodologies for designing real-time applications are outside the scope of this book; the assumption is the design will be completed using either formal or informal techniques and proceed from there.

Once the design of a real-time application has been completed, the second goal of constructing executable code can begin, which is the subject of this chapter. The executable code might be a Human Interface (HI) command that is loaded into memory from a disk file each time it's run, it might be a device driver that enables the OS to work with a new peripheral device, or it might be a new set of system subroutines that will be built into the OS and loaded with it when the system is initially bootstrap loaded.

Development is done in a cycle that includes editing, compiling, linking, and testing stages. Errors can be detected at any stage in the cycle, at which point the cycle returns to the beginning, the editing stage. At each stage a development tool is used to transform a disk file in some way. These development tools include text editors, compilers, linkers or binders (I do not distinguish between these two terms), and symbolic debuggers.

Central to understanding the development process is the subject of program *modules*, and an overview of the types of modules involved in the de-

velopment cycle is presented below. The chapter proceeds with a discussion covering types of modules and disk files involved in developing code.

3.1.1 Program modules

The types of modules used in developing applications are called *source, object,* and *loadable* modules. A source module is an ASCII text file produced by a text editor. An object module is the machine language representation of a source module that a compiler or assembler produces. Each run of a compiler accepts exactly one source module as input and produces exactly one object module for output. Most compilers allow pieces of source code, called *include files,* to be inserted into a source module during the compilation process. An object module must be processed by a linker or binder before it is in a format suitable for execution. The linker or binder normally combines several object modules to produce a single loadable module. This book describes two types of loadable modules, although other types exist. One loadable module is called a *single task loadable* (STL) module, which is the type of module found in an executable disk file, such as an iRMX HI command. The other loadable module is called a *bootstrap-loadable* module, which is loaded for execution without any assistance from the OS, often because it is the OS itself.

The generic term *linkable module* refers to any module that can be processed by a linker or binder. A linkable module can be a single object module produced by a compiler or assembler, or it can be constructed from several individual object modules by a previous execution of the linker or binder. A linkable module is not ready to be loaded into memory for execution; it must be converted to an STL or bootstrap module first. A major difference between a bootstrap-loadable module and an STL module is that a bootstrap-loadable module must be loaded into a fixed part of memory for it to run, but an STL module is relocatable; it can be loaded into any part of memory for execution.

Figure 3.1 shows the steps in the development process for an STL module, and Figure 3.2 shows the steps in the development process for a bootstrap-loadable module. Both figures show the process in terms of the files and development tools involved, with files represented by circles and development tools by rectangles. Source modules and include files are prepared using a text editor, an object module is generated from a source module by a compiler (or assembler), and both STL and linkable modules can be produced by a linker or binder. Figure 3.1 shows the binder being used to produce an STL module that can be run as an HI command, and Figure 3.2 shows the binder producing a linkable module, which is then combined with other linkable modules by a special binder, called the *system builder,* to produce a bootstrap-loadable module.

The structure of a source module is determined by the programming language being used, and the structure of all other types of modules is deter-

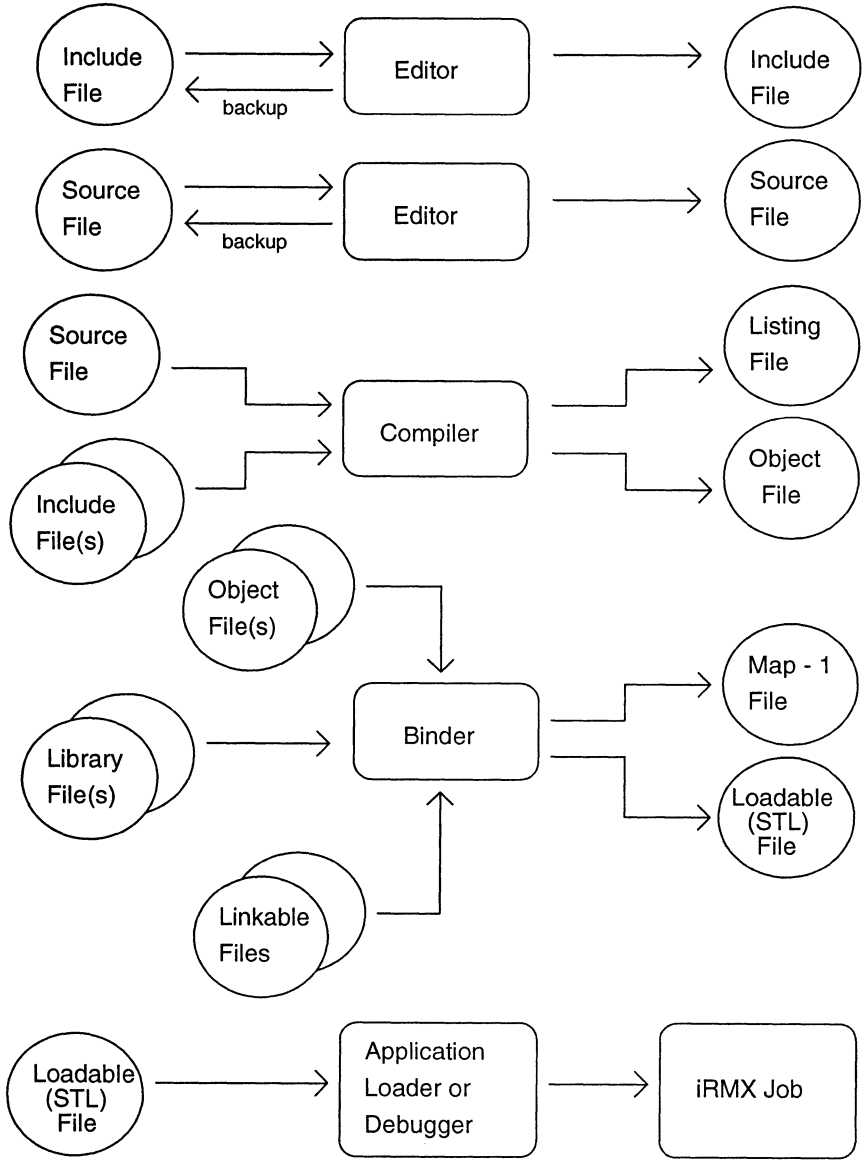


Figure 3.1 Development steps for an STL module, such as an HI command file.

mined by a formal specification called the *Object Module Format* (OMF). Intel publishes different OMF specifications, depending on the architecture of the target-system microprocessor. Although iRMX I and MS-DOS both run on the same microprocessors, the 8086, and compatible architectures, Microsoft chose to use a slightly different OMF from Intel's for both

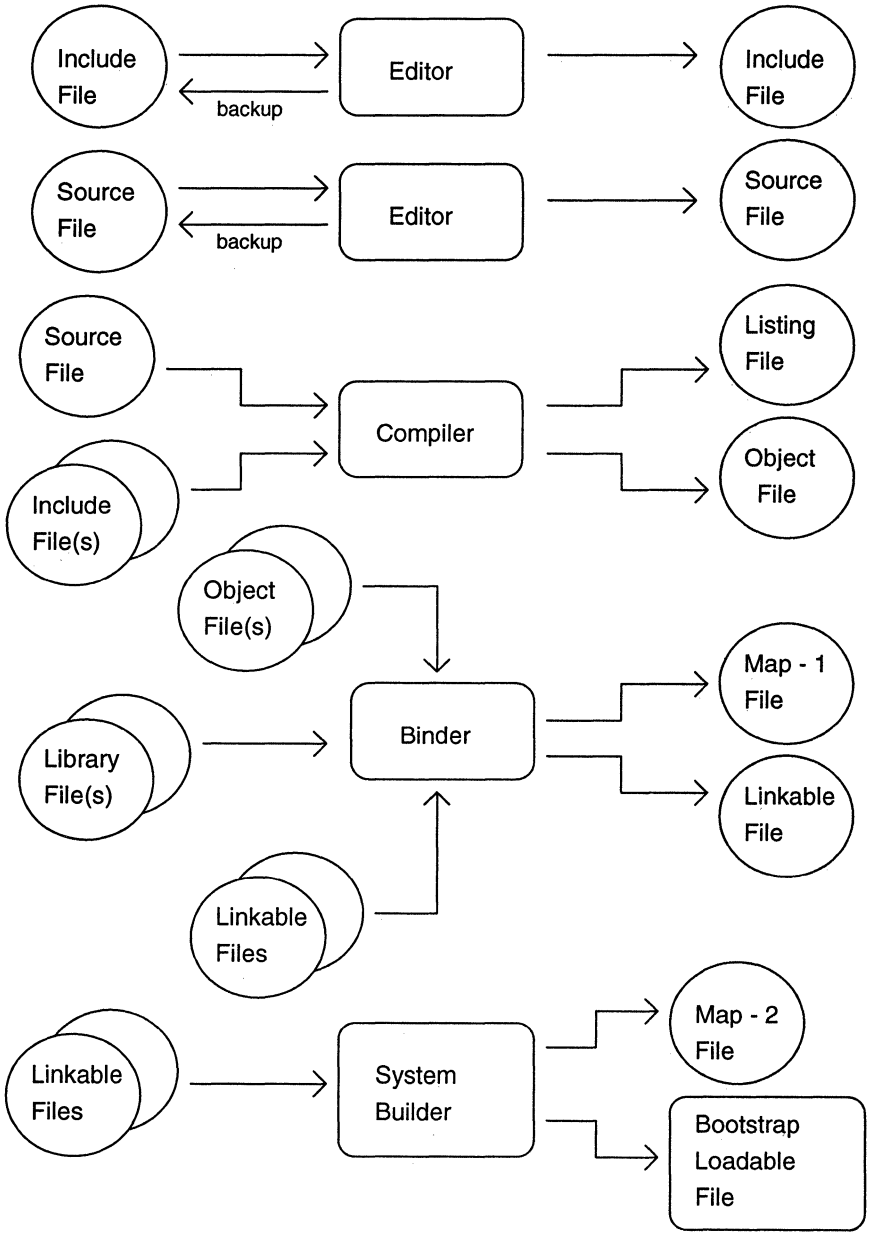


Figure 3.2 Development steps for a bootstrap-loadable module, such as an operating system image file.

object modules and loadable modules. The iRMX I linker will accept object modules produced by DOS compilers, but the iRMX program loader will not accept loadable files that are in Microsoft's OMF. Because almost all programs must make system calls to a particular OS to perform such tasks as I/O and memory management, the inability to load DOS programs under iRMX is not really an issue because a DOS program would fail as soon as it tried to make a DOS system call.

To iRMX users, this means you cannot run your favorite DOS spreadsheet or word processing program on iRMX, just as you cannot run them under any other operating system installed on your PC, such as Unix. With iRMX for Windows, however, you can run your DOS applications on DOS while running your iRMX applications on iRMX because both operating systems are in the PC at the same time.

3.1.2 Development and target environments

Before you can develop any program, you must decide on your development and target environments. The most popular development environment for iRMX applications is a PC running MS-DOS and, optionally, Microsoft Windows. The PC can be used with DOS to run all iRMX development tools, regardless of the target platform and version of iRMX (I, II, or III). Alternatively, an iRMX system can be used as the development system. iRMX I can be used only to run the development tools for iRMX I targets, but both iRMX II and III can be used to develop applications targeted for any of the three versions of the OS. With iRMX for Windows, the DOS and iRMX III development environments are just a keystroke away from each other, so a mixed development strategy can be chosen.

Development tools that run under DOS are sometimes called *DHDT*, which stands for DOS-Hosted Development Tool. Likewise, tools that run on iRMX are called *RHDT* (RMX-Hosted Development Tool), which are also sometimes called *native-mode* tools. Virtually all of Intel's DHDTs can be run on iRMX II or III as well if they are invoked under the control of a special utility program called *run86*. A bit of legerdemain is required to accomplish this feat, which relies on the development tools internally using the special set of system calls, the *Universal Development Interface* (UDI). The *run86* program provides a special UDI-to-iRMX system call translator that is invoked as the development tool runs. Although *run86* enables you to run DHDTs on iRMX, remember that you still cannot run normal DOS applications from the iRMX prompt, because normal DOS applications make DOS system calls, not UDI system calls.

The choice of a target system depends on the application. Small embedded applications (less than 1 MB, including the operating system) that can operate effectively without the benefits of hardware memory protection can be targeted for iRMX I systems. iRMX I applications are called *real-mode programs* in reference to the name for emulating the 8086 architec-

ture with an 80286 or greater microprocessor. (The architecture of Intel microprocessors is reviewed in chapter 5.) Relatively small protected-mode applications (less than 16 MB, including the operating system) can be run effectively on iRMX II systems. However, performance degrades if the code or data for the application exceeds 64 kilobytes (KB) because of a 16-bit limit on memory segment offset values for the 80286 architecture. Choosing an iRMX III target system can significantly improve performance of large applications because the 80386 architecture allows 32-bit segment address offsets (4-GB segment sizes), which essentially removes any size limitations on code and data. Protected-mode applications are called 16-bit or 32-bit, depending on if they were developed for iRMX II or III.

A valuable feature of iRMX III is that it runs 16-bit applications without any changes. If your target is an iRMX III system (including iRMX for Windows), you can choose to develop and run either 16-bit or 32-bit code. The tradeoff for 16-bit code is that the same loadable module can be used on iRMX II, iRMX III, or iRMX for Windows, but it will generally run slower than 32-bit code.

3.1.3 Development steps

Independent of the target system decision is whether your application will be run as an HI command or configured into the OS itself. HI commands are loaded into memory when the user enters the command name at the keyboard, and the memory they use is then freed when the commands terminate. Resident programs, on the other hand, never terminate and continue to occupy memory until the system is rebooted or shut down.

The distinction is primarily whether the application executes under user control or executes under the control of external events. This distinction is closely related to the difference between conventional and real-time applications, but there are many exceptions. First, it is very convenient to run real-time applications as HI commands while they are being developed. When an error is found, the command can be aborted, repaired, and re-run without reloading or rebuilding the OS. The application can finally be configured into the OS after it has been debugged. Another possibility is a hybrid application, in which part of the application acts as an extension to the OS and is made resident, providing functions that can be invoked by dynamically run HI commands. A device driver is an example of this sort of code, and there are many others as well.

Resident programs can be either incorporated into the loadable module that contains the operating system image itself or loaded into memory after iRMX starts running. The latter option requires the use of a special program called *sysload* that is available only for iRMX III and iRMX for Windows systems.

Most of the steps used to develop HI commands and resident applications are identical. Each step in the process consists of running a development tool that reads files as input and generates new files as output. As

Figure 3.1 shows, the steps to develop HI commands are editing, compiling, and binding. The STL file containing the resulting program can then be loaded into memory for execution either by a debugging program or by a part of the OS called the *Application Loader* (AL). A resident application that is to be loaded by *sysload* is built in exactly the same way as an HI command, so Figure 3.1 applies to the development of both HI commands and some resident commands.

A resident application that is to be configured into the operating system, as Figure 3.2 shows, includes an additional step in which the linkable modules produced by the binder use another development tool, the system builder, to incorporate the application into an image of the OS that is loaded into memory when the system is bootstrap loaded. Although the ICU (mentioned in chapter 2) is used specifically to generate a new copy of the iRMX operating system, the system builder is a general-purpose development tool that can build a bootstrap-loadable module for any operating system, or even a standalone application that runs without an OS.

3.1.4 Development tools

The development tools must be able to run on the development system, must all be compatible with each other (since the files output by one tool are used as input to the next tool), and must generate a program that can run on the target system. Let's use the development of a C language application as a concrete example. Depending on whether you want to develop a real-mode, 16-bit, or 32-bit application, you need to use the *iC86, iC286,* or *iC386* compiler, respectively. Versions of these three compilers are available that run on DOS, iRMX II, and iRMX III systems. There is also a C-86 compiler that runs on iRMX I. The files produced by these compilers use different OMFs, so the appropriate linker or binder must be chosen to be compatible with the chosen compiler. *Link86* can process the output of the C-86 compilers, *bnd286* can process the output of the C-286 compilers, and *bnd386* can process the output of both C-286 and C-386 compilers. There are versions of the linker and binders that run on different types of development systems, so the possibility for mismatching development tools may seem likely. In practice, you select your development and target systems, get the one set of tools you need, and go to work.

3.2 A Sample Application

To make the material in this chapter more concrete, the following sections will use the sample PLM program in Figures 3.3* (the main program) and

*The first comment line of all sample code in this book includes the name of the file (`hellormx.plm` in this example) containing the code. The files are available by anonymous *ft* to `ipcl.cs.qc.edv`. Readers can also obtain the files by mailing a diskette to the author: Dr. Christopher Vickery, Computer Science Dept., Queens College of CUNY, Flushing, NY, 11367-0909.

Figure 3.3 Source code for the sample PLM main program.

```
/***>  hellormx.plm  <************************************************
 *
 *      Sample PLM program for iRMX
 *      -- main program
 *
 ************************************************************************/
$title ('Sample PLM Main Program')
hellormx:   DO;
$include (hellormx.ext)
DECLARE
      prompt (*) BYTE PUBLIC INITIAL (0, 'Type something: '),
      reply (81) BYTE,
      Status     WORD_16;

dosub:  PROCEDURE (response$ptr, response$max) EXTERNAL;
DECLARE
        response$ptr    POINTER,
        response$max    WORD_16;
END dosub;

/*
 *  Execution Starts Here
 */

  prompt (0) = size (prompt) - 1;
  CALL dosub (@reply, size (reply) - 1);

  CALL rqc$send$co$response (NIL, 0, @(11, 'You typed: '), @Status);
  CALL rqc$send$co$response (NIL, 0, @reply, @Status);
  CALL rq$exit$io$job (0, NIL, @Status);
END hellormx;
```

3.4 (a subroutine) as an example of an application to be developed into an iRMX HI command. Examples of resident applications are shown in later chapters when you have a better understanding of iRMX. This program is written in Programming Language for Microcomputers (PLM) and will run equally well under any version of iRMX. PLM was developed by Intel specifically to generate code for its microprocessors, and is the traditional systems programming language for iRMX systems.

The C language, however, is quickly supplanting PLM as the language of choice for much of the work being conducted with iRMX, and will be the primary expository language in this book. PLM is used for the sample code in this chapter, however, because it illustrates the concepts of interest here a bit more clearly than the equivalent C program. If you have trouble following the PLM code, you can refer to the equivalent ANSI C version of the same program provided in chapter 4. The two languages are compared in some detail in that chapter.

The main program and subroutine for the sample program are in different files (`hellormx.plm` and `hellosub.plm`). Each file contains vari-

ables referenced by the code in the other file. This structure for the sample program was chosen to illustrate some concepts in the binding step better than if the entire program was in a single file. It is not intended to illustrate an optimal design for the program, nor even to illustrate a typical iRMX application.

The following discussion is quite detailed, so you might wish to refer to Figure 3.5 first to see the complete sequence of iRMX commands that could be used to build the sample program and run it. Lines [1] and [2] of the figure use the editor to create two source files, `hellormx.plm` and `hellosub.plm`. Lines [3] and [4] compile the two source files to create two object module files, `hellormx.obj` and `hellosub.obj`. Line [5] binds the object module files with the necessary library module producing an executable file named `hellormx`, and line [6] runs the program.

Figure 3.4 Source code for the sample PLM subroutine.

```
/***>  hellosub.plm  <*********************************************
 *
 *      Sample PLM program for iRMX
 *      -- subroutine
 *
 ************************************************************************/
$title ('Sample PLM Subroutine')
hellosub:   DO;
$include (hellosub.ext)
DECLARE
        prompt (*)  BYTE EXTERNAL,
        Status      WORD_16;

dosub: PROCEDURE (resp$ptr, resp$max) PUBLIC;
DECLARE
        resp$ptr    POINTER,
        resp$max    WORD_16;

  CALL rqc$send$co$response (resp$ptr, resp$max, @prompt, @Status);
  RETURN;

END dosub;
END hellosub;
```

Figure 3.5 A sequence of iRMX commands that could be used to build the sample program in this chapter.

```
iRMX> aedit hellormx.plm                                              [1]
iRMX> aedit hellosub.plm                                              [2]
iRMX> plm386 hellormx.plm compact debug                               [3]
iRMX> plm386 hellosub.plm compact debug                               [4]
iRMX> bnd386 hellormx.obj, hellosub.obj, &
** /rmx386/lib/rmxifc32.lib, :lang:plm386.lib rc(dm(0,0FFFFFFFFh)) &
** ss(stack(8192)) rn(code32 to code) oj(hellormx)                     [5]
iRMX> hellormx                                                        [6]
```

3.3 Text Editing

At the top of Figure 3.1, a source file is processed by an editor to produce a new source file. The original source file is optionally saved as a backup file in case the user wants to undo an entire editing session.

Any text editor can be used to prepare files for use with iRMX but there is just one text editor that runs on both DOS and iRMX development systems — *aedit*. *Aedit* is a good, full-screen editor designed for use with terminals connected to the system through a serial communication link. As such, it does not support colors or mouse menus; however, it is very easy to use, very powerful, and, because it is available for both iRMX and MS-DOS (using either the PC's console or a terminal connected to a serial port), it can provide a useful consistency in editing as you switch between the two operating systems. Even if you will not be using *aedit*, you should still read the following paragraphs, but then skip the Aedit Usage Summary section.

The end of each line in an iRMX text file ends with the ASCII <cr><lf> (carriage return, linefeed, 0x0D, 0x0A) sequence.[1] There is no end-of-file character in text files. *Aedit* handles these conventions automatically, but using files developed on other systems or exporting iRMX text files to other systems might require minor adjustments. In particular, Unix systems terminate lines with just the <lf> character, and Unix refers to that character as a "newline" (<nl>).

To understand the difference between these two methods of ending lines, let's review the roles that the ASCII control codes <cr> and <lf> played on mechanical terminals: the <cr> character initiated the movement of the printing head to the left margin, an operation that took considerable time to complete, and the <lf> character advanced the paper up one line. By sending <cr> and <lf> to mechanical terminals in that sequence, the paper-up movement could occur while the carriage was moving left, and the terminal would be ready to print the next character when it arrived. That is, a new line operation required two characters, and these characters had to be received in the proper sequence for some terminals to work properly.

CRT displays still use the same two ASCII codes to move the cursor left and down as independent operations, but the order is irrelevant because there are no mechanical timing constraints. Unix systems save file space by storing just the <lf> code at the end-of-text-file lines, but they must also generate the <cr> character whenever text is displayed to make it look right. Just to confuse matters a little bit, there will be places where you

[1] Items in <> represent single keys. The name of an ASCII code, a letter, the normal name on the keycap, or a letter with a modifier might appear inside the angle brackets. The modifiers used are alt- for the Alt key, and ^ for the control key. Modifier keys work like shift keys.

will see `aedit` refer to the <cr><lf> combination as <nl>, the same notation that Unix uses for the ASCII <lf> character.

The result of all this is that when you look at a Unix file on iRMX, the lines might walk across the page

something like this

because

iRMX systems expect the file to contain the <cr> characters that tell when to return to the left margin. Most development tools for both iRMX and Unix are indifferent to the presence or absence of <cr> characters in practice, but you can always convert an entire Unix file to an iRMX file in *aedit* with the command sequence, <j><s></><r><^R><0><a><esc><Enter><esc>. An English translation is "Jump to Start of file, Replace all 0x0A characters with <cr><lf>.[2]

The other text file compatibility issue is the MS-DOS convention of inserting a <^Z> character (0x1A) at the end of text files. Most software for both iRMX and DOS is indifferent to the presence or absence of this character as well. In *aedit* it looks like a <?> on the screen and can be deleted just like any other character in the file.

3.3.1 Aedit usage summary

If you will not be using *aedit* as your editor, you can skip this section.

This summary of *aedit* operations is not a tutorial, simply a guide to the features of the editor to speed learning it. The *Aedit User's Guide* that accompanies the system explains all the details. (The manual is Volume 12 of the iRMX for Windows documentation set.)

When *aedit* starts, it reads an initial set of commands from a file called `aedit.mac`. That file is in the same directory as *aedit* by default, but users can keep a personal copy in `:home:`. `Aedit.mac` can contain commands that you would type at the keyboard as well as macro definitions.

Intel supplies a set of macro definitions in a file called `useful.mac` that you can copy into your `aedit.mac` file. These macros really are useful if you use *aedit* very much, and they are documented in the *Aedit User's Guide*. The following discussion assumes that none of the *aedit* defaults have been changed by commands in your `aedit.mac` file.

Aedit is always in one of three modes: insert, exchange, or command. Insert and exchange are standard insert and overstrike modes for entering text. Use the arrow keys as usual. To go as far as possible horizontally or vertically, press an arrow key followed by the <Home> key. As far as possible horizontally is to the left or right end of the line; the maximum vertically is forward or backward one screen length. <Backspace> erases the

[2] If you export an iRMX file to Unix, *vi* will show <^M> at the end of each line (<cr> is the same as `control-M` in ASCII). The *vi* command `:1,$s/^VM//` removes all the <cr> characters in the file.

character to the left of the cursor, <Del> or <^F> erases the character under the cursor. <^Z> erases the current line, and <^A> erases from the cursor to the end of the line. <^R> allows you to insert characters by giving their numeric values in hexadecimal (used in the carriage return example above). Note that *aedit* and the Command Line Interpreter (CLI) both use the same keys for editing a single line, except for `aedit`'s <^A>, which is accomplished by <cr> in the CLI.

In command mode, a menu of commands is shown at the bottom of the screen, and <tab> is used to scroll through the menu. (One of the `useful.mac` macros lets you use the spacebar in place of <tab>.) Typing the first letter of a command either executes the command, brings up a submenu, or produces a prompt for more information, such as text to be searched for or the name of a file to retrieve. Commands are terminated with the <esc> key, but for terminals without an <esc> key, `aedit` can be configured to recognize some seldomly used character, such as <`> (backquote) for <esc>. Commands can be canceled by typing <^C>.

Of course, the first command to learn is the last one, the one to exit the editor. For `aedit`, that command is `quit`, achieved by typing <Q> in command mode. The <Q> command leads to a submenu with the following choices:

- <A>, Abort the editing session without changing any disk files.
- <E>, Exit the program, writing the newly edited file to disk. The original file being edited, if there was one, is also saved with the extension .BAK.
- <W>, Write the file to disk, but stay in `aedit`.
- <I>, Begin editing a new file without leaving `aedit`.
- <U> Same as Exit, but stay in `aedit`.

Aedit supports editing two files at once. The <O> (Other) command switches between the two. This feature is particularly well suited for editing a source file and a listing file for the same program during the compilation stage (discussed in the next section). The <W> (Window) command splits the screen into top and bottom portions so you can view and edit two parts of one file or two different files at the same time. After splitting the screen into two windows, the <W> command is used to switch the cursor between the two, and <K> (Kill) returns the screen to a single window. The *make* utility (discussed in section 3.7.2) uses these commands to invoke an *aedit* session whenever syntax errors are discovered during compilation. The top part of the screen shows the statement in error with the compiler's error or warning message, and the lower part of the screen shows the source code with the cursor placed on the line that caused the error. The user can step through a sequence of syntax errors by typing the *aedit* command sequence <E><esc>.

To copy a block of text while in command mode, move the cursor to one

end of the block of characters you want to copy, type a <B>, and the cursor will become an <@>. Move the <@> cursor to the other end of the block you want to copy, and type <B> again to copy the block into an internal *aedit* buffer. Now move the cursor to where you want the text copied, and type <G><Enter> or <^B> to insert a copy of the internal buffer into the file. To move a block instead of copying it, just use <D> instead of <B> in the above description. The <G> command used to insert the internal text buffer can also be used to insert the entire contents of a file into the one you are editing. Simply type the name of the file you want between the <G> and the <Enter>.

The feature of *aedit* that you most want to carefully study is the macro facility. With a bit of practice, macros become very easy to define and use. Just to give a quick example, try typing the following sequence when *aedit* is in command mode:

```
<M><C><PgUp><Enter><up arrow><Home><M>
```

Those seven keystrokes create a macro named <PgUp> that will scroll up a screenful at a time every time you press the <PgUp> key. If you want to save the macro in your `aedit.mac` file so it is automatically defined in each editing session, select `aedit.mac` for editing, create the macro definition, and type <M><S><PgUp><Enter>. The macro is saved as a sequence of codes that the editor interprets as keyboard characters when it reads `aedit.mac`. (*Aedit* for DOS recognizes the <PgUp> and <PgDn> commands directly, so this example works only in *aedit* for iRMX.)

3.4 Compiling

The same command line can be used to run a compiler regardless of the development environment, although an alias might be required for the command name on some systems, and certain logical names or environment variables might need to be established before a tool will run. Such details, however, are normally handled automatically by the installation process.

The iRMX> prompt is used for the examples, even though users of the DOS-hosted compilers will see C> at the beginning of the line (and will be using <\> instead of </>in path names). Assume that our sample PLM program is going to run as a 32-bit application on an iRMX III or iRMX for Windows target. That means that you must use the PLM-386 compiler, and the command line would look like line [3] presented in Figure 3.5:

```
iRMX> plm386 hellormx.plm compact debug                    [3]
```

The `compact` and `debug` parameters are *compiler controls*, and could alternatively be placed inside the source code file by using `$compact debug` as the first line of the file. The `compact` control is explained more in the

section on memory segmentation models later in this chapter, and the debug control is discussed in the section on debugging below.

3.4.1 Source modules and source files

The sample program is split into two source modules, contained in the two source files hellormx.plm and hellosub.plm. Inside each source file is a DO block with a label that is the same as the base part of the source file name. For example, hellormx.plm contains:

```
hellormx: DO;
    . . .
END hellormx;
```

A source module is defined as all the PLM statements inside the outermost labeled DO block in the source file. Each file that is to be processed by the PLM compiler must contain exactly one source module. The $title statements in the sample source files are directives to the compiler rather than source code statements, so they do not need to be inside the DO block. The compact and debug controls tell the compiler how to compile the source module, so they *must* appear before the labeled DO block if they appear inside the file. The label on the DO statement is the name of the source module. Generally, you use the same name for the source module as the base part of the source code file name so that you are compatible with the SoftScope debugger (discussed below).

The C language does not have any syntactic structure that delimits a source module the way PLM does, but there is a C compiler control called modulename (abbreviated mn) that can be used to set the name of a module. If the mn control is not used, the compiler uses the base part of the source code file name as the source module name.

3.4.2 Include files

The code in a source module can come from more than one file by using the include directive provided by both C and PLM. The sample source files both contain include controls, one for the file hellormx.ext and the other for the file hellosub.ext. Before the compiler actually compiles a file, it creates a temporary file that contains all the statements from the source file with the code from any include files inserted in the appropriate location. Strictly speaking, this temporary file is the source module that is compiled, not just the contents of the source file itself. Include files for C programs typically have an extension of .h, and are called *header files*. They are discussed more in chapter 4.

An important function of include files is to provide code that declares the names and argument types for subroutines that are not otherwise defined within the source module being compiled. An example of this type of decla-

ration appears for the subroutine *dosub()* in the code for `hellormx.plm`. That same source module contains references to two other external subroutines, the system calls *rqcsendcoresponse()* and *rqexitiojob()*. Note that the PLM compiler ignores the dollar sign character in symbolic names.

The iRMX operating system provides text files with the external procedure declarations for every iRMX system call for each of the various programming languages used for iRMX development. For PLM, the file is called `/rmx386/inc/rmxplm.ext`, and for C the corresponding header file is called `:include:rmxc.h`. Some versions of iRMX, notably iRMX for Windows versions 2.0 and later, come with an additional header file for C programs called `:include:rmx_c.h`. This header file provides aliases for system call names that insert underscore characters for improved legibility, analogous to the dollar sign in PLM names. For example, `rmx_c.h` changes *rqcsendcoresponse()* and *rqexitiojob()* to *rq_c_send_co_response()* and *rq_exit_io_job()*.

The appropriate include file should be included in every source file that contains iRMX system calls, but because these files contain the declarations for every system call supported by the OS, compiling all this included source code takes quite a bit of time. It can be worthwhile to build customized include files that contain just the declarations for those system calls actually referenced by a particular source module. The utility program *extgen*, available from iRUG, generates such customized include files automatically for PLM programs. It was used to generate the files `hellormx.ext` and `hellosub.ext` by the command

```
iRMX> extgen hellormx, hellosub                                    [7]
```

The code for `hellormx.ext` and `hellosub.ext` is listed in Figure 3.6 and 3.7, respectively. These files include some boilerplate code that declares new data type names as literal substitutions for some of the data

Figure 3.6 Included file, `hellormx.ext`, for the sample PLM main program.

```
$save nolist
/*  This file was generated by EXTGEN  */
DECLARE TOKEN    LITERALLY 'SELECTOR',
        BOOLEAN LITERALLY 'BYTE',
        TRUE     LITERALLY '0FFh',
        FALSE    LITERALLY '000h';

$if WORD16
DECLARE WORD_16 LITERALLY 'WORD';
$else
DECLARE WORD_16 LITERALLY 'HWORD';
$endif

RQ$ExitIoJob:
      PROCEDURE(
```

Figure 3.6 (*Continued*)

```
            user$fault$code,
            return$data$ptr,
            except$ptr) EXTERNAL;
      DECLARE
            user$fault$code WORD_16,
            return$data$ptr POINTER,
            except$ptr POINTER;
END RQ$ExitIoJob;

RQ$C$SendCoResponse:
      PROCEDURE(
            response$ptr,
            response$max,
            message$ptr,
            except$ptr) EXTERNAL;
      DECLARE
            response$ptr POINTER,
            response$max WORD_16,
            message$ptr POINTER,
            except$ptr POINTER;
END RQ$C$SendCoResponse;

$restore
```

Figure 3.7 Included file, `hellosub.ext`, for sample PLM subroutine.

```
$save nolist
/*  This file was generated by EXTGEN  */
DECLARE TOKEN   LITERALLY 'SELECTOR',
        BOOLEAN LITERALLY 'BYTE',
        TRUE    LITERALLY '0FFh',
        FALSE   LITERALLY '000h';

$if WORD16
DECLARE WORD_16 LITERALLY 'WORD';
$else
DECLARE WORD_16 LITERALLY 'HWORD';
$endif

RQ$C$SendCoResponse:
      PROCEDURE(
            response$ptr,
            response$max,
            message$ptr,
            except$ptr) EXTERNAL;
      DECLARE
            response$ptr POINTER,
            response$max WORD_16,
            message$ptr POINTER,
            except$ptr POINTER;
END RQ$C$SendCoResponse;

$restore
```

types recognized by the compiler. The data types `WORD_16` and `WORD_32` are conditionally defined depending on the setting of the `WORD16` or `WORD32` control for the PLM-386 compiler. The PLM-386 compiler uses different data type names for 16- and 32-bit values, depending on which compiler control is in effect. The `WORD16` control is used for PLM-286 compatibility. All PLM programs in this book use `$include` to include a declaration file that was generated by *extgen.*

3.4.3 Listing files

As indicated in Figure 3.1, the compiler produces a listing file and an object file. These two files have the same base name as the source file, with an extension of `.LST` for the listing file and `.OBJ` for the object file. For example, the two sample source files are `hellormx.lst`, and `hellormx.obj`. The object file is the one carried on to the binding stage of the development process, but the listing file is important now, before the object file is ready for use, because the listing file is where the compiler puts error messages indicating problems it had compiling the source file.

When you get errors from the compiler, you must re-edit the source file to correct them while consulting the listing file to see what the errors were. *Aedit*'s two editing buffers are very useful for this, because you can use the window command to view both the source and listing files at the same time. For convenience, you could use the following *`aedit`* command line to load `hellormx.lst` into *`aedit`*'s main editing buffer and `hellormx.plm` into the other editing buffer at the same time.

```
iRMX> aedit hellormx.lst-plm                                    [3]
```

Aedit running on DOS versions prior to 5.0 does not support this form of command line. If you try it, only `hello.lst` will be loaded for editing and you will have to use the command sequence `other, quit, init` to load `hello.plm` into the other buffer for editing. Keep in mind the following hints:

- All error messages and warnings from the compilers start with three or four asterisks followed by a blank. If you remember not to use that sequence of characters in your programs' comments or character strings, it is a convenient string to search for in the listing file as you look for error lines to fix. The angle brackets in the first line of each sample program in this book are there to prevent the line from containing the "***" string, for example.
- Be careful to edit the source file, not the listing file, as you make your corrections!

The listing file tells the lexical level for each source statement, which can be helpful when tracking down error messages from the compiler about il-

legal nesting or mismatched block identifiers. In addition, the listing file can be augmented with information useful for debugging binder and run-time errors. Look up the `symbols` and `code` compiler controls in your language's *Users Guide* for more information.

At the end of the listing file is a summary of how much memory is needed for each part of program: code, data, and stack. This information can be particularly valuable for real-mode and 16-bit applications that must keep track of segment sizes, as well as any other application with memory-size constraints.

Finally, the listing file is designed for maintaining hard copy program documentation. For example, the `title` directive (and for PLM, the `subtitle` directive) generates a new page in the listing with header information that can make it easier for readers to follow the code.

C programmers accustomed to the traditional Unix environment or to an integrated development environment will probably already have developed work habits that do not include using listing files because Unix compilers traditionally do not produce them, and development environments provide error and source windows automatically linked to each other. *Aedit's* double file editing using the compiler's listing file is the closest thing available for development on iRMX systems. With Windows, you can keep multiple windows open for the editor and compiler, but you must keep their contents synchronized manually.

3.4.4 Object files and object modules

The compiler translates the source module into an object module and then places it into an object file. The compiler names the object module using the name of the source module, and it names the object file using the source file's base name plus the extension `.OBJ`. For C, the base name of the source file, the base name of the object file, the name of the source module, and the name of the object module are always the same unless you use the `modulename` compiler control. The same is also true for PLM programs, provided you explicitly name the source module the same as the base part of the source file's name. Thus, the source file `hellormx.plm` contains a source module named `hellormx`, and compilation produces an object file named `hellormx.obj`, which contains the object module named `hellormx`.

If you compile using PLM386, the object module will adhere to the OMF-386 specification, which is compatible with the BND386 binder. For an iRMX I target, PLM86 would generate an OMF-86 format object module, which would be processed by the LINK86 linker. As you might expect, PLM286 generates OMF-286 object modules, and the corresponding binder is BND286 for iRMX II. In addition, however, BND386 can process OMF-286 object modules, and iRMX III can run OMF-286 STL modules.

3.5 Segmentation Models

Each program that runs on a processor with an x86 architecture consists of three types of memory segments, called `code`, `data`, and `stack`.[3] Code segments contain executable machine instructions, data segments contain data variables, and stack segments contain memory for parameters, return addresses, and local variables for subroutines. Data constants can be stored in either code or data segments. This segmented memory architecture is discussed more thoroughly in chapter 5.

The compiler generates machine code for exactly one code, one data, and one stack segment for each object module that it produces. When the compiler generates machine language instructions, it must use different types of memory pointers depending on how the segments from the object modules that compose the program will be combined by the binder. The different rules the binder can use to combine separate object modules are called either *memory segmentation models* or *models of compilation*, and all object modules that will be linked by the binder must use the same model. The compiler must be told which segmentation model the binder will use so the compiler can generate the correct types of pointers in the object module.

iRMX I and II programs use one of two different segmentation models, *compact* or *large*, while iRMX III and iRMX for Windows programs almost always use the compact model. The compilers, on the other hand, can generate modules using models called *small* or *medium* as well as compact or large. For 32-bit bootstrap-loaded applications, another model called *flat* is also available. You must explicitly tell the compiler which model to use because both the PLM386 and C386 compilers default to the small model, which will not work for iRMX systems. The compact model was specified when compiling the source files for the sample program.

The different models are described further in the next section on the binding stage of the development process, but a full understanding must wait until chapter 5, which describes the relevant features of x86 microprocessor architectures further. The whole subject is even more complicated because compilers support a facility called *extended segmentation*, which allows the careful programmer essentially to mix compilation models within a program.

The most commonly used segmentation model is compact, which uses less memory for addresses and runs faster than the large model for real mode and 16-bit applications. The large model must be used for real mode

[3] PLM-86 programs might have another segment called *memory* that serves somewhat the same function as *blank common* does for Fortran programs. All PLM programs that reference a built-in array named `memory` refer to locations within this segment.

and 16-bit applications that need more than 64 KB of code or data. For 32-bit applications, the PLM-386 and iC-386 compilers treat large and compact models the same because each code and data segment can contain up to 4 GB.[4] Thus, our example, which was compiled using the PLM-386 compiler, would have produced the same object module if it had been compiled with the large model.

When combining modules compiled using the compact model, the binder will produce a single code segment that contains all the code from all of the object modules it processes, plus one data segment containing all the data from all of the object modules it processes, plus one stack segment that is shared across all modules. For 16-bit applications, the large model still has a single stack segment, but the binder maintains separate code and data segments for each object module processed.

Readers familiar with the segmentation models used for Microsoft and Borland C compilers will find that some of the Microsoft and Borland names for various models are the same as Intel's, but that the actual definitions of the models are different. Intel compilers, however, do not provide models named tiny or huge. The following list of corresponding names for segmentation models is only relatively accurate, but you can get an idea of how the segmentation model names do *not* match up across vendors.

Intel	Borland	Microsoft
small	tiny	
compact	small	small
medium	compact	compact
medium	medium	
large	large	large
huge	huge	

3.6 Binding an HI Command

Once all the source modules for an application have been compiled without errors, the resulting object modules must be combined with other object modules to construct the program. Since our sample program is going to be a 32-bit application, we will use BND386 to do the binding.

3.6.1 Input files: object files and libraries

The format of a BND386 command line consists of an input file list followed by a number of parameters known as *binder controls*. For the sample program, the command line might be:

---

[4] Although the iC-386 and PLM-386 compilers seem to treat large and compact models the same, they set the combine-types of segments differently for the two modules. The binder then combines segments differently. It is also sometimes necessary to differentiate between near and far procedures and pointers. As described in chapter 5, the distinction involves whether selectors are involved in accessing a memory location or not. The C language provides the key words *near* and *far* for dealing with this issue. Both PLM-386 and iC-386 can also handle such situations using extended segmentation controls.

```
iRMX> bnd386 hellormx.obj, hellosub.obj, &
** /rmx386/lib/rmxifc32.lib, :lang:plm386.lib &
** rc(dm(0,0FFFFFFFFh)) ss(stack(8192)) &
** rn (code32 to code) oj(helloplm)                                    [4]
```

Note that this is the same [4] in Figure 3.5. Remember, <&> tells the iRMX CLI that a command is continued on the next line and "**" is the normal iRMX prompt for continuation lines. Long command lines such as this can also be put into a file referenced with the `cf` (command file) binder control, like this:

```
iRMX< bnd386 cf(bnd386.cf)                                              [5]
```

The command file `bnd386.cf` would contain:

```
hellormx.obj, hellosub.obj, &
/rmx386/lib/rmxifc32.lib, :lang:plm386.lib &
rc(dm(0,0FFFFFFFFh)) ss(stack(8192)) &
rn (code32 to code) oj(helloplm)                                        [4]
```

This technique can be particularly useful for DOS-hosted development tools because DOS does not support continued command lines. Note that everything starting with `rc(dm(...` is a list of binder controls, which are described in the next section. For now, the focus is on the input file list.

The input file list consists of all the file names that the binder will combine to build the executable program. For the sample program, the list consists of the two object files plus two library files called `/rmx386/lib/rmxifc32.lib` and `:lang:plm386.lib`. The name of the first library file tells a bit about what it contains. `''rmxif''` indicates that this library contains iRMX interface procedures. That is, this file contains the subroutines to allow programs to access the system call subroutines that are part of iRMX. Chapter 6 introduces the iRMX system call mechanism, and chapter 10 explains the mechanism in some detail. The next part of the file name, "`c32`," indicates that this library can be used with 32-bit applications that are compiled using the compact segmentation model. As the sample source modules were both compiled using the `compact` control, and all modules processed by the binder must be compiled using the same model, the compact library must be used. For 16-bit applications, there are two other interface libraries, `/rmx386/lib/rmxifc.lib` for compact model programs and `/rmx386/lib/rmxifl.lib` for large model programs, that would be used instead. The other input file is the PLM runtime library, which is described further in chapter 4. It is not actually needed for most iRMX application (including this one).

Libraries are simply files that contain more than one object module. A special code in the first byte of the file, not the name of the file, tells the binder whether the file contains one object module or a library of object modules. The OMF-386 (or -286 or -86) specification describes the internal structure that both libraries and object modules must use. You can create a

library and add your own object modules to it (or remove or replace them) using the *lib386*, *lib286*, or *lib86* librarian that comes with the system. The librarians can work interactively and have a good built-in *help* command.

Selecting object modules to combine. The binder uses the object modules from every object file named in the input path list as it constructs a program, but it is selective about modules it uses from libraries. The reason for being selective is to save memory and binding time. There is no need to include every object module from a big library just to get the few that are needed for a particular application.

To understand how object modules are selected from libraries, first consider the process the binder uses to deal with the variables and procedures generally declared public and external in object modules.[5] In the sample program, the procedure `dosub` is declared `external` in `hellormx.plm` and is declared `public` in `hellosub.plm`. Figure 3.8 represents the `hellormx` object module.

When the compiler encounters the `CALL dosub` statement, it generates part of a machine language `call` instruction, but it cannot fill in the address of `dosub` needed to complete the instruction. Instead, the compiler leaves space for the address in the instruction's code, indicated by a box with a question mark in the figure. The compiler includes the ASCII name `dosub` in the Fixup List that is part of the object module, along with a link from the ASCII name to the incomplete address in the code segment.[6]

There will be one entry in the Fixup List for each external symbol referenced by the module. When the binder processes the `hellormx` object module, it copies the information from the Fixup List into an internal symbol table that it builds, and marks the symbol `dosub` as unresolved. The symbol `prompt` has been declared public, so it appears in the Public Symbols List that is also part of the object module. The names of such public symbols are also entered into the binder's internal symbol table, along with the address of where in the module's data segment the symbol is defined. This address is said to be the value of the symbol, and it is this value that will be used to fix up incomplete instructions that reference the symbol. The binder will also find the external names `rqcsendcoresponse` and `rqexitiojob` in the `hellormx` object module and enter

[5] C programs are not as explicit about declaring things to be public or external as PLM programs. Chapter 4 provides more information on this topic.

[6] The terms *address*, *pointer*, and *link* are imprecise at this point. Chapter 5 will deal with the nature of pointers and memory addresses in more detail. There are two parts to a pointer (selector or base and offset). An address may consist of either a selector (or base) and an offset, or just an offset. Fixups sometimes have to be applied to the selector or base and sometimes to the offset. We use the term *link* to refer to pointers that the binder needs for its own housekeeping, to distinguish them from the pointers or addresses that will become part of the code that is output by the binder.

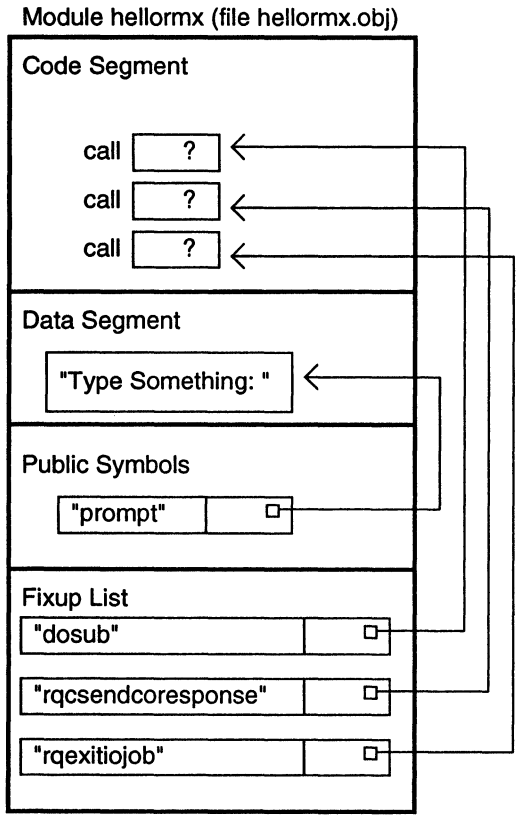


Figure 3.8 Structure of object module for sample PLM main program.

them into its symbol table as well. Thus, after processing the first object module, the binder will have constructed a symbol table that looks something like this:

Symbol Name	Symbol Value
`dosub`	unresolved
`rqcsendcoresponse`	unresolved
`rqexitiojob`	unresolved
`prompt`	address in data segment

When the binder processes the next file in its input path list, `hellosub.obj`, it combines the code segments from the two object modules into a single module, and does the same for the data segments as well (see below). The binder finds that the compiler has left the ASCII name `dosub` in the `hellosub` module's Definition List (the list of public symbols de-

fined within the module), along with a pointer to the place in that module's code segment where the dosub subroutine begins. The binder now can fix the call instruction that referenced dosub so that dosub points to the subroutine, and the binder resolves the value of dosub in its symbol table to be the address of the subroutine in case the binder needs the value again during the binding sequence. When the compiler built the hellosub object module, it was unable to generate the complete data address of prompt in the call to rqcsendcoresponse because of the external declaration. When the binder processes the hellosub object module, it resolves this symbol (prompt), and can fix the instruction that referenced prompt (an instruction to push the address onto the stack) immediately. The binder's symbol table now looks something like this:

Symbol Name	Symbol Value
dosub	address in code segment
rqcsendcoresponse	unresolved
rqexitiojob	unresolved
prompt	address in data segment

At this point, the loadable module that is being built looks like Figure 3.9. The question marks represent links to the unresolved external symbols.[7]

With all this background about the binding process, it is quite simple to tell which modules the binder will include from a library: those modules with public declarations for symbols marked unresolved in the binder's symbol table. To make this process efficient, each library contains a master dictionary of all the public symbols that are defined in it along with links to the object modules that contain them.

The binder's symbol table is very dynamic. Including one object module to satisfy an unresolved symbol can result in new unresolved symbols that are referenced by the newly included module. The binder resolves these second-level references from the same library, using the master dictionary, if possible, and then moves on to the next file in its input list for processing.

It is possible to construct libraries that make circular references to each other, such as library1 containing a reference to a module in library2, which contains a reference to another module in library1. The binder does not automatically go back to a file once it has processed it, so for a circular reference, you must list the same library file (in this case, library1)

[7]The subroutines in the rmxifc32.lib library do not actually perform the functions of rqc$send$co$response, which is to issue a prompt and read a reply, and rq$exit$io$job, which is to terminate the program. Rather, they act as interface procedures to the actual subroutines that are part of the OS itself. (The letters *if* in the file names of the libraries stand for *interface*.) These interface procedures are covered in detail in chapter 10, when we cover the techniques for adding system calls to the OS. The system calls themselves are covered in chapter 7.

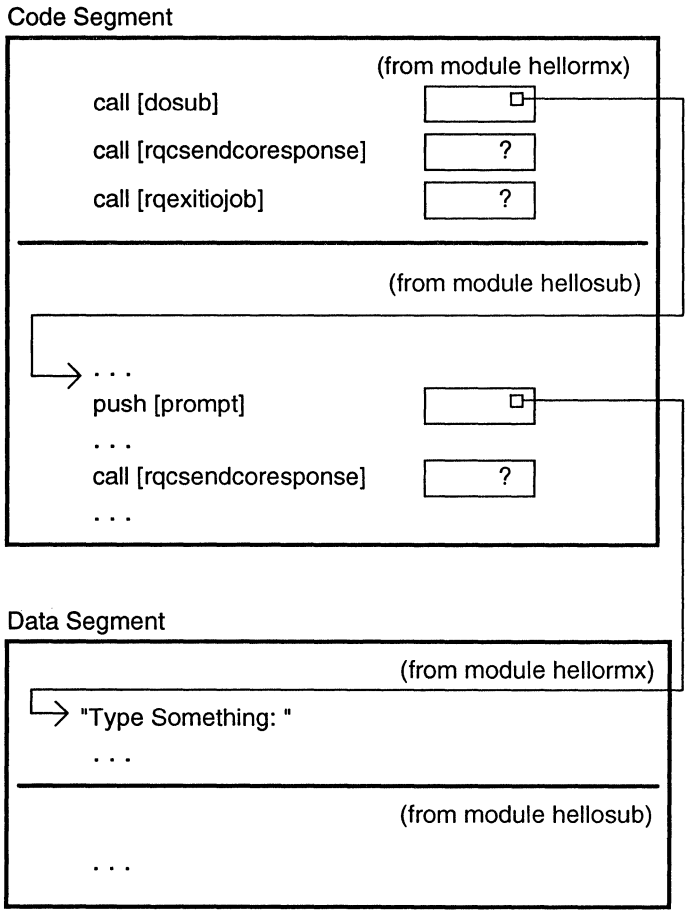


Figure 3.9 STL module (executable file) for sample PLM program. Note that the binder has linked modules `hellormx` and `hellosub`, but has not yet linked any of the libraries.

multiple times on the binder's input list. Alternatively, you could force the binder to include modules from `library1` even though they have not been referenced yet. You can accomplish this using the syntax `library1(module1, module2)`, which would force the binder to include modules named `module1` and `module2` from the file `library1`.

3.6.2 Output files: the map and load files

As Figure 3.1 indicates, the binder produces two files, a map file and a loadable file. The map file, which normally has the same base name as the first object file on the binder's command line and an extension of `.mpl`, contains any error messages generated by the binder, the sizes of the various segments generated, a list of the object modules included, and a list of any

external symbols left unresolved at the end of the binder's execution. For example, `hellormx.mpl` includes these lines:

```
SEGMENT MAP
                                                    COMBINE     COMBINE
LIMIT          ACCESS      ALIGN      USE          TYPE        NAME
0000018DH      ER          DWORD      USE32        NORMAL      CODE
0000007AH      RW          DWORD      USE32        NORMAL      DATA
FFFFDFFFH      RW          DWORD      USE32        STACK       STACK
INPUT MODULES INCLUDED:

HELLORMX (HELLORMX.OBJ)
HELLOSUB (HELLOSUB.OBJ)
HXSNCR (:SD:RMX386/LIB/RMXIFC32.LIB)
EXJEXJ (:SD:RMX386/LIB/RMXIFC32.LIB)
ERT14N (:SD:RMX386/LIB/RMXIFC32.LIB)
HRT1CN (:SD:RMX386/LIB/RMXIFC32.LIB)
RQCERR (:SD:RMX386/LIB/RMXIFC32.LIB)
RQCSEX (:SD:RMX386/LIB/RMXIFC32.LIB)
NRT1CE (:SD:RMX386/LIB/RMXIFC32.LIB)
NUCLER (:SD:RMX386/LIB/RMXIFC32.LIB)
```

The first line of the segment map is for a segment that is 0x018D bytes long, has executable and readable memory protection attributes (`ER`), starts on a doubleword memory boundary (`DWORD`), was generated by a 32-bit compiler (`USE32`), and is the result of combining all segments named code in the typical method (i.e., according to the rules for the compact segmentation model).[8] The size of the stack segment looks strange with a decimal value of -8193 (0xFFFFDFFF); this value is negative because a stack segment grows downward in memory as data is pushed onto the stack.

The second part of this `.MPl` file shows the names of the object modules and the files from which they come. The first two modules are from the object files the compiler generated from the source files. The next two modules contain the subroutines `rqcsendcoresponse` and `rqexitiojob`, respectively. All other modules listed were included because they are modules that satisfied external declarations made by some module earlier in the list. In particular, they were included to support the OS's exception handling mechanism for system calls, mentioned in chapter 2.

The most common message in a `.MPl` file is "unresolved external symbol." This message is considered a warning, rather than an error, by the binder because of the possibility that the output file will later be bound with other object modules that will provide public declarations for the unresolved symbols (discussed further below). It should be considered an error when you are producing a file intended to be executed, as in the

[8]For segment size, hexadecimal constants end with the letter <h> in PLM, and this convention carries over into numerical parameters for several iRMX commands, including the binder. The C language format is used for hexadecimal consants in the text of this book.

present example. Trying to run an HI command that generated this warning when bound will almost certainly result in a hardware fault.

The second file generated by the binder is its output file, which can be either in the STL format suitable for loading into memory for execution as an HI command, or a linkable module, which can be used later as input to another run of the binder or as input to the system builder (BLD386), as described below.

The first part of an STL file is a header that tells the minimum amount of memory the program will need to be loaded, the maximum amount of memory the program will need during execution, the types and sizes of all the memory segments that compose the program (see chapter 5 for information on memory segmentation), and information needed to initialize the CPU's registers when the program is loaded for execution, such as the address of the first instruction to be executed. After the header comes information explaining which information goes into which memory segments.

The information in an STL file is just what iRMX needs to load a program into memory and start it executing, but the same file could be run by any OS that recognizes OMF-386 structures. Although iRMX is a multitasking operating system, all iRMX programs start execution with a single task, or thread of execution, but might create additional tasks during execution as needed. However, iRMX tasks are not hardware tasks in the sense intended by the term *single task loadable*, and the STL format is perfectly suited for loading iRMX multitasking applications. Chapter 5 discusses hardware tasks further, and chapter 6 discusses the nature of iRMX tasks.

Instead of an STL module, the binder can produce a file that contains a linkable module. By default, such files are given file names ending in `.lnk`, although it is the content of the file rather than its name that determines its nature. For all practical purposes, a linkable file looks like the output of a compiler, especially if the linkable file uses the compact segmentation model. In this case, the linkable module contains one combined code segment and one combined data segment, just as if all the object modules that were linked together had originally come from one big source module. The situation is a bit different for the large segmentation model, however, because in this case the linkable module will contain multiple code and data segments, one from each of the linked object modules. Compilers never generate multiple code or data segments.

There are three reasons for generating a linkable module. The first reason is that this format is used for input to the system builder, *bld386* (or *bld286* or *loc86*). The operating system is constructed from relatively independent layers that do not share code or data with one another through public or external variables. Each layer can be independently bound into a linkable module without concern for any public or external symbol names that might clash with such symbols used in another layer. The resulting

link files are then processed by the system builder to construct a bootstrap loadable file on disk, the process summarized in Figure 3.2.

A second reason for generating a linkable file is to control public and external symbols when building a single HI command. For example, if it is necessary to bind a program with two different libraries that both contain public symbols for subroutines with the same names, the application can be bound first with the library containing the desired version of the subroutine, and that public symbol can be purged from the resulting linkable file using the `publics except` binder control. (See the section Producing Linkable and Bootloadable Modules below for more information on this process.) The linkable file can then be bound to the second library without any "duplicate public symbol" messages from the binder.

A third reason for working with linkable files is for applications that mix 16-bit and 32-bit code. The binder combines all segments with the same name (such as `code32`), but it is an error to combine segments with different attributes (16-bit and 32-bit, for example). Separate linkable files could be built, one containing only 16-bit segments and the other containing only 32-bit segments. The combined segments can then be given different names using the `rn` control (described below), and the linkable modules can be bound together without error.

3.6.3 Binder controls

This section provides a summary of the four binder controls included on the sample *bnd386* command line. For further details, consult the binder section in the *Intel386 Family Utilities User's Guide*, volume 17 of the iRMX for Windows documentation set.

`rc(dm(0,0FFFFFFFFh))`. This control identifies how much memory the program will need when it is loaded and executed. `rc` originally stood for RMX Configure, but the documentation for BND386 generalizes this to "an 80386 operating system." `dm` stands for dynamic memory. The first of the two hexadecimal values represents the minimum amount of memory that must be free to load the program, and the second value limits how much memory the program can allocate from the OS during execution.[9] The values specified here (0 and 4 GB) are synonyms for no limits, but could be adjusted either to ensure there is enough memory for the program to complete once it is loaded, or to limit the amount of memory the program will use as it runs. If you omit this control, *bnd386* will still generate an STL module, but it will set both the minimum and maximum values to zero, and the system will substitute reasonable values when the program is

[9]These two values correspond to the minipool and maxpool arguments to the CLI's *background* command mentioned in section 2.4.2. The *background* command arguments can be used to override the `rc(dm())` arguments to the binder.

run. If you omit this control for *bnd286* you will generate a linkable module rather than an STL module.

`ss(stack(8192))`. The `ss` control manipulates the size of segments in the load module. In this case, the size of the stack segment is set to 8 KB. Compilers place information in object modules identifying how much stack space is needed for the code in each module for calling subroutines and allocating local variables, but the compilers cannot know how much stack space will be needed for nested subroutine calls, such as recursive subroutines or OS system calls. The figure 8K is very generous, and allows enough stack space for use by both system calls and the SoftScope debugger.

Note that SoftScope uses the same stack as the application program that it is debugging in iRMX I and II. SoftScope for iRMX III and iRMX for Windows runs with its own separate stack, so the application's stack may not need to be so large.

`rn (code32 to code)`. This control renames segments with the name `code32` to `code`. The binder will only combine segments that have the same name, and will issue error messages if the same symbol appears in different segments that it cannot combine. Since the compiler names the code segment `code32` but the library functions have code segments called `code`, this rename operation must be performed for a successful bind.

`oj(helloplm)`. This control specifies the name for the file that receives the load module produced by the binder. The full name for this control is `object`, which does not relate well to our terminology for types of modules. By default, the binder places the load module in a file that has the same name as the base part of the first file name in the input list, and no extension. That is, `oj(helloplm)` was superfluous in our example. It will be crucial for some C programs, however, so it was used here to establish a good precedent.

3.7 Automating the Process

Typing and retyping long command lines as you iterate through the steps of the development process can be tedious, and there are ways to improve the situation. The CLI's alias command and command history buffer facility, discussed in chapter 2, can do much to improve the situation, but for real production work, stronger solutions are needed.

3.7.1 Command files

Sequences of commands can be typed into command files for invocation by the *submit* command, and the parameter substitution feature of both

alias and *submit* can help in the construction of general purpose tools. For example, here is a command file that will compile, bind, and execute a single-source-module PLM program:

```
plm386 %0.plm compact debug
bnd386 %0.obj,/rmx386/lib/rmxifc32.lib &
  rc(dm(0,0FFFFFFFFh)) ss(stack(8192)) &
  rn(code32 to code) oj(%0)
%0
```

Expanding a bit on the "doit" example from chapter 2, assume the preceding lines were entered into a text file named `cbe.csd`, and the following alias is defined:

```
iRMX> alias g=submit cbe(#)                                    [6]
```

Now, a PLM program, say `myprog.plm`, can be compiled, bound, and executed with the single command line:

```
iRMX> g myprog                                                 [7]
```

Two problems exist for automating the development process with this approach. One, it does not account for the possibility that the compilation might fail due to syntax errors, thus making the binding step inappropriate (or binding might fail, thus making execution inappropriate). Two, no provision exists for different numbers of source modules contributing to different load modules, making the file a somewhat inflexible tool.

The first problem can be handled by the HI command *esubmit*, which is an extended version of *submit* that supports testing the results of one step before continuing to the next step in the command file. Below is an *esubmit* command file that performs the same function as the *submit* file above, but skips the binding step if compilation fails and skips execution if binding fails. It also includes a crude mechanism for handling either one or two source files.

```
$reset eok
$reset quit

run86 -fixplm :lang:plm386 %0.plm compact debug
$if not commandexcep = eok
$set quit
$endif
$ifexist %1.plm
run86 -fixplm :lang:plm386 %1.plm compact debug
$if not commandexcep = eok
$set quit
$endif
$endif

$if not quit
```

```
run86 :lang:bnd386 %0.obj, &
$ifexist %1.obj
  %1.obj, &
1$endif
  :sd:rmx386/lib/rmxifc32.lib &
  rc(dm(0,0FFFFFFFFh)) ss(stack(8192)) &
  rn(code32 to code)
  $if commandexcep = eok
%0
$endif
$endif
```

One of the reasons this file looks so complex is, as an HI command, *esubmit* does not have access to the CLI's aliases, so the invocation of the compiler and binder must be explicit. Until now, the sample command lines have assumed an alias exists, which conceals the fact that the *plm386* command actually involves invoking the compiler using the *run86* utility.

3.7.2 The *make* command

The *esubmit* command is a relatively recent addition to iRMX, and future versions might make this a stronger tool. Meanwhile, a version of the Unix *make* command for iRMX is available that works very well for automating the development process.[10] The basic idea of *make* is that you specify the name of a file to be created on the command line and *make* invokes exactly those HI commands needed to create the file. The file to be created is called the target, and *make* is supplied with a set of rules that identify which commands to run to build targets based on their file names.

For example, if you tell *make* to create a file called `myfile.obj`, it would look for a source file to compile. If, on the other hand, you told *make* to create a file called simply `myfile`, it would invoke the binder. The two powerful features of *make* are that it performs only those operations actually necessary, based on the time and date of the last modification the OS stores with each file, and its macro capability, which gives it a great deal of flexibility.

The rules that *make* follows are stored in two text files, called `:lang:builtins.mk` and `:$:makefile`. Both files contain the same information, but if there is any conflict between them, `makefile` takes precedence over `builtins.mk`. For a single-source-module program written in either C or PLM and a properly set-up `builtins.mk`, you should be able to just type a `make` command with the name of the program as an argument to compile and bind the program, even without a `makefile` present. If the compilation stage fails, edit the source file to fix the problem, and issue the same *make* command again.

If more than one source file constitutes a program, *make* proves to be

[10]The command is called *mk* in some versions of iRMX.

very powerful, but it does have to be told by statements in `makefile` which modules compose the program. The technique is to enter a line in the `makefile`, which takes the general form

```
<target> : <dependencies>
```

The `<target>` is the name of the file to be built. After the colon is a list of files (separated by spaces, not commas) that must be older than the target. For example, a possible `makefile` for our sample program is the following:

```
hello : hellormx.obj hellosub.obj
<tab>$(BND) hellormx.obj,hellosub.obj, &
<tab>:SD:rmx386/lib/rmxifc32.lib &
<tab>rn(code32 to code) &
<tab>ss(stack(4096)) rc(dm(0,0FFFFFFFFh)) oj(hello)
```

The first line of the file says that the target `hello` depends on the two object files listed after the colon. If the `.obj` files are not older than `hello`, *make* treats them as targets themselves and consults the rules in `builtins.mk` to learn how to compile the corresponding source files if necessary. Then, if either object file is newer than the target file `hello`, all the lines that start with a `<tab>` character will be executed to create a new version of the target. In this case, there is just one command, an invocation of *bnd386*. Thus, with this `makefile`, the command line

```
iRMX> make hello                                                    [8]
```

would cause the following sequence of events to occur, depending on the relative ages of the files involved:

- If `hellormx.plm` is younger than `hellormx.obj`, or if `hellormx.obj` does not exist, compile `hellormx.plm`.
- If `hellosub.plm` is younger than `hellosub.obj`, or if `hellosub.obj` does not exist, compile `hellosub.plm`.
- If `hellormx.obj` or `hellosub.obj` is younger than `hello`, or if `hello` does not exist, execute the commands following the dependency line that start with `<tab>`.

It is possible that *make* will find nothing to do (the object files are older than the target file, and the source files are older than the object files), and will issue a message saying that `hello` is "up to date," and exit without doing anything. *Make* will try to execute the first target it encounters in `makefile` if no argument is given on the command line, so line [8] could simply be entered as

```
iRMX> make                                                          [9]
```

An example of a make *macro* in this makefile is `$(BND)`, which is a reference to a macro named BND. That macro was defined in `builtins.mk` to be the command to invoke the binder. This use of *make* macros is similar to the use of the CLI's alias feature, but it is not limited to redefining command names at the beginning of a line. For example, consider using a macro named MODEL to determine the segmentation model to be used when compiling. You can give a value to a macro four different ways, which *make* uses in the following sequence:

1. Put the macro definition in `builtins.mk`.
2. Put the definition in `makefile`.
3. Define the macro name as an environment variable.
4. Define the macro on the *make* command line.

As an admittedly far-fetched example that demonstrates all four cases, consider the following hypothetical file contents:

```
:lang:builtins.mk
  MODEL = small

:$:makefile
  MODEL = medium

:prog:r?env
  MODEL = large
```

The last file demonstrates the technique used by iRMX to set up environment variables. With these three files in place, the following command line could be used to invoke *make* with the actual value of MODEL ultimately being equal to `compact`:

```
iRMX> make MODEL = compact hello                                    [10]
```

That is, the command line definition of MODEL is "compact," which overrides the setting of the environment variable MODEL to "large," which in turn overrides the makefile definition of the macro to be "medium," which overrides the `builtins.mk` definition of the macro to be "small." Assuming that the rule telling *make* to invoke the compiler contained a reference to `$(MODEL)` on its command line, the compiler would be invoked with the `compact` control in this example.

Full details on how to use *make* are given in the help file provided with the program. Compatible versions of *make* are also available for both DOS and Unix. For example, the sample code for this chapter has been successfully built using the DOS-hosted development tools for iRMX and the version of *make* provided with the Borland C++ development system.

3.8 Debugging HI Commands

Once the binder has produced a file containing an STL module, it can be executed by simply typing the name of the file on the command line. The HI will locate the file either by examining the directories in its search path list or by following an explicit path name, and will pass the file to the Application Loader (AL), resident in iRMX, to load into memory for execution. It is the AL that detects the errors such as "bad header" or "not enough memory while loading command" that occasionally show up at this stage.

In addition to loading the command through the AL, two other commands can be used to load HI commands specifically for aiding program debugging. These commands are the system command *debug* and the utility debugging program, *ss* (SoftScope). The following subsections introduce all three of these techniques used for debugging HI commands.

3.8.1 Using only the application loader

The simplest way to debug a program is just to run it. If the program causes an exception, the default exception handler will manage the situation, hopefully by providing you with enough information to fix the problem. The default exception handler is the one supplied by the HI, which issues an error message and terminates the program that encounters an exception.

Whether an incorrect program causes an exception or simply produces the wrong results, the only tool available for debugging when running solely under the application loader (AL) is to add output statements to the source code to display values of key variables or execution flow information. This technique is a very primitive and appropriate only for small programs.

The highest form of this lowly technique is to provide either compile-time or run-time switches for turning debugging output on or off. A run-time debugging switch can be valuable for helping users unfamiliar with a program figure out what they are doing wrong when they run it. Otherwise, the technique has two significant problems associated with it: (1) Each debugging run involves the time-consuming process of modifying source files, recompiling, and rebinding, and (2) The debugging statements must later be removed from the program after it is debugged.

Any change to source code raises the possibility of introducing new errors, and nothing is more frustrating than having to fix errors introduced by taking out debugging statements, except perhaps having to fix errors introduced by putting in comments.[11]

[11] My advice: Write the comments first. If you know what you are doing, they will always be right and won't need to be changed. Ignore those who suggest that the solution is to leave out the comments!

3.8.2 Using the debug monitor

The system debug monitor enables you to examine the state of your program as it is executing without modifying the code. Several versions of this monitor are available, and they all have different names, depending on the platform being used. The version you most likely will encounter is called SDM. It either resides in ROM or is loaded with the operating system. iRMX for Windows loads the SDM into RAM by means of a *sysload* command when the system initializes.

There are three ways to enter the debug monitor. On some systems, there is a front-panel interrupt button (*not* the reset button) that causes an immediate break to the monitor. iRMX for Windows achieves the same effect on a PC by using the <^alt^Break> key combination (Press and hold <Cntrl>, then press and hold <Alt>, then press <break>. This technique can be useful for interrupting programs that enter an endless loop, rather than simply typing <^C> to abort the program completely.

The second technique of breaking to the monitor involves modifying the source code. For this technique, you insert a statement that will be compiled into a machine language *int 3* instruction. This instruction is the one that debugging programs use to set execution breakpoints in code. The debug monitor is normally configured as the default debugging program that receives control when an *int 3* instruction is executed, so you would place the *int 3* instruction wherever you want the program to break to the monitor. Both PLM and C allow you to insert an *int 3* instruction with the same statement `causeinterrupt (3);`

This technique, of course, inherits all the disadvantages of putting output statements in the source code, but it can be valuable when the code to be debugged is not accessible to SoftScope, such as resident code in iRMX I and II systems.

The third technique associated with the debug monitor is the *debug* system command, which loads the command file, displays information about where memory segments have been loaded, and then breaks to the debug monitor. This technique incurs the overhead of loading *debug* into memory with the application, but it requires no modification of the program to accommodate debugging and requires less memory than SoftScope.

The command prompt from the debug monitor is one dot (.) if the processor is in real mode (iRMX I systems), or two dots (..) if the processor is in protected mode (iRMX II and III systems). You can enter commands to display the processor's registers and memory, set breakpoints, and step through your code one machine instruction at a time. This tool can therefore be very useful for debugging code written in assembly language. Our focus, however, is on developing applications using high-level languages, and those readers interested in using a debug monitor can work from the appropriate manual.

3.8.3 Using the system debugger

The system debugger (SDB) is an extension to the debug monitor loaded into memory along with the OS. It can provide extremely valuable information for iRMX developers, including those working with high-level languages. You can access the SDB the same way as the debug monitor, and you can enter SDB commands at the same prompt as the monitor (. or ..). Furthermore, you can issue SDB commands from the SoftScope debugger while doing source code debugging of programs written in a high-level language.

The full value of the SDB will not be clear until chapter 6, which explains the design of iRMX as an object-based operating system. For now, note that the SDB allows you to view iRMX objects, both those created by your application and those created by the OS for its own use. All SDB commands start with the letter *v* (for view) and are fully documented in the *iRMX System Debugger Reference Manual.* They are also documented in chapter 9 of the *SoftScope III Debugger User's Guide*, volume 13 of the iRMX for Windows documentation set.

3.8.4 Using SoftScope

By far the most effective tool for debugging iRMX applications is the SoftScope source code debugger. The basic debugging features are all keyed to the source code: source statements can be displayed as they are executed, execution breakpoint addresses are specified in terms of source code functions or line numbers, and variables are displayed by name, with full recognition of data types, structures, and source language scope rules. In addition, SoftScope provides debug monitor-like access to the processor's memory and registers, as well as access to the SDB commands for viewing iRMX objects.

What sets SoftScope apart from other source code debuggers is its support for multitask application debugging and multiuser debugging. These two features are available only with SoftScope III. The versions of SoftScope for iRMX I and II are somewhat more limited in this regard.

Debuggers insert an execution breakpoint into code being executed by substituting a machine language interrupt instruction (the *int 3* instruction mentioned earlier) for one of the instructions originally in the code.[12] When execution reaches the interrupt instruction, the CPU branches to the code designated as the interrupt handler for this class of interrupt. When SoftScope starts running, it takes over this role from the resident debug monitor. The breakpoint handler determines where the interrupt occurred in case there were multiple breakpoints set, then issues a message

[12] SoftScope III uses the 80386 microprocessor's hardware trap feature to accomplish the same objective without modifying the code being debugged. This technique allows breakpoints to be set in ROM and allows for efficient breaking on data accesses as well.

and waits for the user to enter commands. What makes SoftScope III an excellent debugger is that it properly handles the situation in which more than one task executes the same breakpoint interrupt. It not only tells the user if an interrupt has been reached by more than one task (SoftScope I and II do that as well), but SoftScope III also tells the user exactly which tasks have reached the breakpoint, and then gives the user the ability to control further execution by each task independently.

When SoftScope starts running, it connects to certain software hooks in the OS, a process that can occur only once. In a multiuser environment, only one person can run SoftScope at a time for versions I and II. For SoftScope III (used with iRMX III and iRMX for Windows systems), a program called *sskernel* connects to the OS hooks. *Sskernel* is run just once, as a background job or by the *sysload* mechanism. After that, multiple users can invoke SoftScope independently at the same time to debug different applications.

The DHDT version of SoftScope III can debug iRMX applications running on a separate target system connected to the PC by a serial link. In this configuration, the monitor, called *iM*, must reside in ROM on the remote target system. The DHDT version of SoftScope III features a nicer screen display and better use of the PC's keyboard and mouse than the iRMX-hosted version. Currently, a "WHDT" (Windows-Hosted Development Tool) version of SoftScope III is being developed for iRMX for Windows. This version will provide users with full debugging access to iRMX applications from Windows, using the standard Windows user interface of mouse, pulldown menus, and the like.

To use SoftScope, it is first necessary to tell the compiler to insert debugging information into the object module. This information, which is specified by the `debug` compiler control (see the section on compiling above), consists essentially of the entire symbol table that the compiler builds while compiling the program. That is, this information includes the name, data type, and address of every variable (including field names and data types of structures) as well as the address of every function and source statement line number. Unlike some debugging systems, the debugging information is totally separate from the code and data segments that the compiler generates. In terms of the OMF specification, it goes in a debug segment (called *debug records* in OMF-86).

The significance of this fact is that the impact of including debugging information is minimized. During binding, you can choose to retain the debug segments in the load module or not (the default is to retain them), and, if they are retained, they remain associated by name with the object modules from which they were derived. When the load module is loaded by the AL or by *debug*, any debug segments are simply discarded, and the program loaded into memory is exactly the same as the one that would have been loaded if the debug segments had never been created. The penalties for including the debug segments in these cases are the extra disk space re-

quired to hold the additional information in the object modules and the load module, and the extra time to extract the program's code and data segments from the file and load them into memory. When a program is run under control of SoftScope, the debugger first loads the program treating the code, data, and stack segments in exactly the same way as the AL or *debug*.[13] After the program is loaded, when the user refers to the code or data in a particular object module, SoftScope uses the information in the debug segment of the module to provide a symbolic interface for the user.

Thus, only when a program is running under SoftScope does the debug information actually occupy any space in primary memory. Of course, at that time, the program being debugged is also sharing memory with SoftScope itself, and the development system must supply enough extra RAM to accommodate these extra requirements.

A useful concept to understand is how SoftScope displays source statements while stepping through a program being debugged. The actual source file is not a part of the object module or load module. Instead, SoftScope uses the listing file produced by the compiler for displaying source code statements. It uses the listing file instead of the source code file because the listing file contains line numbers that can be matched against the line number addresses in the debug segment of the module. SoftScope makes two assumptions about the path name of any listing file it needs: (1) The file is in the same directory as the load module file being debugged, and (2) The name of the listing file has the same base name and object module name, with an extension of `.LST`.

The second assumption is automatically matched by C language programs, as described earlier, but PLM programmers must ensure their source module name matches the base part of the source file name for SoftScope to work. If SoftScope cannot find a listing module it needs, it will prompt the user to type in the proper file name.

To debug a program using SoftScope I or II, just enter the command name (usually *sscope*) followed by the command line normally used to run the program. For example, to debug a program called *myprog* that takes three command line arguments, the command might be

```
iRMX> sscope myprog arg1 arg2 arg3                    [11]
```

If the program uses HI command line parsing, SoftScope I or II must be informed of this by entering `option parse =rmx` as the first SoftScope command after the program is loaded. If the program uses C language command line parsing, the `option parse` command can be omitted because the default option will work. In both cases, the situation is a bit messy

[13] SoftScope I and II load the program themselves, and SoftScope III uses the iRMX AL to load the program.

because the parsers will see `sscope` as the first item on the command line, not `myprog`.

For SoftScope III, the SoftScope kernel must be running to use the debugger. The usual technique is to run the kernel as a background command or with the *sysload*:

```
iRMX> background sskernel > :bb:                    [12]
```

or use *sysload*:

```
iRMX> sysload :utils:ss                              [13]
```

After entering either line `[12]` or `[13]` once, you can invoke the debugger by a command such as the following:

```
iRMX> ss myprog arg1 arg2 arg3                       [14]
```

SoftScope III has no `option parse` command, and programs that use C language parsing will run correctly with `myprog` as the first command line argument (`argr[0]`). The version of SoftScope III that is current at the time of this writing, however, does not support HI command line parsing at all.

Each version of SoftScope comes with complete documentation, including sample tutorial sessions, and there is an interactive help command as well. A summary of SoftScope III commands is provided in Appendix A.

3.9 Producing Linkable and Bootstrap-Loadable Modules

This section reviews some of the controls for *bnd386* typically used when generating a linkable file, and gives a brief overview of the steps used to build a standalone application — one that can be loaded for execution by a bootstrap loader.

3.9.1 Binder controls for linkable modules

The first binder control to discuss is `rc`, which must be omitted to produce a linkable module. In its place, the `noload` control is used, which can be abbreviated `nolo`, to tell the binder that the resulting module will not be loaded for execution under the control of an operating system.

To control which public symbols are carried forward into the linkable module, use the `nopublics except` or `publics except` binder controls, which are abbreviated `noply ec` and `ply ec`, respectively. After `ec`, put a comma-separated list of public symbols enclosed in parentheses in the command. For example, to bind `myprog.obj` with an interface library

to produce a linkable module that excludes all public symbols except `main_task` and `main_data`, use this command:

```
iRMX>bnd386 myprog.obj, /rmx386/lib/rmxifc32.lib &
**nolo nopl ec (main_task, main_data)                    [15]
```

The linkable module will be placed in a file named `myprog.lnk`. The name of the file could be changed using the `oj` control described earlier.

3.9.2 Adding a linkable module to the OS

Two types of code can be added to the iRMX OS: device drivers and resident applications. A device driver provides new DUIBs (which were discussed in the section on using floppy disks in chapter 2) and the code to handle the associated I/O device. A resident application is one that is loaded with the OS and starts executing when the OS initializes. This type of code is properly called a first-level job, which is explained further in chapter 6. One significant feature of both device drivers and first-level jobs is that they do not run as HI commands, are not associated with any particular logged on user, and thus have no access to a user's console input or console output device.

There are two techniques for adding code to the OS, depending on the version of iRMX being used. For iRMX III and iRMX for Windows, a command called *sysload* is used to load STL modules that are to remain resident in memory after control returns to the CLI. The differences between using *sysload* and the CLI's *background* command are the following:

- Programs that run from *sysload* cannot do console I/O, even through command line redirection.
- Programs that run from *sysload* cannot terminate by calling *rqexitiojob()* as the sample program did. They can terminate by making another system call, *rqdeletejob()*, or they can be terminated by means of a user-written HI command, somewhat like the CLI's *kill* command for terminating background programs.[14]
- Background programs have the same privileges as the user who issued the *background* command, but *sysload*-ed programs always have the privileges of the super user. This distinction is important for programs that access disk files, which involves checking a user's access rights.

The other technique for adding code to the OS is called *system configuration*. A special program called the Interactive Configuration Utility (ICU)

[14] iRMX for Windows `2.0c` introduced a version of *sysload* that can unload programs using the -u switch.

is used to edit a series of screens that describe various features of the operating system, including the pathnames of any linkable modules the user wants to add to the OS. The ICU generates a number of files, including source code files that must be assembled or compiled and linked into new linkable modules; a control file that tells *bld386*, the system builder, how to construct a bootstrap-loadable file; and a command file. The command file contains all the commands needed to compile the generated source files; to bind the new object modules, linkable modules supplied by the user, and configuration libraries supplied with the OS; and to build a new bootloadable copy of the OS in a disk file.

At the time of publication, *sysload* is available only for iRMX for Windows and iRMX III, and the ICU can be used only with configurable versions of the OS, which does not include iRMX for Windows. Many of the configuration parameters normally set by the ICU can be specified when an iRMX for Windows system is loaded into a file called `:config:rmx.ini`.

3.10 Debugging First-Level Jobs

The difficulty with debugging first-level jobs and device drivers is that the program is loaded by *sysload* or by the bootstrap loader rather than by the debugger. For iRMX I and iRMX II, this means that the only way to debug the code is to use the debug monitor and system debugger. The usual way to handle this situation is to insert a `causeinterrupt(3)` statement in the code so that the system will break to the monitor when the code is executed. The user is then left with the rather tedious process of debugging the code at the machine-language level.

For iRMX III and iRMX for Windows, SoftScope III can be used to debug first-level jobs and device drivers symbolically. Because the code to be debugged already has been loaded when SoftScope is started, the *ss* command, `load symbols`, is used to get the symbolic debugging information from the boot-loadable file. You may find that the ICU strips debugging information from the linkable or bootstrap-loadable module by inserting the `nodebug` control on the *bnd386* or *bld386* statements in the command file it generates. That control must be removed from the command file manually before submitting it for SoftScope to support symbolic debugging.

Chapter

4

Development Languages

Choosing the programming language for implementing an iRMX application or systems program profoundly affects how easy it is to develop the code and how efficiently the code performs when executing.

4.1 Overview

This chapter examines how a programming language affects a programmer's productivity through features of the source code language itself, how a programming language affects the run-time efficiency of a program via the run-time environment provided by the language, and how languages interact with the operating system. This chapter also mentions how the need to interact with the iRMX operating system affects the programming techniques used to implement algorithms, which can differ from the method of coding an algorithm using the same language, but a different operating system.

The three most commonly used languages for developing iRMX applications are assembler, PLM, and C. FORTRAN compilers are also available for all versions of iRMX, and Pascal compilers are available for some versions of iRMX, but these two are not as widely used as assembler, PLM, and C, so the focus is on these three.

The following table shows how the three languages can be ranked according to the criteria of programmer productivity, run-time efficiency, and ease of mastery, with rank of 1 being best:

Rank	Programmer Productivity	Run-Time Efficiency	Ease of Mastery
1	C	assembler	PLM
2	PLM	PLM	C
3	assembler	C	assembler

This table makes some assumptions in the rankings for run-time efficiency and ease of mastery that should be mentioned, even though the assumptions do not affect the discussion that follows. The assumption for run-time efficiency is that code for both PLM and assembler is written by programmers who are experts in both languages. The average assembly language programmer, however, does not generate code as efficiently as the PLM compiler running with all optimizations enabled. Assembly language programs are ranked as more efficient, however, because of the possibility of hand-tuning by an assembly-language expert. The assumption about ease of mastery is that the programmer does not already know any of the languages listed. Obviously, this assumption is often inappropriate, invalidating that part of the ranking.

Many people develop iRMX applications in assembly language, and a rudimentary knowledge of assembly language is just about essential for debugging iRMX applications. Regardless of these two facts, assembly language is dropped from consideration now for two reasons. First, assembler, as a rule, provides very little advantage over PLM and C in terms of which functions can be implemented. Note that the one exception to this rule is the direct manipulation of a processor's registers, which can be done only in assembler. Assembler is used in chapter 10 to perform some register operations, but otherwise, PLM and C provide all the functions needed to develop iRMX code, including device drivers. The second reason for leaving assembler aside is that the run-time efficiency benefits of assembly language programming are typically so small compared to the productivity benefits of programming in PLM or C that it is better to pursue high-level programming techniques instead.

The compilers and assemblers available for iRMX are all compatible with one another, so it is common to code different parts of an application using different languages. We focus on PLM and C separately in this chapter, looking at the technical issues involved in using the languages rather than on trying to decide which language is better than another. Note that this chapter is not a tutorial on the syntax for either PLM or C.

4.2 Source Language Issues

The sample program in chapter 3 used the iRMX system call `rqcsendcoresponse()` to perform console I/O, as well as another iRMX system call, `rqexitiojob()`, to terminate the program's execution. Instead of coding that program in PLM and using iRMX system calls, the same program could have been coded in C using the standard C library functions *printf()* and *gets()* for console I/O, and a *return* statement from *main()* to terminate the program. The code for such a C program is given in Figures 4.1 and 4.2. These two figures provide some interesting comparisons with Figures 3.3 and 3.4, which is considered in the material that follows. Before

Figure 4.1 C version of *hellormx* main program using C library functions.

```
/***>  hellormx.c  <**************************************************
 *
 *      Sample C program for iRMX
 *      -- main program using C library functions
 *
 ************************************************************************/

#include <stdio.h>

char *prompt = "Type something: ";

int
main (int argc, char *argv[]) {

char        reply[80];

  dosub (&reply);
  printf ("You typed: %s\n", &reply);
  return 0;
  }
```

Figure 4.2 C version of *hellormx* subroutine using C library functions.

```
/***>  hellosub.c  <**************************************************
 *
 *      Sample C program for iRMX
 *      -- subroutine using C library functions
 *
 ************************************************************************/

#include <stdio.h>

extern char *prompt;

void
dosub (char * inbuff) {
  printf ("%s", prompt);
  gets (inbuff);
  return;
  }
```

launching that comparison, however, it is important to note that these two pairs of source code files represent just 2 of 16 ways in which our sample application might have been constructed.

The 16 ways to construct the application are derived from the fact that the following four Boolean decisions could be made independently of one another.[1] The:

1. Main program could have been written in PLM or in C.

[1]Later in this chapter two versions of the C library (shared or non-shared) are discussed, both of which can be used, adding at least 12 more ways in which the sample program could be constructed.

2. Subroutine could have been written in PLM or in C.
3. Main program could have used either iRMX system calls or C library functions.
4. Subroutine could have used either iRMX system calls or C library functions.

In particular, consider Figures 4.3 and 4.4, which are C versions of the main program and subroutine that use the same iRMX system calls as Figures 3.3 and 3.4. Later in this chapter you will see two PLM versions of the program that call functions from the standard C library instead of making direct iRMX system calls. With three sets of sample programs (Figures 3.3 and 3.4, 4.1 and 4.2, 4.3 and 4.4), you are now ready to consider the first of the language issues, that of *include files.*

4.2.1 Include files

Both PLM and C include files are used to insert prototypes for external functions into a source module. Function prototypes tell the compiler how many parameters should be passed to a function or procedure, what the data types of those parameters should be, and what type of value, if any, will be returned. With this information available, the compiler can verify that references to these functions are coded correctly. Without function prototypes, the compiler would have to accept references to functions that are syntactically correct (balanced parentheses, etc.) but error inducing

Figure 4.3 C version of *hellormx* main program using iRMX system calls.

```
/***>  hellormx.c  <*********************************************
 *
 *      Sample C program for iRMX
 *      -- main program using iRMX system calls
 *
 ****************************************************************/

#include <rmxc.h>        /*  Header file for iRMX system calls.  */
#include <string.h>      /*  Header file for udistr(), etc.  */

char *prompt = "Type something: ";

int
main (int argc, char *argv[]) {

char         reply[80], *youTyped = "You typed: ";
WORD         Status;

  dosub (&reply);
  rqcsendcoresponse (NULL, 0, udistr (youTyped, youTyped), &Status);
  rqcsendcoresponse (NULL, 0, reply, &Status);
  rqexitiojob (0, NULL, &Status);
  }
```

Figure 4.4 C version of *hellormx* subroutine using iRMX system calls.

```
/***>  hellosub.c  <************************************************
 *
 *      Sample C program for iRMX
 *      -- subroutine using iRMX system calls
 *
 ************************************************************************/

#include <rmxc.h>              /*  header file for iRMX system calls  */
#include <string.h>

extern char *prompt;

void
dosub (char * inbuff) {

WORD    Status;

  rqcsendcoresponse (inbuff, 80, udistr (prompt, prompt), &Status);
  return;
  }
```

when the code the compiler generates is finally executed. A general principle in software development is that the earlier errors are detected in the development process, the easier they are to correct.

Traditionally, C compilers allow function prototypes to be omitted, making assumptions about functions referenced but not defined. ANSI C compilers, such as those used for iRMX development, provide for careful type checking through function prototypes. C programs should always code function prototypes explicitly to develop the most robust applications possible. PLM compilers always insist on a function prototype for every external procedure referenced by a source module.

For both languages, the issue is to know which header file to include to obtain the proper set of function prototypes for the functions actually called by the program. The PLM language supplies no standard header files, but the iRMX operating system provides a single file (`/rmx386/inc/rmxplm.ext`) that provides function prototypes for all iRMX system calls. For C, iRMX provides an equivalent header file, `:include:rmxc.h`, which is included in Figures 4.3 and 4.4. (Intel C compilers look in the `:include:` directory to find the include files, which are named inside angle brackets, such as <`rmxc.h`>.) The traditional C program shown in Figures 4.1 and 4.2, uses the standard I/O functions, *printf()* and *gets()*, to perform input and output operations, so these programs include the standard I/O header file, `:include:stdio.h`, instead. The prototypes for various C functions are kept in a number of different header files, and the documentation for each function tells which header file to include to obtain the corresponding prototype. Documentation for virtually all functions used for iRMX applications can be found in one of four places, described as follows.

Any book that documents the ANSI standard library for C programs, such as Harbison and Steele (1991) or Plauger (1992) provides documentation on C functions. Programs that use only those functions defined by the American National Standards Institute (ANSI) standard are most easily ported from one operating system to another.

The library reference manual that accompanies the compiler also describes these functions. For example, Intel C compilers for DOS and iRMX provide a number of functions for use with those operating systems that are not part of the ANSI standard. An example of such a function is *udistr( )*, which is used in Figures 4.3 and 4.4 and is discussed in the section on character strings later in this chapter. The manual that documents these functions is the *iC-86/286/386 Library Supplement*, volume 15 of the iRMX for Windows documentation set.

The functions that provide access to iRMX operating system services (iRMX system calls) are documented in the *iRMX System Call Reference*, volume 9 of the iRMX for Windows documentation set. The same documentation is available in hypertext format as a DOS command, *rmxhelp*, which is part of iRMX for Windows. The iRMX system calls are introduced in chapter 6 and beyond in this book. This book also provides C language function prototypes for the iRMX system calls discussed here.

The functions that provide access to iRMX networking services are documented in the *iRMX Network Programmer's Reference*. Network programming is introduced in chapter 11 of this book. This book also provides function prototypes for both the PLM and C languages.

In addition to function prototypes, both C and PLM allow include files to contain arbitrary pieces of source code, such as boilerplate comments or data structure declarations that are shared across several source modules. It is possible to include executable code in header or include files in both languages, but this option should be avoided if you plan to use a source-level debugger, such as SoftScope, which cannot trace such code. This restriction does not apply to macro definitions in C header files (see the next section for information on macros).

C header files are usually not displayed in the listing file produced by the compiler, but this default can be overridden by the `listinclude` compiler control, abbreviated as `lc`. PLM include files do appear in the listing file by default, but most PLM include files, including those produced by *extgen*, start with the `save` and `nolist` compiler controls to suppress the listing, and end with the `restore` compiler control to resume listing of the source code. *Extgen* is the utility program introduced in chapter 3 that generates small include files for PLM programs based on the actual system calls the programs reference.

4.2.2 Macro preprocessing

Both C and PLM provide a text substitution facility, but the C macro capability is much more powerful than that available for PLM. PLM's macro

facility is provided by the LITERALLY clause, which does not make any provision for arguments and is restricted to a maximum of 255 bytes of text per macro. The include files for the sample PLM program in chapter 3 (Figures 3.6 and 3.7) illustrate typical uses for PLM's LITERALLY clause, such as providing symbolic names for commonly used data constants (TRUE and FALSE) and giving alternative names to data types (TOKEN and WORD_16). The operating system supplies PLM include files that provide symbolic names for commonly used data type and constant declarations (/rmx386/inc/common.lit, for system call condition codes /rmx386/inc/error.lit, and for various other constants useful in PLM programs for iRMX).

The C language macro facility provides for parameter substitution within a macro body and uses a declaration and invocation syntax that makes macro invocations look like function calls. This feature lets the designer of a C function decide whether to optimize for either memory usage or execution speed. Functions use less memory because the machine code for a function is generated only once regardless of from how many places it is called, whereas machine code for a macro is inserted into a program each place the macro is referenced. Macros execute faster because they avoid the overhead of passing parameters and manipulating return addresses. Programmers might not know whether they are using a macro or calling a function when the macro definition or function prototype is part of a header file and not displayed in the source code listing.

4.2.3 I/O support

The most significant difference between the two languages is I/O processing. The C language is well-known for its standard library of I/O functions, codified by the ANSI and adopted by POSIX.1. C programs that use these functions are automatically portable across all operating systems that support ANSI C. This standardization of I/O functions by no means implies that the I/O functions are implemented equivalently on different systems. For example, the reference to stdin in the sample C program is actually a macro reference, which is defined in :include:stdio.h for iRMX as:

```
#define stdin (&_stdio_ptr()->_stdin)
```

When the program is run under DOS using Borland C, the corresponding stdio.h file contains the definition:

```
#define stdin (&_streams[0])
```

A corresponding Unix header file contains:

```
#define stdin (&iob[0])
```

Thus, ANSI C programs are portable with respect to I/O operations at the source code level, provided the header files provided with the compiler are not considered part of the source code.

The contrast with PLM could not be greater. *The PLM language does not support I/O at all*, except for very primitive operations used for coding device drivers. There is nothing like C's *printf( )*, nothing like Pascal's *writeln*, nothing. So what is *rqcsendcoresponse( )*, which was used in the sample PLM program in chapter 3? It is a system call provided by iRMX, not part of the PLM language.

At this point, the difference between the two languages regarding I/O might still appear to be minor. In the sample programs, the PLM program used one system call, *rqcsendcoresponse( )* to accomplish what the C program in Figures 4.1 and 4.2 did with two of its standard I/O functions, *printf( )* and *gets( )*. The difference between the two seems even more trivial when you see that C programs can make iRMX system calls directly just as PLM programs can, and that PLM programs can call all the standard C library functions as well. A PLM version of the sample program that calls functions from the C run-time library is given in Figures 4.10 and 4.11, for example.

The sample programs blur the distinctions between the I/O facilities of C and PLM because they happen to do only character string I/O. Consider instead a program that operates with floating point data.

4.2.4 Floating point support

Both languages support floating-point data types and computations using an x87 numeric hardware coprocessor, which should be initialized before being used, or an emulator. PLM programs explicitly initialize the coprocessor by calling the built-in procedure *initrealmathunit( )*. Several options can be set when using the coprocessor, such as its rounding mode and error reporting. The PLM built-in procedure *setrealmode( )* can be called to change these options for particular applications, and there are other built-in procedures for testing, saving, and restoring the status of the coprocessor.

Although Intel numeric coprocessors implement the IEEE-754 standard for floating-point computations, ANSI C does not specify a standard method for controlling the options to control floating-point operations. Thus, for ANSI C programs, the coprocessor is automatically initialized, and the various options cannot be changed. The Intel C compilers, however, provide the same built-in functions as the PLM compilers for initializing the coprocessor, changing its options, and managing its status, if desired.

At this point, a significant difference between the I/O processes of the languages emerges: C programs can read in and output floating-point values easily using the `%f` conversion format with standard I/O functions

such as *printf()* and *scanf()*, but no such conversions exist for PLM. This point is made in the context of floating-point conversions, but the situation is the same for integers as well: PLM programs can make system calls to do I/O, but the only data type that can be input or output is an array of bytes, the equivalent of C's `unsigned char[]`. This issue is obscured for the sample programs, because they only do I/O with character strings, which map fairly directly onto arrays of characters (bytes) in both languages.

PLM programs that use numbers to communicate with human users must provide routines to convert the numbers from their external representation as character strings to their internal representation as integers or reals and back. These conversion routines are either written by the application developer or drawn from a library of such routines. Such a library is designed for use with assembly language programs, but it can be accessed from PLM programs as well, in `/lib/ndp387/dc387f.lib`. Figure 4.5 is a listing of a PLM program that uses two of the routines from this library to input two floating point numbers and display their sum. As you can see, it takes quite a bit of code to convert between character and floating-point representations of real numbers. The code that performs equivalent conversions generally adds quite a bit of overhead to C programs that work with `float` or `double` data.

Figure 4.5 Sample PLM program that illustrates floating-point I/O using conversion routines from `/lib/ndp387/dc387f.lib`.

```
/***>  floatest.plm  <*********************************************
 *
 *  Sample PLM program to illustrate floating-point I/O
 *  This code illustrates the use of the routines mqcdec_bin and
 *  mqcbin_declow from the library /lib/ndp387/dc387f.lib.
 *
 ************************************************************************/
$title ('Sample Program to Illustrate Floating-Point I/O')
$compact (exports mqcdec_bin, mqcbin_declow)

floatest: DO;
$include (floatest.ext)
DECLARE cr          LITERALLY   '0Dh',
        lf          LITERALLY   '0Ah';
DECLARE Status      WORD_16,
        (x,y,z)     REAL,
        index       BYTE,
        prompt1 (*) BYTE INITIAL (' Enter first value: '),
        prompt2 (*) BYTE INITIAL (' Enter second value: '),
        answer (*)  BYTE INITIAL ('+zzzzzz.E+zz',cr,lf),
        outbuf (81) BYTE INITIAL (' The sum of '),
        string (21) BYTE;

/*  External Procedure Declarations
 */
mqcdec_bin: PROCEDURE (dcb$ptr) EXTERNAL;
```

Figure 4.5 (*Continued*)

```
DECLARE dcb$ptr     POINTER;
END mqcdec_bin;

mqcbin_declow: PROCEDURE (adcb$ptr) EXTERNAL;
DECLARE adcb$ptr    POINTER;
END mqcbin_declow;

/*  Data Structures Used by Conversion Routines
 */
DECLARE dcb     STRUCTURE ( /*  Decimal Conversion Block  */
        b_buffer    POINTER,    /*  Binary Buffer  */
        precision   BYTE,
        d_length    BYTE,
        d_buffer    POINTER);   /*  Decimal Character Buffer  */
DECLARE adcb    STRUCTURE ( /*  Augmented Decimal Conversion Block  */
        b_buffer    POINTER,    /*  Binary Buffer  */
        precision   BYTE,
        d_length    BYTE,
        d_buffer    POINTER,    /*  Decimal Character Buffer  */
        scale       SHORTINT,   /*  True Decimal Exponent  */
        sign        BYTE);      /*  plus or minus character */
DECLARE single      LITERALLY   '0', /* Codes for precision */
        double      LITERALLY   '2',
        extended    LITERALLY   '3';

$subtitle ('Main Program')
/*  Coprocessor Initialization.
 *  No need to use INIT87 or INITFP with PLM.
 */
    CALL init$real$math$unit;

/*  Get the external representation of the first value...
 */
    prompt1(0) = length (prompt1) - 1;
    CALL rqcsendcoresponse (@string, 20, @prompt1, @status);

/*  ...and convert it to floating-point format.
 */
    index = skipb (@string(1), ' ', string(0));
    dcb.b_buffer = @x;
    dcb.precision = single;
    dcb.d_length = string(0) - index - 2;
    dcb.d_buffer = @string(index + 1);
    CALL mqcdec_bin (@dcb);

    CALL movb (dcb.d_buffer, @outbuf(12), dcb.d_length);
    CALL movb (@(' and '), @outbuf(12 + dcb.d_length), 5);
    outbuf(0) = 17 + dcb.d_length;

/*  Get the external representation of the second value...
 */
    prompt2(0) = length (prompt2) - 1;
    CALL rqcsendcoresponse (@string, 20, @prompt2, @status);

/*  ...and convert it to floating-point format.
 */
    index = skipb (@string(1), ' ', string(0));
    dcb.b_buffer = @y;
```

Figure 4.5 (*Continued*)

```
      dcb.precision = single;
    dcb.d_length = string(0) - index - 2;
    dcb.d_buffer = @string(index + 1);
    CALL mqcdec_bin (@dcb);

    CALL movb (dcb.d_buffer, @outbuf(outbuf(0)), dcb.d_length);
    outbuf(0) = outbuf(0) + dcb.d_length;
    CALL movb (@(' is '), @outbuf(outbuf(0)), 4);
    outbuf(0) = outbuf(0) + 4;
/*  Computation section
 */
    z = x + y;

/*  Generate the result string
 */
    adcb.b_buffer = @z;
    adcb.precision = single;
    adcb.d_length = 6;
    adcb.d_buffer = @answer(1);
    CALL mqcbin_declow (@adcb);
    answer(0) = adcb.sign;
    IF adcb.scale < 0 THEN DO;
      answer(length(answer) - 5) = '-';
      adcb.scale = -adcb.scale;
      END;
    answer(length(answer) - 3) = BYTE(adcb.scale mod 10 + '0');
    answer(length(answer) - 4) = BYTE(adcb.scale / 10 + '0');

/*  Display the result and exit
 */
    CALL movb (@answer, @outbuf(outbuf(0)), length(answer));
    outbuf(0) = outbuf(0) + length(answer);
    CALL rqcsendcoresponse (nil, 0, @outbuf, @Status);
    CALL rqexitiojob (0, nil, @Status);
END floatest;
```

4.2.5 16- and 32-bit targets

Recall from earlier chapters that there are three versions of iRMX: iRMX I, which runs in real mode on microprocessors with 16-bit words; iRMX II, which runs in protected mode on microprocessors with 16-bit words; and iRMX III, which runs in protected mode on microprocessors with 32-bit words. iRMX for Windows is iRMX III with added software to support concurrent operation of iRMX III, DOS, and Windows.

Chapter 5 discusses the architectural significance of different word sizes and processor modes. Chapter 12 describes the features that iRMX for Windows adds to iRMX III. The focus of this section is how different word sizes affect code written in PLM or C for the different versions of the operating system. For present purposes, there is no need to differentiate between iRMX III and iRMX for Windows.

Different compilers allocate different amounts of memory for variables declared identically. For example, a variable declared to be of type WORD in

PLM-86 or PLM-286 is 16 bits of memory, but a variable declared the same way in PLM-386 is allocated 32 bits. Likewise, an `int` is 16 bits in iC-86 and iC-286, but 32 bits in iC-386. The problem of variable sizes depending on the word size of the target computer is a general one, and not restricted to the compilers used to develop code for iRMX. If source code is being developed for use with only a single target architecture, the problem is minimal, except for ensuring that the correct parameter types are passed to library or system call functions.

If source code is designed to run on different versions of iRMX, the problem is more significant, but can still be dealt with using straightforward techniques. Regardless of how many different architectures are to be the target of code being developed, several options exist for dealing with target word size. C programmers have the following options:

- The C data types `short` and `long` are always 16 and 32 bits long, respectively, regardless of the version of the compiler. This consistency is part of the ANSI standard for the C language, so all programs can rely on it.
- The Intel C compilers provide a predefined macro called `_ARCHITECTURE_` that returns values of 86, 286, and 386 for iC-86, iC-286, and iC-386, respectively. Conditional compilation (using the preprocessor directive `#if`) can be used to cause different code to be compiled depending on the architecture of the target processor.
- The header file for iRMX system calls, `:include:rmxc.h`, includes definitions for certain PLM data types. The term *PLM data types* is a bit of a misnomer, as will be seen directly. Nonetheless, the data types `BYTE`, `WORD`, and `DWORD` can always be used for 8-, 16-, and 32-bit unsigned integers respectively. Other PLM data types are declared in `:include:rmxc.h`, but the three listed here work specifically with different sizes of unsigned values, and appear frequently in the sample code in this book.
- The `rmxc.h` header file also defines a data type called `NATIVE_WORD`, which is a 16-bit unsigned integer for iC-86 and iC-286, or a 32-bit unsigned integer for iC-386.

In PLM, a variable declared to be of type `WORD` will be allocated 16 bits by the PLM-86 or PLM-286 compiler, but is allocated 32 bits of memory by the PLM-386 compiler. When using the PLM-386 compiler, there are two ways to allocate 16 bits to a variable: either declare the variable to be of type `HWORD`, or use the `WORD16` compiler control, which causes the compiler to use 16 bits for all `WORD` variables in the program. When you use the `WORD16` compiler control, you can test a compiler variable, also named `WORD16`, with a `$if` compiler control, as seen in the boilerplate code generated by *extgen* in chapter 3. The sample PLM code in this book uses the data types

WORD_16 and WORD_32 established by *extgen* to declare variables with proper word sizes.

Programs that make iRMX system calls must know the word size of the target processor, whether the code is intended to run on a single version of the operating system or on different versions. Fortunately, including the proper header files in the program allows the compiler to verify that the proper data types are being passed to the operating system functions. A properly coded program can be compiled to run without any syntax errors for any version of iRMX using the techniques previously listed. Certain system calls are unique to specific versions of the OS, so a program designed to run on different versions of iRMX must also take this factor into account.

The relationship between iRMX II and iRMX III applications deserves special note. Loadable iRMX II modules (16-bit code developed using PLM-286 or iC-286 and BND286) execute under iRMX III without any need to recompile or rebind. The secret to this binary compatibility lies in the libraries linked to 32-bit applications that make iRMX system calls. All parameters are passed to subroutines on a pushdown stack in memory, which uses 16-bit words or 32-bit words, depending on the development tools and libraries used to construct the application. The 32-bit libraries include code that passes a flag variable to the operating system to indicate the use of a 32-bit stack segment. If an iRMX III operating system function does not find this flag variable on the stack, it automatically adjusts to work with a 16-bit stack. One reason 32-bit code can't be run on iRMX II is that iRMX II does not adjust to the 32-bit stack size. More important, the iRMX II Application Loader (AL) does not recognize the 80386 Object Module Format generated by *bnd386*.

4.2.6 Scoping rules

In PLM, all data and procedure names declared within a source module are private to the module unless explicitly declared to be public. In addition, the only way a PLM program can reference data or procedures declared in a different source module is to declare the symbolic name to be external. Of all the object modules combined by the binder, exactly one can contain a public declaration of a symbol, and any number of other modules can refer to the symbol as an external.

PLM variables are allocated storage in the data segment of the module. Although variables declared inside a procedure cannot be accessed from outside the procedure, the values of the variables are retained between calls to the procedure by default. The exception to this rule is any procedure declared to be *reentrant*. In this case, local variables are allocated storage on the run-time stack when the procedure is entered, and when the procedure

returns, the storage is freed, so the values of local variables are not preserved across calls.

The C language essentially operates in a complementary fashion to PLM. For C, all data declared outside any function and all functions themselves are automatically either public or external unless explicitly restricted to a single module by being declared *static*. Whether a non-static symbol is public or external depends on whether the symbol is being defined (public) or referenced (external). To complete the complementary pattern, all C functions are reentrant by default, which means that all variables declared local to the function are allocated storage on the run-time stack unless declared to be static.

The character string `prompt` is declared and defined in the main module of our sample C program (Figures 4.1 and 4.3), thus being a public symbol in that object module. It is declared with the `extern` storage class in the subroutine module (Figures 4.2 and 4.4). If `prompt` had not been declared `extern` here, it would have been considered a defining declaration, and *bnd386* would have issued a warning about duplicate public declarations.

4.2.7 Function prototypes

In the sample C program, no function prototype for the function *dosub()* exists in the main module (Figures 4.1 and 4.3). In this situation, the compiler provides the function with a prototype based on the form of the reference to *dosub()* in the code. For the sample program, the ANSI standard specifies that the function be prototyped as returning an `int`, even though the function actually returns nothing, as established by its declaration as type `void` in the second module. This inconsistency results in a conflict between the two declarations in the object modules `hellormx` and `hellosub`. Technically, the binder should issue a warning for this type mismatch, even though the code will run without error because the main program does not actually try to use any return value from the function. Current versions of *bnd386* do not issue any warning or error for this specific type mismatch, probably because it is a benign problem characteristic of many C programs.[2]

If this type of symbolic mismatch is a problem, a development tool called *map386* can be useful. The tool was not previously mentioned in chapter 3 because it is not a required step in the development process. *Map386* produces a complete symbolic map for a load module. The command

```
iRMX> map386 hellormx                                    [1]
```

generates a file named `hellormx.map`, which lists information about

[2]The current iC-386 compiler issues a "remark" about the missing function prototype when it compiles `hellormx.c`.

every public symbol for all linkable modules combined to produce a load module. This map file can be very useful for tracking down linking errors that are not clear from the binder's `.mpl` file. For example, the `.map` file for the sample program includes a warning for the symbol type mismatch of *dosub()*, even though the `.mpl` file produced by *bnd386* does not.

This description of C declarations omits some details. See chapter 4 in Harbison and Steele (1991) for more information. The main issue here is to point out some of the differences between PLM and C that can lead to unexpected problems during the binding stage if they are not understood.

4.3 Run-time Considerations

This section discusses the run-time environments provided by C and PLM. (Other run-time considerations specifically relating to iRMX are covered in the next section.)

Run-time efficiency can be optimized for either speed or memory requirements. Both C and PLM compilers allow the developer to select the level of optimization for generating object code, using a value from 0 (no optimization) to 3 (maximum optimization). Most object code optimizations improve the speed of the program, often by eliminating superfluous instructions, which improves the memory requirements of the program as well. Both compilers are expected to produce object code of equivalent size and speed when working at the same optimization levels.

The two languages differ most in the size of the code that must be bound together to produce an executable program. Figure 4.6 represents the memory structure of a loadable iRMX application, such as an HI command. The lower part of the figure is the OS itself, which is always completely resident in memory. The upper part of the figure represents one loadable application. Systems with multiple users or users running multiple programs simultaneously (using the CLI's *background* command or the HI *sysload*

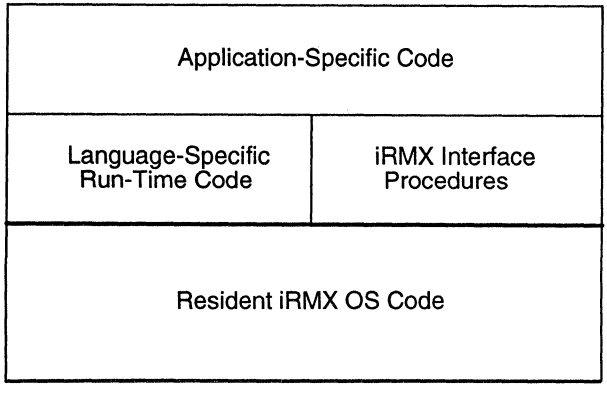


Figure 4.6 Main memory structure for a loadable iRMX application program.

command) might have multiple loadable applications in memory at the same time. Each application consists of three parts: application-specific code, language-specific code, and interface procedures.

Application-specific code is derived from the source code written by the application programmer and compiled into object modules by the compiler. This code is made part of an application by listing the object module files for the application first in the input path list of a *bnd386* command line. The application-specific code for all versions of the sample program is in the files `hellormx.obj` and `hellosub.obj`.

Interface procedures are used to access system calls, which are subroutines located in the resident part of the OS. The interface procedures come from a library, such as `/rmx386/lib/rmxifc32.lib` for our sample program, which is supplied as part of the OS. The library of interface procedures for system calls is language independent, and must be bound with all application programs, whether written in PLM, C, or assembler. It will be necessary to understand the internal logic of interface procedures in chapter 10 when adding new system calls to the operating system are discussed, but consider these interface procedures to be black boxes for now.

Currently, you should pay attention to the language-specific run-time libraries that must be bound with programs written in high-level languages. These libraries contain functions and procedures that are called by the application-specific code explicitly, implicitly, or indirectly.

Explicit functions. Explicit functions are routines called by an application but considered part of the programming language. The entire standard C library consists of explicit functions, including *printf( )* and *gets( )* for the sample program in Figures 4.1 and 4.2. The PLM-386 language includes very few explicit functions, only a few routines for manipulating bit arrays.

Implicit functions. Implicit functions are routines that do not appear by name in an application's source code but for which the compiler inserts machine language *call* instructions to invoke them. These functions provide operations supported by the language but implemented as subroutines by the compiler rather than with in-line code. For example, both PLM and C support multiplication and division of doubleword integers, but both compilers use implicit functions to perform these particular computations.

Indirect functions. Indirect functions are routines that are called by functions called by the application's code. A language-independent example of an indirect function is *rqerror( )*, which is referenced by the interface procedures for all iRMX system calls. *rqerror( )* is called by an interface procedure if the system call returns an error condition. The PLM language does not use any indirect functions, but the C language explicit functions make many indirect function references. For example, the *printf( )* function indirectly calls other functions to format the characters for output and then

perform the actual printing. The presence of indirect function references helps determine the order in which library files are listed on the binder's command line. For example, if the C run-time library makes indirect references to the iRMX system call interface library, the C library must appear before the iRMX interface library so the binder can resolve the indirect references successfully.

The run-time routines for the implicit functions referenced by the PLM and C compilers are found in the library files `/intel/lib/plm386.lib` (`:lang:plm386.lib` for iRMX versions other than iRMX for Windows) and `:lang:ic386.lib` for PLM and C, respectively.[3] These run-time libraries are not only very small, but are actually used only for applications that perform 64-bit multiply and divide operations. For example, although it was included in the input file list for binding the sample PLM program in chapter 3, no modules from `plm386.lib` were actually included in the loadable file produced by the binder. The same is true of `ic386.lib` for the sample programs in this chapter. Implicit functions never make indirect function references, so the language-specific run-time libraries are always listed last in the input file list for the binder.

No library of explicit functions for PLM programs exists other than the library of interface procedures for iRMX system calls, so the following discussion is specific only to C-language programs or PLM programs that call routines in the C run-time library.

The explicit and indirect functions called by a C program come from two different files. The first is a single object module called the *C start-off code*, and the second is the C run-time library itself. The C language does not provides a method to generate an execution starting point as PLM and assembler do. PLM programs start executing at the first executable statement that appears outside of any procedure; there must be only one such statement among all the object modules bound together to create an executable program. Assembler programs use the `END` directive to specify the execution starting point.

The C compiler never specifies an execution starting point in the object modules it produces. Rather, the convention is for execution to begin at the function named *main()*, which must be called by an assembly language or PLM program known as the *start-off program*. The code in the start-off module performs a number of initializations (described as follows), and then calls *main()*, passing command-line arguments in the traditional `argc` and `argv` parameters. If *main()* returns to the start-off module, it will encounter an indirect call to *cq_exit()*, which in turn will make an indirect call to the iRMX system call *rqexitiojob()* to terminate execution.

---

[3]The implicit library for C is in the interface library for the shared C library for those systems, including iRMX for Windows, that support that library. The shared C library, also called the C layer, is described later in this section.

There are two ways to handle the run-time library for C programs. Each C program can be bound to the necessary parts of the library, or the entire library can be loaded with the operating system so that C programs can invoke any library function the same way they call iRMX system calls — by calling interface procedures. Figure 4.6 represents the case in which each application is linked to the C library, and Figure 4.7 represents the case in which a single resident copy of the run-time library is shared among all C applications. The shared copy of the C run-time library is pre-linked to the iRMX system call interface procedures that actually perform operating system functions such as I/O and program termination, so those interface procedures are shown as part of the resident OS. If an application program made iRMX system calls in addition to C run-time library calls, interface procedures for those system calls would need to be included in the application program's memory in the upper part of the figure.

Because Figures 4.6 and 4.7 are not drawn to scale, the major difference between the two techniques is not immediately obvious: The shared resident version of the library results in much smaller application programs at the expense of more memory dedicated to the resident portion of the OS. In the current version of iRMX for Windows, the resident C library adds 107 KB to the memory requirements of the operating system.

The size of the C nonshared run-time library is measured in the hundreds of thousands of bytes, and virtually all C programs extensively use the routines it provides. The C version of the *hellormx* program that calls *gets()* and *printf()* in Figures 4.1 and 4.2, for example, binds to 75 modules directly or indirectly from the C run-time library `crmxnf3c.lib`. Furthermore, the functions in these modules reference an additional 51 modules from `rmxifc32.lib`, resulting in a loadable command file of over 80 KB.

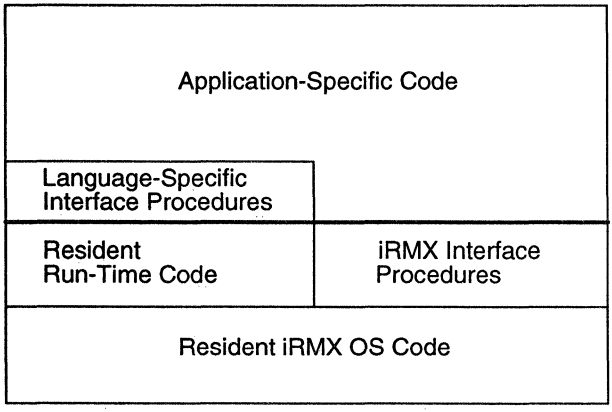


Figure 4.7 Main memory structure for a loadable iRMX application using the resident C run-time library.

If the same program is bound to the library of interface procedures for the resident version of the library instead, only 11 modules are included from that library, and no indirect references are made to the iRMX interface procedure library. The loadable command file is only about 10 KB. The PLM version of the same program (Figures 3.3 and 3.4), incidentally, is less than 7 KB after binding to the iRMX interface procedure library.

Table 4.1 summarizes the size requirements for the PLM and C versions of the *hellormx* program presented in this chapter and chapter 3, including the PLM programs that use C library functions, which are examined later in this section. For the programs, I/O and program termination were coded either to make iRMX system calls directly or to call functions in the C run-time library. Programs that call functions in the C run-time library were linked to either the shared or the non-shared version of the library. The number of object modules linked from the C library is given as $1 + n$ for those programs that require the C start-off code, and as n otherwise. The *Loadable* columns give the sizes of the loadable files (executable STL files) with and without debugging information retained. The *Code + Data* column gives the sum of the code and data memory segments for each program, but does not include memory requirements for the program's stack segment, nor for data segments created as the program executes. All compilations were performed using the compiler's `compact` and `debug` controls. All C programs were compiled using the compiler's `nosrclines` control.

The table points out the ambiguity involved in citing the size of a program. As can be seen, the sizes of the loadable files vary greatly, depending on whether they include debugging information. As discussed in chapter 3, this debugging information is discarded by the iRMX AL, but is used by source language debuggers such as SoftScope. The actual amount of memory used by a program is more accurately given by the *Code + Data* column of the table, which shows the actual amount of memory occupied by the code and data segments of the programs. Even this column is not a totally accurate representation of a program's memory requirements, however, because both iRMX system calls and C run-time library functions create additional data memory segments as a program executes.

TABLE 4.1 Sizes of Various Versions of the *Hellormx* Program

Source Language	Functions	C Library	C Modules	iRMX Modules	Loadable w/Debug	Loadable w/o Debug	Code + Data
PLM	iRMX	none	0	7	6,873	4,380	455
C	C	shared	1 + 11	0	10,791	8,787	5,161
C	C	not shared	1 + 75	51	80,821	46,912	52,369
C	iRMX	shared	1 + 10	7	12,025	9,062	5,430
C	iRMX	not shared	1 + 72	53	75,900	42,916	48,369
PLM	C	shared	6	0	6,355	4,386	376
PLM	C	not shared	1 + 75	51	81,478	46,907	52,449

The following two command lines illustrate the two ways to build the version of *hellormx* that uses the C run-time library:

```
iRMX> bnd386 &
** :lib:cstrmx3c.obj, hellormx.obj, hellosub.obj, &
** :lib:crmxnf3c.lib, /rmx386/lib/rmxifc32.lib, &
** rn(code to code32) ss (stack(8192))
rc(dm(0,0FFFFFFh)) &
** oj(hellormx)                                          [2]
```

```
iRMX> bnd386 &
** :lib:cstart32.obj, hellormx.obj, hellosub.obj, &
** :lib:clibxf32.lib &
** rn (code to code32) ss(stack(8192))
rc(dm(0,0FFFFFFh)) &
** oj (hellormx)                                          [3]
```

Command line [2] illustrates the use of the non-shared version of the library, and command line [3] illustrates the use of the shared version. The logical name `:lib:` in both command lines should not be taken literally because the locations of the various libraries sometimes change positions from one release of iRMX to another, or across versions of the OS. For iRMX for Windows 2.0a, `:lib:` refers to the directory `/rmx386/new/intel/lib` in command line [2] and refers to the directory `/intel/lib` in command line [3].[4]

The two command lines reference different start-off files as well as different run-time libraries. The start-off code for the non-shared version of the library in line [2] (`cstrmx3.obj`) performs much more initialization, to be described in the section on multitasking, than the start-off code for the shared version of the library in line [3] (`cstart32.obj`). Note that the binder normally generates its output in an STL file that has the same name as the first file of its input file list with the extension of the file name dropped. If the `oj()` control had not been used on the command lines, the binder would have produced its output in a file in the `:lib:` directory. Aside from not wanting the executable program to be named `:lib:cstrmx3` or `:lib:cstart32`, the bind commands would probably fail on any iRMX file systems where most users do not have write permission for the `:lib:` directory.

Several versions of non-shared run-time libraries are supplied with the C compilers for iRMX. The name of the one used in command line [2] indicates that it is used to interface C programs to iRMX (`crmx...`), that it

---

[4]The names and directories for library files have changed several times during the evolution of the iRMX for Windows operating system. For example, the command to build the hellormx program using the shared C library in iRMX for Windows version 2.0c is:

```
iRMX> bnd386 &
**  /intel/lib/cstart32.obj, hellormx.obj, hellosub.obj, &
**  /intel/lib/cifc32.lib &
**  rn(code32 to code) ss(stack(8192)) &
**  rc(dm(0,0FFFFFFFFh)) oj(hellormx)
```

provides no floating-point support (`...nf...`), that it is for iRMX III (`...3...`) and that it is used for programs compiled using the compact segmentation model (`...c.lib`). At the time of this writing there is just one version of the shared run-time library available. It is found in the file `/rmx386/jobs/clib.job`, which can be called from any iRMX III application regardless of its model of compilation, and which includes floating point support. There is presently no library of interface procedures to this library for 16-bit applications.

4.4 Congruence with iRMX

This section discusses issues that must be handled properly to make iRMX system calls from C and PLM programs. The issues mainly involve passing arguments to system calls and receiving results back from the OS, but there are also considerations that focus specifically on multiple tasks and multiple jobs for C programs.

4.4.1 Character strings

iRMX system calls that receive character string arguments or return character string values, such as `rqcsendcoresponse()` in the sample PLM program, expect the character string to be represented as an array of bytes. The first element of the array contains an unsigned binary number specifying the number of characters in the string, and the next bytes in memory contain the ASCII codes for the characters in the string. For example, if a user of the PLM version of *hellormx* typed `ABC<cr>` in response to the `Type something:` prompt, the first six bytes of the array `Reply` would be filled with the following sequence of values, given in hexadecimal:

```
05 41 42 43 0D 0A . . .
```

The first byte contains the length of the string, which is five characters long. The next three bytes contain the ASCII codes for the letters A, B, and C, and the next two bytes contain the ASCII codes for carriage return (0x0D) and linefeed (0x0A). The remainder of the bytes in the array would not be affected by this call to *rqcsendcoresponse()*. PLM has no character string data type, so arrays of bytes are used for all iRMX strings. PLM programs must explicitly set the length of strings that will be passed to iRMX system calls, by using the built-in *length()* function, for example, to compute the length to be stored in array element zero, as shown in the statement: `Prompt(0) = length (Prompt) -1;` in the PLM version of *hellormx*.

The C language does have a character string data type, which is represented in memory as an array of bytes terminated by a byte containing binary zeros (known as *null-terminated string*). This representation of strings can be used as long as the C run-time library only is involved in any string operations, such as in the C version of *hellormx* in Figures 4.1 and 4.2. When iRMX system calls are made from C programs, however, C

strings must be converted to iRMX strings and vice versa. The C run-time library supplied with Intel C compilers provides functions named *udistr()* and *cstr()* for these conversions. The function prototypes for these functions are given in `:include:string.h`, along with the function prototypes for the ANSI string-handling functions. The code in Figures 4.3 and 4.4 illustrates using *udistr()* to convert C strings to iRMX strings. (The name *udistr()* refers to the Universal Development Interface (UDI) layer of iRMX, but all layers of the OS use the same representation for strings.)

iRMX strings cannot be longer than 255 bytes because of the limitation of storing the length of the string in a single byte, but each byte of the string can contain an arbitrary bit pattern, including all zeros. C strings can be longer, but cannot contain any bytes with all zeros within the string. C programmers must remember these differences when using strings with iRMX system calls.

The C language allows strings to include control characters using an escape mechanism. For example, the C string `''hello\n''` uses <\n> to insert a new-line (ASCII linefeed) character into the string. PLM strings do not support such an escape mechanism. Rather, arbitrary values can be used to initialize byte arrays by simply placing their value in the initialization list. The PLM initialization list, `('hello', 0Ah, 0)` is equivalent to the preceding C string. Note that the byte of zeros at the end of this list must be given explicitly in PLM, but is automatically appended to the end of all C-language strings.

One final issue concerning strings that C programmers need to be aware of is the difference between signed and unsigned characters. By default, the `char` data type is a signed data type, according to the ANSI standard for the C language. (Watch for a sign extension when a `char` value greater than 0x7F is promoted to an `int`.) The function prototypes for the ANSI string manipulation functions all specify signed characters (or pointers to them) for their arguments. iRMX system calls that take string arguments, however, are prototyped to take unsigned `char` arguments. The PLM `BYTE` data type defined in `:include:rmxc.h` is for unsigned characters, and it is sometimes necessary to be aware of the difference between the two data types when mixing iRMX system calls and C string operations. The `:include:rmxc.h` header file defines the `STRING` data type to be signed characters and can be used for C applications that need signed characters.

4.4.2 Parameter passing

When establishing protocols for passing parameters to subroutines and receiving returned values, you must consider the specific architecture of the CPU being used, the programming languages involved, and the operating system. An early example of treating all these issues uniformly was the IBM S/360 architecture introduced in the 1960s. IBM specified that all programming languages and all operating systems that ran on the S/360

architecture would use the general purpose registers of the CPU in a standard way when calling and returning from subroutines.[5]

iRMX, C, and PLM all use the stack system of the x86 architecture for managing parameters, return addresses, and return values for subroutine calls. Differences exist, however, between how C uses the stack and how iRMX and PLM use the stack, and these differences must be taken into consideration. Readers familiar with making system calls to other operating systems such as DOS or OS/2 should remember that those two systems use a register-based scheme for passing parameters to system calls rather than the stack scheme used by iRMX. Also, Unix programmers have not dealt with this issue at all because Unix and the implementations of C on Unix always use a single calling convention, which is stack-based on those processors that support it.

On stack-based systems, each subroutine call results in a data structure called a *stack frame* put on the top of the pushdown stack kept in memory. A CPU register called the *stack pointer* (`sp`) always points to the top of this stack, and another register called the *base pointer* (`bp`) keeps track of nested stack frames. Figure 4.8 shows the general structure of a stack frame during a subroutine call. Note that stacks grow downward; `sp` always has the lowest memory address occupied by the stack. The three parts of a stack frame are constructed in three separate steps:

1. A sequence of machine language *push* instructions pushes the arguments to be passed to the subroutine onto the stack.
2. A machine language *call* instruction pushes the return address onto the stack. The return address is the address of the next machine language instruction in the calling program after the *call* instruction.
3. Finally, for re-entrant subroutines, space is reserved on the stack for the local variables of the subroutine by subtracting the number of bytes needed for local variables from the `sp` register. Subroutines address incoming parameters by specifying positive offsets relative to the `bp` register and local variables using negative offsets relative to `bp`.

The difference between PLM and C is that both the PLM compiler and all iRMX system call subroutines assume that parameters are pushed onto the stack in a left-to-right sequence, whereas C compilers normally push parameters onto the stack in a right-to-left sequence. The C technique always puts the first parameter, which sometimes specifies the number of actual parameters there are for the call, at a fixed location in the stack next to the return address.

---

[5]For the S/360 architecture, register 14 contained the return address, register 1 contained the address of a parameter list in memory if the arguments could not be passed in registers, and register 13 was used as the link field in a linked list of register save areas for nested subroutine calls. There was no hardware support for pushdown stacks in memory, so stacks were not an issue. The same conventions were retained in the S/370 architecture.

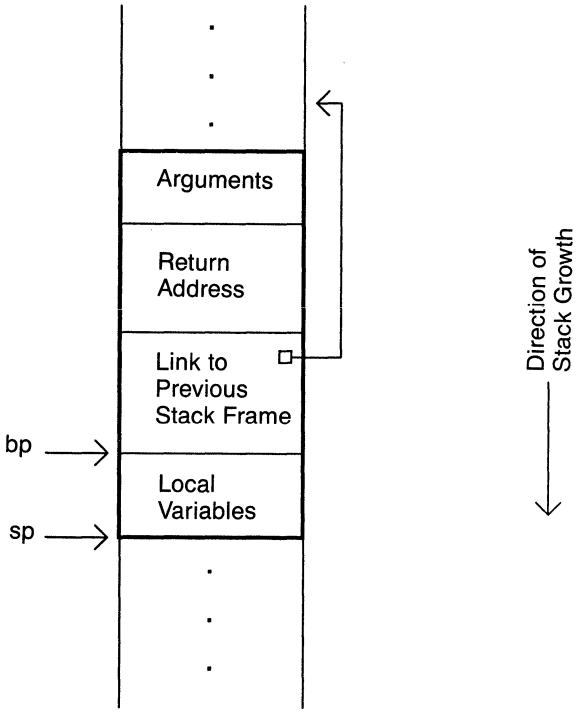


Figure 4.8 Structure of a stack frame during a subroutine call.

For example, the first parameter to *printf()* is a string that contains formatting codes; the number of formatting codes in the string tells the function how many additional parameters it should expect to find on the stack. The PLM technique makes it impossible to implement variable-length argument lists to subroutines, but executes more efficiently on the x86 architecture because the fixed number of arguments allows the subroutine to update the stack frame pointer, return to the calling program, and drop the incoming parameters from the stack in one or two machine instructions (`leave` *count*, or pop bp; `return` *count*). This method is opposed to the variable argument list technique, which requires the parameters to be dropped as a separate instruction by the calling program after the subroutine returns.

Intel C and PLM compilers can each use the other's calling convention. This feature allows the C run-time library to use the more efficient PLM model wherever possible, allows C programs to make iRMX system calls (which require the PLM argument passing model), and makes it possible to develop mixed-language applications as well. Intel C compilers recognize a pair of pragmas, `fixedparams` and `varparams` to identify which calling convention is to be used for individual functions. PLM compilers provide

the `interface` control for forcing routines to be called using the C convention.

Figure 4.9 shows a sample PLM program that calls routines from the C run-time library. Note that the program is coded as a procedure named *main( )*, which will be called from the C start-off code. This construction is necessary when linking to the non-shared version of the C run-time library because the start-off code must complete certain initializations before any function in the library is called. When using the shared version of the li-

Figure 4.9 PLM version of *hellormx* main program using non-shared C library functions.

```
/***>  hellormx.plm  <*********************************************
 *
 *      Sample PLM program for iRMX
 *      -- main program using non-shared C library functions
 *
 ************************************************************************/
$title ('Sample PLM Main Program')
$interface (C = printf)

hellormx:   DO;
DECLARE
      prompt (*) BYTE PUBLIC INITIAL ('Type something: ',0),
      reply (81) BYTE;

$if WORD16
DECLARE WORD_16 LITERALLY 'WORD';
$else
DECLARE WORD_16 LITERALLY 'HWORD';
$endif


printf: PROCEDURE (ptr) EXTERNAL;
DECLARE ptr     POINTER;
END printf;

dosub:  PROCEDURE (response$ptr, response$max) EXTERNAL;
DECLARE
        response$ptr    POINTER,
        response$max    WORD_16;
END dosub;

/*
 *  Execution Starts Here
 */
main: PROCEDURE (argc, argv) WORD_16 PUBLIC;
DECLARE
        argc            WORD_16,
        argv            POINTER;

  CALL dosub (@reply, size (reply) - 1);

  CALL printf (@('You typed: %s', 0Dh, 0Ah 0), @reply);
  RETURN 0;
END main;
END hellormx;
```

brary, this initialization is automatically performed when the first call to any function in the run-time library is called, and the equivalent PLM program can be coded as shown in Figure 4.10. Either version of the main program, using the appropriate version of the run-time library, could be linked to a single version of the subroutine, which is shown in Figure 4.11.

Figure 4.12 provides the *bnd386* command that will build the program under iRMX. This command involves linking the same files as any C pro-

Figure 4.10 PLM version of *hellormx* main program using shared C library functions.

```
/***>  hellormx.plm  <***********************************************
 *
 *      Sample PLM program for iRMX
 *      -- main program using shared C library functions
 *
 ************************************************************************/
$title ('Sample PLM Main Program')
$interface (C = printf, exit)

hellormx:   DO;
DECLARE
        promptStr (*)    BYTE INITIAL ('Type something: ',0),
        prompt           POINTER PUBLIC INITIAL (@promptStr),
        reply (81)       BYTE;

$if WORD16
DECLARE WORD_16 LITERALLY 'WORD';
$else
DECLARE WORD_16 LITERALLY 'HWORD';
$endif


printf: PROCEDURE (ptr) EXTERNAL;
DECLARE ptr     POINTER;
END printf;

exit: PROCEDURE (code) EXTERNAL;
DECLARE code    WORD_16;
END exit;

dosub:  PROCEDURE (response$ptr, response$max) EXTERNAL;
DECLARE
        response$ptr    POINTER,
        response$max    WORD_16;
END dosub;

/*
 *  Execution Starts Here
 */

  CALL dosub (@reply, size (reply) - 1);

  CALL printf (@('You typed: %s', 0Dh, 0Ah, 0), @reply);
  CALL exit (0);

END hellormx;
```

Figure 4.11 PLM version of *hellormx* subroutine using C library functions.

```
/***>  hellosub.plm  <*********************************************
 *
 *      Sample PLM program for iRMX
 *      -- subroutine using C library functions
 *
 ************************************************************************/
$title ('Sample PLM Subroutine')
$interface (C = printf, gets)

hellosub:   DO;

$if WORD16
DECLARE WORD_16 LITERALLY 'WORD';
$else
DECLARE WORD_16 LITERALLY 'HWORD';
$endif


printf: PROCEDURE (ptr) EXTERNAL;
DECLARE ptr     POINTER;
END printf;

gets: PROCEDURE (ptr) EXTERNAL;
DECLARE ptr     POINTER;
END gets;

DECLARE prompt  POINTER EXTERNAL;

dosub: PROCEDURE (resp$ptr, resp$max) PUBLIC;
DECLARE
        resp$ptr    POINTER,
        resp$max    WORD_16;

  CALL printf (@('%s', 0), prompt);
  CALL gets (resp$ptr);
  RETURN;

END dosub;
END hellosub;
```

Figure 4.12 The command to bind the PLM version of *hellormx* using the non-shared C library.

```
iRMX> /rmx386/new/intel/lib/cstrmx3c.obj, hellormx.obj, hellosub.obj, &
** /rmx386/new/intel/lib/crmxnf3c.lib, /rmx386/lib/rmxifc32.lib &
** rn (code32 to code) ss(stack(8182)) oj (hellormx) rc(dm(0,0FFFFFFh))
```

gram. The resulting loadable file is approximately 81 KB. Using the *bnd386* command shown in Figure 4.13 to link with the interface procedures for the shared library (and the object module for Figure 4.10) reduces the size of the load module file to approximately 6 KB. Table 4.1 compares the actual sizes for these two versions of the program in terms of files and actual memory usage.

Finally, note that PLM and C programs can build compatible stack frames with the codes shown in Figures 4.14 and 4.15. Figure 4.14 is a C

Figure 4.13 The command to bind the PLM version of *hellormx* using the shared C library.

```
iRMX> hellormx.obj, hellosub.obj, &
** /intel/lib/clibxf32.lib, /rmx386/lib/rmxifc32.lib &
** rn (code32 to code) ss(stack(8182)) oj (hellormx) rc(dm(0,0FFFFFFh))
```

Figure 4.14 C version of *hellormx* main program that can call the PLM version of the subroutine given in Fig. 4.11.

```
/***>  hellormx.c  <**************************************************
 *
 *      Sample C program for iRMX
 *      -- main program using C library functions
 *      -- calls subroutine using PLM calling conventions
 *
 ************************************************************************/

#include <stdio.h>

unsigned char *prompt = (unsigned char *) "Type something: ";


#pragma noalign (dosub)
void dosub (unsigned char *, unsigned short);


int
main (int argc, char *argv[]) {

unsigned char   reply[80];

  dosub (&reply[0], 80);
  printf ("You typed: %s\n", &reply);
  return 0;
  }
```

Figure 4.15 C version of *hellormx* subroutine that can be called from the PLM main program given in Fig. 4.10.

```
/***>  hellosub.c  <**************************************************
 *
 *      Sample C program for iRMX
 *      -- subroutine using C library functions
 *      -- subroutine is called using PLM calling conventions
 *
 ************************************************************************/

#include <stdio.h>

extern unsigned char *prompt;

#pragma noalign (dosub)
void
dosub (unsigned char * inbuff, unsigned short limit) {
  printf ("%s", prompt);
  gets ((char *) inbuff);
  return;
  }
```

main program that can call a PLM version of *dosub( )*. The `fixedparams` pragma tells the compiler to use the PLM calling convention for that routine. The subroutine in Figure 4.15 can be called from a PLM program, such as the one in Figure 4.10 or from a matching C main program, such as the one in Figure 4.14.

4.4.3 Pointers

Memory address pointers can take four different forms in iRMX environments. Depending on the memory segmentation model, pointers are classified as *near* or *far*. Depending on the architecture of the processor running the application and the OS, pointers reference 16- or 32-bit segments. Thus, the four types of pointers are 16-bit near, 16-bit far, 32-bit near, and 32-bit far. Near pointers are used when the item being addressed is in one of the memory segments directly accessible by the CPU, and consist of a 16-bit or 32-bit offset value for identifying a location in the segment. Far pointers are used when an item being addressed is not in one of the memory segments directly addressed by the CPU. They consist of a 16-bit segment identifier, plus a 16-bit or 32-bit offset. Pointers and memory addressing in general are discussed further in chapter 5.

Both PLM and C support parameter passing by value only, but both languages accomplish the equivalent of reference parameters by passing pointers as values. Parameters passed by value cannot be modified by a subroutine because only a copy of the argument is actually available to the subroutine. For C and PLM, a copy of the parameter value is pushed onto the stack in this instance. Passing a pointer, on the other hand, enables the subroutine to modify variables in the calling program's memory by indirect addressing. The PLM language supports indirect addressing through its based-variables mechanism. Indirect addressing is accomplished in C by declaring a variable to be a pointer, either by using the asterisk declarator or by declaring the variable to be array.

Although it is not an iRMX issue, you should note that a major difference between pointers in C and PLM is the type checking performed by the compiler. All PLM pointers are generic in the sense that the language includes no mechanism for specifying the data type of the memory location addressed by a pointer. The C language, however, requires the data types of the variables addressed by pointers to be declared explicitly, and the compiler performs full type checking for all uses of pointers.

The distinction between 16- and 32-bit pointers is not as significant when making iRMX system calls as one might expect. All programs that run on iRMX I and iRMX II must use 16-bit code just to run, so there is no question about 32-bit pointers for those operating systems. For iRMX III, we have already noted that the OS can accept either 16- or 32-bit pointers because it can determine the word size of the stack segment at run time (the interface procedure pushes a flag for 32-bit stacks). As long as a program

uses all 16-bit code or all 32-bit code, PLM386 and iC386 will cause all segments, including the stack, to use the same word size, and pointers will automatically match that chosen word size.

The rule for passing near or far pointers to iRMX system calls is simple: all pointers must be far pointers because the operating system occupies separate memory segments from applications. For PLM programs, simply use that language's @ operator, which always generates far pointers. In certain circumstances, the PLM compiler will issue a warning message if you create a far pointer to information normally accessed through a near pointer, such as a pointer to a subroutine when using the compact memory segmentation model. This warning can be eliminated by using an extended segmentation model to force the compiler to use the large model for accessing particular subprograms. The *floatest* application in Figure 4.5 illustrated the use of this extended segmentation technique for the *mqcbin_declow()* and *mqcdec_bin()* functions.

For C, the situation is somewhat more complicated. The & operator will generate either a near or far pointer, depending on the memory segmentation model being used and whether the referenced item is in the program's code or data, as summarized in this table:

Segmentation Model	Code Reference	Data Reference
Compact	near	far
Large	far	far

Furthermore, the C compiler will generate a pointer without use of the & operator when arrays or functions are passed as arguments to functions. For example, the C compiler generates exactly the same code for the following two function calls without issuing any warning or error messages:

```
char    a_string[20];
foo (a_string);
foo (&a_string);
```

In summary, C generally produces far pointers, which is what is required for passing pointers to iRMX system calls. The one exception is a pointer to code when using the compact segmentation model. For this situation, use the compiler's `far` type qualifier when declaring a function, use a `far` cast on the pointer to the function, or use extended segmentation to make the function far. The far `type` qualifier is only recognized by the compiler when the `extend` compiler control is used.

PLM-like extended segmentation for C programs uses the exact same syntax as for PLM programs, but the segmentation definition must be in a separate file from the source module, and that file must be named using the `subsys` compiler control. The following is an example of two files that illustrate all three techniques. Note that this example is triply redundant; only one of the three techniques is needed:

File `foo.sys`:

```
$compact (exports bar)
```

File `foo.c`:

```
#pragma subsys (foo.sys)
#pragma extend

void far bar(void);
main( ) {
 foo ( (far *) bar);
 }
```

Because the `subsys` and `extend` compiler controls are specified with pragmas in the source module file (`foo.c`), and the segmentation model is specified in the subsystem declaration file (`foo.sys`), this example could be compiled with the simple command line:

```
iRMX> ic386 foo.c                                    [4]
```

Because of the triple redundancy in declaring *bar( )* to be a `far` procedure, this example would be unchanged if the far keyword were omitted from the function prototype for *bar( )*, if the (`far *`) cast were omitted from the call to *foo( )*, or if the `subsys` pragma were omitted entirely. The `subsys` pragma is particularly useful for mixed-language applications, which can maintain a single set of subsystem definitions used by all parts of the application, regardless of language.

4.4.4 I/O connections for C programs

C programs have three choices for identifying I/O channels: file descriptors, streams, and iRMX connections. The first technique uses small integers to identify different channels: the value 0 is used for the standard input channel, 1 for the standard output channel, and 2 for the standard error channel. File descriptors historically have been used for Unix and DOS programs, but should be avoided for new applications because they are not part of ANSI C, and thus, are not portable. Streams, which are defined as part of ANSI C, are referenced by a variable declared as a pointer to the data type FILE, which is a typedef for a data structure declared in `stdio.h`.

The iRMX C run-time library provides a function called *fdopen( )* for converting file descriptors to stream file pointers and a function *fileno( )* for converting file pointers to `int`s that can be used as file descriptors. The C

run-time library internally translates both file descriptors and file pointers into iRMX objects called *connections* so that the library can implement I/O operations using the iRMX I/O system. iRMX connections are identified to the OS by variables of type TOKEN, which is a typedef declared in the header file `rmxc.h`. This header file contains all typedefs and function prototypes needed by any C program that makes iRMX system calls. The functions *_get_rmx_conn()* and *_put_rmx_conn()*, which are also prototyped in `rmxc.h`, can be used to convert C file descriptors to iRMX connections and vice versa. The use of iRMX connections for I/O is discussed in chapter 8.

4.4.5 Multitasking and multiple jobs for C programs

Even though this section pertains only to C programs, it might also be of interest to programmers familiar with Unix systems programming because a rough correspondence exists between iRMX jobs and Unix processes and between iRMX tasks and POSIX.4 threads. (iRMX jobs and tasks are described more thoroughly in chapter 6. POSIX.4 is the IEEE standard for real-time Unix introduced in chapter 1.)

A Unix process has a defined part of memory that it can access (its "address space") and a single thread of execution. A Unix thread is a thread of execution that shares an address space with a process. Both processes and threads inherit all open I/O connections from their parent process, but can create their own private I/O connections as well. A major distinction between iRMX and Unix is that iRMX jobs do not inherit any I/O connections from their parents, although they can create their own connections based on those of their parent or any other job. For example, when the HI creates a job to run a command, the HI provides the job with connections corresponding to `stdin`, `stdout`, and `stderr` (connections with iRMX logical names `:ci:`, `:co:`, and `:term:`, respectively). Since these connections belong to the job that created the command rather than to the command itself, the command must create its own connections based on these connections before using them. The creation of I/O connections for `:ci:` and `:co:` is handled automatically by the *rqcsendcoresponse()* system calls in our sample programs, and the process is demonstrated explicitly when we examine I/O programming in detail in chapter 8. Although I/O connections cannot be shared across jobs, all tasks within an iRMX job can share I/O connections.

The iRMX C run-time library follows the POSIX.1 model of providing separate environments for jobs and tasks created by C programs. The shared version of the C library sets up this environment the first time a task makes a call to one of the library functions. The non-shared version of the library intercepts any iRMX system calls that create or delete iRMX

jobs and tasks so that it can cause the start-up code within the library to be executed before passing control to the actual code for the job or task specified by the application program. This start-up code is a subset of the start-off code that is executed for all C programs running as iRMX HI commands. For most C programs, this behavior is invisible and provides a very clean way to implement applications with multiple jobs or tasks in C, but is less efficient than the shared library implementation.

C applications not run as HI commands must be aware of this behavior on the part of the nonshared run-time library, however. Such applications include those resident applications configured into the OS using the ICU described in chapter 9. Such applications must be bound with a version of the `cstrmx.obj` file, which has been modified to eliminate any calls to routines that initialize command line parsing (there is no command line to parse) and to set up `stdin`, `stdout`, and `stderr` (there is no default terminal associated with such applications).

Furthermore, such applications cannot have their system calls for creating jobs and tasks intercepted by the C run-time library, or those same initialization routines will be called and will then fail for the reasons indicated. One approach to solve this problem is to use the non-shared C run-time library and do an incremental bind, first binding the application with the iRMX interface library (`/rmx386/lib/rmxifc32.lib`) so that system calls that create and delete jobs and tasks will call the OS directly without going through the C run-time library. The resulting linkable module can then be bound with the C run-time library to link in other routines from that library that the application might call. Such applications must not call run-time routines that do I/O using `stdin`, `stdout`, or `stderr`, however, because these I/O streams cannot be set up for resident applications.

The assembly language source code for generating `cstrmx.obj` is provided with the compiler in a file named `/lib/ic386/cstart.asm`. The release notes provided with the compiler give instructions for reassembling this file to produce a version of the start-up code tailored to the needs of resident applications.

The situation is simpler for resident iRMX III and iRMX for Windows applications that use the shared run-time library. These programs are loaded by *sysload* rather than as part of the OS itself. They still do not have access to a user's login terminal, but they can do command-line argument processing. Since the shared run-time library initializes access to the `stdin`, `stdout`, and `stdout` I/O connections when first referenced rather than from within the start-off code, there is no need to create a special version of `cstart.obj` for these applications. Furthermore, the shared run-time library does not intercept the iRMX system calls to create and delete iRMX tasks and jobs, so there is no need to perform an incremental bind of applications that make these calls.

4.5 Debugging

This section considers two ways in which the choice of language affects the debugging process: exception handling and compatibility with the SoftScope debugger.

4.5.1 Exception handling

System call condition codes and the exception handler procedure are covered in detail in chapter 6. Basically, abnormal conditions detected by the OS during system calls either cause a procedure called the *exception handler* to be called or cause a nonzero condition code value to be returned to a variable passed to the system call as a reference parameter. The sample programs in this book generally use a 16-bit unsigned integer variable named `Status` to receive this condition code value. Every iRMX task has its own exception handler procedure and sets its own mode for handling exceptions, either by testing the condition code variable after each system call (referred to as *in-line handling*), or by having the exception handler procedure called automatically when exceptions occur. A default exception handler and mode setup exists for each iRMX job, which is used for each task within the job. Two system calls are available for determining and changing the current mode and procedure for individual tasks as they execute.

Most iRMX systems are configured with either the System Debugger (SDB) or the HI's exception handler as the default handler, and the exception handling mode is set to call the handler whenever exceptions occur. This behavior is what C and PLM programs typically encounter when they run. For exceptions that occur during functions provided by the C run-time library, the iRMX condition code is mapped into a numeric value for the `errno` variable provided for each task. Portable C programs can test for standard error conditions by comparing `errno` to the symbolic constants defined in `<errno.h>`.

4.5.2 SoftScope debugging

Under iRMX I and II, SoftScope sets the default exception handler for a job to a routine that it supplies. Although applications can still use system calls to change the handler for any task, the default behavior for both C and PLM programs is for iRMX applications to break to SoftScope whenever an exception occurs. Version 1.0 of SoftScope III operates in the opposite manner: all exceptions must be handled in-line, regardless of whether the program is coded in PLM or C. A later version of SoftScope III is supposed to change this behavior to match the iRMX I and II versions.

SoftScope III operates differently from SoftScope for iRMX I and II for command-line processing as well. Under iRMX I and II, all programs that process command line arguments work properly even though the actual

command line given to start the debugging session has the *sscope* command name as the first part of the command line. A C program that refers to the string at `argv[0]`, for example, will find the command's name, not SoftScope's. For SoftScope III, C programs have their command lines parsed the same way (properly) whether they are run under SoftScope or directly as HI commands. Version 1 of SoftScope III does not support HI command line parsing at all. In fact, Version 1 of SoftScope III does not support several critical system calls to the HI layer of iRMX, such as *rqcsendcoresponse()*, which was used in the sample programs in chapter 3 and this chapter. To support these system calls would require SoftScope III to implement some of the functionality of the HI itself. One way to resolve this issue is to add system calls to the HI that SoftScope itself could call to perform the necessary functions. This approach is superior to the alternative of having SoftScope implement these functions itself, because changes in the internal logic of the HI in future versions of the OS would not require parallel changes in SoftScope.

Chapter

5

The Intel x86 Architecture

5.1 Overview

Ideally, it would be possible to develop systems programs and real-time applications using a high-level language such as C or PLM without considering the computer architecture. A computer's architecture consists of those features of the processor's design visible to the machine- or assembly-language programmer.

High-level language programmers should not have to be concerned with architectural issues such as CPU registers, memory addressing and protection, or interrupt handling. Real-time and systems programmers do not live in an ideal world, however, and an understanding of the architecture of the processor used is necessary to producing efficient applications or even functional code. In chapter 4, for example, we already considered the various formats that memory pointers can take.

This chapter presents those microprocessor architectural features relevant to iRMX applications development. This discussion is not a complete survey of the Intel x86 architecture, and it does not claim to include enough information to program in assembly language. The chapter should, however, provide the necessary background for understanding some key concepts of the iRMX operating system, which has been specifically designed to take advantage of x86 architectural features to provide reliable and efficient real-time performance.

5.1.1 CPU Registers

The architecture of a processor is its appearance to the machine- or assembly-language programmer. Thus, it includes the processor's instruction set, data formats, addressing mechanism, interrupt mechanism, I/O mechanism, and register set. The goal of this chapter is to discuss those parts of the architecture that a systems programmer, working in a high-level sys-

tems implementation language such as C or PLM, must work with, but not those parts of the architecture specific to assembly language programming. Very little about the processor's instruction set is discussed, for example, except the names of a few instructions. High-level language programmers do not need to work with processor registers either, but, some register names are mentioned. It is worthwhile to show the entire set of registers to provide context for the ones discussed.

Figure 5.1 shows the processor registers visible to the machine- or assembly-language programmer for 80386 and 80486 microprocessors.[1] All registers, except the segment registers, include subregisters that can be manipulated independently. For example, Register `eax` is a 32-bit register, but bits 0 through 15 can be modified by referencing Register `ax` without affecting bits 16 through 31 of Register `eax`. Furthermore, bits 8 through 15 of `eax` can be manipulated as an independent register by referencing Register `ah`.

The registers shown in Figure 5.1 are the same as the 80286 registers and earlier processor registers, with the following exceptions:

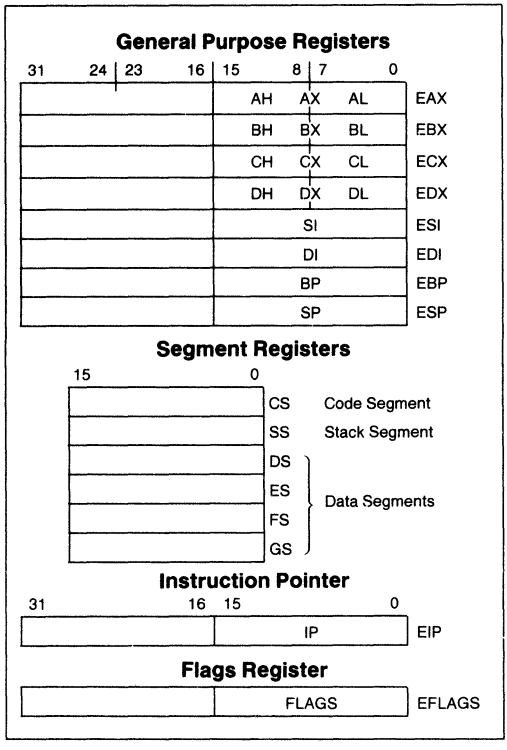


Figure 5.1 Processor registers for the 80386 microprocessor. (From the i486™ Microprocessor Handbook, Intel order number 240440-001. Reprinted by permission of Intel Corporation, © Intel Corp. 1989.)

[1]Various Intel microprocessors also provide additional registers: CR*n* for control functions, DR*n* for debug functions, and TR*n* for test functions.

- All registers on the 80286 and earlier processors are 16 bits. The 32-bit registers, such as `eax`, `eip`, *etc.*, do not exist on those processors, only their 16-bit parts, `ax`, `ip`, *etc.*
- The `fs` and `gs` segment registers do not exist on the 80286 and earlier processor.

5.2 Memory Segmentation

The x86 microprocessors all provide access to memory with a segmented architecture. The actual memory attached to an x86 processor consists of a linear array of bytes, which are accessed by the microprocessor for reading and writing using a physical memory address.[2] Segmentation allows the programmer to generate controlled references to this linear array that can be checked for basic correctness by the processor. Segmentation, however, does not affect the basic linear nature of the memory system connected to the processor.

An anology might be to think of physical memory as a large piece of graph paper with each box, representing one byte, containing one value. The bytes are accessed by specifying their position (linear address) on the graph paper. Segments would be like boundary lines drawn on a clear piece of plastic set on top of the graph paper. Programs specify memory addresses for instructions and data relative to these segment boundary lines, and the processor transparently transforms these segmented addresses into linear addresses for accessing the actual memory location.

Each memory segment is identified to the CPU by a 16-bit quantity called a *selector*. At the least, a selector is a value used by the CPU to compute the starting address of a segment in the physical RAM attached to the processor, and some x86 processors use the selector to determine other segment characteristics as well. The CPU can access information in a segment only if that segment's selector has been loaded into one of the CPU registers specifically intended to hold selectors. The segments that have their selectors loaded into one of the CPU selector registers at any particular moment are called the *currently accessible segments*, or just the *current segments*. All members of the x86 family provide segment registers for addressing one code, one stack, and one data segment at a time. These registers are named `cs`, `ss`, and `ds`, respectively. The data segment selected by the `ds` register is the default data segment, and various x86 microprocessors provide additional segment registers for accessing extra data segments, with names like `es`, `fs`, and `gs`. Any time the processor must access a code, stack, or default data segment different from the current data seg-

[2]For now, the terms *physical memory address* and *linear address* are used interchangeably. There is a difference between the two when paging is used, which is discussed later in this chapter.

ments, the processor must load a new selector into the corresponding CPU segment register. The processor can access extra data segments without reloading `ds`, however, by using a selector from `es`, `fs`, or `gs`. Loading segment registers requires time and should, therefore, be avoided whenever possible.

Figure 5.2 shows how an x86 processor accesses physical memory. The first step is to develop a logical address that consists of a selector register and an effective address. The various components of an effective address are not always used. For example, a far pointer is a logical address that consists of just a selector and a displacement, with no base register or index. The segmentation unit combines the two parts of a logical address to produce a linear address, which is transformed into a physical memory address by the paging unit. If the paging unit is not used or is not present (for instance, on processors earlier than the 80386), the linear address is the same as the physical address. On 80386 processors and later, the effective address, the linear address, and the physical address are all 32 bits wide. A separate data path exists for transferring information between the memory location specified by a physical address and the registers inside the processor, as shown in the lower part of Figure 5.2.

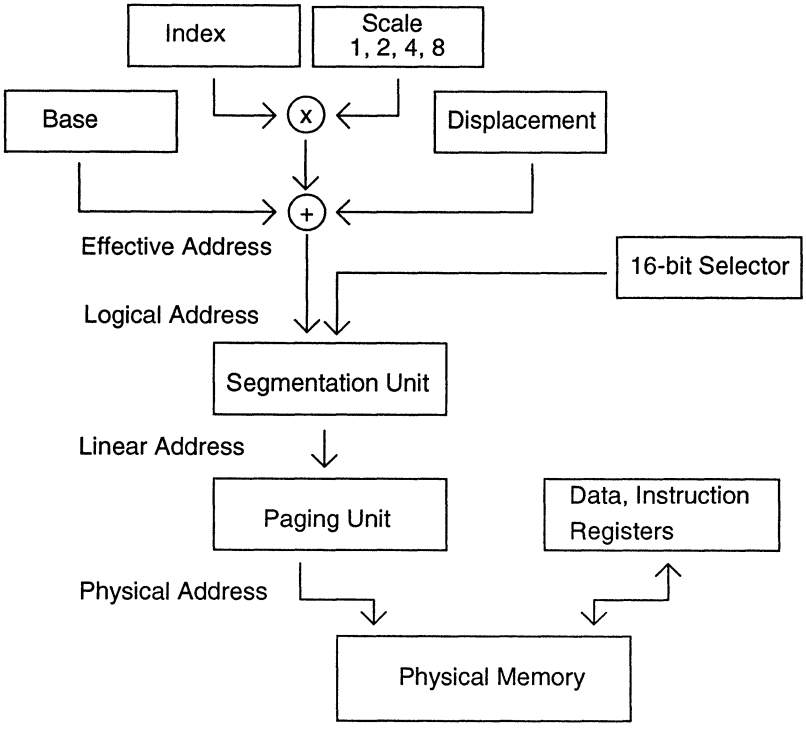


Figure 5.2 Protected-mode address calculation.

A linear address is the sum of a segment base address and an effective address. The segmentation unit computes the segment base address from the segment selector in one of two ways, depending on the operating mode of the processor. In real mode, which is supported by all x86 processors, the segment base address is 16 multiplied by the value of the selector. That is, it is computed by shifting the selector 4 bits to the left and filling in on the right with zeros. Thus, the real-mode base addresses are constrained to be 16 + 4 = 20 bits wide, which places a limit of 2^{20} = 1 MB of memory that can be accessed in real mode.[3]

In protected mode, which is supported by 80286 and later processors, the base address of a segment is taken from a data structure called a segment descriptor, and the selector is used to select the appropriate descriptor from a table of descriptors. (Hence the name *selector*.) For the 80286 microprocessor operating in protected mode, segment descriptors provide 24-bit base addresses, allowing the CPU to address up to 16 MB of memory. Descriptors for 80386 and later microprocessors can contain 32-bit base addresses, allowing access to as much as 4 gigabytes (GB) of memory when operating in protected mode. A descriptor must be loaded into the CPU every time a segment register is changed in protected mode, which is why the penalty for changing segments is so great in iRMX II. The problem is not as severe for iRMX III applications, simply because the segment registers do not need to be changed as often (discussed later in the chapter).

The maximum effective address (offset) that can be added to a segment's base address is 16 bits wide, except for the 80386 or later microprocessors. The 80386 and later microprocessors operating in protected mode might have a particular bit of the current code segment descriptor (called the `D` bit) set to 1, in which case the offsets are 32 bits wide.[4] Thus, the maximum size of a memory segment is 64 KB for all x86 processors before the 80386, and 4 GB for the 80386 microprocessor and after. iRMX I and II applications always have 64K segments, and iRMX III applications can have either 64K or 4G segments, depending on the `D` bit setting, which will be set to 1 if compiled with the PLM-386 or iC-386 compiler and bound with *bnd386*.

The x86 architecture allows segments to overlap each other in memory. For example, for the small memory segmentation model, the `cs`, `ss`, and `ds` registers are all loaded with the same selector, constraining such programs to work with a total of 64 KB of memory for code, data, and stack in real and 80286 protected modes. A special case of the small memory segmentation

---

[3]The actual limit is 0x0FFFF0 plus the maximum value of an offset, 0x0FFFF (i.e., 0x010FFEF = 1M + 64 KB). However, on processors below the 80286, addresses above 1MB (0x0100000) wrap around to low memory for a limit of 1M. On the 80286 and later, the additional 64 KB above 1M is accessible in real mode and the amount of memory available is 1M + 64 KB. The 64 KB above 1M is called the *High Memory Area* (HMA).

[4]D stands for Default Operation Size; 1 = 32 bits, 0 = 16 bits.

model is the flat model. For flat memory addressing, all segment registers of an 80386 or later operating in protected mode are loaded with a single selector for a 4-GB segment that starts at physical memory location 0. In this case, the entire notion of memory segmentation essentially disappears because any location throughout physical memory can be accessed using any segment register and an offset value equal to the desired physical memory address. The problem with overlapping segments is that they circumvent the hardware's memory protection mechanism by allowing multiple methods of accessing the same memory location, possibly using different access rights. This override feature is essential for such situations as loading code into memory for execution (the code must be treated as data while being loaded), but the override feature can cause error conditions to go undetected. The 80386 microprocessor's paging mechanism offers an alternate way to provide memory protection when segmentation is not used.

The iRMX operating system maintains separate, non-overlapping segments for an application's code, data, and stack segments. That is, iRMX supports only the compact and large models and does not support the small or flat segmentation models. This approach to using memory allows iRMX to take full advantage of the memory protection facilities provided by the segmentation hardware of protected-mode processors, and real-mode applications are automatically upwardly compatible with protected-mode versions. The main advantage of operating in protected mode for real-time systems is that protected mode automatically detects coding errors early in the development process, leading to more robust systems.

To summarize, all memory is accessed using the segmentation mechanisms of the x86 architecture under the iRMX operating system. Various versions of the operating system operate with different segment implementation as follows:

The iRMX I operating system operates in real mode. A maximum of 1 MB of addressable memory exists and segments can be up to 64 KB. Addresses within the current set of addressable segments can be referenced with near pointers consisting of 16-bit offset values, and addresses outside the current set of addressable segments are referenced with far pointers that consist of a 16-bit offset value and a 16-bit segment base address expanded to 20 bits by multiplying by 16.

The iRMX II operating system operates in protected mode. A maximum of 16 MB of addressable memory exists, and segments can be up to 64 KB. Near pointers consist of 16-bit offsets, and far pointers consist of a 16-bit offset and a 16-bit selector that identifies a descriptor containing a 24-bit segment base address.

The iRMX III operating system and iRMX for Windows operates in either 16-bit or 32-bit protected mode. In the latter case, there is a maximum of 4 GB of addressable memory, segments can be up to 4 GB, near pointers consist of 32-bit offsets, and far pointers consist of a 32-bit offset and a 16-bit selector that identifies a descriptor containing a 32-bit segment base address.

From a programming perspective, near pointers are always 16 bits, and far pointers are always 32 bits for iRMX I and II as well as for iRMX III code that operates in 16-bit mode. Near pointers are 32 bits, and far pointers are 48 bits (16-bit selector plus 32-bit offset) for iRMX III code that operates in 32-bit mode.

5.2.1 iRMX segmentation rationale

It may seem contradictory that an operating system developed by Intel to exploit the x86 architecture would support only the compact and large models. Indeed, this chapter introduces additional architectural features of the x86 architecture not used by the iRMX operating system, so the issue is not limited to memory segmentation.

The explanation for this apparent anomaly lies in the design goals for a real-time operating system: deterministic behavior, robustness, and efficiency. For memory segmentation models, the small and flat models are not robust in the sense that overlapping segments defeat the ability of the hardware to detect memory-protection violations. In this case, either an application must check for such violations in software, an undesirable performance penalty, or applications are subject to failure due to unchecked invalid memory access errors, an undesirable compromise to a system's robustness.

Another explanation hinges on what it means to say that the operating system supports (or does not support) a particular segmentation model. The issue is simply a matter of what types of pointers (near or far) must be used for pointer arguments to system calls and what types of machine language call and return instructions are used for system calls. All iRMX system calls require far pointers for arguments, regardless of the segmentation model being used by the code, but both near and far calls and returns are supported for system calls. Programs can be compiled using any segmentation model, provided only that the programs adhere to these requirements when making system calls. Naturally, the development tools used to build iRMX applications, the compiler and binder, must support the segmentation model used by the operating system. A fuller understanding of the issues involved here is developed in chapter 6, where the iRMX system call mechanism is presented.

5.2.2 Procedure calls and stack segments

Procedures, functions, subprograms, and *subroutines* are all terms used to refer to code invoked by a machine language *call* instruction, and which can resume execution at the next instruction after the *call* by executing a machine language *return* instruction. The semantic differences among the four terms listed above, if any, are imposed by high-level programming languages and concern such matters as whether the procedure returns a value to the calling program or not. In this section, the architectural support provided by the x86 architecture for calling and returning from procedures is presented without regard to high-level language constructs. The material in this section is useful both for developing the iRMX system call mechanism in Chapter 6 and for understanding the rationale behind the x86 segmentation mechanism.

The processor uses a register called the *Instruction Pointer* (`ip`) to hold the offset into the current code segment of the next instruction to be executed.[5] As one instruction is fetched, the processor automatically increments `ip` by the length of the instruction so that it always contains the address of the next instruction. The common way to represent the selector and offset parts of an x86 pointer is to separate them by a colon; `cs:ip` refers to the address formed from the selector in the `cs` register and the offset in the `ip` register. A machine language *call* instruction must preserve the value of `cs:ip` when the call is made, and the CPU restores this preserved value when a procedure executes a *return* instruction so that the processor can execute the instruction following the *call.*

The x86 architecture uses a memory stack segment for holding preserved values of `cs:ip` as well as for passing parameters to procedures and allocating memory to local variables defined only within the scope of a procedure. The `ss` register is used to hold the selector for the stack segment, and two more registers, the `sp` and `bp`, are used to mark distinguished positions within the stack segment. The `sp` register is called the Stack Pointer, and always contains the offset of the current top of the pushdown stack maintained in the stack segment. Initially, the pushdown stack is empty, and `sp` contains the offset of the end of the stack segment. As information is pushed onto the pushdown stack, `sp` is used as the offset for storing the information in the stack segment, and is then decremented to point to the new top of the stack. If `sp` ever reaches zero, it means that the pushdown stack has overflowed its stack segment, and the processor raises a protection fault (interrupt level 12).

The processor automatically uses `ss:sp` as the address for storing `cs:ip` in memory whenever a *call* instruction is executed, and again when

[5]For 80386 microprocessors and above, the corresponding 32-bit register is called `eip`. The naming convention of prefixing 32-bit register names with `e` holds for all the registers mentioned in this chapter.

the processor restores `cs:ip` during execution of a *return* instruction. By using a pushdown stack for the `cs:ip` values, the processor automatically accommodates nested subroutine calls and returns.

Of course, both `cs` and `ip` do not need to be saved and restored if the procedure called is in the current code segment, as is the case for the compact memory model of compilation. In this case, only the `ip` register must be preserved and restored, which reduces the time for protected mode calls and returns considerably (see the next section for more information). Two different machine language *call* instructions and corresponding *return* instructions must be used, depending on whether the full `cs:ip` value or just the `ip` value is to be pushed onto the stack and restored. These instruction pairs are referred to as *far call* and *far return* for far calls or *near call* and *near return*, for near calls. Compilers for PLM and C generate the proper *call* and *return* instructions automatically, provided that the segmentation models and function prototypes used in the modules to be linked together are declared consistently.

Data operands. A procedure can access data operands from one of four locations in memory:

- Static global data
- Local data
- Procedure parameters
- Pointers

Static global data are variables and constants located in the current data segment. For PLM and C, these data are declared outside of the body of any procedure or function, which is sometimes referred to as *the module level.*

Local data are variables and constants that exist when a procedure or function is called and cease to exist when the procedure returns. The data can be either static or dynamic.

Static local data occupies memory permanently allocated from the current data segment. The data cannot be referenced by code outside the procedure that contains the data declarations because it is semantically incorrect for a compiler to generate code to do so.

For *dynamic local data*, storage is allocated on the stack when the procedure starts executing. This allocation occurs by decrementing the `sp` register by an amount equal to the total size of the local variables for the procedure. The `sp` register is incremented by an equal amount when the procedure returns, so the data can no longer be accessed by any code. In fact, its memory will be overwritten by the local variables of the next procedure called after the first procedure returns. The C language refers to this type of data as *auto storage* data.

In C, variables and constants declared inside a function are dynamic local data unless their declarations include the *static* modifier. For PLM,

variables declared inside a procedure are static unless the procedure is declared to be *reentrant*. In this instance, the C language is more flexible than PLM; C allows individual variables declared within a function to be either static or dynamic, but PLM requires local variables for a procedure to be either all static or all dynamic.

Arguments to a procedure are pushed onto the stack before the procedure is called, and *procedure parameters* can be accessed from within the procedure by accessing memory locations with the appropriate stack segment offsets. The `bp` register is used to facilitate access to these values, as described next.

Pointers, often passed as parameters to procedures, can be used to access data through indirect addressing. The selector part of the pointer is loaded into one of the extra segment registers, and the offset is used to complete the effective memory address. Near pointers consist only of the offset part of a logical address; the selector is automatically taken from the current value of the `ds` register. Pointers can be used to access any of the other three types of data, or even to reference memory areas not occupied by the program being executed.

Stack frames. With the aid of the x86 architecture, compilers construct a standard data structure in the stack segment for each procedure, called an *activation record* or *stack frame*. Figure 5.3 shows how a stack frame is developed as a procedure is called.

In Figure 5.3a, the `bp` register holds the offset of the calling procedure's stack frame, and the `sp` register holds the offset of the current top of the pushdown stack. (Don't let the term *top of stack* confuse you; it refers to the offset of the last item pushed onto the stack and takes on numerically lower values as the stack grows downward within the stack segment.)

When a compiler translates a procedure call (function reference), it generates a series of machine language *push* instructions to push the procedure's arguments onto the top of the stack. These arguments can be either copies of data values or pointers to data values. After these *push* instructions have executed, the situation looks like Figure 5.3b.

The next instruction the compiler generates is either a near *call* or a far *call*, and the stack looks like Figure 5.3c when the processor starts executing code inside the procedure. The return address in (c) is either just an `ip` value or a full `cs:ip` pointer, depending on the type of *call* instruction used.

The compiler generates machine instructions, called the *procedure prologue*, at the beginning of each procedure which completes the construction of the activation record for the procedure. For most x86 processors, the activation record can be done in a single machine instruction called *enter*, but early processors, such as the 8086 and 8088, require a few separate instructions. The procedure prologue performs three functions: (1) it pushes a

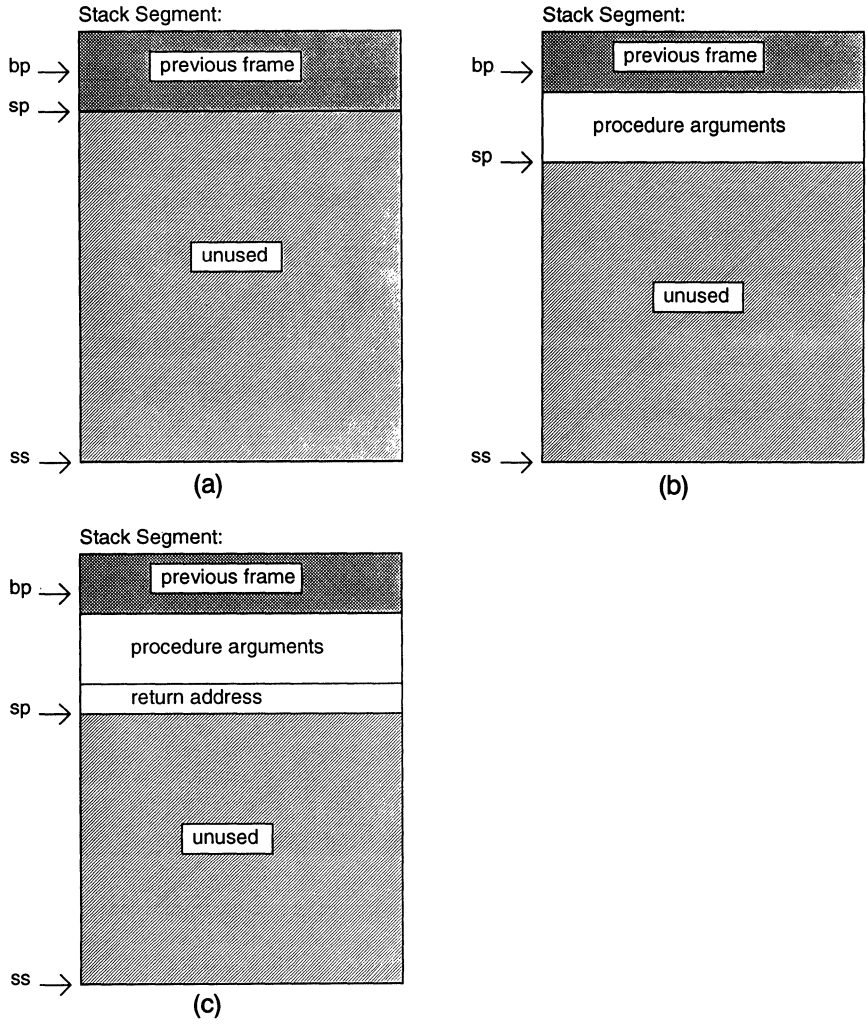
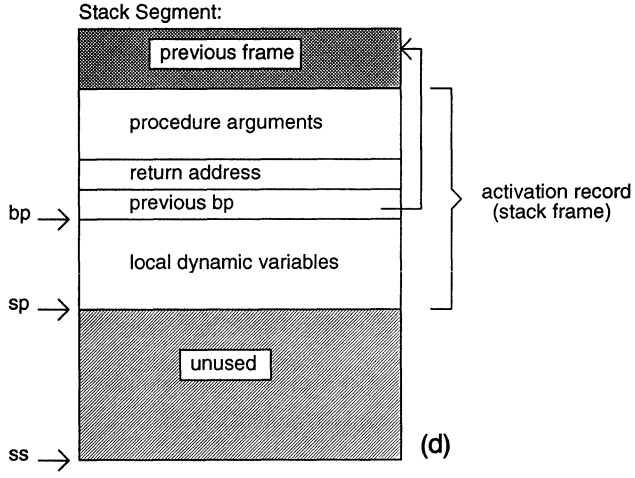


Figure 5.3 Development of an activation record (stack frame) for a procedure call. (a) The stack segment before the call. The selector in the ss register is used to derive the base address of the stack segment. The sp register contains the offset of the top of the pushdown stack, and the bp register contains the offset of a known position in the current procedure's activation record. (b) The situation just before the *call* instruction is executed. The arguments to be passed to the procedure have been pushed onto the stack, and the sp register has been decremented accordingly. (c) Just after the *call* instruction is executed, but before the procedure's prologue code. The return address has been pushed onto the stack, and sp has been decremented. (d) After the procedure's prologue code has been executed. The previous value of the bp register has been saved on the stack, and the bp register has been set to the offset of that stack position. The sp register has been decremented to allocate memory for the procedure's dynamic local variables. From within the procedure, parameters can now be accessed using positive offsets from bp. Local dynamic variables can be accessed using negative offsets from bp. When the procedure returns to the caller, the stack must be restored to the condition shown in (a).

Figure 5.3 (*Continued*)



copy of the bp register onto the top of the stack, (2) it copies the value in the sp register into the bp register, and (3) it decrements the sp register by the amount of storage needed for the procedure's local dynamic variables. The situation after the prologue code has executed looks like Fig. 5.3d.

With the activation record complete, a procedure can access the various parameters passed to it by adding various constants to the value in the bp register. The procedure can access its local variables using stack segment offsets computed by subtracting various constants from the value in the bp register.

To support nested procedure calls, a procedure's activation record must be removed from the stack when the procedure returns to the caller. To do this, the compiler generates code called the *procedure epilogue* that is executed before a procedure returns to a caller. The epilogue code increments the sp register to drop the local variables, pops the value now at the top of the stack into the bp register, thereby restoring the caller's stack frame pointer, and returns to the caller (using a near or far *return* instruction as appropriate), which increments the sp register to an offset just above the return address pushed by the caller.

To complete the removal of an activation record, the parameters that were passed to the procedure must also be dropped from the stack by incrementing the sp register by the appropriate amount. If a procedure is always called with the same number and types of arguments (such as C functions named in a `fixedparams` pragma and PLM procedures not listed in an `interface` control), the value to be added to sp can be specified as part of the *return* instruction so that the parameters are dropped as part of the execution of that instruction. If the compiler cannot tell how many bytes might be passed as arguments to a procedure (such as for normal C functions like *printf()*), the *return* instruction cannot drop the arguments, and

the calling program must include the code to do so as a separate instruction after the procedure returns.

Threads of execution. At this point something can be said about what information an operating system must maintain to support independently scheduled threads of execution, such as iRMX tasks or Unix processes. For each thread, a separate `cs:ip` value must exist, as well as a separate stack segment.

When one thread is to assume control of the processor from another, the address of the next instruction to be executed by the first thread (its `cs:ip`) must be saved and then restored when that thread is scheduled to use the processor again. We call this address the thread's *continuation address* because it tells where the thread is to continue its execution when allowed to use the CPU again. A pushdown stack cannot be used to hold continuation addresses unless the different threads of execution are always scheduled in a nested fashion, like procedure calls. Since this condition does not generally hold for multi-threaded operating systems, it is not possible to share a single pushdown stack for continuation addresses across threads. (iRMX task scheduling is discussed in chapter 6.) By similar reasoning, threads cannot share a pushdown stack for their activation records either. Each thread of execution must have its own `cs:ip` and its own stack segment.

5.2.3 Memory protection

The segment descriptors used in protected-mode addressing contain information to provide for hardware memory protection as well as the segment base address. Figure 5.4 shows the format of descriptors for code and data segments in the 80386 and 80486 processors. The format is the same for 80286 microprocessors, except that the 16 bits labeled `base 31 . . . 24, G, D,` and `limit 19 . . . 16` are all zeros for the 80286. The two items to notice in the descriptor format for now are the 20-bit limit field constructed by concatenating `LIMIT 19 . . . 16` and `SEGMENT LIMIT 15 . . . 0)` and the access-rights byte.

The limit field of a descriptor tells how large the segment is. If an offset used to access information in the segment exceeds this value, it signifies that an attempt has been made to access memory outside the segment, and the hardware then raises a general protection violation. As indicated earlier, allowing segments to overlap can defeat this protection mechanism, and overlapping segments are not normally used for iRMX applications. Protected mode versions of iRMX do, however, provide system calls that allow users to create descriptors for segments that occupy arbitrary parts of physical memory, including overlap with other segments, and to allow users to set their access-rights bytes in arbitrary ways.

To accommodate 4-GB segments with a 20-bit limit field, the `G` (granularity) bit is set to 1, and the processor internally appends twelve 1s to the

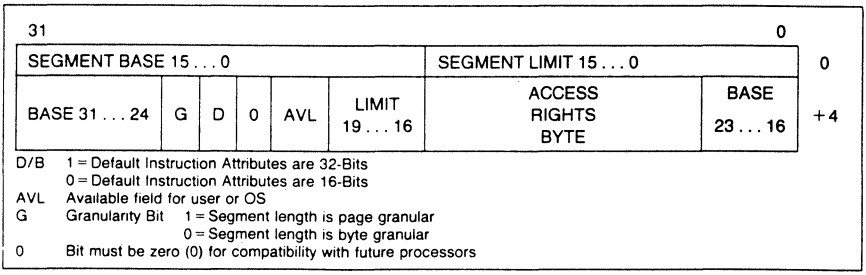


Figure 5.4 Descriptor format for code and data segments. Reprinted by permission of Intel Corporation, © Intel Corp. 1989.

right of the limit value from the descriptor, for a total of 32 bits, which implies that the size of segments larger than 1 MB must be a multiple of $2^{12} = 4$ KB in size.

The access-rights byte tells the processor whether a descriptor is for code, a stack, a data segment, or one of the other types of segments discussed later in this chapter. For code segments, the access-rights byte also signifies whether read accesses are allowed in addition to instruction fetch accesses, and, for data segments, whether write accesses are allowed in addition to data read accesses. For protected-mode versions of iRMX, the memory management software always makes code segments both executable and readable and data segments both readable and writeable.

5.2.4 Other types of descriptors

The x86 protected-mode architecture uses descriptors for more than just describing code, data, and stack segments. There are also *system segment descriptors* and *gate descriptors*. System segments are memory segments used by the microprocessor itself. Typically, these segments are initialized by the operating system and then accessed and updated by the microprocessor as it runs. Gates are special descriptors used to provide controlled access to system calls and interrupt handlers in protected mode.

System segments. There are four types of system segments: the global descriptor table (GDT), local descriptor tables (LDTs), the interrupt descriptor table (IDT), and task state segments (TSS).

Only one global descriptor table exists, but there can be many local descriptor tables. Each descriptor table contains up to 8,192 descriptors. Two CPU registers hold the physical memory addresses of the global and current local descriptor tables. Each time a segment register is loaded with a selector, bit 2 of the selector (third from the right) is used to indicate whether the corresponding descriptor is in the global or local descriptor table, and another 13

bits are used to index into the proper table to obtain the correct descriptor.[6] The descriptor is then loaded into a CPU register associated with the selector (code, stack, data, extra) and is not changed until the segment register is loaded again. Performance of the 80286, 80386, and 80486 microprocessors suffers a bit because these processes do not contain actual registers to hold the current segment selectors — reloading the same selector results in loading the same segment descriptor from the descriptor table again. This behavior is logically necessary in case the contents of a descriptor table in memory changes between loads of a selector. An on-chip cache of recently used selectors and their descriptors could improve performance significantly.

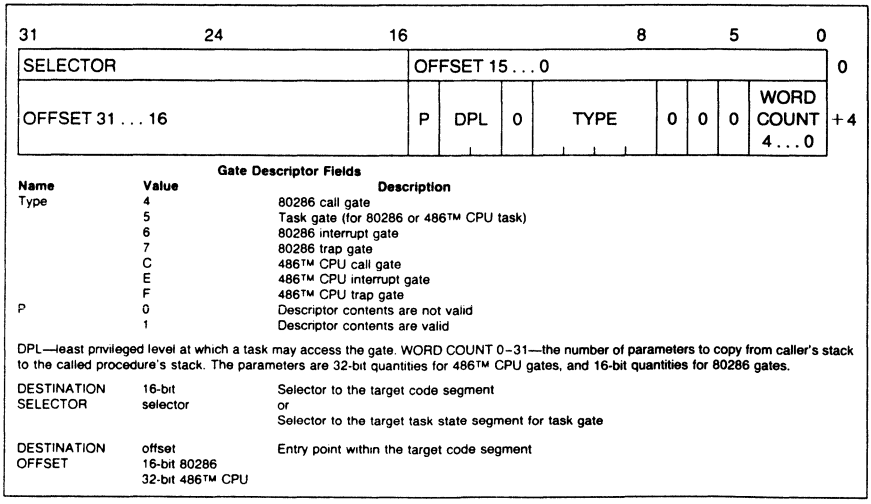
The memory management software of the OS builds descriptors and inserts them into the appropriate table in memory. To accomplish this routine, the OS keeps a descriptor for a writeable data segment that overlaps the descriptor table segment, allowing the operating system to modify the descriptor table memory.

Although the x86 protected-mode architecture provides direct support for multitasking applications, such as the provision for local descriptor tables, iRMX II and III do not currently use these features to implement its multitasking operations. It is more efficient for the operating system to perform a subset of the architecture's multitasking operations in software than to let the CPU do full multitasking in hardware. At this point, the relevant issue is that current implementations of iRMX II and III use the global descriptor table for all code, data, and stack segments.

Task state segments (TSS) are also associated with hardware support for multitasking. Each TSS contains all the information the CPU needs to interrupt and resume a thread of execution. This information includes the state of all CPU registers, such as the `cs`, `ip`, `ss`, `sp`, and `bp` registers mentioned above in the discussion of procedure calls and multithreading. TSSs also provide space for any housekeeping information an operating system might want to maintain about a thread, such as its priority and scheduling state. Although iRMX II and III do not use CPU management for multitasking, they do maintain similar data structures to hold the information needed for iRMX tasks.

Gate descriptors. As indicated above, the three types of descriptor tables, (global, local, and interrupt) contain special descriptors known as gates. There are call gates, trap gates, interrupt gates, and task gates. The structure of these gates is given in Figure 5.5. Except for task gates, gate descriptors contain a pointer to a routine that will receive control when the

[6] These uses account for 14 of the 16 bits in a selector. The right-most two bits are a privilege level (valve 0 to 3), mentioned below.



Gate Descriptor Fields

Name	Value	Description
Type	4	80286 call gate
	5	Task gate (for 80286 or 486™ CPU task)
	6	80286 interrupt gate
	7	80286 trap gate
	C	486™ CPU call gate
	E	486™ CPU interrupt gate
	F	486™ CPU trap gate
P	0	Descriptor contents are not valid
	1	Descriptor contents are valid

DPL—least privileged level at which a task may access the gate. WORD COUNT 0–31—the number of parameters to copy from caller's stack to the called procedure's stack. The parameters are 32-bit quantities for 486™ CPU gates, and 16-bit quantities for 80286 gates.

DESTINATION SELECTOR	16-bit selector	Selector to the target code segment or Selector to the target task state segment for task gate
DESTINATION OFFSET	offset 16-bit 80286 32-bit 486™ CPU	Entry point within the target code segment

Figure 5.5 Descriptor format for call, trap, and interrupt gates.

gate is referenced. The pointer consists of a selector for the code segment that contains the routine, and the field that would contain the limit for segment descriptors is used as the offset part of the pointer. The use of these gate descriptors is described further in section 5.4.

5.2.5 Privilege levels

The x86 protected-mode architectures support rings of privilege, which are used to provide controlled access to OS functions from applications. The code being executed at any moment has a privilege level associated with it (held in the low-order two bits of the `cs` selector register), and every descriptor has a 2-bit descriptor privilege level used to determine whether or not the currently executing code is allowed to access the item referenced by the descriptor.

A separate 2-bit privilege level is loaded into a CPU register to control the use of machine language instructions that perform I/O operations. This privilege level is known as the I/O Privilege Level (IOPL) field of the Flags register. The current code selector's privilege level is compared to the I/O privilege level to check whether a program is allowed to execute I/O instructions or not. These instructions are described in section 5.6.

Current versions of iRMX II and iRMX III do not use the privilege levels provided by the processor. They effectively turn this feature off by setting the privilege levels for all code segments to 0 (the most privileged ring) and the I/O privilege level of the CPU to 3 (I/O instructions can be executed by

any code regardless of privilege). This design decision makes sense because iRMX is not intended as a timesharing system where tasks compete for access to CPU resources in an uncontrolled manner, or where there is the possibility that malicious processes might try to compromise the system. Rather, iRMX is designed as a real-time system where tasks cooperate to complete jobs, and the overhead of managing privilege levels could interfere with real-time performance.[7]

5.2.6 Paging

The x86 architecture supports memory paging for 80386 and later processors. Paging is a mechanism whereby the memory addresses generated by a program, called *virtual memory addresses*, are mapped by memory management hardware in the processor to different physical memory addresses. To accomplish this mapping, the physical memory is divided into fixed-size blocks (4KB for x86 processors) called *page frames*. Memory management hardware includes a memory map, which contains a list of page frame numbers. Virtual memory addresses are conceptually divided into two parts, a logical page number (the leftmost 20 bits of the address) and an offset into the logical page (the rightmost 12 bits of the address). The logical page number is used as an index into the memory map to obtain a page frame number, which is concatenated with the offset part of the address to generate the physical memory address that is actually used for reading or writing memory. Figure 5.2 illustrated this simplified description of paging.

Paging for the x86 architecture is performed by the processor after the segmentation processing, as was shown in Figure 5.2. This implementation is sometimes referred to as *paging under segmentation*. Like segmentation, paging does not require any particular features of the memory system itself, just access through linear physical addresses. The clear sheet of plastic analogy used earlier for describing segmentation could be extended to paging by imagining a series of optical fibers connecting the graph sheet cells with arbitrary locations on the clear plastic segmentation overlay. By looking at a location on the segmentation sheet, one might actually see any location in physical memory. The analogy, however, is perhaps a bit overburdened to be useful. For example, paging can map two different virtual memory locations to the same physical memory location, which does not correspond well to a plastic sheet and fiber optics model.

The actual x86 memory map is implemented using memory-resident tables called *page tables* and *page directories*. The issue involved is the need

[7] iRMX for Windows does use privilege rings to help isolate the DOS and Windows environments from the iRMX environment. Refer to the discussion of interrupt virtualization in Chapter 12 for more information on how this is implemented.

to manage 2^{20} or 1M of virtual pages for 32-bit virtual addresses and 4 KB pages. Such a large memory map cannot be implemented on the CPU chip itself using current technology. Even if a single such map could be accommodated, many paging systems require separate memory map images for different processes or threads of execution, all of which would have to be held in primary memory (or disk) anyway.

Paging is enabled by turning on the pg bit of the processor's control register 0 and loading control register 3 with the physical base address for a page directory table. The page directory table contains 1,024 entries, each of which has the physical address of a page table. Each page table then contains 1,024 entries for page frames. When paging is enabled, the processor uses the 32-bit virtual address computed by the effective address and segmentation units to access the paging mechanism. The high-order 10 bits of the virtual address index into the page directory, which contains the base address of a page table. The second 10 bits of the virtual address (bits 12 through 21) index into this page table, which contains the base address of a page frame in memory. The remaining 12 bits of the virtual address are added to the page-frame base address to obtain the physical address of the memory location to be accessed.

To reduce the overhead associated with paging, the processor maintains a cache of the most recently computed virtual-to-physical page address mappings in its translation lookaside buffer. Access to locations in these pages are made without accessing either the page directory or a page table in primary memory.

Both the page directory entries and the page table entries contain a bit (the p bit) that signifies whether the corresponding information (page table or page frame) is actually present in primary memory. If this bit is off when an access is made through the entry, the processor generates an interrupt number 14, which must be connected to the page fault handler to handle the condition. Usually the handler reads a copy of the referenced information in from disk to replace some other directory or frame. Other bits in the page table entries control whether a page is readable, writeable, or executable. Systems that use hardware tasking can easily assign different page directories and page tables to different programs to implement hardware-enforced memory protection across tasks.

It is the responsibility of the operating system, or possibly the application program in the case of iRMX, to allocate memory for the page directories and tables, to initialize their p bits appropriately, to initialize the read/write/execute bits as desired, and to load control register 3 with the address of the page directory. After that, the operating system (or iRMX application) must handle page faults as they occur by supplying the proper code to handle interrupt number 14s. Both PLM and C provide statements for loading control registers. In PLM-386, loading the control registers is performed by referencing the built-in array of words named

`control$register`. iC-386 provides functions named *getcontrolregister()* and *setcontrolregister()*.

Paging is a powerful tool for memory management. Although not used in any versions of iRMX at the time of this writing, paging might be incorporated in iRMX III or iRMX for Windows at some future time. Consider the following three situations and how they could be handled for iRMX III and iRMX for Windows:

1. Virtual memory smaller than physical memory.
2. Virtual memory and physical memory are the same size.
3. Physical memory is smaller than virtual memory.

Virtual memory smaller than physical memory. An inevitable process in the evolution of computer systems seems to be the appearance of the small memory problem. This problem refers to the inability of a computer architecture to address as much primary memory as people would like to attach to the processor.

For example, the small memory problem manifests itself in the x86 architecture in the 20-bit limit placed on real-mode memory addresses and, to a lesser extent, on the 24-bit limit for 80286 protected-mode addresses. As DOS applications were developed that needed to use more than 1MB of memory (DOS runs in real mode only), the small memory problem became acute. The solution was a software implementation of paging called Expanded Memory Management (EMM). For x86 microprocessors that do not include paging support (i.e., the 80286 and below), one or more pages are reserved in the upper area of memory between 640K (the official top of DOS's memory) and 1M + 64K (the top of real-mode addressing). Applications make special system calls to signify what physical page frame they want mapped to a particular upper memory page. The system calls then interface with an expanded memory board to map the desired page frame to the selected upper memory page, and the application can then access the expanded memory through the addresses in the upper memory page.

With processors that support paging (80386 and later), EMM can be implemented more efficiently using what is called *extended memory* and paging. Extended memory is simply memory that can handle 24-bit or larger addresses, but contains no address mapping mechanism of its own. In this situation, software called EMM386 puts the processor into protected mode (so it can generate addresses greater than 1M + 64K) and enables the processor's paging mechanism. When an application needs to reference expanded memory, it makes the same system calls it would have made for conventional expanded memory, but EMM386 receives the calls and uses the information to update the processor's page tables. Subsequent accesses

to the upper memory pages are then mapped to the appropriate page frames above 1M.

iRMX for Windows supports expanded memory provided it is implemented using expanded memory hardware. iRMX for Windows does not support expanded memory using paged extended memory because it cannot start if the processor has already been put in protected mode by EMM386.

Virtual memory and physical memory are the same size. Sometimes, it is inconvenient or impossible for the operating system to allow an application to reference physical memory most naturally. Paging allows the operating system to establish an arbitrary mapping between an application's memory references and the actual physical memory it uses.

For example, memory segments by definition are physically contiguous in main memory. The problem is that an application might try to create a large memory segment for which there is enough free memory, but which is fragmented due to earlier patterns of allocating and freeing managed memory space. (Chapter 6 covers the iRMX memory management policies.) iRMX III and iRMX for Windows could handle the allocation of large memory segments by using the paging mechanism to map a noncontiguous set of physical pages to a contiguous memory space within a segment. Since paging is done under segmentation, the mapping would be transparent to the application; segments would appear to be contiguous, as expected. There would be problems for DMA device controllers, however, which access memory without going through the processor's paging unit. The operating system would need to provide device drivers for such controllers with the virtual-to-physical mapping information.

Large segments are defined as those over 1 MB in size. Segmentation granularity must be used for segments this large anyway, making them multiples of 4 KB in size and thus conforming nicely with the x86 page size. This desirable feature may be incorporated in a future version of iRMX for Windows.

Physical memory is smaller than virtual memory. This situation is closely associated with the notion of virtual machines that has given logical memory its alternate name of virtual memory. In this situation, an application program can be as large as necessary, even larger than the amount of physical memory available for execution. An initial set of pages that compose the application is loaded into memory, and execution begins. When the program references a page not currently available in primary memory, the operating system intervenes to bring the desired page into memory from disk, a process called *demand paging.*

Demand paging is a very effective technique for improving multiprogramming performance in time-sharing systems in addition to its value for

managing memory use by programs larger than available physical memory. An operating system can keep the active pages of several different programs in primary memory at the same time if the OS uses a reasonable page replacement policy.[8] By keeping just the active pages of several programs in memory, the operating system can readily schedule another process to use the CPU when the running process blocks for an I/O operation. Without demand paging, primary memory holds fewer programs at once, and the operating system must go to disk to find code that the CPU could execute when a running process is blocked.

The discussion in chapter 1 on the importance of deterministic response times in real-time systems should make it clear why demand paging is not implemented in iRMX. Some real-time operating systems deal with the indeterminacy arising from unpredictable page fault patterns by allowing real-time processes to have their pages locked in memory, which excludes these processes from consideration for replacement by the operating system's page replacement algorithm, while allowing non-real-time processes to be paged. iRMX assumes that all processes are real-time and not paged, while allowing the application developer the option of adding paging for non-real-time processes.

For example, DOS running with Windows is an application program as far as iRMX for Windows is concerned.[9] Windows running in enhanced mode conducts a form of demand paging among the programs it runs using the paging mechanism of the 80386 and later microprocessors. This use of demand paging by an application program is allowed by iRMX, but requires some type of coordination by Windows and iRMX in the use of page tables before iRMX for Windows will be able to support enhanced-mode Windows.

5.3 Interrupt Processing

The interrupt processing mechanism of the x86 architecture has already been mentioned several times. This section describes the mechanism from the CPU's perspective. Chapter 9 provides more details on implementing the software used to interface with interrupts in iRMX systems.

Interrupt requests can be initiated either by hardware external to the CPU or by software using the machine language *int* instruction. There are two types of hardware interrupts, non-maskable (NMI) and maskable. The NMI is used for abnormal situations, such as impending loss of elec-

[8]The page replacement policy is the algorithm the operating system uses to decide which page to remove from primary memory when the running process references one of its pages not currently in memory.

[9]The proper terminology is that DOS is an iRMX task, as discussed in chapter 6.

trical power, and is not considered further here. In the case of maskable interrupts, external circuitry called a programmable interrupt controller (PIC) arbitrates among the various devices causing interrupts, manages priorities and sequencing in the case of simultaneous requests, and informs the CPU of the interrupt level (a number between 0 and 255) when an interrupt occurs. Software interrupts appear to be of the same type as maskable interrupts, but the *int* instruction supplies the interrupt level number rather than the PIC.

Software and hardware interrupts are handled differently by the CPU. Hardware interrupts are acknowledged only when the processor is between machine instructions and has its interrupt enable bit (one of the bits in the CPU's Flags register) set to 1, but software interrupts can never be disabled.

The PIC used with x86 processors can receive interrupt signals from eight different wires, which are normally connected to different device controllers. (Device controllers are described below.) When one or more interrupt signals arrives at the PIC, it sends a signal to the CPU on the INTR (interrupt request) wire to indicate the condition. When interrupts are enabled in the CPU and the CPU is between machine instructions, it acknowledges the interrupt by sending a signal called INTA (interrupt acknowledge) to the PIC and reading the interrupt level of the highest priority interrupting device back from the PIC.[10] Before interrupt processing begins, the processor must program the PIC by sending it commands signifying which interrupt level to associate with each of its eight interrupt sources. The PIC decides which level number to send to the processor by evaluating the relative priorities of the sources requesting interrupts at the time it receives the CPU's INTA signal.

A single master PIC can have up to seven slave PICs connected to it in cascade fashion as shown in Figure 5.6, for a total of up to 57 external interrupt sources.[11]

Once the processor acknowledges an interrupt request and determines the interrupt number, it pushes its flags register plus the `cs:ip` of the next instruction to be executed onto the stack of whatever program is running, disables further interrupts, and branches to the interrupt handler associated with the interrupt number. Every task or process in a multithreaded system must have a stack segment large enough to accommodate the information that might be pushed onto it by an interrupt, in addition to its own stack requirements for activation records.

In real mode, as for DOS or iRMX I, an array of 256 pointers is kept in the first 1,024 bytes of memory, and the address of the appropriate inter-

---

[10]Certain instructions that take a long time to execute, such as instructions that perform an operation on an entire array of bytes, can be interrupted during their execution and resumed later from where they left off.

[11]Input number 0 of a master PIC cannot be connected to a slave PIC.

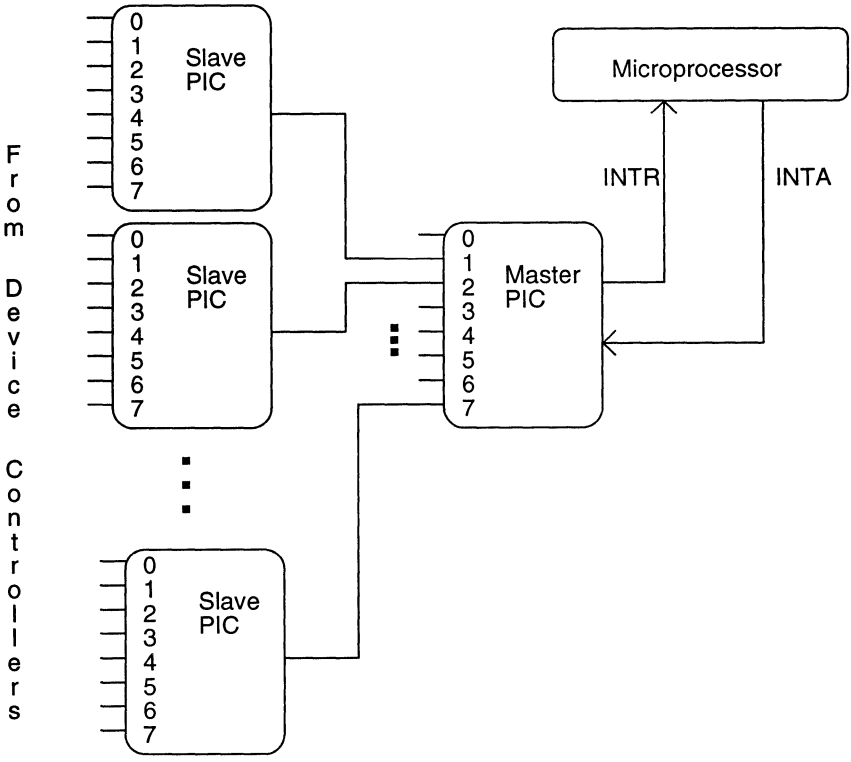


Figure 5.6 Connections between Programmable Interrupt Controllers (PICs) and the CPU.

rupt handler is determined by indexing into this array with the interrupt number. In protected mode, this array of pointers is replaced by a structure called the interrupt descriptor table (IDT), and the interrupt number is used as an index into the IDT. The IDT contains task, interrupt, or trap gates, described in the next section. Typically, the IDT contains interrupt descriptors, which consist of a pointer to the interrupt handler plus a type code telling the CPU that the CPU should disable interrupts before jumping to the interrupt handler code.

When an interrupt handler procedure starts executing, it does so in the context of the program running at the time of the interrupt. Since this activity is transparent to the interrupted program, it is important for interrupt handlers not to disturb the state of the CPU as they do their work. To accomplish this, all processor registers that an interrupt handler modifies must be saved by pushing them onto the stack of the interrupted program, and then they must be restored when the interrupt handler finishes its work and is ready to let the interrupted program resume. The processor's flags, `cs`, and `ip` registers are automatically pushed onto the stack by the CPU's interrupt mechanism, but any other registers must be pushed explicitly.

Finally, when an interrupt handler completes its processing, it must restore the contents of any registers it has modified to their original values. By including the `cs` and `ip` registers in this process, the interrupt handler effectively branches back to the interrupted program. The cleaning up is done in two stages: first, any register values that were pushed onto the stack when the interrupt handler started executing are popped back into their original registers. Then, the `cs`, `ip`, and flags registers are popped from the stack by a special form of *return* instruction called *iret* (interrupt return). Both the PLM and Intel C compilers generate all the special code for interrupt handlers if they are declared to be interrupt procedures or functions. In PLM, this is done by specifying the `interrupt` keyword in the procedure declaration, and in C it is done using the `interrupt` pragma or compiler control.

5.4 Call, Task, Interrupt, and Trap Gates

Gates are protected-mode descriptors used to provide controlled access from application programs to operating system software, such as system calls and interrupt handlers. The four types of gates, call, task, interrupt, and trap, can be found in either the GDT or LDT, and all but call gates can be found in the IDT.

Gates are accessed from the IDT when an external interrupt is acknowledged, when a software *int* instruction is executed, when a trap occurs (such as division by zero), when a fault is detected (such as a memory protection violation), or when an abort occurs (when a fault occurs while processing a fault). Table 5.1 shows the interrupt levels automatically used by the microprocessor for various types of events. I/O interrupts use interrupt levels above 32.

Gates are accessed from the GDT or LDT when a program executes a far call machine instruction. A far call is one that uses a far pointer (selector and offset) as the address of a subroutine. Normally, a far call causes the selector part of the pointer to be loaded into the processor's `cs` (code segment) register, and the corresponding descriptor to be loaded into the code segment descriptor register. The offset part of the pointer is then added to the base address found in the descriptor to calculate the linear address of the next instruction to be executed. If the selector part of the pointer selects a descriptor for a gate (determined by the type code in the descriptor), however, the offset part of the instruction's pointer is discarded, and a far pointer to the actual code is obtained from the gate itself. The far pointer from the gate is then used as just described for operating with the pointer part of a regular far *call* instruction.

Call gates are normally used to make system calls. Two properties of call gates support the rings-of-privilege feature of the processor mentioned earlier. The first property is an automatic, but controlled, shift in CPU privilege between the application and the OS. The gate contains a privilege

TABLE 5.1 Interrupt Vector Assignments.

Function	Interrupt Number	Instruction Which Can Cause Exception	Return Address Points to Faulting Instruction	Type
Divide Error	0	DIV, IDIV	YES	FAULT
Debug Exception	1	Any instruction	YES	TRAP*
NMI Interrupt	2	INT 2 or NMI	NO	NMI
One Byte Interrupt	3	INT	NO	TRAP
Interrupt on Overflow	4	INTO	NO	TRAP
Array Bounds Check	5	BOUND	YES	FAULT
Invalid OP-Code	6	Any illegal instruction	YES	FAULT
Device Not Available	7	ESC, WAIT	YES	FAULT
Double Fault	8	Any Instruction That Can Generate an Exception		ABORT
Intel Reserved	9			
Invalid TSS	10	JMP, CALL, IRET, INT	YES	FAULT
Segment Not Present	11	Segment Register Instructions	YES	FAULT
Stack Fault	12	Stack References	YES	FAULT
General Protection Fault	13	Any Memory Reference	YES	FAULT
Page Fault	14	Any Memory Access or Code Fetch	YES	FAULT
Intel Reserved	15			
Floating Point Error	16	Floating Point, WAIT	YES	FAULT
Alignment Check Interrupt	17	Unaligned Memory Access	YES	FAULT
Intel Reserved	18–32			
Two Byte Interrupt	0–255	INT n	NO	TRAP

*Some debug exceptions may report both traps on the previous instruction, and faults on the next instruction. Reprinted by permission of Intel Corp.

level that must be matched or exceeded by the privilege level of the code being executed. A gate with a privilege level of zero could be called only by code running at ring zero (determined by the privilege level of the current code descriptor), but a gate with a privilege level of three could be called by any code, for example. If a program is allowed to call a gate, the selector part of the gate is used to access a new code segment. The CPU shifts to the privilege level of this new segment (typically level zero) when it starts executing the OS code, and automatically reverts to the original privilege level when the OS returns to the application.

The second property of call gates is the provision for separate stacks for different privilege levels. A 5-bit field in a call gate descriptor signifies how many words (parameters) to copy from the application's stack to the called routine's stack before transferring to the called routine. The performance overhead incurred by parameter copying across stacks is one of the main reasons that iRMX II and III do not use the privilege-rings feature of the

x86 protected-mode architecture.[12] iRMX II and III do, however, use call gates for accessing sytem calls from application programs, as seen in Chapter 6.

Traps are essentially interrupts that occur because of the execution of certain instructions. Examples of traps include division by zero, invalid operation codes, memory protection violations, and the like. In fact, the CPU treats software interrupt instructions as traps. Both trap and interrupt gates contain pointers to interrupt service routines and are normally found in the IDT. Unlike call gates, neither trap nor interrupt gates cause any parameters to be copied from the interrupted program's stack to the operating system's stack. The difference between trap and interrupt gates is that an interrupt gate resets the CPU's interrupt enable flag, but trap gates do not.

Hardware tasking, activated by calls to task gate descriptors, is a particularly intriguing feature of the x86 protected-mode architecture for a multitasking operating system such as iRMX. A hardware task is a complete execution context including the address for a thread of execution (`cs` and `ip` registers), stack registers for all privilege levels, plus data and extra data segment registers, and a pointer to an LDT table for providing a memory context. All this information is kept in the Task State Segment (TSS) memory segment. A task gate contains a selector for a TSS; a far call that references a task gate causes the processor to save the current state of the processor in the current TSS (a CPU register has the current TSS's selector in it) and to load all the CPU registers from the new TSS.

Although iRMX II and III use a data structure that has the same format as a TSS to hold the states of the tasks it manages, current implementations of II and III do not use task gates, and the entire operating system and all applications run as a single machine task. The i486 data book claims that that processor can completely switch from one hardware task to another in just 17 microseconds, but iRMX engineers have found that system performance is even better if they switch task contexts in software rather than if the operating system uses multiple hardware tasks. By using a data structure similar to the TSS structure for iRMX tasks, however, the option remains to change the operating system relatively painlessly to use multiple hardware tasks at some time in the future.

5.5 Virtual 8086 Mode

The x86 protected-mode architecture provides a feature called *virtual 8086 mode* (VM86) to allow real-mode programs, including operating systems such as DOS, to run while the processor is in protected mode. To the program running in VM86 mode, the processor appears in every way to be

[12]The other reason is the overhead of loading the descriptor for a new stack segment and restoring it again when the call completes.

operating in real mode. The CPU appears to use base and offset pointers rather than selectors and offsets, memory is limited to 1M + 64K bytes, interrupts appear to be vectored through pointers in the first 256 doublewords of memory, and so on.

In reality, a VM86 program runs as a privilege-level 3 (the lowest privilege ring) machine task in protected mode. A protected-mode operating system (such as iRMX for Windows) must provide the software glue to make the VM86 real-mode illusion work. All interrupts and traps, including the software *int* instructions typically used to make system calls from real-mode operating systems, cause the processor to generate a general protection (GP) fault (protected-mode interrupt number 13) when the CPU is operating in VM86 mode. The GP faults are handled by the protected-mode operating system, and one of the bits in the CPU's flags register (the vm bit) tells whether or not the interrupt was generated while the processor was in VM86 mode. If the processor was in VM86 mode, the protected-mode operating system can choose either to handle the interrupt or trap on behalf of the VM86 program or to pass control back to the VM86 program to handle the interrupt itself, whichever is appropriate. Chapter 12 describes the operation of the iRMX for Windows code that handles VM86 mode interrupts, called the *VM86 Dispatcher*.

If paging is enabled, the protected-mode operating system can transparently map the VM86 program's 1 MB of memory into any part of available RAM, including memory above the first 1 MB of real memory. Thus, for example, multiple copies of DOS could be running simultaneously in different parts of RAM as different simultaneous VM886 programs. (iRMX for Windows does not enable paging.)

The VM86 architecture also provides flexible control over I/O processing. The protected-mode operating system can selectively allow a VM86 task to access I/O ports (see Section 5.6) directly, or can program the processor to cause a GP fault whenever an I/O port is accessed. The selection of which port accesses cause GP faults is based on the I/O permission bitmap in the TSS of the VM86 program. The bitmap has one bit for each I/O port that tells whether the VM86 program is allowed direct access to the port or if accessing the port will cause a GP fault. For example, iRMX for Windows, which runs DOS programs in VM86 mode, uses the I/O permission bitmap to control access to the computer's PIC, timer, and serial ports. More details on the implementation and features of iRMX for Windows is covered in chapter 12.

5.6 I/O Processing

CPU accesses to memory and I/O devices are closely related, so this section begins with an overview of how the microprocessor reads and writes all types of information, and then examines the specifics of how I/O operations occur. This section is of particular interest to those who will be devel-

oping device drivers, the software that interacts with device controllers. Although the concepts presented in this section focus on hardware issues and the corresponding machine-language constructs, PLM and C compilers provide full access to these resources from a high-level language. The actual design of iRMX device driver software is covered in chapter 9.

The microprocessor is connected to both memory systems and I/O device controllers by means of a bus, which consists of address, data, and control wires. Various systems provide different buses, called the CPU bus, the local bus, or the system bus. The CPU bus refers to the wires attached directly to the electrical contacts on the microprocessor's housing and provides the highest speeds for moving information into and out of the CPU. (The electrical contacts are normally called pins.) The CPU bus is limited, however, in terms of the lengths of wires that can be used for it and the number of memories or device controllers that can be connected to it.

The system bus is logically similar to the CPU bus, but includes additional circuitry so that it can be reliably connected to a larger number of memories or device controllers over greater distances. A system bus also normally includes additional signal wires compared to the CPU bus so that the system bus is suitable for connecting different CPU types to device controllers and memories in a standard way. Examples of system buses include the ISA, EISA, Microchannel, Multibus I, and Multibus II buses mentioned in chapter 2.

A local bus is either a synonym for the CPU bus or a third level of wires intermediate between the CPU bus and the system bus in terms of electrical loading and speed characteristics. The distinctions among the three types of buses are not really important for our purposes, and the material here is presented in terms of just one bus, which will be treated as if it were the CPU bus without any real loss of generality.

The CPU must perform six different types of information transfers using the bus: (1) read an instruction from memory, (2) read data from memory, (3) write data to memory, (4) read data from a device controller, (5) write data to a device controller, and (6) read an interrupt level number from an interrupt controller.

For the i486, the CPU has three pins used to identify which type of bus operation it is performing, called M/IO#, D/C#, and W/R#.[13] M/IO# is true if the microprocessor is reading or writing memory data or code, and false if the microprocessor is reading or writing I/O data.[14] D/C# is true if the information being read or written is data, and false if the information is a machine instruction or an interrupt vector. W/R# is true if the micropro-

[13]These particular pin names and their meanings are unique to the i486, but other processors use similar pins with similar functions.

[14]This section describes conventional I/O operations. An alternative is called *memory mapped I/O*, in which M/IO# is always true, and device controllers respond to particular memory addresses as if they were I/O addresses.

cessor is writing (sending information from itself to a memory or device controller), and false if the microprocessor is reading. There are 2^3 or 8, ways these three bits can be set, of which we are concerned with the six combinations that correspond to the six types of bus transfers mentioned above. The following table is a summary of the six types of bus operations.

M/IO#	D/C#	W/R#	Type of Bus Transfer
0	0	0	Get Interrupt Vector
0	1	0	I/O Read
0	1	1	I/O Write
1	0	0	Instruction Read
1	1	0	Memory Read
1	1	1	Memory Write

These three pins of the microprocessor are connected to three of the wires that constitute the control part of the bus. They tell all the device controllers, memories, and the interrupt controller what the CPU wants to do with the bus at all times. Two other pins of the microprocessor connected to the bus are called ADS (address status), which is true whenever the microprocessor wants to start a bus transfer, and RDY, which is used by the memories and device controllers to let the microprocessor know when a bus transfer is complete. One other pin of the microprocessor that connects to a wire in the control portion of the bus is INTR, which is used by the interrupt controller to send a signal to the microprocessor telling it that there is a request for an interrupt.

The number of address and data wires in the bus depend on the particular microprocessor. The i486 can access up to 4 GB of memory, and can access 1, 2, or 4 bytes of information per bus transfer. To do this, it has 30 address pins (A_{31} through A_2) that can select any of 1G 4-byte words, plus four additional pins (BE_3 through BE_0) for specifying which byte(s) within the word are involved in the transfer.

Another way to look at this would be as if there were 32 address pins plus two more pins to tell how many bytes are to be transferred. The 80286 microprocessor has 24 address pins, and the 8086 microprocessor has 20 address pins, corresponding to the maximum memory capacities of 16 MB and 1 MB of those two CPUs, respectively. Likewise, each microprocessor has a number of pins devoted to carrying the information to be read or written, called data pins, even though instructions and interrupt vector numbers are transferred through these pins in addition to data. The i386 and i486 have 32 data pins; the 8086 and 80286 have 16. The address and data pins of the microprocessor are connected directly to the address and data wires of the bus.

For memory read and write operations, the microprocessor calculates a 32-bit physical address using the segmentation, and possibly paging, mechanisms as shown in Figure 5.2. This 32-bit physical address is sent out

over the address wires of the bus at the same time that the M/IO#, D/C#, and W/R# wires are set to their appropriate values, and ADS is asserted (made true) to signal the start of a bus cycle. Every device controller and memory attached to the bus examines the address wires simultaneously, and one memory unit will recognize the address as belonging to itself. That memory unit will then either store the data that the microprocessor supplies on the data wires into the proper place (a memory-write operation) or will supply the data from the proper memory location on the data wires (a memory-read operation). The memory unit signals the completion of either type of bus transfer by asserting RDY. A crucial concept here is that everything connected to the bus includes logic circuitry for comparing an address on the bus to the addresses to which it will respond, and for comparing M/IO#, D/C#, and W/R# to the combination of bits that signals its own type of device. Those controllers or memories that do not match an address value for a particular bus cycle simply ignore all further activity on the bus until ADS becomes true again.

From this discussion of how a memory cycle operates, it should be clear that there are similarly structured addresses for both memory and for device controllers. The significant differences between device controller addresses and memory addresses are the following:

- The CPU computes memory addresses by using the segmentation and paging mechanisms described earlier whenever it fetches instructions, as well as when it uses the effective address of a machine instruction to access an operand from data or stack memory. Device controller addresses are generated directly from special I/O machine language instructions that never involve paging or segmentation.
- Device controller addresses are limited to 16 bits for all x86 architectures, whereas memory addresses are 20, 24, or 32 bits, depending on the CPU model. Thus, there is always a maximum of 64K device controller addresses.

The question arises of what it is that device controller addresses refer to. The answer is that each device controller contains a number of registers that can be read or written by the CPU, using exactly the same technique for reading and writing individual memory locations, but with the M/IO# pin set to false. These registers fall into three categories: data buffers, command registers, and status registers. Each of these registers is assigned a 16-bit I/O port address used by the CPU for reading or writing the register. By the way, the term *I/O port* can be used to refer to an entire device controller, such as a serial I/O port. Such a device controller could use several I/O port addresses for the registers and buffers the CPU can access.

To understand this issue, first look at Figure 5.7, which shows the connections between the CPU, bus, memory units, device controllers, and device units. *Device unit* is the iRMX term for an individual I/O device, such

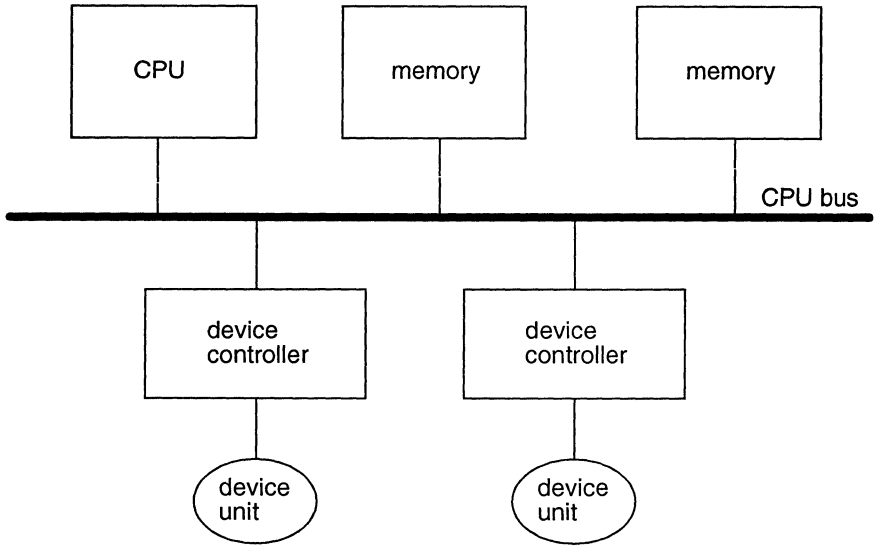


Figure 5.7 Connections between the CPU, bus, memory, device controllers, and device units.

as one terminal or one disk drive. Figure 5.8 focuses on a single device controller, showing the command and status registers and some data buffers. Each of these registers uses a single I/O port address, although some device controllers might share one I/O port address for two registers. For example, writing to an address might store a command in a device controller's command register, but reading from the same address might read the contents of the status register. Figure 5.8 shows only the connections between the data wires of the CPU bus and the registers and buffers in the device controller. The address and control wires are implicit in that diagram.

A command register is used for receiving commands written from the CPU to a device controller. Each device controller has its own set of commands that it can process, analogous to the operation codes of the CPU's machine language. Before using a device controller, a CPU program must write one or more of these commands to the controller's command register to set the various options available for it, such as whether to generate interrupts or not.

The data buffers accommodate the vast difference between the speed with which the CPU can read and write information using its bus compared to the time it takes to transfer information to or from typical I/O devices themselves. To output data to a device unit, the CPU first writes the data to a device controller's data buffer, and the device controller then transmits the data to the device unit in the appropriate manner for the particular device. The transfer of data from the CPU to the device controller can be just as fast as storing information in memory (the same bus is used), but transferring data from the device controller to the device unit is a very slow pro-

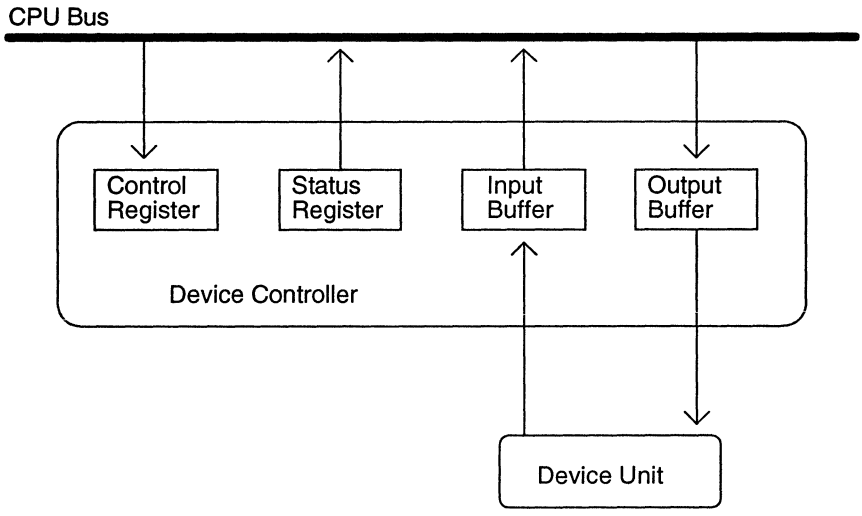


Figure 5.8 Device controller.

cess in comparison. For example, to write a character to a terminal operating at 9600 baud might take less than 100 nanoseconds for the transfer from CPU to device controller, but it would take approximately 1 millisecond (10,000 times as long) for the transfer from the device controller to the terminal. Some device controllers, however, operate at significantly slower clock rates than the CPU and thus mitigate this speed difference.

The CPU continues to execute other code while a device controller performs data transfers to or from a device unit, and there are two methods for the CPU to find out when such a transfer is completed. The first method is called *polling*, in which the CPU periodically reads a device controller's status register. The device controller dynamically changes the setting of one or more bits in the status register to indicate when data transfers are complete. The CPU can read this status register at any time and then test the transfer-complete bit(s) to know when more data can be written to the device controller or when data has arrived at the device controller that can now be read into the processor. The device controller also sets bits in the status register that the CPU can test to determine if any errors occurred as the device controller transferred information to or from its device unit.

The second way for the CPU to know when a data transfer is complete is through the interrupt mechanism. Device controllers can be programmed to generate an interrupt request every time one of their transfer-complete bits becomes true. In this case, the software that responds to the interrupt reads the device controller's status register and tests whatever bits are necessary to verify that the interrupt indicates a data transfer completed nor-

mally rather than because of an error condition. The software can then read or write more data to or from one of the device controller's data buffers.

A direct memory access (DMA) device controller is able to compete with the CPU for use of the system bus. To perform a long data transfer, registers on the DMA controller are initialized with the physical addresses of the beginning and end of the memory buffer to be read or written, and the controller then proceeds to complete the data transfer between the device unit and the buffer, generating a single interrupt to the CPU (or setting a status bit that can be polled) only when the transfer is complete. The controller's address registers are initialized by the CPU using normal I/O instructions. Although the CPU and DMA controller compete with each other for use of the system bus during a DMA data transfer, the technique can be particularly attractive if the CPU has an on-chip code or data cache that reduces its demands on the bus.

The choice between polled and interrupt-driven I/O can be a critical one for high performance real-time systems. Chapter 1 introduced interrupt response time (IRT) as a critical parameter affecting a real-time system's performance. Perhaps surprisingly, the best IRTs are obtained if you don't use interrupts at all! A system that uses polling can respond to the change in a device controller's status register most rapidly by executing a tight loop that continuously reads the status register and tests the bit or bits that indicate an I/O operation has completed. The system responds to the event that would normally give rise to an interrupt request by the device controller without incurring the overhead of saving and restoring the CPU's state.

Of course, polling is not limited to a single device controller. A polling loop can be constructed that tests the status registers of a number of device controllers in succession, branching off to a processing routine each time an event is found to process.

The problem with a polled sytem, however, is that the entire CPU is devoted to the polling loop, so it can do no other useful work while waiting for events to process. An interrupt-driven system can perform time-critical operations in response to interrupts, and defer less critical processing to those times when no interrupt handling routines are active. For systems of even moderate complexity, the overhead associated with saving and restoring CPU state in response to interrupts is offset by the improved manageability afforded by the ability to partition the code executed in response to events by means of separate tasks with different scheduling priorities. Of course, if a single CPU is to be used for both real-time and non-real-time processing, as when DOS and/or Windows is used to provide the user interface to a real-time process using iRMX for Windows, there is no choice but to use an interrupt-driven system.

There is no direct support for polling in iRMX. Polling can be done, but it would interfere with all the task management functions central to the design of iRMX, and will not be considered further in this book.

The programmable interrupt controller mentioned earlier is an example of a special-purpose device controller. It is programmed by writing encoded binary values to its command port addresses, and it outputs the 8-bit interrupt level number onto the data wires of the bus when the CPU initiates a get-interrupt vector bus cycle (M/IO#, D/C#, and W/R# equal to 0, 0, 0).

Intel's PLM and C compilers all support I/O port operations. In PLM, byte output to ports occurs by assigning values to a built-in 64K element array called `output`. There are corresponding arrays for outputting 16-bit and, in PLM-386, 32-bit values as well. Port input of 8- or 16-bit data occurs by invoking built-in functions named `input` and `inword`, respectively, which take port addresses as arguments and return input data as their result. Again, PLM-386 provides for 8-, 16-, and 32-bit data transfers. In C, both input and output occur via functions. The *outbyte()* function takes a port number and a data value as arguments and returns nothing. The *inbyte()* function takes a port number as an argument and returns a byte of input data. Again, additional functions exist for doing 16-bit and, in iC-386, 32-bit transfers as well. There are also machine-language constructs that allow a single instruction to input or output an entire array of bytes, and corresponding C and PLM constructs as well: the *blockinbyte()* function and its cousins for C and the *blockinput()* procedure and its cousins for PLM.

Part

2

iRMX Concepts and Features

Chapter

6

Fundamental iRMX Objects and Structures

6.1 Overview

You can look at the design of an operating system (OS) in various ways: the functions and relationships among the units that constitute the OS, the layout of memory when the OS is running, the different interfaces the OS presents to the application programmer or the user, and other ways as well. This chapter discusses the first three of these views: the software layers that constitute the structure of the OS, memory organization and management, and the system call mechanism used by application programmers to access iRMX services. The common thread across all of these topics, the object-based nature of the operating system, is also discussed.

The Nucleus layer of iRMX is the particular focus of this chapter. The Nucleus is the one required layer of all iRMX systems. A process called *system configuration* can be used to build a copy of iRMX that omits other layers of the system if they are not needed for a particular situation, but every configuration must include the Nucleus layer. A crucial feature of the Nucleus is that it provides the basis for the object-based nature of iRMX. All of the optional layers of the OS build on the resources provided by the Nucleus in ways that preserve this object-based design philosophy.

The chapter begins with a discussion of object-based and object-oriented systems, providing a background for the terminology and concepts used to characterize iRMX. It then presents the three fundamental iRMX object types managed by the Nucleus (jobs, tasks, and segments), and introduces the memory management and system call facilities provided by the Nucleus. The chapter ends with a description of the iRMX system call mechanism that relates both to the object-based nature of the operating system and to the features of the x86 architecture introduced in chapter 5.

6.2 Object-Based Systems

The ideas of object-oriented and object-based systems date back to the 1960s, when the programming language Simula provided programmers with the means to create entities called *classes*. A class is a programmer-defined data type and set of operations that may be applied to items declared to be *instances* of the type. Examples of other languages that support the same concept of classes include Ada (the concept is referred to as *packages*), C++ (referred to as *classes*), and Smalltalk (also referred to as *classes*). Note that the C-language typedef facility is not an example of an object-oriented system because the language provides no built-in mechanism to associate typedefs with the functions that operate on variables of the defined type.

iRMX described itself as object-oriented before that term evolved and became more restricted than would properly apply to iRMX. Following the lead of Finlayson (1991),[1] the term *object-based* is used for iRMX, which refers to a subset of the features of a true object-oriented system.

An object is defined as a data structure combined with a set of functions that provide access to the data structure. An *object type* is the format of the data structure, and an *object instance* is an occurrence of an object type in the computer's memory. For each type of object in an object-based system, the set of procedures for managing individual objects of that type is called a *type manager*. Each object type has its own type manager. All type managers include two procedures: one for allocating memory to hold the data structure that represents an instance of the object type and for initializing the data structure, and another for deleting the object instance and freeing the memory that instance occupied. In addition, each type manager provides a set of functions to appropriately manipulate the objects the type manager manages according to the object type. In a pure object-based system, an application program cannot access the memory occupied by an object except by calling type-manager functions. The set of function definitions provided by the type manager is known as the *Application Program Interface* (API) for the type.[2]

Figure 6.1 shows the relationship between application programs, type managers, and objects for object-based systems. The application programs at the top of the figure can access the objects at the bottom of the figure only by calling the functions provided by the type managers shown in the middle of the figure. In the figure, arrows indicate allowed accesses, either through function calls or direct memory manipulation. Each type manager provides its own set of functions and manages its own separate set of ob-

---

[1] "To parallel Wegner's definition of an object-oriented *programming language* (Wegner, 1987), we say that in order for an operating system to be object-*oriented* rather than object-*based*, it must also support some form of inheritance."

[2] The term *API* is also used in other contexts besides object-based programming.

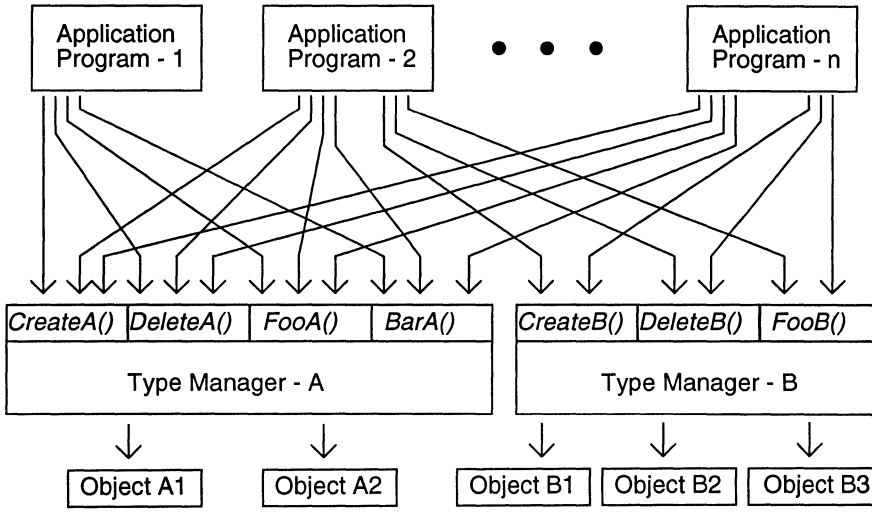


Figure 6.1 Relationships among application programs, type managers, and objects in an object-based system.

jects. The API for Type Manager A consists of the four functions *CreateA()*, *DeleteA()*, *FooA()*, and *BarA()*.

You should note two important features of object-based systems. The first is that object-based systems simplify the work of application programmers, who work only with an API rather than the internal data structures of objects. The second important feature is that object-based systems allow the developer of the type manager the flexibility to implement objects in a variety of ways, provided only that the implementation is consistent with the API. Thus, the implementation can be changed, for whatever reason, without affecting any of the applications that use the type manager, as long as the API specification does not change.

Sometimes, especially when an object's data structure is quite rich, the API does not provide all of the functionality needed by some application programs for dealing with objects of a particular type. This situation results in *cheating*, which is defined as an application program directly accessing the data structure for an object rather than going through the type manager's API. Although cheating might seem innocuous when accomplishing the goal of getting something to work, it is just as iniquitous in software development as it is in real life. Once an application is coded to cheat, it is doomed to work properly only as long as the underlying object implementation does not change. If the object is managed by an operating system, for example, the application might need to be recoded every time the operating system is updated by the vendor.

The only proper solution to the need to cheat is to augment the type manager with additional functions to provide controlled access to the desired information. When done correctly, the new functions do not affect

existing applications that do not use them, and are designed so that they provide a consistent interface to applications despite possible future changes in the internal representation of objects.

Any object-based system needs some mechanism to allow a type manager to discriminate among the various object instances that it has created at one time. In Figure 6.1, how is function *FooA()* to know whether it is supposed to deal with object A1 or object A2 when called? The answer is that the type manager's *create()* function returns an identifier, called a *token* in iRMX, to the application. Thereafter, when the application wants to perform an operation on a particular object, it calls the appropriate type manager function using the token as one of its arguments.

Internally, the type-manager functions use these tokens to identify the part of memory that contains a particular instance of the object. The token could be an index into an array of data structures, a pointer to a data structure in memory, an index into a table of data structure descriptors, or whatever the type manager deems appropriate. In iRMX, all tokens are memory segment selectors (or segment bases in iRMX I) which are used to access memory segments through the Global Descriptor Table (GDT) described in chapter 5[3]. In PLM programs, iRMX tokens are stored in variables declared to be of type `SELECTOR`, for which `TOKEN` is usually declared to be a literal equivalent. The Intel C compilers also provide a `selector` data type, provided you include any one of a number of different header files (`rmxc.h`, `i486.h`, `i386.h`, `i286.h`, `il86.h`, `i8086.h`, or `i86.h`). The `rmxc.h` header file provides the `selector` data type as well as literal equivalence among `selector`, `SELECTOR`, and `TOKEN`.

To summarize, in an object-based system, objects are data structures that occupy memory. A type manager exists for each class of object supported by the system. Each type manager provides functions to allocate memory for a new object, to free memory for objects no longer needed, and perform any other functions needed to support the class of objects represented by the type. Individual objects are said to be *encapsulated*, which means that the memory of these objects cannot be examined directly or modified by application programs, only by functions provided by the type manager.

iRMX is an object-based system. When an iRMX object is created, a memory segment and a slot in the GDT are allocated to the object, and the selector for that segment becomes the token for the object. Because iRMX II and III applications operate at privilege ring 0, object encapsulation cannot be enforced by the hardware; given a token for an object, it is possible for a program to directly access the object memory without resulting in a hardware trap. But to do so is cheating, of course. Several iRMX object types and associated type managers exist, with each type manager consist-

[3]Future versions of iRMX might change to allow LDT-based objects.

ing of a set of system calls for creating, deleting, manipulating, and obtaining information about objects of its particular type.

There are also iRMX system calls that have nothing to do with objects. That is, iRMX does not coerce all operating system functions into the object-based model. For example, there are system calls for obtaining or changing the system's time-of-day clock, but the clock is not treated as an object instance or type of object because it does not make sense to create or delete instances of the time.

6.3 Object-Oriented Systems

As the terminology has evolved, the term *object-oriented* has come to be applied only to systems that include certain features in addition to those of object-based systems like iRMX. Because iRMX is not object-oriented according to present usage of the term, this section describes some of the differences between object-based and object-oriented systems to help keep iRMX in proper perspective. Although the features described in this sec tion are not built into iRMX itself, iRMX could easily be used as the basis for an efficient implementation of any of these features by iRMX applications.

6.3.1 Polymorphism

Polymorphism, also known as operator overloading, refers to using a single function name to provide different functionality, depending on the type of object passed to the function for manipulation. A standard example is the plus operator in most programming languages.[4] If the objects to be summed are real numbers, the plus operator invokes the processor's floating-point unit to perform the operation, but if the objects are integers, plus invokes the processor's fixed-point hardware instead. In most programming languages, the overloading of the plus operator is handled by the compiler when it translates the source code into machine language. In an object-oriented system that supports the mechanism called late binding, overloaded functions can determine the types of objects during program execution rather than when the program is compiled because the objects themselves are not created until run-time.

The main advantage of polymorphism is that it provides the application programmer with a convenient and consistent way to handle objects that have logically equivalent functionality but different implementations. Addition of real numbers and addition of integer numbers are logically equivalent, but have different implementations.

iRMX does not provide polymorphism. Each function provided by each

[4]Syntax is the only significant difference between an infix operator such as + and a function such as *plus()*, so plus can be thought of as a function that is provided as part of the type manager for a data type.

type manager has a unique name, and tokens are validated when system calls are made to ensure that each function deals only with objects of its own proper type. Although polymorphism is a convenient mechanism for the programmer, its greatest advantage comes when it is coupled with late binding, which implies a level of overhead inconsistent with high-performance real-time systems.

6.3.2 Derived classes, inheritance, and code reusability

One of the most appealing features of object-oriented systems is their provision for using existing object types (classes) as the basis for defining new types. Either existing functions in the API are redefined to meet the needs of the new type, or new functions are added to the set provided by the original type. In both cases, all of the functions provided by the original type manager remain available in name, but with possibly altered semantics. Using standard terminology, one says that the derived class (new type) inherits the function set of the base class (original type).

Inheritance is appealing because it promises developers the Holy Grail of code reusability. If you build a good set of type managers for one application, you can use the same code in other applications, yet be able to extend them to match the new application's needs.

This concept is simply not present in iRMX. Instead, iRMX provides users with the ability to create new objects composed of aggregates of existing objects. In addition, the same mechanisms used by OS type managers for interacting with the operating system are also available for user-written type managers. Each type manager, however, must define its own set of system calls (the API), and it must make explicit calls to the appropriate type managers to manipulate its object's subobjects. An object that consists of other objects is called a *composite object* in iRMX terminology. A type manager for composite objects reuses code in the sense that it makes calls to the type manager functions for component objects to manipulate those component objects, but it does not inherit those functions in the object-oriented sense of the term.

6.3.3 Message passing

A scheme commonly used by object-oriented systems, but not unique to them, is the inversion of the relationship between objects and their API to produce what is called a *message-passing system*. In a message-passing system, the application sends a message to an object telling it what operation to perform, rather than calling a function provided by an object type's API to manipulate the object. The object itself is no less encapsulated by this approach; it simply provides a convenient mechanism for finding the code to perform an operation in the case of derived classes. The Objective C lan-

guage is an example of an object-oriented language based on this message-passing model, and the documentation provided with NeXT computers provides a good description of its implementation.

iRMX supports message passing as an important mechanism for tasks that share information and synchronize execution with each other. However, no relationship exists between this feature of the operating system and the message-passing mechanism of some object-oriented systems. All operations on an iRMX object are performed by passing the object (a token for the object) to a type manager function rather than by passing some sort of function, in the form of a message, to an object.

6.4 Survey of iRMX Layers

Before discussing the Nucleus layer in more detail, it is useful to have an overview of all of the iRMX layers. These OS layers are the following:

- Nucleus
- Basic I/O System (BIOS)
- Extended I/O System (EIOS)
- Application Loader (AL)
- Human Interface (HI)
- Universal Development Interface (UDI)
- Shared run-time library for C (introduced in chapter 4)

Each layer provides one or more type managers (object types and associated system calls) and, possibly, other system calls that provide services not part of any type manager. All of a layer's system calls are implemented as subroutines that occupy the area of primary memory called system memory. The code for the system calls is loaded into system memory when iRMX first runs and remains resident as long as the operating system runs, i.e., until the computer is turned off or rebooted.[5]

The idea of the entire operating system always being resident in memory contrasts with many other operating systems, which reduce demands on primary memory space by making certain parts of the operating system transient. That is, parts of the operating system in RAM can be overwritten by application programs or other parts of the operating system, and then reloaded from disk when needed again. This strategy can be very ef-

[5]System memory can be implemented as read-only memory. In this case, the code for the system calls is burned into ROM chips once and does not need to be loaded when power is applied to the system. In many embedded systems, the application's code is loaded into memory at the same time as the OS. These systems also encompass the case of application code burned into ROM.

fective for improving overall throughput for many types of systems, but it is generally inappropriate for a real-time system such as iRMX. The reason this strategy is inappropriate is because of the time required to bring parts of the operating system into memory introduces an indeterminacy into the response time of the system that would be unacceptable for real-time work.

6.4.1 The Nucleus

The Nucleus layer provides the system calls that

- Support the fundamental object-based nature of the operating system.
- Provide object types for intertask synchronization and communication.
- Manage the connections between the hardware and software of the interrupt system.
- Create new object types.
- Add new system calls to the system.

These features of the Nucleus are described later in this chapter. The ability to create new object types and to add system calls are particularly relevant to this survey of iRMX layers because those functions are used by all the other parts of the operating system to establish themselves as iRMX layers. Because system calls are available to application programmers as well as the layers supplied with the operating system, application programmers can easily add their own layers to the operating system or replace layers that do not match their requirements with custom versions that do. Chapter 10 tells how to use these facilities to add customized layers to iRMX.

The discussion of the Nucleus layer later in this chapter will include a description of the three fundamental iRMX object types: memory segments, jobs, and tasks. The other types of objects managed by the Nucleus include mailboxes, semaphores, regions, extensions, and composites, which are discussed in chapter 7. Nucleus functions for interrupt management are covered in chapter 9 when the topic of adding I/O device drivers to an iRMX system is explained.

Nucleus operations must meet both speed and robustness criteria to provide a sound basis for real-time applications. The iRMX III and iRMX for Windows Nucleus layer is implemented using the iRMK real-time kernel for some of its fundamental operations. iRMK functions can be called directly by applications that need to do so for performance reasons. Such applications, however, lose some of the robustness provided by the Nucleus' object-based design. iRMK is also an object-based system, but it is easier for errors in applications that use iRMK functions to compromise the system's integrity than for those that restrict themselves to Nucleus

functions. The iRMK functions available to iRMX applications are in the following categories:

- Task management.
- Time management.
- Mailboxes and semaphores.

Task management. Applications can use iRMK functions to provide the explicit control over task scheduling that cannot be achieved using the Nucleus task management functions described in Section 6.5.3. iRMK functions exist to turn task scheduling on and off and to notify procedures every time a task switch occurs. Nucleus task objects are iRMK task objects, which is not exactly true for other iRMK object types.

Time management. iRMK functions can provide applications with millisecond-granularity timing functions, whereas the standard Nucleus granularity is 0.01 second. In addition to a *sleep( )* function analogous to the one provided by the Nucleus, iRMK provides functions for elapsed-time measurement and alarms, which are user-written routines called at specified times.

Mailboxes and semaphores. These objects are used for intertask communication and synchronization. These iRMK objects provide only subsets of the functionality provided by the corresponding Nucleus objects, as well as the Nucleus' region object type, but the iRMK functions operate faster. The Nucleus' task synchronization and communication functions are covered in section 7.2.

The iRMK functions available to iRMX applications are documented in the manual *iRMK Kernel for the iRMX Operating System.*

6.4.2 The Basic I/O System

The services provided by the Basic I/O System (BIOS) system calls fall into the categories of time-of-day management, data operations, and file system management. The inclusion of time-of-day management in the BIOS layer rather than the Nucleus, where task scheduling is based on real-time clock interrupts, is based on the fact that time-of-day clocks are typically implemented as I/O devices. If there is a battery-backed clock in a system, the BIOS obtains the current time of day by reading that device when the system initializes, and then maintains its record of the current time based on real-time scheduling of a BIOS task by the Nucleus. The BIOS system calls for setting and reading the battery-backed clock are called *rqsetglobaltime( )* and *rqgetglobaltime( )*, while the system calls for setting and reading the time of day maintained by the BIOS time-of-day

task (often called the local time) are called *rqsettime( )* and *rqgettime( )*. If no battery-backed clock exists in the system, the BIOS initializes the local time from the time and date at which the computer's file system was last accessed.

This section introduces some of the key concepts of the BIOS's data operations and file system management, which are covered in detail in chapter 8. In addition to code invoked directly by system calls, the BIOS includes the code for device drivers (software that acts as an intermediary between hardware device controllers and the remainder of the software that constitutes the system), and data tables that describe I/O device characteristics. These features of the BIOS are described in more detail in chapter 9. Some of the important features of the BIOS layer include those described in the following subsections.

The BIOS presents a device-independent interface to the code that calls its functions. The same system calls are used whether an I/O operation involves a disk, a tape drive, a file on a network, a terminal, a printer, or a custom device that has been incorporated into the system. Programs do not need to be changed to work with different devices.

I/O operations can be invoked from application programs or from any layer of the operating system. In all cases, however, actual I/O operations are performed by code in the BIOS layer of the operating system. For example, the system call *rqcsendcoresponse( )* used in the sample programs in chapters 3 and 4 is part of the HI layer of iRMX, but the HI implements the call by making calls to the EIOS layer, which in turn implements its functions by making system calls to the BIOS.

BIOS operations are generally performed asynchronously with respect to the program that invokes them. This means that a program can use a BIOS system call to add an item to the BIOS's queue of I/O operations to be performed without waiting for the operation itself to complete. At some later time the program makes another system call to determine if the operation has completed and its outcome. This feature allows a program to enter several operations into the BIOS's various work queues simultaneously and to overlap computation with I/O operations in general.

You can build a copy of iRMX that does not include the BIOS. Although all system calls that involve I/O operations require the BIOS layer for their operation, it is possible to build a copy of iRMX that does not include the BIOS or any other layer that depends on it. In this case, the application program itself must supply all the code for I/O operations.

The BIOS implements an iRMX object type called an I/O user object. This object type is used to help determine the access rights that programs have to

individual files and I/O devices, based on the identity of the user who invoked the program.

The BIOS implements a second iRMX object type called an I/O connection that is used for all I/O operations. BIOS I/O connections are analogous to file handles in DOS systems and file descriptors in Unix systems. The C run-time library maps file descriptors and I/O streams to iRMX connections for programs that use standard C functions for I/O. Two special library functions, *_get_rmx_conn()* and *_put_rmx_conn()*, in the C run-time library can be used by C programs to translate between file descriptors and iRMX connections if needed.

6.4.3 The Extended I/O System

Essentially, the Extended I/O System (EIOS) is a synchronous interface to the BIOS. Almost every BIOS system call has a corresponding EIOS system call that performs the same function, but the EIOS version provides a simpler interface to the application program. The simplification sometimes involves supplying default or built-in values for certain system call parameters, but the primary simplification is the synchronous interface.

When a program makes an EIOS system call, code in the EIOS initiates the I/O operation by making the corresponding BIOS system call. The EIOS code then waits for the BIOS operation to complete, checks the result of the operation for exceptional conditions, and only then returns to the calling program. That is, when control returns to a program that made an EIOS system call, all activity associated with the operation has completed. This means, for example, that any data buffer involved in a data transfer either contains new data (in the case of a read operation) or can safely be overwritten with other information (in the case of a write operation).

This fact might not seem remarkable because it is the normal mode of operation for programs that use the standard I/O facilities of most high-level languages, but it is important in the context of the asynchronous interface provided by the BIOS for iRMX. The problem with synchronous I/O is that EIOS calls constrain the degree of throughput that can be achieved by a single thread of execution. Those applications that must overlap computation with I/O or need to schedule concurrent I/O operations would use BIOS functions instead of EIOS functions. Where these features are not important, however, an application can benefit by using the simpler EIOS calls instead. Furthermore, EIOS system calls can be used to implement automatic overlap of I/O operations with application execution using techniques called *read-ahead* and *write behind*. If an application processes files sequentially, it can tell the EIOS to use its own buffers to implement this feature as described in chapter 8, providing the advantages of asynchronous BIOS interface along with the simpler form of synchronous EIOS calls.

The EIOS manages a type of iRMX object called an *I/O job*. An I/O job is a normal iRMX job as implemented by the Nucleus (discussed in Section 6.5.2) that has been augmented to have an I/O user object associated with it, along with certain other details. The EIOS also manages its own type of connection object based on BIOS connection objects that have been augmented with housekeeping information. The EIOS needs this housekeeping information to support its simplified interface to application programs. Section 7.4 covers I/O jobs in greater detail.

6.4.4 The Application Loader

As its name implies, the Application Loader (AL) loads application programs into primary memory for execution. One system call provided by the AL simply loads code into memory but does not start its execution. This system call requires only the BIOS layer of the operating system for its implementation. Other AL system calls create an I/O job for the program being loaded and start the program running; they require the EIOS layer of the operating system to create I/O jobs.

There are both asynchronous and synchronous versions of the AL system calls to create I/O jobs, which operate in ways analogous to the asynchronous and synchronous versions of I/O system calls. In the case of the AL, the asynchronous version of the call returns to the calling program before the application program is actually loaded into memory, and the synchronous version returns to the caller only after the new program has been loaded successfully. Both versions, however, return before the loaded program completes its execution, and a mechanism is provided for determining when the loaded program has completed and for checking its final status.

The files loaded into memory by the AL must be in standard STL object module format, as prepared by the binder (discussed in chapter 3), and can include either memory overlays or monolithic program images. The AL layer does not provide a type manager for any iRMX object types.

6.4.5 The Human Interface

Three types of system calls are provided by the Human Interface (HI) layer of iRMX: (1) command invocation, (2) command-line parsing, and (3) console I/O.

When an iRMX system starts running, the HI determines how many terminals are available for user access to the system by reading the `:config:terminals` file. For those terminals with static logon users, the HI logs a particular user onto the system automatically, creates an I/O job for the user, and has that job start running the Command Line Interpreter (CLI).[6] For terminals with dynamic logon users, the HI issues a prompt for

[6]Although the CLI is typically loaded into system memory with the resident layers of the system, it is not considered a layer itself because it does not provide any system calls to the system, which the other layers do.

a user's name and password, and then creates an I/O job for the user when the logon process is successful.

When the HI creates an I/O job for a user, it augments that job with information that allows the job to use HI system calls to perform console I/O and command line parsing. An I/O job that has been augmented in this way is called an *HI job*, although this is not standard iRMX terminology. An example of the HI's console I/O system calls was *rqcsendcoresponse( )*, seen in the sample programs in chapters 3 and 4. The command-line-parsing system calls allow an application program to determine what the user typed on the command line that started the program running and to interpret that information in the standard way outlined in chapter 2 (input file list, preposition, output file list, parameters).

Like the AL, the HI layer does not provide a type manager for any iRMX object types.

6.4.6 Universal Development Interface

The Universal Development Interface (UDI) layer provides a way of implementing portable application programs. This same layer of software has been implemented for iRMX, DOS, VAX/VMS, and Unix System V, so programs that make no system calls except those supplied by this layer can be run unchanged on all of the operating systems for which the UDI programming interface is available. Intel has used this interface in implementing all of its x86 software development tools so that just a single copy of each tool runs on all the different supported development platforms. For VAX/VMS, the development tools had to be built to execute using DEC's VAX machine language instructions, but for operating systems that run on the x86 architecture, the same binary file can run under the different operating systems. The UDI is the mechanism used to implement the common DOS-hosted and iRMX-hosted editors, compilers, and binders mentioned in chapter 3.

Two things are of particular note in the UDI. First, most iRMX applications are not intended to be portable. The main reason for developing them to run under iRMX is to take advantage of the specific real-time features of this operating system. Thus, the UDI is not used for real-time applications, but rather is reserved for use with utility programs and development tools.

Second, there are alternatives to the UDI for portable applications. A primary example is the use of a standard high-level language with its own run-time library, such as C using the POSIX.1 library, which was introduced in chapter 1. POSIX.4 even offers the hope for portable real-time applications at the source-code level. Presently, the advantages of the UDI are that it provides a simpler interface to the I/O system than the EIOS, and it provides a mechanism by which a single binary file can run on both DOS and on iRMX.

6.5 iRMX Fundamental Objects

The previous survey of iRMX layers has suffered from the use of some vague and as-yet-undefined terms. In particular, just what is a job, and what does it mean to augment it to make it into an I/O job or an HI job? Also, how is a BIOS I/O connection object augmented to become an EIOS connection object? This section presents the background information needed to solidify these concepts.

Three fundamental object types are provided by the iRMX operating system: memory segments, jobs, and tasks. These object types could be presented here in the manner appropriate to all object types, by describing the set of procedures available for working with these objects (the API for the objects). For these three types of objects, however, it is important first to see how they relate to all the other object types managed by the operating system, and that is the primary focus in this section. These object types, as well as the remainder of the object types managed by the Nucleus, are covered from the application programmer's viewpoint in chapter 7, which surveys some of the system calls provided by the various layers of the operating system.

For an iRMX system to function, there must be objects, and there must be operations performed on those objects. There are computations that applications and the operating system itself perform that have nothing to do with objects, but without objects these other computations would operate in a vacuum and thus serve little or no useful purpose.

To have objects, memory must exist to hold the information about the objects, a mechanism must exist to allocate the memory used by objects in an orderly way, and an execution entity must exist that can cause objects to be created, manipulated, and disposed of. In iRMX, the *memory segment* object type provides the memory that holds objects, the *job* object type provides the basis for allocating memory for objects (jobs are said to own the memory occupied by objects), and the *task* object type provides the execution entities of the system. These three object types are described next.

6.5.1 Memory segments

There are only two type manager functions (system calls) for memory segments. One function creates a segment, and the other deletes a segment. Assuming `myseg` has been declared to be of type `TOKEN`, and `Status` has been declared to be of type `WORD`, the following statement makes an iRMX system call to create a segment:

```
myseg = rqcreatesegment (5280, &Status);
```

This is a C statement, but it would be the same in PLM if the `&` was changed to `@`. The first argument is the size of the segment, in bytes. For iRMX I and

II, the maximum size is 64K, but for iRMX III, it is 4G. The second argument is a pointer to a 16-bit unsigned value that will be set to a completion code for the system call. A value of 0 means the call completed successfully.[7]

Unlike any other type of iRMX object, memory segments are completely unstructured as far as the operating system is concerned, and there are no restrictions on what an application does with the memory contents. For this one type of object, the only reason for creating an object is, in the object-based sense of the term, to cheat with it!

Figures 6.2 and 6.3 present equivalent PLM and C programs that illustrate a possible use of an iRMX segment object. The main program creates a segment, reads characters into it from the keyboard, and passes it to a subroutine that displays the characters on the screen. The subroutines access the information in the segment by either explicitly or implicitly using the token received from *rqcreatesegment( )* as the selector part of a pointer to the information. The code seems a bit messy in the subroutines because of the logic to check that the message to be displayed will actually fit into the memory reserved for an output buffer. These programs should be viewed only as demonstrations of how segments can be created and accessed rather than as illustrations of proper programming style for iRMX. There are certainly much more efficient ways to share information between a main program and a subroutine than to incur the overhead of creating an iRMX memory segment object!

Fundamental about memory segment objects is that all other iRMX objects are constructed from them. For example, if a program creates a job object, the type manager for jobs creates a memory segment to hold the data structures that it maintains about the job, initializes the data structures in the segment, and returns the selector for the segment as the token for the job. Likewise, if a program creates a task object (or a mailbox, a semaphore, or any other type of iRMX object), the type manager for tasks (or mailboxes, semaphores, or whatever) calls *rqcreatesegment( )* to obtain the memory in which the type manager places information about the task (or mailbox, semaphore, or whatever), and returns the selector for the segment as the token for the new object.

When an application needs to do something with one of its objects, it passes the token for the object to the appropriate type manager function, and the type manager's code uses the fact that the token is a selector to access the data structure for the object in much the same way that *mySub( )* accessed the contents of the input buffer segment in Figures 6.2 or 6.3. It is

[7]For more information on how to code this and all other system calls, you should consult either the on-line help facility provided with iRMX for Windows, or the reference manual for the appropriate part of the operating system, such as the *iRMX System Call Reference*, volume 9 of the iRMX for Windows documentation set.

Figure 6.2 Sample PLM program illustrating the use of an iRMX segment.

```
/***>  segexamp.plm  <***********************************
 *
 *   Sample program to illustrate the use of an iRMX segment
 *
 ************************************************************/
$title ('Sample Program Illustrating Use of an iRMX Segment')

segexamp: DO;
$include (segexamp.ext)
DECLARE
        SEGSIZE     LITERALLY   '81',
        myseg       TOKEN,
        Status      WORD_16;

/*
 *   Subroutine to display an iRMX string contained in a segment.
 *   ------------------------------------------------------------
 */
segsub: PROCEDURE (theSeg);
  DECLARE
        theSeg                  TOKEN,
        theBuf BASED theSeg (1) BYTE,
        outBuf (81)             BYTE;

  CALL movb (@(11, 'You typed: '), @outBuf, 12);
  CALL movb (@theBuf(1), @outBuf(12), theBuf(0));
  IF (outBuf(0) + theBuf(0)) < size (outBuf) - 1 THEN
    outBuf(0) = outBuf(0) + theBuf(0);
  ELSE
    outbuf(0) = size (outBuf) -1;
  CALL rqcsendcoresponse (NIL, 0, @outBuf, @Status);
  RETURN;
END segsub;

/*  Main program starts here
 *  ------------------------
 *  Read a string from the keyboard into a segment, and pass it
 *  to a subroutine for display.  Assume that system call errors
 *  are checked by the default exception handler.
 */
myseg = rqcreatesegment (SEGSIZE, @Status);
CALL rqcsendcoresponse (buildptr (myseg,0), SEGSIZE - 1,
     @(16,'Type something: '), @Status);
CALL segsub (myseg);
CALL rqexitiojob (0, NIL, @Status);
END segexamp;
```

not cheating for the type manager to access the memory for an object in this way; rather, that is its purpose. Such accesses correspond to the lower set of arrows in Figure 6-1.

At this point, what an application might want to do with an object has not yet been covered, but that is because other types of iRMX objects have not yet been considered. As each object type is introduced, the functions that can be performed with objects of that type need to be presented as well.

Figure 6.3 Sample C program illustrating the use of an iRMX segment.

```
/***>  segexamp.c  <************************************************
 *
 *  Sample program illustrating the use of an iRMX segment
 *
 ************************************************************************/
#pragma title ("Sample Program Illustrating Use of an iRMX Segment")
#include <rmxc.h>
#include <string.h>

#define SEGSIZE 81

TOKEN   mySeg;
WORD    Status;

/*  Main program starts here
 *  ------------------------
 *  Read a string from the keyboard into a segment, and pass it
 *  to a subroutine for display.  Assume that system call errors
 *  are checked by the default exception handler.
 */
int
main (int argc, char * argv[]) {

mySeg = rqcreatesegment (SEGSIZE, &Status);
rqcsendcoresponse (mySeg, SEGSIZE - 1,
  udistr ("Type something: ","Type something: "), &Status);
mySub (mySeg);
rqexitiojob (0, NULL, &Status);
}

/*
 *  Subroutine to display an iRMX string contained in a segment.
 *  ------------------------------------------------------------
 */
void
mySub (TOKEN theSeg) {
char    outBuf[81] = "You typed: ";
char    *theBuf;

  theBuf = theSeg;
  strncat (outBuf, &theBuf[1],
    (sizeof (outBuf) > (strlen(outBuf) + theBuf[0])) ?
    theBuf[0] : sizeof (outBuf));
  udistr (outBuf, outBuf);
  rqcsendcoresponse (NULL, 0, outBuf, &Status);
  }
```

6.5.2 Jobs

Each object in an iRMX system is owned by some job. This relationship has three important ramifications.

1. Each job has a memory pool. The iRMX memory management policy is to allocate a contiguous block of free space memory to a job when it is created. Free space is an area of RAM that has not yet been turned into seg-

ments in the sense that there are no descriptors in the GDT or LDTs that could be used to access this memory. When a memory segment or other object belonging to a job is created, a new descriptor is constructed that defines a piece of this memory pool as a segment, and the record of the job's memory pool (one of the data structures in the job object itself) is updated to reflect that this piece has been taken from its pool.

2. Objects cease to exist when their owning job is deleted. Because the memory for an object is allocated from the memory pool for the job that owns the object, and because deleting a job implies deleting its memory pool, all objects that belong to a job are automatically deleted when the job is deleted.

3. There is a tree-structured hierarchy of jobs. All objects are owned by jobs and jobs themselves are objects, implying that jobs are also owned by jobs, which is true. Each job has one parent job and zero or more child jobs. There is one distinguished job, called the root job, that has no parent. This parent-child relationship with a root node gives rise to the tree-structured hierarchy of jobs in an iRMX system. The form of an iRMX job tree can be very dynamic as jobs are created and deleted during the time a system is running. In practice, however, much of the job tree is quite static.

A corollary of the tree-structured hierarchy of jobs is that there is a tree-structured hierarchy of memory pools. Because jobs are owned by other jobs, it makes sense that the memory pools for jobs should be allocated from the memory pools of their parent jobs; however, this is only partially correct. There are two memory pool values associated with each job, called the *minimum* and *maximum* memory pools. When a job is created, a contiguous block of memory equal to the size of the job's minimum pool is allocated from its parent jobs' pool. During the job's lifetime, it can request additional memory by means of the *rqcreatesegment()* system call or by creating other objects whose type managers call *rqcreatesegment()*. These segments will be taken from the job's minimum memory pool if possible, but, if the job's minimum memory pool is either exhausted or simply too fragmented to provide a contiguous block of memory to satisfy the request, the segment can be allocated from the memory pool of the job's parent by a process called *borrowing*.

If the parent job's memory pool is too small or too fragmented, the segment can be allocated from the grandparent job's pool, and so on up the job tree to the root job. A job's maximum memory pool is the limit for how much memory can be borrowed from a job's ancestors. A significant difference between a job's minimum pool and its maximum is that the former is always contiguous, but the latter can be scattered among several ancestor's pools.

Memory segments must always be internally contiguous, but this requirement might be impossible to meet for very large segments when jobs' memory pools become fragmented, even though the total number of bytes

needed for a large segment might be available. As mentioned in chapter 5, iRMX III could use the paging facilities of the 80386 and later microprocessors to map disjoint pages into contiguous segments to accommodate applications' needs for very large segments, but this feature has not been implemented at the time of this writing. Recall that paging is done by the microprocessor after segmentation, which would make it transparent to an application that references the contents of a segment through the standard selector and offset pointer mechanism (logical addresses). Recall also that segments larger than 1 MB must be multiples of 4 KB in size.[8] Thus, a proper match always exists between the size of large segments and the page size used by the processor, which is also 4 KB, and no additional memory fragmentation or wastage occurs due to the use of the paging mechanism in this context.

It might help to consider the sequence of events that occurs when an iRMX system starts running. First, a copy of the entire operating system is loaded into an area of memory known as *system memory*. This loading process can be done by a bootstrap loader that reads the operating system from a local or network disk file, or the operating system can be burned into PROMs so that it is available as soon as electrical power is applied to the system. The code loaded into system memory consists of all the type managers initially defined (others can be added as the system is running), a GDT that includes call gates for all the type-manager functions that have been loaded, plus initialization code that starts executing as soon as control passes to the operating system.

In addition to system memory, there must be some RAM that can be used for the dynamic memory requirements of the various jobs in the system. This RAM is known as the *free-space memory*, and it is managed by a software module in the Nucleus called the Free Space Manager (FSM). Free-space memory cannot be accessed initially because no descriptors for segments in this part of memory exist yet.

The initialization code first calls the FSM to create a segment that will contain the root job object, and it fills in that segment with the information about the job. The job information of concern here is the memory pool information for the root job, which is set up to have a small minimum memory pool as well as a maximum memory pool limited only by the amount of physical memory available. In essence, the root job owns the entire free-space memory, and it is conceptually equivalent if its pool minimum is set to be equal to the size of the entire free space area. The situation at this point is shown in Figure 6.4. The root job has been created, and its maximum memory pool encompasses all of FSM. The root job object occupies a memory segment that has been allocated from free space memory and charged against its own maximum memory pool.

---

[8]Only 20 bits are available in descriptors for the segment size, so 12 zeros are appended to the 20-bit limit field in the descriptor for such large segments.

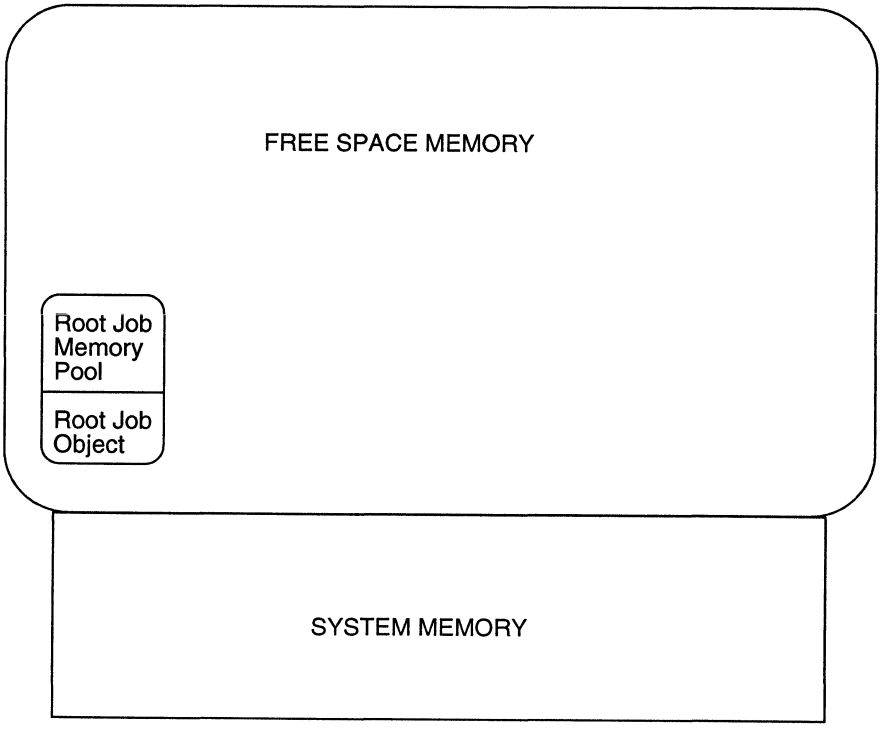


Figure 6.4 Primary memory during system initialization.

The lower part of memory is shown containing system memory, and the upper part is free-space memory, all of which belongs to the root job. A piece of the free-space memory has been allocated as the root job's minimum memory pool, and a segment has been created in the root job's minimum memory pool to hold the root job object itself. The memory for an object is normally allocated from the owning job's memory pool, but this one object must be treated specially because of the distinguished nature of the root job itself; it has no parent, so it owns itself.

After the root job has been created, initialization continues with the creation of a set of jobs that are the immediate children of the root job. These jobs are called first-level jobs, and their memory pool requirements are defined when the operating system is configured. These first-level jobs consist of jobs needed by the various layers of the operating system as well as application-specific jobs. These jobs are created in a well-defined sequence, which is also defined when the operating system is configured, so that any job that depends on another job's existence will be created only after that other job has been created and has completed execution of its initialization code.

Each first-level job is given a minimum memory pool which is taken from the root job's pool. Figure 6.5 represents the situation in which there are

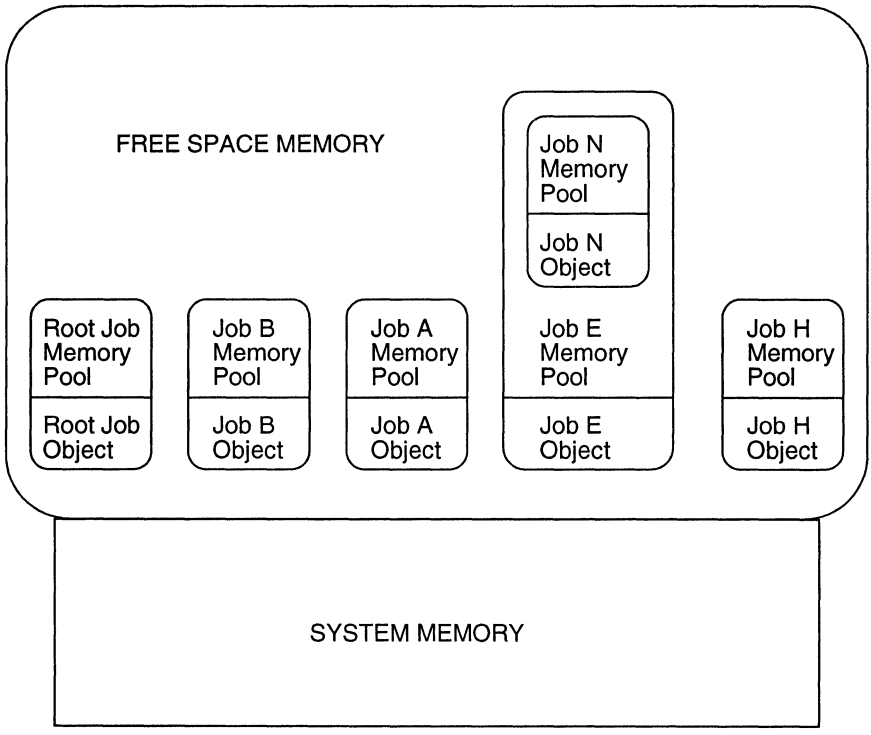


Figure 6.5 Primary memory after the root job has created first-level jobs B, A, E, and H.

four first-level jobs, named B, A, E, and H, in the figure. To make matters interesting, Job E is shown having created a child job of its own (Job N), which has taken some of Job E's minimum pool. Jobs B, A, E, and H have been allocated memory pools from the root job's maximum pool (FSM). Job N has had its memory pool allocated from job E's pool. Jobs B, A, E, and H can all borrow memory from the root job's maximum pool, and job N can borrow memory from either its parent (job E), or its grandparent (the root job).The letters used to name the jobs in this figure are meant to be suggestive of the jobs' natures, which is explained next. The relative sizes of the boxes, however, do not indicate the relative sizes of the memory pools or job objects in the figure.

Among the first-level jobs created are jobs for some, but not all, of the layers of iRMX. In Figure 6.5, jobs B, E, and H represent jobs created for the BIOS, EIOS, and HI layers of the operating system. Job A represents an application-specific job created after the BIOS job and before the EIOS job. One might assume that this particular application needs to be able to make Nucleus and BIOS system calls, but not EIOS or HI calls, based on the order in which it was created relative to the other jobs. Job N, the child job of the EIOS, is an example of a job created by a first-level job during

system initialization. An example of a job like this on some systems is the iRMX-Net job, which is why it was named N in this example.[9]

In a typical iRMX system, each first-level job is given an unlimited maximum memory pool. The effect of this is to allow the root job and all first-level jobs to compete equally for use of free space memory. First-level jobs need to draw on the resources of free-space memory for two different reasons. One regards the unique requirements of the HI, and the other is based on the relationship between first-level jobs and iRMX layers. These two matters are discussed next.

The HI job's memory pool. For those configurations that include the HI layer of the operating system, the section of the job tree rooted at the HI job is the most dynamic segment of the tree. The HI creates a new job each time a user logs on to an iRMX system, and another job is created every time a user runs an HI command or certain CLI commands (such as *background*). All of these jobs endure only during execution of the specified command or, in the case of a logon job, until the user logs off.

All of these descendant jobs of the HI compete for resources within the free-space memory, since the HI job's minimum memory pool is not large enough to meet any but the smallest attempts to borrow from it. To provide some degree of fairness in multi-user systems, the HI is normally configured to supply each logon user job with a memory pool large enough to run most commands and development tools, and a maximum memory pool small enough to ensure that some free memory will be available for other users who log on later. These memory pool settings are established by the files in the `:config:users` directory. For each user, there is a file in this directory that specifies the user's minimum and maximum memory pool, as well as other information, such as the pathname to the user's home directory. For example, a system with 4 MB of free space memory might set each user's minimum memory pool to 512K and maximum memory pool to 2M. This way, there would always be enough memory for at least two users to log on without any problem, and a system with light memory demands could support as many as eight users simultaneously.

The relationship between first-level jobs and layers. The second type of demand on free-space memory is made directly by other first-level jobs besides the HI. To understand these demands, first consider the implications of the hierarchical nature of the job tree and memory pools on the lifetimes of iRMX objects.

A crucial concept for understanding the iRMX operating system is the following: when a job is deleted, all of its memory pool is returned to the

[9]The iRMX-Net layer of the operating system is described in chapter 11.

job(s) from which it was obtained, and all memory segments that had been established within that memory pool are deleted from the GDT. This means, without exception, that *all objects owned by a job cease to exist when the job terminates.* A corollary to this rule follows from the fact that it is possible for one job to obtain a token for an object that belongs to another job. In general, objects can be shared across jobs, but if the job that owns an object terminates, any further attempt to use the token for that object from another job will result in an error[10].

So, what does this have to do with first-level jobs for OS layers, and why do some layers have first-level jobs, and other layers do not? The answer is that some layers need to create permanent dynamic objects, and others (the AL and UDI, in particular) do not. The AL and UDI provide system calls that might result in the creation of objects, but those objects are allocated from the memory pool of the application program that made the system call. The other layers, however, create objects that must continue to exist after the job that triggered their creation terminates. There is a first-level job for each of these other layers so that they can act as the owners for these more-permanent objects.

It might not make much sense to discuss these permanent objects because you do not know what each of the layers does yet, but a few of the objects are described briefly here to explain the rationale for first-level jobs.

The BIOS layer creates system-wide objects every time a device is attached. Even though the action of attaching a device is invoked from a particular job, such as the job spawned by the HI when the user enters an *attachdevice* command, the attachment itself must exist beyond the lifetime of the creating job (the job that runs the *attachdevice* command). Thus, the objects associated with a device attachment (which include an object called a *device connection*, a task for communicating with the device driver software for the device, and perhaps some memory segments for sharing information between the BIOS and the device driver) must be created dynamically, but must continue to exist after the job that calls the BIOS terminates. The solution is for the attachment to be performed by, and hence to be owned by, a permanent job, namely the BIOS job.

Some layers of the operating system create type managers for new object types beyond those provided by the Nucleus. For example, the BIOS layer supplies the type manager for an iRMX object type called a *user object.* (A user object is simply a memory segment containing a list of ID numbers to be associated with a particular user of the I/O system. This list of user IDs is used by the BIOS to determine access rights to files that a program tries to open. You can see a display of your user object by typing the *whoami* command at the CLI's prompt.)

---

[10]Not all objects can be shared across jobs. Objects called *connections*, which are managed by the BIOS and EIOS layers of iRMX, may not be shared across jobs. This restriction may well extend to other types of objects as the operating system evolves.

The idea introduced here is that type managers can be added to iRMX by other layers of the operating system besides the Nucleus. An example is the type manager of the BIOS for user objects. Application programs can add their own type managers to the system that will operate the same as the type managers supplied by the operating system itself. The mechanism for adding a type manager involves creating a type of iRMX object called an *operating system extension* (OSE). When a program (either an OS layer or an application program) creates an OSE object, it specifies a type code for the new object type and, optionally, a deletion mailbox for objects of the new type. The new type manager also creates new system calls for managing objects of the new type so that programs can create, delete, and manipulate objects of the new type. Other parts of the Nucleus use the information specified when the OSE was created to provide an orderly mechanism for deleting a job's objects when the job is deleted.

Thus, a second rationale for having a job associated with an operating system layer is to provide an owner for the OSE objects that a layer needs if it provides type managers. The code for the type managers (the code for the system calls that can be used to create and manipulate object instances of the new type) occupies system memory, but the OS extension object associated with each type manager is created dynamically and owned by the layer's job. The reason the AL and UDI do not need jobs is that they do not provide type managers.

The EIOS layer, on the other hand, needs a job to own the two object types that it creates, logical names for devices (introduced in chapter 3) and a composite object type called I/O jobs. An I/O job consists of a job object, a user object, and some other objects not important here. What is important at this point is that it follows immediately from the existence of these type managers that the EIOS layer needs a corresponding job to own the OSE objects for these types.

There are two reasons for having a job for the HI. One reason is for managing the memory pools for users and their applications already presented. The other is to provide an owner for the tasks that the HI provides. The HI job owns a task for each terminal on which users can log on to the system. When a user logs on, the task associated with the terminal creates a child job of the HI that acts as the unique parent for all jobs created by the user's commands. In addition, a task is used to implement the *sysload* command that allows programs to be loaded and to continue running after the user who issued the *sysload* command logs off the system. Commands run by *sysload* are run as child jobs of the HI job rather than as child jobs of the user's logon job. That is, there is an HI job to act as a permanent owner for the *sysload* task and the logon tasks.

6.5.3 Tasks

Tasks are threads of execution. They are a fundamental object type in iRMX simply because without them no code can be executed. Without

them, the type managers provided by the operating system or its extensions could not be called, and no objects could be created, manipulated, or deleted. Without them, nothing would happen.

Until now, the discussion of jobs and programs had to be a bit imprecise. Although jobs own objects, they do not create objects. An object is created when a task executes a system call that executes a type manager's function to create the object. So how does a job own an object that a task creates? The job that owns the task that creates the object owns the object. How does a job get to own a task? An initial task is automatically created for every job created. Except for being the first task created for the job, this one task is indistinguishable from all other tasks that the job might own during its existence. As mentioned in chapter 5, the iRMX II and III task managers allocate a memory segment structured as a protected-mode Task State Segment (TSS) for every task in the system.

No hierarchy exists among iRMX tasks. All objects created by a task belong to the job that owns the task rather than to the task itself. If one task creates another task the two tasks are siblings: they are both owned by the same job and have no other particular relationship to each other. If a second task creates a third task, that task is equivalent to both the initial task and the task that created it. If a task creates a job, that job is owned by the creating task's job, and the new job's initial task is owned by the created job. The creating and created task in this case simply have the relationship to each other of "do not belong to the same job," and there is nothing distinctive about the fact that one belongs to a job that is the parent of the other's job.

An important distinction to make is the difference between a task and a procedure. A task is a thread of execution; a procedure is a piece of executable code. The proper terminology is to say that a task enters a procedure, executes the procedure's code, and exits. (Of course, a procedure might contain an endless loop, in which case the task would never exit.) More than one task can enter a procedure at the same time, in which case the procedure must be re-entrant — it must allocate separate copies of all incoming parameters, return addresses, and local variables for each task that enters. This allocation is accomplished by giving each task its own stack segment in memory, and allocating memory for parameters, return addresses, and local variables on the calling task's stack[11].

All functions in the C language are re-entrant by default, but PLM procedures and functions must explicitly be declared re-entrant if they are to be used in this way. Both languages use the stack for passing parameters and holding return addresses; the issue is whether local variables are part of the stack frame (activation record) or stored in a static data segment.

---

[11]Refer to chapter 5 for a description of how the x86 uses a stack segment for subroutine activation records containing this information.

When a job or task is created, one of the parameters for the call to the type manager's create function is the address in memory where the task (the initial task in the case of a job) is to begin execution. Unlike other operating systems that require tasks to execute within the address space of the parent process, iRMX places no restriction on what memory can contain the procedures to be executed by a job's tasks. In particular, no requirement exists for any task to start executing at an address within its job's memory pool. In fact, this would be impossible to arrange for a job's initial task because the creating task cannot know the address of a job's memory pool before the job is created. For example, the initial task for an HI command starts executing an AL procedure that loads the command's code and data from a file into the job's memory, and then branches to the initial instruction in the newly loaded code segment.

Once a task has been created, it can execute procedures any place in memory that contains code. Unlike other operating systems that require some type of context switch when a thread of execution makes a system call, iRMX tasks execute system call procedures directly without any context switch at all. The one exception to this principle is asynchronous processing by the BIOS, which is described in chapter 8. The mechanism for branching to the proper location in system memory to execute system calls will be discussed later in this chapter.

6.5.4 Examining iRMX objects

As mentioned in chapter 3, iRMX provides an extension to the debug monitor called the system debugger (SDB) that can be used to examine iRMX objects interactively. The SDB is code placed in system memory with the rest of the operating system when the OS is first loaded. The SDB code is connected to the same command interpreter as the debug monitor (the code that allows interactive examination of hardware facilities), so it can be accessed from either the same prompt as the monitor ("." or ". .", depending on whether you are using iRMX I or a protected-mode version of the operating system) or from the SoftScope command line. Several SDB commands exist for examining objects and certain other data structures maintained by the operating system. They are fully documented in the *iRMX System Debugger Reference Manual.* The *SoftScope III Debugger User's Guide* (volume 13 of the iRMX for Windows documentation set) tells how to use them from a SoftScope session.

SDB commands help application programmers know about the objects created by their program so they can debug effectively, but the programmers should not have to cheat to do so. One of the most commonly used SDB commands is *vt*, which stands for *view token.* Given a token for an object (either the numerical value of the selector for the object or, in SoftScope, the name of a variable that holds a token), *vt* will display all of the

essential information about the object in a format suitable for human examination, but without compromising the encapsulation principle of an object-based system. Think of *vt* as an interface to the type managers for all iRMX object types. It follows from the material presented previously on object-based systems that any changes to the internal representation of an object type by the OS designers must include parallel changes to the routines in the SDB that display information about that type of object.

SDB commands can also be very instructive for people interested in better understanding how iRMX works. If you want to know what information is maintained by the OS for jobs, give a *vt* command with a token for any job object. If you want to know what is known about tasks, give a *vt* command with a token for a task object. For example, Figure 6.6 is the output of a *vt* command for a job, discussed more fully in the next section.

Some SDB commands that might be useful to try out at this time include the following:

vj View Jobs. The hexadecimal value of the token for every job in the current iRMX job tree is displayed with indentation showing the tree structure. You can limit the display to one branch of the tree by giving the token for a particular job as an argument. Within a level, jobs are displayed from the top down in the reverse order of creation (HI job on top, BIOS job on bottom for first-level jobs, for example).

vo View Objects. Given a token for a job, this SDB command displays a list of all the objects owned by that job, arranged by object type.

vh View Help. This command lists SDB command names and syntax.

Figure 6.6 Display produced by the SDB *vt* command, given the token for the root job of a system running iRMX for Windows.

```
Object type = 1  Job1

Current tasks   0003          Max tasks     ffff     Max priority     00
Current objects 0005          Max objects   ffff     Parameter obj    0000
Directory size  00c8          Entries used  0018     Job flags        0000
Except handler  0280:00008850 Except mode   00       Parent job       0000
Pool min        00021fff      Pool max      00021fff
Initial size    00021fff      Borrowed      00000000

   Byte range    | Number chunks  | Largest chunk  |  Total memory
-------------------------------------------------------------------
     22-44H      |    00000001    |    00000030    |    00000030
     44-84H      |    00000000    |    00000000    |    00000000
     84-200H     |    00000000    |    00000000    |    00000000
     200H-1K     |    00000000    |    00000000    |    00000000
     1K-2K       |    00000000    |    00000000    |    00000000
     2K-4K       |    00000000    |    00000000    |    00000000
     4K-8K       |    00000000    |    00000000    |    00000000
     8K-32K      |    00000000    |    00000000    |    00000000
     + 32K       |    00000002    |    00035760    |    00044dd0
```

6.6 More about the Nucleus

With the fundamental nature of memory segments, jobs, and tasks having been introduced, we now further examine some other fundamental features of the Nucleus layer, starting with more information about job and task management.

6.6.1 Job management

This section provides more information about how a job's memory pool is managed by the FSM and how a data structure called an *object directory* can be used for sharing access to objects across tasks or jobs.

Memory management. The basic principles of memory management were already introduced in the previous discussion of jobs and memory segments, but a few details can be elaborated on here.

The FSM uses data structures within a job object as it manages the job's memory pools. Without knowing the actual organization of these data structures or the actual algorithms used by the FSM (in keeping with the encapsulation principle of object-based design), it is nonetheless possible to deduce some useful information about management of a job's memory based on documentation and a bit of experimentation with the SDB. Figure 6.6 is the display produced by the SDB *vt* command for the root job of a system running iRMX for Windows.

The lower portion of Figure 6.6 shows some of the information that the FSM uses. The Pool min, Pool max, Initial size, and Borrowed all refer to the job's memory pool, with values given as numbers of 16-byte units called paragraphs[12]. The three columns at the bottom of the display show how the FSM keeps track of the memory in a job's pool. It keeps a list for each of several chunk sizes. (You can find out that the lists are implemented as doubly linked lists by consulting the SDB manual, but that kind of knowledge comes close to cheating!) For each chunk size, there is a list of contiguous bytes of memory that are of that size range. Presumably, although not shown by the *vt* display, the FSM stores the actual starting address and length for each chunk on the list. Note that all the values shown in the chunk-management part of the display are in bytes rather than paragraphs.

The algorithm actually used for allocating memory segments is not published by Intel, but what follows should be a reasonable approximation:

1. Search for the first nonempty list that might contain a chunk large enough to accommodate a segment of the desired size. Start with the list

[12]Paragraphs have no architectural significance in protected mode, but a paragraph in real mode is the minimum spacing between segment base addresses because of the way base addresses are calculated. This topic is discussed in chapter 5.

for the chunk size range that includes the requested segment size, and continue to the lists for larger chunks until either a nonempty list is found or the set of lists is exhausted.

2. If no list is found in the current job, repeat step 1 for the parent job as long as the current job's pool max has not yet been reached and as long as there is a parent job. If unable to continue, return the condition code E_MEM to the caller.
3. Search the selected list of chunks for one large enough to hold the segment, skipping over chunks within the range of the list but too small to hold the segment. This step could use either a first-fit or best-fit rule. The former would use the first chunk on the list that is large enough, and would execute somewhat faster, whereas the latter would search the entire list for the smallest chunk that would hold the segment, and thus could reduce memory pool fragmentation.
4. If step 3 fails, return to step 1, starting with the list for the next larger range of chunk sizes in the current job.
5. Delete the selected chunk from its list.
6. Subtract the number of bytes needed for the segment from the size of the selected chunk, resulting in a new chunk size. Add the size of the segment to the base address of the selected chunk, resulting in a new chunk base address. Add the new chunk to the appropriate list for the job.
7. Create a descriptor for the new memory segment. Use the base address of the selected chunk as the base address field in the descriptor, and use the requested segment size as the limit field of the descriptor. Set the descriptor type bits to be writeable data.
8. Allocate a slot in the GDT for the new descriptor. If this cannot be done, return E_SLOT to the caller, and restore the job's chunk list to its original condition.
9. Store the descriptor in that slot. Create a selector for the descriptor and return its value as the token for the new segment object.

Two observations about this algorithm are worth noting. First, it is bounded, but nondeterministic. That is, creating segments can lead to variable response times from the system, depending on the current dynamics of the job tree and memory pools. Second, the algorithm does not guarantee to eliminate fragmentation. It is possible for there to be enough free bytes in a job's memory pool to meet the needs of a *rqcreatesegment()* request, but no contiguous chunk that is large enough, which leads to borrowing if possible, or leads to a possible failure to create the segment. The current FSM will not move segments around within a memory pool to eliminate fragmentation, but a future version of the iRMX III FSM might use

paging to map noncontiguous pages within a memory pool into an apparently contiguous segment.

Both indeterminacy and fragmentation can be minimized by creating as many of the necessary memory segments as possible when the job first starts running, rather than incrementally throughout the job's existence. Furthermore, creating these segments in decreasing order of size can reduce fragmentation and the possible need for borrowing. Because all objects occupy memory segments, the strategy applies equally well to memory segments themselves as well as all other types of objects that the job will need. Figure 1.2 implicitly recognized this strategy when it showed the code structure for a typical real-time task: the task first performs initialization (creates the objects it will need), then begins its real-time mission by entering an endless event loop.

Object directories. Job objects contain another data structure called an *object directory* used to facilitate sharing objects among jobs. Various parts of the operating system use object directories extensively for their own purposes, such as EIOS logical names, but the mechanism is equally available for use by application programs. You can look at the object directory for a job with the SDB *vd* command.

An object directory consists of a list of tokens and corresponding names for them. The number of entries that can be made in a job's object directory is a fixed number determined at the time the job is created, and can be as small as zero and as large as several thousand. A typical object directory has room for 50 entries.

To help explain the notion of an object directory, as well as some of the derived concepts that follow, the three system calls related to object directories are introduced. In the context of the discussion of object-based design, these system calls are part of the iRMX job type manager. An object directory is actually implemented as a hash table occupying one of the data structures of a job object, but the system calls you are about to see make the actual implementation irrelevant to the application program.

First, two notes on iRMX system calls. A system call is a procedure residing in system memory executed by a task that belongs to some job. The task that makes a system call is normally assumed to belong to a job associated with an application, although tasks belonging to first-level jobs and other operating system jobs make system calls as well.

Second, C function prototypes are used to introduce iRMX system calls. ANSI C allows, but does not require, variable names in the parameter list of a function prototype. Such variable names are used to facilitate talking about the different parameters, even though they are normally omitted in actual practice. PLM programmers can refer to the on-line help accompanying iRMX for Windows or to the *iRMX System Call Reference* manual (volume 9 of the iRMX for Windows documentation set) for the equivalent information expressed using that language's syntax.

```
void
rqcatalogobject ( TOKEN                job,
                  TOKEN                object,
                  char far *           namePtr,
                  WORD far *           exceptPtr);
```

This system call causes the token stored in variable `object` to be cataloged in the object directory for the job whose token is stored in the variable `job`, using the name stored in the character array `name`. If the system call succeeds, the word pointed to by `exceptPtr` will be set to a value of zero (often referred to symbolically as E_OK). If the call fails, either the word at `exceptPtr` will be set to an exception code or the exception handler for the task that made the system call will receive control, depending on how the task handles exceptions. (Exception handling is discussed in more detail in the following section).

For the system call to succeed, `job` must be a token for a valid job object, enough memory must exist in that job's object directory to hold the new entry, the `object` token must be for an existing iRMX object (of any type), the length of the iRMX string at `name` must be between 1 and 12 bytes, and the string must not match any of the entries already cataloged in the job's object directory[13].

At this point, a number of facts about object directories should be stated explicitly, and are outlined here. First, a program can catalog tokens in any job's object directory, not just its own, provided only that it can obtain a token for the job. For example, the system call *rqgettasktokens()* can be used to get the token for such key jobs as the root job.

Second, the same *token* can be cataloged multiple times either within a single job or across multiple jobs. The only requirement is that every entry in a particular job's object directory have a unique *name* within that directory. The same name can be used in different job directories for the same or different tokens.

Third, not all objects are cataloged in object directories, only those that need to use the directory mechanism for sharing. For example, if two tasks can access a shared variable, that variable can be used for sharing the value of a token, and no object directory needs to be used to do the sharing. Generally, object directories are used for sharing tokens between tasks that belong to jobs loaded into memory separately and that do not share global variables.

Finally, whenever tokens are shared, whether through object directories or by global variables, a token can become invalid. For example, if a token for an object belonging to Job 1 is cataloged in Job 2's object directory, and

---

[13]An iRMX string consists of one byte containing the length of the string as an unsigned value between 0 and 255, followed by a sequence of values occupying the specified number of bytes. The byte values are often ASCII codes, but ASCII is not a requirement. Any 8-bit values can be used.

Job 1 terminates, all the objects belonging to Job 1 are deleted. The token in Job 2's object directory, however, is not automatically deleted. iRMX will not let an invalid token be placed into a job's object directory, but it does not guarantee that all cataloged tokens are still valid at some later time.

```
TOKEN
rqlookupobject ( TOKEN                  job,
                 char far *             name,
                 WORD                   timeLimit,
                 WORD far *             exceptPtr );
```

Given a token for `job` and a match between `name` and one of the names in the object directory for `job`, this function returns a copy of the token cataloged under that name. `timeLimit` specifies the amount of time the calling task is willing to wait if the name is not in the job's object directory when the system call is made. If a value of zero is specified and the name is not found, the call completes immediately with an exception code of 0x0001 (E_TIME), either returning to the caller with the word pointed to by `exceptPtr` set to 0x0001 or to the caller's exception handing routine, depending on the setup. If `timeLimit` is set to 0x0FFFF, the calling task will block until the name appears in the job's directory (placed there by another task) or the program is terminated. (Task scheduling states are discussed in the next section.)

If `timeLimit` is between 0 and 0x0FFFE and `name` does not match an entry in the object directory, the calling task will block until either the entry appears in the object directory or the time limit expires. The time limit is specified in 0.01-second intervals for almost all iRMX systems. That is, the task may block for 0 to 655.34 seconds waiting for another task to catalog an object that uses the matching name.

Because a time limit is associated with this call, the call can be used to synchronize concurrently executing tasks. The synchronization provided by this call is normally used only when an application is initializing and cataloging objects that are to remain in place for the duration of the application. More dynamic synchronization is usually accomplished using other system calls that involve less overhead.

```
void
rquncatalogobject ( TOKEN               job,
                    char far *          name,
                    WORD far *          exceptPtr);
```

This function is used for housekeeping. As its name implies, it removes an entry from a job's object directory. It is good programming practice to uncatalog tokens for objects about to be deleted to avoid the invalid token problem mentioned previously. Also, uncataloging objects ensures that a job's object directory does not fill up over a period of time.

A common use for this call is when an application catalogs an object in another job's directory, terminates unexpectedly (i.e., without uncataloging the object), and is restarted. It must uncatalog the entry from the first time the program executed before it can catalog a new token using the same name the second time it runs.

The following are a few summary points about object directories, which are necessary before discussing tasks.

Object directories are an optional property of jobs. It would be absolutely possible to build a complete iRMX system with all object directories for all jobs empty and of length 0. Methods other than object directories can be used for sharing objects among tasks or jobs.

The iRMX job tree and object directories are not related at all to the tree-structured file system or the directory nodes in a file system.[14] Think about it. For example, you could not have a file tree without directory nodes, but you can have a job tree without object directories. Files and directories reside on disks. Jobs and object directories occupy primary memory (RAM). The job tree is based on a parent-owns-child relationship. File directories do not own the files listed in them — users (people) own files and file directories. No analogies exist between jobs and files. The fact that there are both job trees and file system trees says something about the ubiquitous nature of tree structures in computer science, but does not imply any relationship between jobs and files.

There is an SDB command to view the object directory for a job. View directory, or *vd*, (available from the SoftScope prompt or by typing <alt-Break> under iRMX for Windows[15]) takes a token for the job as its argument. As various layers of iRMX that use the object directory facility are discussed, it can be very instructive for you to poke around in the system you are using to see the actual entries that have been cataloged in various job's object directories.

6.6.2 Task management

The essence of task management is scheduling. Task creation and deletion are part of task management too, but the heart of the matter is scheduling.

In a single-CPU environment, such as the computers on which iRMX runs, exactly one task can be executing at any particular moment. Task scheduling ultimately boils down to a matter of selecting which single task is to execute at any time. In iRMX, tasks can use a rich set of functions for synchronization and communication among themselves. All of these func-

---

[14]Certain objects that the BIOS or EIOS use in managing a file system do show up in object directories (logical names), but the basic statement that the two have nothing to do with each other is correct.

[15]Type <g> to exit the SDB if you enter it using <alt-Break>.

tions should be thought of as extensions to the task scheduler procedure in the Nucleus. When any system call that involves communication or synchronization changes the state of a task, it does not return directly to the code from which it was called (the application program). Instead, it exists to the task scheduler, which determines whether that task or some other task is to execute next.

Two concepts are central to task scheduling. First, every task has a priority number between 0 and 255 assigned to it, with 0 being the highest. The basic scheduling rule used by the iRMX task scheduler is called *preemptive, priority-based scheduling.* This rule says that of all the tasks ready to execute at any moment, the one task with the highest priority will be selected to execute, and it will continue to execute until it either blocks (waits for an event of some sort) or is preempted by another task of higher priority that becomes ready. If two tasks of equal priority are ready at the same time, the first one that became ready is selected for execution.

This first-come-first-served algorithm can be augmented by a round-robin policy in which ready tasks with equal priority are given time quanta of CPU time, and the running task is put at the end of its priority queue when its quantum expires. Round-robin scheduling is optional and most commonly used when an iRMX system is being used as a time-sharing system for development work rather than for real-time applications. The default configuration of iRMX enables round-robin scheduling for tasks with priorities between 140 and 255, with a time quantum of 50 milliseconds. Note that if two ready tasks have different priorities, even if both priorities are in the round-robin range, the higher-priority task is always the one that runs. Round-robin applies only to tasks of equal priority. (Round-robin scheduling is ignored in the description of scheduling that follows, but that does not affect the essence of the discussion.)

Note that all tasks are treated uniformly by the task scheduler, without regard to the jobs to which the tasks belong. The only interaction between jobs and task scheduling is that every job has a maximum task priority associated with it (one is shown, for example, in the upper right corner of Figure 6.6). Thus, no task belonging to a job can be created with a priority higher (numerically lower) than this value. Tasks can change their own or other tasks' priorities during execution (using *rqsetpriority( )*), but each job's priority limit always applies to all the tasks that it owns.[16]

The second concept central to task scheduling is the notion of the scheduling state. At any moment, every task in the system is in one of several states. Figure 6.7 is the standard diagram for iRMX task scheduling states. Exactly one task is always in the running state, and any number of tasks

[16]If you still are not clear about the difference between time-sharing and real-time systems, consider this: If you are using an iRMX system for timesharing and you do not like the impact of the other users on your work, just run a program that calls *rqsetpriority( )* to change all the other users' task priorities to 255!

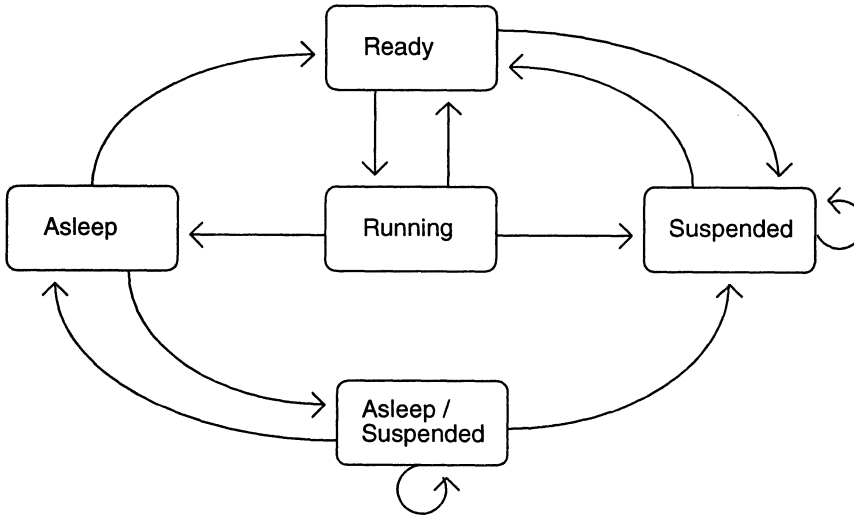


Figure 6.7 State diagram showing possible state transitions for iRMX tasks.

are in each of the other states. The ready state holds all tasks not blocked but with priorities less than or equal to the task in the running state. The asleep state is used for most cases of blocked tasks, with *suspended* and *asleep-suspended* used for a special form of blocking described in the following sections.

Once a task enters the running state, it stays there and continues to have total control of the CPU until one of the following things happens.

The task calls *rqsleep()*. This system call causes the task to block (enter the asleep state) for a specified time interval. At the end of the interval, the task wakes up, enters the ready state, and runs again if its priority is higher than the running task's.

The task calls *rqsuspendtask()* and specifies itself as the task to be suspended. The task then moves to the suspended state and stays there until another task in the running state calls *rqresumetask()*, and then this task enters the ready state and runs again when it has the highest priority.

The task calls *rqresumetask()* and the task being resumed has a higher priority than the running task. The resumed task enters the ready state, is seen to have a higher priority than the running task, and preempts it. The running task enters the ready state, and the resumed task becomes the running task.

The task makes one of the system calls listed in Table 6.1, which blocks because the specified request cannot be satisfied.. If the task blocks, the calling task will remain blocked until either the blocking event occurs or a time

TABLE 6.1 Nucleus System Calls That Might Cause a Task to Block

System call	Event that will end the block
rqforcedelete	Another task calls *rqenabledeletion()* for the object to be deleted. No time limit.
rqlookupobject	A task calls *rqcatalogobject()* for a token with a matching name.
rqreceivecontrol	A task calls *rqsendcontrol()* for the region. No time limit.
rqreceivedata	A task calls *rqsenddata()* for the data mailbox.
rqreceivemessage	A task calls *rqsendmessage()* for the object mailbox.
rqreceiveunits	A task calls *rqsendunits()* for the semaphore.
rqsleep	None. Time limit only.
rqsuspendtask	Another task calls *rqresumetask()* for this task. No time limit.
rqetimedinterrupt	An interrupt of the proper level occurs.
rqwaitinterrupt	An interrupt of the proper level occurs. No time limit.
rqreceive	A task on another processor of a Multibus II system sends a message to the specified port.
rqreceivereply	A task on another processor of a Multibus II system replies to an RSVP message.
rqreceivesignal	A task on another processor of a Multibus II system sends a signal.

limit completes, except as noted in the table. The table also lists the event that can release the block. Most of these system calls include a time-limit parameter, as we saw earlier for *rqlookupobject()*. The task enters the asleep state and remains there until (1) the request can be satisfied, (2) the specified time limit expires, or (3) another task suspends the asleep task, putting it into the asleep-suspended state.[17] Whichever event — (1) or (2) — occurs next, the task then enters the ready state and preempts the running task if it has a higher priority. The task must examine the condition code returned for the system call to determine whether the call completed because the event occurred (E_OK) or because the time limit expired (E_TIME). A task can enter the asleep state only by making a system call itself. Unlike the suspended state, one task cannot put another task into the asleep state.

The task makes one of the system calls listed in Table 6.2, which satisfies a request that had caused another task to block. The blocked task moves to the ready state and preempts the running task if the blocked task has higher priority.

[17]A task that enters the asleep-suspended state returns to the asleep state if it is resumed before its time limit expires, or enters the suspended state if its time limit expires before being resumed.

TABLE 6.2 Nucleus System Calls That Can Cause Another Task to Enter the Ready State

System Call	Task that might preempt the running task
rqcatalogobject	A task that has called *rqlookupobject( )*.
rqcreatejob	The initial task of the created job.
rqecreatejob	The initial task of the created job.
rqcreatetask	The newly-created task.
rqenable	The interrupt task for the enabled interrupt level.
rqenabledeletion	A task that has called *rqforcedelete( )* for the object.
rqendinittask	The initial task in the next first-level job to be created.
rqresumetask	The resumed task.
rqsendcontrol	A task that has called *rqreceivecontrol( )* for the region.
rqsenddata	A task that has called *rqreceivedata( )* for the data mailbox.
rqsendmessage	A task that has called *rqreceivemessage( )* for the object mailbox.
rqsendunits	A task that has called *rqreceiveunits( )* for the semaphore.

The task calls *rqsetpriority( )*. With this command, the task might either raise the priority of another task or lower its own priority so that it no longer has a higher priority than all the tasks in the ready state.

For each case listed so far, the running task makes a system call that causes itself to exit the running state. Each of these system calls is part of the Nucleus layer of iRMX, and each of them finishes its work by calling the iRMX task scheduler, which is the procedure in the Nucleus that selects the next task to run.

Some scheduling state transitions occur without the running task making a system call. These cases are initiated by hardware interrupts. When a hardware interrupt request occurs, the CPU's interrupt logic, described in chapter 5, saves the state of the currently running program (which for iRMX is the task in the running state) and activates a procedure called an *interrupt handler*. It can be said that the interrupt handler is then running in the context of the currently scheduled iRMX task. All iRMX tasks must have a stack large enough to accommodate the CPU state saved during interrupt processing. The stack is restored to its original condition when the interrupt handler terminates. The following items illustrate the general scheduling issues related to interrupt processing.

There is a clock circuit that causes a hardware interrupt, typically 100 times per second for iRMX systems. The clock interrupt handler uses a counter variable to keep track of current time of day. When the clock interrupt handler increments the time-of-day counter to the time limit of a task in the asleep state, the task scheduler is activated, which moves the asleep task to the

ready state, and causes that task to preempt the running task if its priority is higher. Versions of the Nucleus that support iRMK operations (iRMX III and iRMX for Windows) can be based on a finer-granularity clock (typically, 1 millisecond), but the behavior of the system for applications that do not make iRMK calls is the same as described here.

When a hardware I/O device controller causes an interrupt, an interrupt handler is provided as part of the software device driver for that device controller. Neither the iRMX task scheduler nor the task running when the interrupt occurs can know when an interrupt handler is activated. I/O interrupt handlers, however, are allowed to make a special Nucleus system call named *rqsignalinterrupt( )*, which tells the task scheduler to move a task associated with the interrupt into the ready state. This interrupt task thus preempts the running task if the interrupt task has a higher priority.

6.7 Exception Handling

The last (or only) parameter of every iRMX system call is a pointer to a word set to the condition code, sometimes called the *exception code*, for the call. If the call completes normally, this word is set to a value of zero (0x0000). If an abnormal condition occurs during execution of the call, the iRMX exception handling mechanism is called. A mnemonic is associated with each condition code value, such as `E_OK` for 0x0000. Header files are available so that programs can refer to these values by name rather than by number.[18] This section gives an overview of the exception-handling mechanism, and the next section describes how it is implemented in more detail.

6.7.1 Types of exceptions

Before discussing the types of exceptions, let's distinguish among environmental exceptions, programmer exceptions, and faults.

Environmental exceptions. These are abnormal conditions that arise during the execution of a system call but which do not necessarily represent programming errors. Examples include attempting to write to a printer that is out of paper or having the time limit of a call to *rqlookupobject( )* expire without the requested object being cataloged. The first example is clearly not an error, and the second example might or might not be an error condition, depending on the nature of the application program.

Programmer exceptions. These are also conditions that occur during the execution of system calls, but which simply should not happen. They are caused by bugs in the program that makes the call. An example would be

[18] For example, `:includeirmx_err.h` on iRMX for Windows system.

passing something other than a token for a valid iRMX job as the first parameter of a call to *rqcatalogobject( )*.

Faults. Faults are conditions detected by the microprocessor hardware operating in protected mode, as described in chapter 5. The most common fault that programmers encounter is probably the general protection fault (GP fault), which has a fault code of 13.

The other common type of fault is the stack fault, code number 12, which occurs when the current stack segment overflows (for instance, because of an infinite loop that includes a function call that keeps creating new stack frames) or when a re-entrant function (such as any C function not declared to be static) makes an out-of-bounds reference to a local array variable, and thus located in the stack segment rather than a data segment.

If an application program faults while executing its own code, it is because the programmer has coded an illegal memory reference (array out of bounds, attempt to execute data, etc.). If an application program faults while executing a system call, the fault should be handled by the operating system itself. (Otherwise, it would represent a bug in the operating system.) Faults in a user's code and exceptions, which occur only during execution of a system call, are handled a little differently from each other, as you will see in the next section.

Returning to exception handling, every system call has associated with it a set of condition code values that it can generate to indicate exceptions. These values are documented with each call, and that documentation should be consulted when designing and debugging applications that encounter exceptions.[19] The same exception-code value can provide you with subtly different information, depending on what system call caused it to occur.

The decision about what to do if an exception or fault does occur depends on the stage of development of the application and the nature of the particular exception. Some conditions should cause the application to terminate or break to a debugging program, while others might alter the logic flow in the program. Programmers accustomed to developing software in a non-protected environment such as DOS might consider faults and exceptions to be a nuisance at first, but they are actually extremely significant time savers during development. Without faults and exceptions, debugging can involve working from the side effects of an error that occurred thousands of instructions earlier than the point at which the problem became apparent. A fault or exception, on the other hand, generally occurs at the moment the error occurs, making localization of the problem much easier.

---

[19]iRMX for Windows users will find the exception codes associated with each system call listed in the help system provided with that version of iRMX.

6.7.2 Handling exceptions and faults

You can handle exceptions in two ways. One is to have the application task examine the condition-code value after the system call completes, which is called *in-line exception handling*. The other is to establish an exception-handling procedure that automatically receives control when a system call completes with a nonzero condition code value.

Figure 6.8 is a C program that demonstrates in-line exception handling. The two values explicitly tested for Status in this program are 0x0000, which is also known as E_OK (no error), and 0x0002, also known as E_MEM, which means there is not enough memory to satisfy the request. Other condition codes that might be returned include 0x0004 (E_LIMIT) if the job has already created all of the objects it is allowed to, and 0x000C (E_SLOT) if there are no more slots available in the GDT to hold the descriptor for the segment.

Figure 6.8 Sample code illustrating in-line exception handling.

```
/***>  inline.c  <************************************************
 *
 *      Demonstrate in-line exception handling
 *
 ************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <rmxc.h>

/*  Execution starts here
 */
int
main (int argc, char *argv[]) {

DWORD           Size = 100;     /*  Default segment size  */
WORD            Status;
TOKEN           Segment;
EXCEPTIONSTRUCT eh;

/*  Set up for in-line exception handling
 */
  rqgetexceptionhandler (&eh, &Status);
  eh.exceptionmode = 0;  /*  Never call the exception handler  */
  rqsetexceptionhandler (&eh, &Status);

/*  Get the size of a segment to create from the command line
 */
  if (argc == 2) Size = atol (argv[1]);

/*  Create a memory segment object and test to see if the system call
 *  was successful or not.
 */
  Segment = rqcreatesegment (Size, &Status);
  switch (Status) {
    case 0x0000: printf ("Created a %d-byte segment successfully.\n",
                         Size);
                 exit (0);
    case 0x0002: printf ("Not Enough Memory for a segment of size %ld.\n",
                         Size);
                 exit (1);
    default:     printf ("Unable to create segment of size %ld.\n"
                         "Exception code is %4X\n", Size, Status);
                 exit (1);
```

This program explicitly checks for the two most common values for the condition code after calling *rqcreatesegment( )* (normal completion and not enough memory), and lets all the other possible values that the call might return be handled as the default case in the switch statement.

Every iRMX task has an exception handler, and there are several ways to select which procedure is to be the exception handler for a particular task. The most accessible way to associate an exception handler with a task is for the task to use the *rqsetexceptionhandler( )* system call. This same call is also used to select between in-line exception handling and use of an exception handler. The C function prototype is the following:

```
void
rqsetexceptionhandler ( ehstruct *              eh,
                        WORD far *              exceptPtr );
```

`ehstruct` is a structure consisting of a far pointer to the routine to serve as the handler procedure and a mode byte that tells under what conditions the handler procedure should be called. Values for the mode byte follow:

0 Never. The application must handle all exceptions in-line.
1 Programmer Errors. The exception handler is called if a programmer error occurs, but other exceptions are handled in-line.
2 Environmental Conditions. The exception handler is called if an environmental condition occurs, but programmer errors are handled in-line.
3 Always. All system calls that result in a nonzero condition code setting cause the exception handler to be called.

Every job has an exception handler and mode associated with it; a pointer to an `ehstruct` is one of the parameters of the *rqcreatejob( )* or *rqecreatejob( )* system call (see chapter 7 for a summary of many system calls that have not been introduced yet). Each task created for the job automatically inherits that job's default exception handler and mode, and each task can then change its handler and mode by calling *rqsetexceptionhandler( )*.

Sometimes, a task needs to switch between in-line exception processing and the use of its default exception-handler procedure. To do this, the task calls *rqgetexceptionhandler( )*, which fills in an `ehstruct` structure with the current exception handler pointer and mode byte. The task can then change the value of the mode byte in this structure and call *rqsetexceptionhandler( )* to achieve the desired effect. For example, Figure 6.9 is a PLM program that does in-line exception handling for a call to *rqcreatesegment( )* and uses the job's default exception handler to handle any exceptions that occur when it calls *rqexitiojob( )*. Examining this code, you might wonder why some system calls are followed by tests of the variable `Status` and others are not. Let's consider each call in sequence.

Figure 6.9 PLM program illustrating both in-line exception handling and use of the default exception handler for the job.

```
/***>  handle.plm  <************************************************
 *
 *  PLM Program demonstrating switch between in-line and default
 *  exception handling
 *
 ************************************************************************/

handle: DO;
$include (handle.ext)

DECLARE
        E$OK          LITERALLY   '0',
        NEVER         LITERALLY   '0',
        PROGRAMMER    LITERALLY   '1',
        ENVIRONMENT   LITERALLY   '2',
        ALWAYS        LITERALLY   '3',
        CR            LITERALLY   '0Dh',
        LF            LITERALLY   '0Ah',
        ehstruct      STRUCTURE (
            handler   POINTER,
            mode      BYTE),
        Segment       TOKEN,
        Status        WORD;

/*  Get the default exception handler and mode for the job, and
 *  change the mode to 0 -- handle exceptions in-line.
 */

    CALL rqgetexceptionhandler (@ehstruct, @Status);
    ehstruct.mode = NEVER;
    CALL rqsetexceptionhandler (@ehstruct, @Status);

/*  Create a segment and check for exceptions.
 */
    Segment = rqcreatesegment (5280000, @Status);
    IF Status <> 0 THEN
      DO;
        CALL rqcsendcoresponse (NIL, 0, @(15, 'Create Failed', cr, lf),
        @Status);
        IF Status <> 0 THEN
          CALL rqcsendcoresponse (NIL, 0, @(23, 'SendCOResponse Failed',
          cr, lf), @Status);
      END;

/*  Let default exception handler manage anything that goes wrong now
 */
    ehstruct.mode = ALWAYS;
    CALL rqsetexceptionhandler (@ehstruct, @Status);
    IF Status <> 0 THEN
        CALL rqcsendcoresponse (NIL, 0, @(20, 'Set Handler Failed', cr,
        lf), @Status);

    CALL rqexitiojob (0, NIL, @Status);
END handle;
```

When the program starts running, the default exception handler will delete the job if any exception occurs (described in the following section), and the mode is initially set to 3 (also described). The first system call is to *rqgetexceptionhandler( )*, and `Status` is not checked after the call. `Status` is not checked because if the call fails, the default handler will abort the job, so any code following the call will either find `Status` to be 0 or will never be reached. The second call is to *rqsetexceptionhandler( )*. Again, the condition code is not checked. If the call fails, it means the mode did not change, the default handler will be invoked to handle the condition, and the job will be aborted. If the call succeeds, there is no need to check the condition code—it is 0.

The call to *rqcreatesegment( )* is followed by an in-line check of `Status`, and the code handles any error condition by displaying a message. Since the call to *rqcsendcoresponse( )* might fail, `Status` is again checked after that call. Chances are, trying another *rqcsendcoresponse( )* to display a message about the failure of call to the same routine will also fail, but the code is included anyway, and the potentially infinite regress is arbitrarily terminated at that point. (This piece of code has not been fully tested; it is hard to get *rqcsendcoresponse( )* to fail!)

Before exiting the job, the default exception-handling mode is changed back to 3, and *rqexitiojob( )* is called. The condition code is checked after *rqsetexceptionhandler( )*, as failure means that in-line checking is still in effect, but the code is not checked after *rqexitiojob( )*, because mode 3 is in effect for that call and will cause the job to abort if the call fails.

Figure 6.10 illustrates the use of a user-written procedure as a task's exception handler. The procedure takes four parameters: the condition code for the system call that caused an exception, the number of the parameter that was in error (numbered left to right in the system call's argument list, starting with 1), an unused parameter, and a word containing the numeric coprocessor's status word if the condition code is E_NDP_ERROR (0x8007). There is no standard header file that contains a function prototype for exception handler procedures. The *iRMX Nucleus Programming Concepts* manual (volume 3 of the iRMX for Windows documentation set) gives the information needed to code this procedure.

The sample handler simply displays the values of the first two parameters passed to it and returns to the program that caused the exception to occur. The function *main( )* installs the exception handler and then tests it by calling *rqlookupobject( )* with a value for the first parameter of the call taken from a command-line argument. Three interesting cases can be tested with this program, as described as follows.

The command line argument is 0 or omitted. The *rqlookupobject( )* system call interprets a job token with the value of 0 as a reference to the calling

Figure 6.10 Sample user-written exception handler, installed and tested by *main()*.

```
/***> handler.c  <***********************************************************
 *
 *    User-written exception handler example
 *
 ***************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <rmxc.h>

/*      The Exception Handler Procedure
 *      -------------------------------
 */
void far
myHandler (WORD code, WORD param, WORD reserved, WORD npxStatus) {

printf ("Exception 0x%04X occurred in parameter %d.\n", code, param);
return;
}

/*      Code to test the handler starts here
 *      ------------------------------------
 */
int
main (int argc, char *argv[]) {

EXCEPTIONSTRUCT     eStruct;
TOKEN               dummy;
WORD                Status;

/*  Establish handler() as this task's exception handler
 */
    eStruct.offset =  (NATIVE_WORD) (void near *) myHandler;
    eStruct.base = (selector) myHandler;
    eStruct.exceptionmode = 3;
    rqsetexceptionhandler (&eStruct, &Status);

/*  Force an exception to test the handler
 */
    if (argc == 2)  dummy = (TOKEN) atoi (argv[1]);
    else            dummy = (TOKEN) NULL;
    rqlookupobject (dummy, "\x006INARDY", (WORD) 0, &Status);
    printf ("exit from main\n");
    exit (0);
}
```

task's job. (No real iRMX object has a token value of 0; GDT slot 0 holds the descriptor for the GDT segment itself.) Because the program does not have anything cataloged in its job's object directory using the name "INARDY" the system call fails with an E_TIME exception because the object cannot be found within the time limit specified. The messages issued by the program are the following:

```
Exception 0x0001 occurred in parameter 3.
exit from main
```

The numeric code for the E_TIME exception is 0x0001, and the system call reports that the error occurred because of the value of the third parameter (the time limit). The second message demonstrates that the exception handler returns to *main()* successfully.

The command line argument is 1. No iRMX object has the token value of 1, so the call to *rqlookupobject()* fails with an exception code of E_EXIST (token does not exist), which has a numeric code of 0x0006. Since the job parameter is the first argument to the system call, the program issues the following messages:

```
Exception 0x0006 occurred in parameter 1.
exit from main
```

The command argument is 600. Telling you this is cheating, but 600 (0x258) has always been the numeric value of the token for the root job of iRMX systems, and will probably continue to be so for some time.[20] If your system is running with iRMX networking installed (discussed in chapter 11), there will indeed be a token cataloged in the root job's object directory using the name `INARDY`, and the exception handler will not be called if you run the program with 600 on the command line. Only the `exit from main` message will be displayed.

Fault handling also involves passing control to an exception handler routine when the fault is detected by the hardware. iRMX for Windows systems allow two choices for the fault handler. In the `rmx.ini` configuration file, there is a parameter called `DEH` (Default Exception Handler) in the [`Nucleus`] section. If this parameter is set to `000H` (the file uses PLM syntax for hexadecimal values), faults cause a break to the System Debug Monitor (SDB), which the programmer can use to debug the problem. If DEH is `0FFH`, faults cause the job that owns the faulted task to be deleted.

You cannot cause a program's own exception handler to be invoked when a fault occurs in iRMX for Windows systems, but iRMX systems that support the Interactive Configuration Utility (ICU) can do so. (The ICU is introduced in chapter 9.) When a user's exception handler is configured to receive control on faults, the exception code will be 0x800C for stack faults and 0x800D for general protection faults. That is, the code is 0x8000, plus the interrupt level of the fault. As faults are not associated with system calls, the parameter-number argument passed to the handler is zero for faults.

---

[20]It will probably change as soon as someone in the iRMX development group at Intel reads this! The proper way to determine the token for the root job is to use the *gettasktokens()* system call.

6.7.3 The default exception handler for a job

Every job created has an exception handler and mode assigned to it, either by default (a default system exception handler and mode is configured into every iRMX system) or by setting a parameter in the system call that created the job. (The system calls to create jobs are covered in chapter 7.) Any task can change its own exception handler and mode using the *rqsetexceptionhandler( )* system call; doing so does not affect exception handling for other tasks belonging to the same job. This section considers the options available for establishing the default exception handler for a job, which is the handler in effect for each task belonging to the job until the task changes the exception handler.

The examples in Figures 6.8 and 6.9 demonstrate the difference between C and PLM programs with regard to which default exception handler is in place when execution begins. The difference is only superficial: all programs that run as HI commands start with a default exception handler that deletes the job in the event of any exception and a mode of 3 (signifying that the handler is always invoked). C programs, however, do not start executing at *main( )*, they start executing in a start-off routine that performs initialization and then calls *main( )*. This initialization code sets the exception handling mode to 0 before calling *main( )*, so C programs do in-line exception handling by default.[21] Programs run under SoftScope III also have their exception handling mode set to 0, whether they are coded in PLM or C. Earlier versions of SoftScope leave the default mode at 3.

As mentioned earlier, a pointer to an `ehstruct` is one of the parameters of the *rqcreatejob( )* system call. If this parameter is coded as a null pointer, the job inherits the default exception handler procedure and mode for the system. For systems that support the ICU, the system default is chosen at the time the system is configured, with the choices being:

- A handler that deletes the task that encounters an exception.
- A handler that suspends the task.
- A handler that deletes the job that owns the task.
- The system debugger.
- A procedure supplied by the system administrator when the system is configured.

In addition to the default handler procedure, the default mode can be configured to any of the four values (always, never, environment, programmer) using the ICU as well.

---

[21]This behavior changed with iRMX for Windows 2.0c. With this release, the initialization code does not change the exception handling mode. The code in Figure 6.10 is coded to work regardless of the version of the operating system.

Programs run as HI commands have their job's default handler set to the system default procedure and mode, so the preceding statement about the default handler for HI commands is true only if the system default is set to delete the task's owning job and the default mode is Always.

6.8 The iRMX System Call Mechanism

Chapter 5 introduced the machine-language conventions used for making function calls with the x86 architecture and described the call gate and software mechanisms that can be used for accessing OS functions with protected-mode and real-mode operating systems. In addition, chapters 3 and 4 introduced interface procedures, to which an application must be linked to make system calls. This section describes the internal logic of interface procedures in more detail.

When an application includes iRMX system calls, its object module must be linked with a library that contains an interface procedure for each system call referenced by the application. The interface procedure library contains small assembly-language functions that act as the intermediary between the application's code and the actual system call procedures resident in system memory. Although these interface procedures have system call names like *rqcsendcoresponse( )* or *rqcreatesegment( )*, they do not perform the actual work of a system call. They pass parameters from the application to the actual system call, branch to the proper place in system memory for the actual system call, and return result and condition code values from the actual system call to the application task.

The code for making a system call from an application is just like the code for calling any other type of function: the parameters are pushed onto the calling task's stack (in left to right order for iRMX system calls), and a machine language *call* instruction is then used to push a return address onto the stack and branch to the prologue code in the interface procedure, which completes a normal x86 stack frame by pushing the bp register and saving the sp register in bp. Depending on whether the application was compiled using the compact or large model, the *call* instruction will have pushed either a near or far return address onto the stack (offset only or offset plus selector).

Because the actual system call will access the parameters by referring to fixed offsets from the top of the stack, there are two libraries of interface procedures. The interface procedures in the library for compact-model programming include an extra *push* instruction (of a dummy value) before sp is copied into bp to put the parameters at the same relative positions in the stack as for large model programs. The interface procedures in the large model library are the same as the compact model except for this extra *push* instruction and a corresponding difference at the end of the interface procedure, where the compact model must delete the extra word on the stack before returning to the application. The overhead for these extra two in-

structions in the compact-model interface procedures, by the way, is insignificant compared to the overhead that large model programs incur as they load and restore the cs and ds segment registers (and descriptors in protected mode) when calling and returning from their interface procedures.

With the incoming parameters at a fixed location on the stack, the interface procedure now calls the actual system subroutine to do the system call's work. In real mode, this is done by loading a value into a register to indicate which particular system call is being made and executing a machine-language *int* instruction to enter the operating system. A different interrupt number exists for each layer of the OS, so the interface procedure executes the *int* for the appropriate layer, the interrupt vector contains a pointer to an entry routine for the layer in system memory, and the entry routine examines the register value to determine which system call subroutine to jump to.

For protected-mode versions of the OS, the interface procedure uses a far *call* instruction to enter the appropriate system subroutine directly. The offset part of this *call* instruction is ignored, because the selector part always goes to a slot in the GDT that contains a call gate. From chapter 5 you might recall that call gates are special descriptors that contain complete pointers to subroutines, along with information used to change privilege levels and copy parameters from the application's stack to the stack for the new privilege level. The call gates for current versions of iRMX do not invoke privilege shifts for system calls. Both applications and the OS routines operate at privilege level 0 (most privileged level) at all times, so there is no parameter copying involved for iRMX.

Note that the interface procedures do not need to know the actual address of the system subroutines they call. In real mode, only an interrupt number for the OS layer and a code value for the particular system call are needed. In protected mode, only a slot number in the GDT is needed. Thus, the operating system can be reconfigured, revised, or rewritten, and applications bound to interface procedures do not need to be changed as long as this information remains constant.

At this point, the system subroutine executes and returns certain values to the application. The iRMX convention is to return these values in registers, but most system calls are written in a high-level language (usually PLM or C), so they typically call one of several assembly-language procedures called *exit routines*, passing the appropriate values on the stack. The exit routines put the values into the proper registers and return to the interface procedure.

The interface procedure now stores the register that holds the condition code for the system call (the cx register) in the word pointed to by the last parameter of the application's system call and examines its value. If the value is 0x0000, meaning the system call completed normally, the interface procedure now cleans up the stack frame and returns to the application in the usual way.

If, however, the condition-code value is not zero, the interface procedure calls the *rqerror( )* procedure, which determines what the exception-handling mode is for the application task. If the mode is 0 (handle exceptions in-line), *rqerror( )* returns to the interface procedure, which returns to the application, which must test the word pointed to by its last parameter to determine the result of the system call. If, however, the application task is not doing in-line handling of the type of exception that occurred, *rqerror( )* calls the task's exception handler. A user-written exception handler could return to *rqerror( )*, which will return to the interface procedure, which will then return to the application, but none of the exception handlers supplied with the operating system return.

The foregoing description is slightly simplified with respect to *rqerror( )*. The interface procedures for Nucleus-layer system calls under protected-mode versions of iRMX (II, III, and iRMX for Windows) call *nucerror( )* instead of *rqerror( )*. The difference is that *nucerror( )* checks the exception mode and (conditionally) calls the exception handler directly, but *rqerror( )* uses the Nucleus-layer system call, *rqsignalexception( )*, to do the same function.

The use of *rqerror( )* or *nucerror( )* procedures to test for and possibly invoke a task's exception handler provides an alternative mechanism for exception management for iRMX applications. The code for these procedures comes from the interface procedure libraries, so applications can provide procedures with the same names to substitute for those supplied by Intel. As long as a module containing the substitute procedures has been linked by the binder before the interface library, the interface procedure calls to these routines will be linked to the substitute versions, and the standard versions will not be used. If this technique is used, and if the substitute version of *rqerror( )* does not call *rqsignalexception( )*, the setting of the individual tasks' exception handler and mode become irrelevant. Once the code for an application is linked to the substitute version of *rqerror( )*, all tasks that execute that application's code, regardless of their calls to *rqsetexceptionhandler( )* and their jobs' default exception handler, have their exceptions handled as coded in the substitute *rqerror( )* procedure.

The purpose of this section has been to clarify the steps involved in making any iRMX system call. Readers who plan to code their own system calls (and thus their own interface procedures as well) will have to do some assembly language programming. That code is covered in more detail in chapter 10, which covers techniques for adding system calls and type managers to the operating system.

As a summary, Figure 6.11 is a pseudocode diagram for an application program that makes a system call. Execution begins in the application program, enters the interface procedure, transfers to the code in system memory, returns to the interface procedure, where *rqerror( )* is called if the condition code set by the system call is not zero. *Rqerror( )* calls *rqsignalexception( )*, which invokes the task's exception handler unless ex-

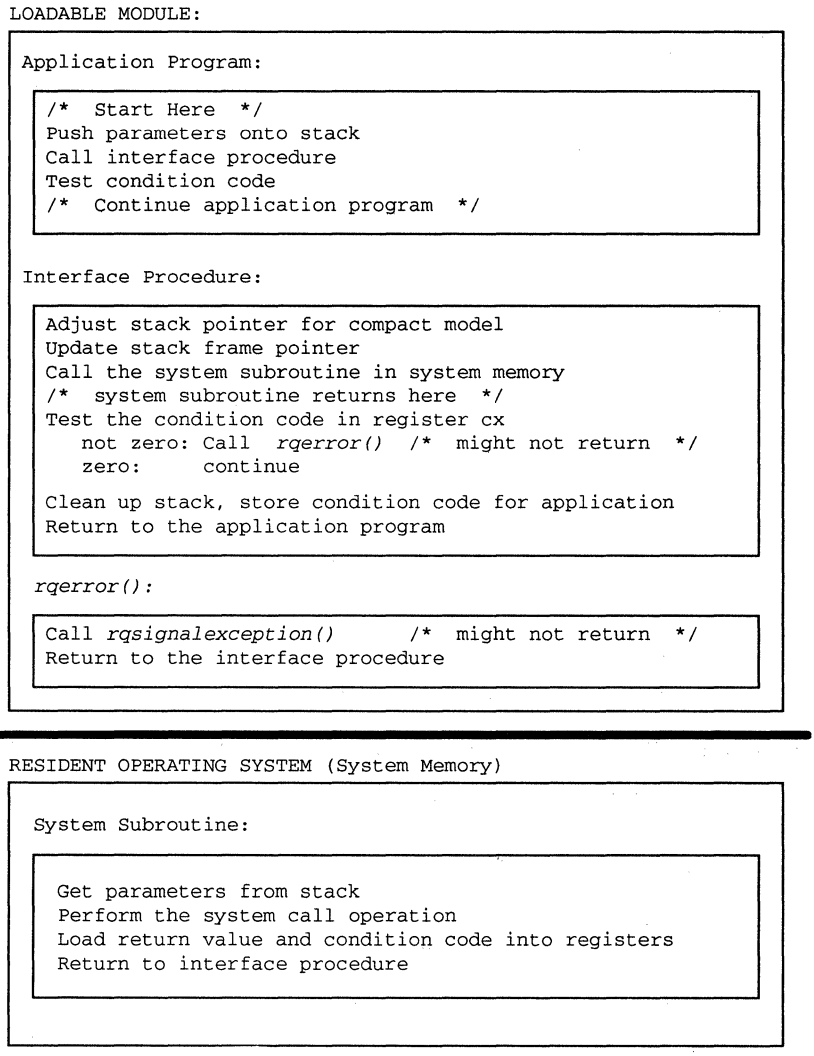


Figure 6.11 Pseudocode for an application program that makes a system call.

ceptions are being handled in line. If they are handled in line, *rqsignalexception()* returns to *rqerror()*, which returns to the interface procedure, which returns to the application. If no exception occurs, the interface procedure returns directly to the application without calling *rqerror()*.

Chapter

7

Basic iRMX System Calls

7.1 Overview

So many system calls are available for use with iRMX that it is often difficult to know how to begin developing a real-time application for the system. This chapter presents the system calls associated with the key concepts of task, memory, and job management used in developing basic real-time applications for iRMX.[1]

System calls for Human Interface (HI) command processing are briefly introduced, but they are not covered in detail because applications developed in C generally do not use them. System calls used for device drivers and interrupt management are covered in chapter 9, and system calls for managing composite objects and for creating new system calls are presented in chapter 10.

For complete information on all the system calls for iRMX II and III, consult the *iRMX System Call Reference* manual, volume 9 of the iRMX for Windows documentation set, or the corresponding volume for other versions of the operating system. The entire text of the iRMX for Windows version of this volume is available in hypertext format from the DOS *rmxhelp* command provided with iRMX for Windows. iRMX I users might prefer to consult the *System Calls* volume of the iRMX I documentation set, which omits material specific to iRMX II and III.

Intel provides system call documentation using both PLM and C syntax. This book presents system calls using C-language function prototypes based on the `:include:rmxc.h` header file provided with the iC386 com-

[1]This chapter omits the calls for deleting objects. For every type of iRMX object there is a system call to delete an object of that type that simply takes a token for the object to be deleted as its first parameter. The names of those calls always take the form of *rqdeletexxx()* with *xxx* equal to the name of the object type.

piler, with descriptive names added for the parameter type specifications. The rationale for using C prototypes rather than PLM prototypes is that C prototypes are more complete; C pointer declarations include the type of variable to which they point, unlike PLM pointers. PLM programmers can easily derive their prototypes from the ones given here, or can consult either the iRMX for Windows on-line help or the system calls reference manual for the PLM prototypes. The C function prototypes use type definitions for various data structures and data types defined in `:include:rmxc.h` or in one of the header files included from there. Most of these typedefs are self-explanatory, but those concerning word sizes and pointer types deserve special attention here.

iRMX I and iRMX II run on microprocessors with 16-bit words, while iRMX III runs on microprocessors with 32-bit words. For the function prototypes given in this book, however, a parameter declared to be of type `WORD` is always 16 bits long, regardless of the operating system and microprocessor being used. Likewise, parameters of type `DWORD` are always 32 bits long. There are, however, some system calls that take certain are 16-bit parameters for iRMX I and II, but 32-bit parameters for iRMX III. These parameters are declared to be of type `NATIVE_WORD`, which is predefined to the appropriate value by the different Intel C compilers.

The preceding situation is handled for PLM by the `WORD16` or `WORD32` compiler-defined symbols and appropriate code in the `rmxplm.ext` include file provided with the system. Coding problems arise when using the PLM386 compiler, however, because it uses 32-bit values for variables declared to be of type `WORD`. All of the sample PLM code in this book uses header files that declare data types called `WORD_16` and `WORD_32`, which are used to generate variables of the correct sizes.

The figures for this chapter include sample programs illustrating the use of some of the system calls discussed in the chapter. You might skip over them for now, and refer back to them to see examples of system calls as they are introduced. Figures 7.1 and 7.2 are equivalent PLM and C programs that illustrate creating a new task within a job. Figure 7.3 illustrates creating an I/O job whose initial task executes a procedure loaded into memory with the application itself. Figure 7.4 is a program that loads an I/O job from a disk file, and Figure 7.5 is a sample program that could be loaded by the program in Figure 7.4. The C programs equivalent to Figures 7.3 through 7.5 are left as exercises for the reader.

Figure 7.1 is a PLM program that displays some information about the job created to run an HI command, creates a new task for that job, synchronizes execution between the initial task and the created task using a binary semaphore, and exits. The output from running this program might look like the following: (The numerical values for the tokens will change from run to run of the program.)

Figure 7.1 PLM program to demonstrate iRMX multitasking.

```
/***>  plmtask.plm  <*********************************************
 *
 *  Sample program illustrating multitasking in PLM.
 *
 ***************************************************************/
$compact (exports mytask)
$title ('Sample Program Illustrating Task Creation')

plmtask: DO;

$include (plmtask.ext)

/*  Global Variables
    ----------------
*/
DECLARE
        CR          LITERALLY   '0Dh',
        LF          LITERALLY   '0Ah',

mess (*)     BYTE INITIAL (0,'This is the initial task: xxxx.', CR, LF,
                           '  I belong to job xxxx.', CR, LF,
                           '  My priority is xxxx.', CR, LF,
                           '  My maximum priority is xxxx.', CR, LF,
                           '  Now I will create a new task.', CR, LF),
taskmess (*) BYTE INITIAL (0, CR, LF, 'This is the new task: xxxx.',
                           CR, LF, '  My priority is xxxx.', CR, LF,
                           '  Now I will send a unit to the semaphore ',
                           'and delete myself.', CR, LF),
waitmess (*)BYTE INITIAL (0, CR, LF, 'This is the initial task again.',
                           CR, LF, '  I created task xxxx.', CR, LF,
                           '  Now I will wait for a unit from ',
                           'the semaphore.', CR, LF),

        hextab (*) BYTE INITIAL ('0123456789ABCDEF'),
        (myjob, mytoken, newtask)   TOKEN,
        syncSem                     TOKEN,
        myprio                      BYTE,
        maxprioptr                  POINTER,
        maxprio BASED maxprioptr    BYTE,
        (unitsLeft, Status)         WORD;

/*  Procedure to Convert a Hexadecimal Value to ASCII Characters
    ------------------------------------------------------------
*/
word2hex: PROCEDURE (value, where);
DECLARE
        value       WORD,
        i           INTEGER,
        where       POINTER,
        xxxx BASED where (1) BYTE;

  DO i = 3 TO 0 BY -1;
    xxxx(i) = hextab(value AND 0Fh);
    value = shr (value, 4);

  END;
END word2hex;
```

Figure 7.1 (*Continued*)

```
/*  Procedure to be Executed by the New Task
    ----------------------------------------
*/
mytask: PROCEDURE PUBLIC;
DECLARE
        mytoken     TOKEN,
        mypriority  BYTE,
        Status      WORD;

  mytoken = rqgettasktokens (0, @Status);
  mypriority = rqgetpriority (selector$of(NIL), @Status);
  CALL word2hex (WORD (mytoken), @taskmess(25));
  CALL word2hex (mypriority, @taskmess(49));
  CALL rqcsendcoresponse (NIL, 0, @taskmess, @Status);

  CALL rqsendunits (syncSem, 1, @Status);
  CALL rqdeletetask (selector$of(NIL), @Status);

END mytask;

/*  Initial Task Starts Here
    ------------------------------------------------------------
*/
  mess(0) = length (mess) -1;
  taskmess(0) = length (taskmess) -1;
  waitmess(0) = length (waitmess) -1;

  mytoken = rqgettasktokens (0, @Status);
  myjob = rqgettasktokens (1, @Status);
  myprio = rqgetpriority (selector$of(NIL), @Status);
  CALL word2hex (WORD (mytoken), @mess(27));
  CALL word2hex (WORD (myjob), @mess(52));
  CALL word2hex (myprio, @mess(76));
  maxprioptr = buildptr (myjob, 018h);  /* Illustrate cheating */
  CALL word2hex (maxprio, @mess(108));
  CALL rqcsendcoresponse (NIL, 0, @mess, @Status);

  syncSem = rqcreatesemaphore (0, 1, 0, @Status);
  newtask = rqcreatetask (myprio, @mytask,
                          selector$of(NIL), NIL, 4096, 0, @Status);

  CALL word2hex (WORD (newtask), @waitmess(53));
  CALL rqcsendcoresponse (NIL, 0, @waitmess, @Status);

  unitsLeft = rqreceiveunits (syncSem, 1, 0FFFFh, @Status);
  CALL rqcsendcoresponse (NIL, 0, @(46, CR, LF,
    'Unit received by initial task.', CR, LF,'  Exiting.', CR, LF),
    @Status);
  CALL rqexitiojob (0, NIL, @Status);

END plmtask;
```

Figure 7.2 C program equivalent to Fig. 7.1.

```
/***>  ctask.c  <*************************************************
 *
 *  Sample C program to demonstrate iRMX multitasking
 *
 ************************************************************************/

#include <rmxc.h>
#include <string.h>

#define MYTASK  0
#define MYJOB   1

char    hextab[] = "0123456789ABCDEF",
        xxxx[] = "xxxx";
TOKEN   syncSem;

/*  Utility to Generate ASCII Representation of a Hex Value
 *  -------------------------------------------------------
 */
void
word2hex (WORD a_word) {
int i;

  for (i=3; i >= 0; i--) {
    xxxx[i] = hextab [a_word & 0x0f];
    a_word = a_word >> 4;
    }
  }

/*  Sample Procedure to be Used as a Task
 *  -------------------------------------
 */
void far
mytask (void) {
WORD    Status;
BYTE    myprio = rqgetpriority ((selector) NULL, &Status);
TOKEN   mytoken = rqgettasktokens (MYTASK, &Status),
        myjob = rqgettasktokens (MYJOB, &Status);
char    mess[256];

  strcpy (mess, "This is a new task: ");
  word2hex ((WORD) mytoken);
  strcat (mess, xxxx);
  strcat (mess, ".\r\n  I belong to job ");
  word2hex ((WORD) myjob);
  strcat (mess, xxxx);
  strcat (mess, ".\r\n  My priority is ");
  word2hex ((WORD) myprio);
  strcat (mess, xxxx);
  strcat (mess, ".\r\n  Now I will send a unit to the semaphore \
and delete myself.\r\n\n");
  udistr (mess, mess);
  rqcsendcoresponse (NULL, 0, mess, &Status);
  rqsendunits (syncSem, 1, &Status);
  rqdeletetask ((selector) NULL, &Status);
  }

/*  Main Task Starts Here
```

Figure 7.2 (*Continued*)

```
 *  ---------------------
 */
int
main (int argc, char *argv[]) {
WORD    Status;
BYTE    myprio = rqgetpriority ((selector) NULL, &Status), maxprio;
TOKEN   mytoken = rqgettasktokens (MYTASK, &Status),
        myjob = rqgettasktokens (MYJOB, &Status),
        newtask;
char    mess[256];

  syncSem = rqcreatesemaphore (0, 1, 0, &Status);

  strcpy (mess, "This is the initial task: ");
  word2hex ((WORD) mytoken);
  strcat (mess, xxxx);
  strcat (mess, ".\r\n  I belong to job ");
  word2hex ((WORD) myjob);
  strcat (mess, xxxx);
  strcat (mess, ".\r\n  My priority is ");
  word2hex ((WORD) myprio);
  strcat (mess, xxxx);
  strcat (mess, ".\r\n  My maximum priority is ");
  rqsetpriority ((selector) NULL, 0, &Status);
  maxprio = rqgetpriority ((selector) NULL, &Status);
  rqsetpriority ((selector) NULL, myprio, &Status);
  word2hex ((WORD) maxprio);
  strcat (mess, xxxx);
  strcat (mess, ".\r\n  Now I will create a new task.\r\n\n");
  udistr (mess, mess);
  rqcsendcoresponse (NULL, 0, mess, &Status);

  newtask = rqcreatetask (0,
                          mytask, (selector) NULL,
                          NULL, 4096, 0, &Status);

  strcpy (mess, "This is the initial task again.\r\n  I created task ");
  word2hex ((WORD) newtask);
  strcat (mess, xxxx);
  strcat
    (mess, ".\r\n  Now I will wait for a unit from the semaphore.\r\n");
  udistr (mess, mess);
  rqcsendcoresponse (NULL, 0, mess, &Status);

 rqreceiveunits (syncSem, 1, 0xFFFF, &Status);
  strcpy (mess, "Unit received by initial task.\r\n  Exiting.\r\n");
  udistr (mess, mess);
  rqcsendcoresponse (NULL, 0, mess, &Status);
  rqexitiojob (0, NULL, &Status);

  return 0;
  }
```

Figure 7.3 Sample program demonstrating the creation of an I/O job, and receiving a termination message from the child job using a mailbox.

```
/***> IOJOB.PLM  <*****************************************************
 *
 *  This is an HI command that demonstrates creation of a child I/O job
 *  and passing information from the child to the parent through the job
 *  completion mailbox.
 *

*************************************************************************/
$compact (exports mytask)
$title ('Sample Program to Create an I/O Job')

iojob: DO;
$include (iojob.ext)

/*  Global Variables
    ----------------
*/
DECLARE
        CR          LITERALLY   '0Dh',
        LF          LITERALLY   '0Ah',

mess (*)     BYTE INITIAL (0,'This is the initial task: xxxx.', CR, LF,
                           '  I belong to job xxxx.', CR, LF,
                           '  My priority is xxxx.', CR, LF,
                           '  My maximum priority is xxxx.', CR, LF,
                           '  Now I will create a new I/O job.', CR, LF),
exitmess (*)BYTE INITIAL (0, CR, LF, 'This is the initial task again.',
                          CR, LF, '  I created I/O job xxxx.', CR, LF,
                           '  It''s exit code was xxxx.', CR, LF,
                           '  Now I will print its exit message ',
                          'and exit myself.',
                          CR, LF),
taskmess (*)BYTE INITIAL (0, CR, LF, 'I/O job: xxxx.', CR, LF,
                           '  Task: xxxx.', CR, LF,
                           '  Priority: xxxx.', CR, LF,
                           '  Max priority: xxxx.', CR, LF, LF),

        hextab (*) BYTE INITIAL ('0123456789ABCDEF'),
        (myjob, mytoken,
        newjob, jobmbx, exittok)    TOKEN,
        exitstruct BASED exittok STRUCTURE (
            termination$code    WORD_16,
            user$fault$code     WORD_16,
            job$token           TOKEN,
            return$data$len     BYTE,
            return$data (1)     BYTE),
        myprio                      BYTE,
        maxprio                     BYTE,
        Status                      WORD_16;

/*  Procedure to Convert a Hexadecimal Value to ASCII Characters
    ------------------------------------------------------------
*/
word2hex: PROCEDURE (value, where);
DECLARE
        value       WORD_16,
```

Figure 7.3 (*Continued*)

```
        i           INTEGER,
        where       POINTER,
        xxxx BASED where (1) BYTE;

  DO i = 3 TO 0 BY -1;
    xxxx(i) = hextab(value AND 0Fh);
    value = shr (value, 4);
  END;
END word2hex;

/*      Procedure to be Executed by the Initial Task of the I/O Job
        -----------------------------------------------------------
*/
mytask: PROCEDURE PUBLIC;
DECLARE
        (mytasktoken, myjobtok)         TOKEN,
        mypriority                      BYTE,
        maxpriority                     BYTE,
        Status                          WORD_16;

  taskmess(0) = length (taskmess) -1;
  mytasktoken = rqgettasktokens (0, @Status);
  myjobtok = rqgettasktokens (1, @Status);
  mypriority = rqgetpriority (selector$of(NIL), @Status);
  CALL rqsetpriority (selector$of(NIL), 0, @Status);
  maxpriority = rqgetpriority (selector$of(NIL), @Status);
  CALL word2hex (WORD (myjobtok), @taskmess(12));
  CALL word2hex (WORD (mytasktoken), @taskmess(27));
  CALL word2hex (mypriority, @taskmess(46));
  CALL word2hex (maxpriority, @taskmess(69));

  CALL rqexitiojob (01234h, @taskmess, @Status);

END mytask;

/*
    Initial Task of the Parent Job Starts Here
    ------------------------------------------
*/
  mess(0) = length (mess) -1;
  exitmess(0) = length (exitmess) -1;

  mytoken = rqgettasktokens (0, @Status);
  myjob = rqgettasktokens (1, @Status);
  myprio = rqgetpriority (selector$of(NIL), @Status);
  CALL word2hex (WORD (mytoken), @mess(27));
  CALL word2hex (WORD (myjob), @mess(52));
  CALL word2hex (myprio, @mess(76));
  CALL rqsetpriority (selector$of(NIL), 0, @Status);
  maxprio = rqgetpriority (selector$of(NIL), @Status);
  CALL word2hex (maxprio, @mess(108));
  CALL rqcsendcoresponse (NIL, 0, @mess, @Status);
  jobmbx = rqcreatemailbox (0, @Status);
  newjob = rqcreateiojob (1024, 0FFFFFFFFh, NIL, 0, 0, @mytask,
            selector$of(NIL), NIL, 4096, 0, jobmbx, @Status);
  exittok = rqreceivemessage (jobmbx, 0FFFFh, NIL, @Status);

  CALL word2hex (WORD (newjob), @exitmess(56));
```

Figure 7.3 (*Continued*)

```
  CALL word2hex (exitstruct.user$fault$code, @exitmess(84));
  CALL rqcsendcoresponse (NIL, 0, @exitmess, @Status);
  CALL rqcsendcoresponse (NIL, 0, @exitstruct.return$data$len, @Status);

  CALL rqexitiojob (0, NIL, @Status);

END iojob;
```

Figure 7.4 Sample program that uses the AL to create a child job and load it from a disk file.

```
/***>  loadjob.plm  <************************************************
 *
 *  Sample PLM program to load an I/O job from file 'newjob'
 *
 ***********************************************************************/
$title ('Sample Program to Load an I/O Job')

loadjob: DO;
$include (loadjob.ext)

/*      Global Variables
        ----------------
*/
DECLARE
        CR          LITERALLY   '0Dh',
        LF          LITERALLY   '0Ah',

mess (*)    BYTE INITIAL (0,'This is the initial task: xxxx.', CR, LF,
                          '  I belong to job xxxx.', CR, LF,
                          '  My priority is xxxx.', CR, LF,
                          '  My maximum priority is xxxx.', CR, LF,
                          '  Now I will create a new I/O job.', CR, LF),
exitmess (*)BYTE INITIAL (0, CR, LF, 'This is the initial task again.',
                          CR, LF, '  I created I/O job xxxx.', CR, LF,
                          '  It''s exit code was xxxx.', CR, LF,
                          '  Now I will print its message and exit.',
                          CR, LF),
hextab (*) BYTE INITIAL ('0123456789ABCDEF'),

        (myjob, mytoken,
        newjob, jobmbx, exittok)    TOKEN,
        exitstruct BASED exittok STRUCTURE (
            termination$code    WORD_16,
            user$fault$code     WORD_16,
            job$token           TOKEN,
            return$data$len     BYTE,
            return$data (1)     BYTE),
        myprio                      BYTE,
        maxprio                     BYTE,
        Status                      WORD_16;

/*  Procedure to Convert a Hexadecimal Word to 4 ASCII Characters
    ------------------------------------------------------------
 */
word2hex: PROCEDURE (value, where);
DECLARE
        value       WORD_16,
```

Figure 7.4 (*Continued*)

```
      i           INTEGER,
      where       POINTER,
      xxxx BASED where (1) BYTE;

DO i = 3 TO 0 BY -1;
  xxxx(i) = hextab(value AND 0Fh);
  value = shr (value, 4);

  END;
END word2hex;

/*  Initial Task of Parent Job Starts Here
    --------------------------------------
*/
  mess(0) = length (mess) -1;
  exitmess(0) = length (exitmess) -1;

  mytoken = rqgettasktokens (0, @Status);
  myjob = rqgettasktokens (1, @Status);
  myprio = rqgetpriority (selector$of(NIL), @Status);
  CALL word2hex (WORD (mytoken), @mess(27));
  CALL word2hex (WORD (myjob), @mess(52));
  CALL word2hex (myprio, @mess(76));
  CALL rqsetpriority (selector$of(NIL), 0, @Status);
  maxprio = rqgetpriority (selector$of(NIL), @Status);
  CALL word2hex (maxprio, @mess(108));
  CALL rqcsendcoresponse (NIL, 0, @mess, @Status);

  jobmbx = rqcreatemailbox (0, @Status);
  newjob = rqsloadiojob (@(6,'newjob'), 0, 0, NIL, 0, 0,
           0, jobmbx, @Status);
  exittok = rqreceivemessage (jobmbx, 0FFFFh, NIL, @Status);

  CALL word2hex (WORD (newjob), @exitmess(56));
  CALL word2hex (exitstruct.user$fault$code, @exitmess(84));
  CALL rqcsendcoresponse (NIL, 0, @exitmess, @Status);
  CALL rqcsendcoresponse (NIL, 0, @exitstruct.returndatalen, @Status);

  CALL rqexitiojob (0, NIL, @Status);

END loadjob;
```

Figure 7.5 Sample program that could be loaded as an I/O job by the program in Fig. 7.4.

```
/***>  newjob.plm  <************************************************
 *
 *  Sample program to be used as a child I/O job for the loadjob program
 *
 ************************************************************************/
$title ('Sample Program to Serve as a Child I/O Job')

newjob: DO;
$include (newjob.ext)

/*  Global Variables
    ----------------
*/
```

Figure 7.5 (*Continued*)

```
DECLARE
        CR              LITERALLY   '0Dh',
        LF              LITERALLY   '0Ah',

taskmess (*)BYTE INITIAL (0, CR, LF,
                          'Job: xxxx.', CR, LF,
                          '  Task: xxxx.', CR, LF,
                          '  Priority: xxxx.', CR, LF,
                          '  Max priority: xxxx.', CR, LF,
                          '  Exit.', CR, LF),
        hextab (*) BYTE INITIAL   ('0123456789ABCDEF'),
        (mytoken, myjobtok)             TOKEN,
        mypriority                      BYTE,
        maxpriority                     BYTE,
        Status                          WORD_16;


/*  Routine to Convert a Binary Word to 4 Characters Representing
    its Hexadecimal Value
    ------------------------------------------------------------
*/
word2hex: PROCEDURE (value, where);
DECLARE
        value       WORD_16,
        i           INTEGER,
        where       POINTER,
        xxxx BASED where (1) BYTE;

  DO i = 3 TO 0 BY -1;
    xxxx(i) = hextab(value AND 0Fh);
    value = shr (value, 4);
  END;
END word2hex;

/*  Initial Task of the I/O Job Starts Here
    ---------------------------------------
*/
  taskmess(0) = length (taskmess) -1;
  mytoken = rqgettasktokens (0, @Status);
  myjobtok = rqgettasktokens (1, @Status);
  mypriority = rqgetpriority (selector$of(NIL), @Status);
  CALL rqsetpriority (selector$of(NIL), 0, @Status);
  maxpriority = rqgetpriority (selector$of(NIL), @Status);
  CALL word2hex (WORD (myjobtok), @taskmess(8));
  CALL word2hex (WORD (mytoken), @taskmess(23));
  CALL word2hex (mypriority, @taskmess(42));
  CALL word2hex (maxpriority, @taskmess(65));

  CALL rqexitiojob (01234h, @taskmess, @Status);

END newjob;
```

```
This is the initial task: D018.
  I belong to job AFB0.
  My priority is 008E.
  My maximum priority is 008D.
  Now I will create a new task.
This is the initial task again.
  I created task D048.
  Now I will wait for a unit from the semaphore.
This is the new task: D048.
  My priority is 008E.
  Now I will send a unit to the semaphore and delete myself.
Unit received by initial task.
  Exiting.
```

Figure 7.1 also illustrates cheating as defined in chapter 6. To determine the maximum task priority for the job, the program examines a location within the job object's data structure. Although the code works for current versions of iRMX III and iRMX for Windows, it does not work for current versions of iRMX I and iRMX II. Furthermore, it may very well fail to work in future versions of iRMX III and iRMX for Windows. Thus, the program illustrates the "wrong" way to code an iRMX application. It is coded this way simply to illustrate the fact that the internal data structure of an iRMX object really is accessible to application programs. Such cheating should never be done in actual iRMX applications.

The C version of the program in Figure 7.2 illustrates the proper way to code this application. It uses the *rqsetpriority( )* system call to set the initial task's priority to 0. The value 0 for the new priority is interpreted by this system call to mean the maximum priority allowed for the task. The task then calls *rqsetpriority( )* to determine what priority it has, and thus determines what the maximum priority is for the job. The output from running this program might look like this:

```
This is the initial task: E040.
  I belong to job BA28.
  My priority is 008E.
  My maximum priority is 008D.
  Now I will create a new task.
This is a new task: F310.
  I belong to job BA28.
  My priority is 008D.
  Now I will send a unit to the semaphore and delete myself.
This is the initial task again.
  I created task F310.
  Now I will wait for a unit from the semaphore.
Unit received by initial task.
  Exiting.
```

The output from running the program in Figure 7.3 might look like this:

```
This is the initial task: B010.
  I belong to job AED8.
  My priority is 008E.
  My maximum priority is 008D.
  Now I will create a new I/O job.
```

```
This is the initial task again.
  I created I/O job B040.
  It's exit code was 1234.
  Now I will print its exit message and exit myself.
I/O job: B040.
  Task: B058.
  Priority: 008D.
  Max priority: 008D.
```

The internal logic of some of the system calls given in this chapter and elsewhere should be described here. The descriptions are based on the published documentation for the system calls, augmented by a bit of sleuthing with the System Debugger's *υ-* commands that display information about iRMX objects. They are not based on examination of the operating system's source code nor on disassembly of any part of the operating system's memory. Thus, these descriptions do not necessarily describe how the OS is actually implemented, but rather how it *could* be implemented. The idea is to build a realistic model of how iRMX works rather than simply present a set of rules that you apply by rote when writing iRMX programs. The model should help you develop iRMX code more effectively without compromising the encapsulation provided by the operating system's object-based design.

7.2 Task Synchronization and Communication

Real-time systems need efficient and powerful multitasking facilities to support programs that use separate threads of execution to manage the different classes of events that drive the logic of most real-time applications. Of course, many nonreal-time applications are event-driven too, notably those with graphical user interfaces. Thus, the multitasking features described in this section make iRMX appealing for both real-time and general application development.

Before you can see how iRMX supports task synchronization and communication, you need to see how tasks are created, explicitly placed into certain scheduling states, and deleted. The following is the function prototype for the Nucleus call to create a task:

```
extern TOKEN
rqcreatetask (          BYTE                priority,
                        far *               startAddress,
                        TOKEN               dataSeg,
                        WORD far *          stackPtr,
                        NATIVE_WORD         stackSize,
                        WORD                taskFlags,
                        WORD far *          exceptPtr);
```

This function returns a token for a new task object. The new task is a sibling of the task that made the system call in the sense that both belong to the same job. No ownership or parent-child relationship exists among the tasks of an iRMX system.

`taskPriority` specifies the initial priority for the task. A value of 0 gives the task the maximum priority for the job that owns the task that made the call to *rqcreatetask()*. Unless it is 0, the value of this parameter must not be less than the maximum priority for the job. If this parameter is 0, the new task is given a priority equal to the highest allowed priority (numerically lowest value) allowed for the current job. If the new task has a higher priority than the task that creates it, the new task will preempt the creating task. If the new task has a priority less than or equal to the task that creates it, the new task is made ready and executes when it becomes the task with the highest priority on the ready queue.

The `startAddress` parameter is a pointer to the first instruction to be executed by the new task. This parameter is normally coded as a pointer to a function or procedure, but this is not strictly necessary. (See the following description of the `dataSeg` parameter for one reason for using a pointer to a procedure, though.) Note that there is no restriction on where in memory the new task must start executing. It can start in the same code segment as the creating task (as in Figures 7.1 and 7.2), or in a different code segment. If it starts in a different code segment, there is no requirement for that segment to belong to the current job, although it usually does.

The `dataSeg` parameter is a selector for the data segment to be used for the new task. There are two different ways to handle this parameter. One way is to supply the token for an iRMX memory segment that is to serve as the new task's data segment. This approach is most often used when the task is going to execute code in the same module as the creating task, and the compact model of compilation is being used. In this case, the value to use for the `dataSeg` parameter is obtained by extracting the selector part of a far pointer to any static variable in the application. For PLM programs, this might be coded as `selector$of(@Status)`, and for C programs as `(selector) &Status`, assuming `Status` is a static variable (i.e., not in the calling task's stack segment).

The second way to handle this parameter is to have the new task execute code that initializes the `ds` register itself and to code this parameter as the selector part of a null pointer (16 bits of zeros). Far procedures always include this initialization code in their prologues, so this technique works in any of the following situations:

- If the `startAddress` parameter is a pointer to a procedure or function that was declared public using the large model of compilation (either PLM or C).
- If the `startAddress` parameter was explicitly declared to be a far function in C.
- If the `startAddress` parameter was made a far function by exporting it from a compact subsystem using the compilers' extended segmentation features.

The `stackPtr` and `stackSize` parameters are needed because every iRMX task has its own stack segment, as described in chapter 6. If `stackPtr` is a valid pointer (not a null pointer), the new task will be given a stack in the memory segment specified by the selector part of the pointer, and the `sp` (top of stack pointer) register will be set to the sum of the offset part of the pointer and the value of the `stackSize` parameter. This option is useful for applications that want the new task to use part of an existing segment for its stack. For example, an application could declare an array of words to be used as the stack for a task and pass a pointer to the beginning of the array as `stackPtr` and the size of the array (in bytes) as the `stackSize`. The problem with this technique is that the new task can easily overwrite other information in its stack segment without the microprocessor detecting an error, even in protected-mode systems. On the other hand, this technique could be used to pass parameters to a new task. Doing so would require some tricky coding.

The other way to handle the new task's stack is to code `stackPtr` as a null pointer. In this case, the Nucleus creates a segment for the new task's stack with a size equal to the `stackSize` parameter. The advantage of this technique is that the new task will cause a stack fault (fault number 12) if it overflows its stack segment, which is trapped by the hardware. Remember, hardware traps like this are good: they let you know as soon as a bug occurs so it can be located easily and fixed.

Except for programs that use the stack for recursion, the size of a stack segment that a task needs depends on the following:

- The procedure calls it makes itself (this quantity is given at the end of the listing file produced by the compiler).
- Space for additional stack frames for nested calls.
- Space to store the state of the processor if an interrupt occurs.

For example, a call to an EIOS function requires stack space for the initial stack frame, plus one for the corresponding BIOS call that the EIOS makes, plus another level of frames for calls to the Nucleus from the BIOS. *iRMX Programming Techniques and Examples* (volume 11 of the iRMX for Windows documentation set) includes an appendix that can be used to determine how much stack space a task actually needs. In practice, tuning this parameter is important only for applications with a large number of tasks and a limited amount of memory. Creating a few 8 kilobytes (KB) or 16 KB stacks (which are generous values) is insignificant in a system with multiple megabytes of RAM.

The `taskFlags` parameter contains only one bit of information. The value should be set to 1 if the task contains floating-point instructions, and set to 0 otherwise. The setting of this bit affects the time the system needs to perform a context switch involving the task because of the separate set of registers for floating-point operations (found in the CPU for 80486 pro-

cessors, and in the coprocessor for other CPUs). If the task does not perform floating-point operations, these registers do not need to be saved in memory when the task is preempted, and they do not need to be reloaded when the task runs again. The time savings can be significant for real-time applications.

The `exceptPtr` parameter is the ubiquitous last parameter of every iRMX system call, as described in the exception handling section of chapter 6.

A task can put itself into the asleep scheduling state by calling *rqsleep( )*:

```
extern void
rqsleep (                WORD                timeLimit,
                         WORD far *          exceptPtr);
```

The `timeLimit` parameter is the amount of time the calling task is placed in the asleep scheduling state. When the time limit expires, the task is placed on the ready queue, and is then scheduled to run again when it has the highest priority of all ready tasks. (The various scheduling states were described in chapter 6.)

The amount of time the task spends asleep is equal to the product of `timeLimit` and the resolution of the system's real-time clock, normally 10 milliseconds (0.01 sec). There are two exceptions, however.

First, a value of 0 simply moves the running task to the end of the portion of the ready queue for its own priority. That is, all tasks on the ready queue with the same priority are arranged in first-in, first-out (FIFO) order, and a `timeLimit` of 0 puts the task at the end of its portion of the queue. If no other tasks have the same priority on the ready queue, the running task just continues to run.

Second, a value of 0xFFFF is illegal for this system call. For system calls that include a time limit parameter, such as *rqlookupobject( )* presented in chapter 6, a value of 0xFFFF means "indefinitely" or "forever." Since the only way a task completes a call to *rqsleep( )* is for its time limit to expire, it makes no sense for it to sleep forever. If a task does not want to execute any more, it should delete or suspend itself. A task can suspend itself or any other task by calling *rqsuspendtask( )*:

```
void
rqsuspendtask (          TOKEN               task,
                         WORD far *          exceptPtr);
```

A task can suspend itself by using the selector part of a null pointer for the task parameter. This strategy is just a convenience: the task would have to call *rqgettasktokens( )* to get the token for itself otherwise. The only way a suspended task can execute again is for another task to call *rqresumetask( )*:

```
void
rqresumetask (           TOKEN               task,
                         WORD far *          exceptPtr);
```

If a task is suspended multiple times, it must be resumed an equal number of times to become ready again. This feat can be accomplished in a scenario in which one task, called the lead task, partitions chunks of work into subchunks that are assigned to other tasks for processing. The lead task is suspended once for each subchunk of work and resumed once by each task that completes processing a subchunk. When the lead task runs again, it partitions the next chunk of work into another set of subchunks, and so on.

Another fundamental way to change a task's scheduling state is to change its priority:

```
void
rqsetpriority (          TOKEN              task,
                         BYTE               priority,
                         WORD far *         exceptPtr);
```

Again, the selector part of a null pointer causes the calling task's priority to change, subject to the limits for its job. A value of 0 sets the task's priority to the maximum allowed for its job. Note that any task can change the priority for any other task for which it can obtain a token, including tasks that belong to other jobs.

As a rule, tasks do not change their priorities dynamically to control execution sequences. The synchronization mechanisms described in the following sections (semaphores, mailboxes, and regions) are used for that purpose. In fact, there is surprisingly little use for this call by most applications, and misuse of it can negatively impact overall system performance.

There are two situations in which the Nucleus changes the priority of a task automatically. One is a dynamic priority change related to regions (described later in the chapter). The other is a static change for tasks that become interrupt tasks by calling *rqsetinterrupt()*. Interrupt tasks take on a priority associated with the particular interrupt level that are associated with, as described in chapter 9. This priority change is a static change because the task's priority is never changed again (unless it enters a region).

To handle situations in which a task must to have a higher priority than allowed for its job, there is a system call to change the maximum priority for a job:

```
void
rqesetmaxpriority (      TOKEN              job;
                         BYTE               priority,
                         WORD far *         exceptPtr);
```

Setting the maximum task priority for a job does not change the actual priority of any tasks, it simply makes it possible for tasks belonging to the job to have higher priorities than otherwise possible. This call cannot be used to lower the maximum task priority for a job. It was added to the operating system to support loadable device drivers. Normally, jobs loaded while the system is running do not have high enough maximum task prior-

ities to allow them to own interrupt tasks, but this call makes it possible for them to do so. See chapter 9 for more information on this topic.

7.2.1 Semaphores

Semaphores are the first of the task synchronization and communication objects to be examined. They, along with mailboxes and regions, fall into the generic category of *exchanges* in the iRMX documentation. Exchanges are mechanisms managed by the Nucleus that enable tasks to synchronize with each other and/or pass information from one to another. Because these mechanisms are managed by the Nucleus, tasks that use them do not need polling loops, which would overload the CPU, to test when other threads of execution have triggered events.

Semaphore objects include a counter of the number of units that reside in the semaphore. Tasks can send units to a semaphore and request to receive units from it. A task that asks to receive more units than are available at the semaphore blocks (enters the asleep state) until either enough units accumulate at the semaphore to satisfy the request or until the time limit specified with the request expires. While a task is asleep, it makes no demands on the CPU. If enough units exist at the semaphore to satisfy a request when it is made, the counter is simply reduced by the number of units requested, and the calling task continues running.

Classically, there are two types of semaphores, *binary* and *counting*. All iRMX semaphores are counting semaphores, but the application can place an upper limit on the number of units the semaphore can hold at one time when it is created. With a limit of 1, a counting semaphore is the same as a binary semaphore.

Like other iRMX objects, no intrinsic limit is placed on the number of semaphores a job can own other than a possible limit on the total number of objects the job can own. This limit is specified at the time the job is created (see below). Semaphores can be shared across jobs, provided you can communicate the token for the semaphore between jobs. When it is necessary to share a semaphore this way, the usual technique is to catalog the token for the semaphore using an agreed-upon name in an agreed-upon job's object directory.

The following are the system calls for semaphores:

```
Extern TOKEN
rqcreatesemaphore (          WORD              initialValue,
                             WORD              maxValue,
                             WORD              semaphoreFlags,
                             WORD far *        exceptPtr);
```

The `initialValue` and `maxValue` semaphores initialize the semaphore counter and its upper limit, respectively. The proper settings for these parameters depend on how the application will use the semaphore. For a binary semaphore used to protect a shared-memory variable, for ex-

ample, the values would both be set to 1, indicating that the variable is available to the first task that wants to access it. A different example is shown in Figures 7.1 and 7.2, where a semaphore with initial and maximum values of 0 and 1 is used so the creating task will sleep until the created task signals when it has completed its initialization phase.

The `semaphoreFlags` controls the order of multiple tasks queued at the semaphore. A value of 0 means by FIFO, regardless of priority, and a value of 1 means by priority, with tasks of equal priority arranged in FIFO order. The task at the head of the queue always has its request satisfied first, even if enough units exist at the semaphore to satisfy a request for fewer units, but made by a task further down the queue.

```
extern void
rqsendunits (           TOKEN           semaphore,
                        WORD            units,
                        WORD far *      exceptPtr);

extern WORD
rqreceiveunits (        TOKEN           semaphore,
                        WORD            units,
                        WORD            timeLimit,
                        WORD far *      exceptPtr);
```

The first parameter for these two calls is the token for a semaphore returned by an earlier call to *rqcreatesemaphore()*. The number of units sent to a semaphore must never make the semaphore's counter exceed its `maxValue`, or *rqsendunits()* will fail with the condition code set to 0x0004 (E_LIMIT).

The value returned by *rqreceiveunits()* signifies how many units remain in the semaphore after removing the units requested. An application that wants to test a semaphore without blocking and without encountering an exception can ask to receive 0 units from a semaphore.

The `timeLimit` parameter is the standard time-limit parameter for many Nucleus system calls. It specifies the number of 0.01-second clock ticks the task is willing to wait for the requested units to arrive at the semaphore. A value of 0xFFFF means the task is to sleep for as long as necessary for the units to arrive.

7.2.2 Mailboxes

Mailboxes provide an efficient mechanism for tasks both to synchronize their execution and exchange data. Tasks send messages (consisting of iRMX tokens) or data (up to 128 bytes of arbitrary data) to a mailbox. Other tasks can then receive the messages or data at a later time. Each mailbox has a queue, which is one of the following:

- Empty.
- A list of messages or data items sent to the mailbox but not yet received.

- A list of tasks that have attempted to receive a message or data but have been put to sleep because there is nothing to receive yet.

The availability of an efficient message-passing mechanism fundamentally impacts the design of iRMX multitasking applications compared to, for example, process synchronization in Unix systems. Consider an event-driven program that waits for input from any of several devices, processes whichever device generates data first, and then waits for the next input. In BSD Unix, this is accomplished using the *select( )* system call, which allows one process to block until the kernel determines that one of the selected channels is ready for I/O. The process then tests the value returned by *select( )*, reads from the proper channel, processes the input, and returns to *select( )*.

An iRMX application handles the equivalent situation by having a set of tasks monitor the various I/O connections. Each monitoring task sends a message to a single mailbox when it completes a data transfer. The event-processing task waits at this mailbox for messages, and processes each event when it arrives. If additional messages arrive while the event-processing task is busy with an event, these messages are automatically queued at the mailbox. The efficiency of mailbox operations and the low overhead associated with iRMX multitasking make it possible to achieve very high processing throughput using this technique.

Each mailbox can handle either messages or data, but not both. The choice is determined by a parameter when the mailbox object is created. Data mailboxes provide no new functionality compared to message mailboxes because a message mailbox can always be used for exchanging memory segment objects that contain data. So why are there two types of mailboxes? Generally, message mailboxes are more efficient to use than data mailboxes even when they are used for sending memory segments. The message bytes are not copied to and from a message mailbox as they are for data mailboxes (only the two-byte token is copied), and no limit exists on the size of message segments that can be sent to a message mailbox. (There is a limit of 128 bytes on data items.)

On the other hand, data mailboxes are useful for applications that perform a lot of one-way message passing. A significant amount of overhead exists, especially for protected-mode versions of the operating system, if a memory segment must be created for each message sent and if the receiving task must then delete each segment it receives to avoid depleting the sending job's memory pool. For such applications, a buffer pool for managing message segments provides a good way to mitigate this problem, but data mailboxes provide a simpler solution for cases where the data traffic is relatively low and the messages are small. Buffer pools are described later in this chapter. The following are the iRMX system calls for mailboxes:

```
extern TOKEN
rqcreatemailbox (          WORD              mailboxFlags
                           WORD far *        exceptPtr );
```

`mailboxFlags` determine three features about the mailbox being created:

1. The queuing rule to be used when tasks are queued at the mailbox.
2. Whether the mailbox is to be used for sending and receiving messages (tokens for iRMX objects) or data (byte arrays).
3. The size of the high-performance queue to be created for the mailbox.

The value for the `mailboxFlags` is the sum of three values: 0 for a message mailbox or 32 for a data mailbox, plus 0 for a FIFO task queue or 1 for a priority-base task queue, plus a value for the size of the high-priority message queue. The high-priority queue applies only to message mailboxes and signifies how much memory to reserve for the queue of messages waiting for tasks to receive them.

iRMX always reserves room for at least eight messages, but you can increase the size (thereby saving the time needed to create a memory segment if the queue overflows) by adding to this parameter a value that is one-half the number of objects you want the queue to hold. Note that the value you add must be an even number. Data mailboxes are always created with room to queue three 128-byte data items, but the queue is automatically expanded if necessary. To send a token to an object mailbox, use *rqsendmessage( )*:

```
extern void
rqsendmessage (            TOKEN             mailbox,
                           TOKEN             object,
                           TOKEN             response,
                           WORD far *        exceptPtr );
```

The token passed as `object` is sent to the message mailbox, `mailbox`. iRMX provides a return-receipt mechanism for message mailboxes through the `response` parameter of this call and a corresponding parameter for *rqreceivemessage( )*. The sending task specifies a token for an iRMX semaphore, mailbox, or region (usually a semaphore) for this parameter, and this token is delivered to the receiving task, along with the token passed using the `object` parameter. The tasks must adopt the convention that the receiving task will send something back to the `response` exchange (typically one unit to the semaphore) to let the sender know when the message has been delivered.

The response mechanism can also be used as a simple alternative to the buffer pool mechanism described later in this chapter. In this case, `response` is a token for another mailbox to which segment tokens received at this mailbox are sent for recycling when the receiving task finishes with

them. Applications that do not want to use this feature of message mailboxes code this parameter as the selector of a null pointer.

```
extern TOKEN
rqreceivemessage (          TOKEN              mailbox,
                            WORD               timeLimit,
                            TOKEN far *        responsePtr,
                            WORD far *         exceptPtr);
```

The preceding function returns the next token available on the message queue of the indicated `mailbox`. `timeLimit` indicates how long the task is willing to wait for a message, in 0.01-second units, if no message is present when the call is made. A value of 0xFFFF indicates no time limit. If the sender specified a token for a response exchange, and if this call is coded with a valid pointer for `responsePtr`, the token for the exchange will be stored in the location pointed to by `responsePtr`.

```
extern void
rqsenddata (                TOKEN              mailbox,
                            BYTE far *         dataPtr,
                            WORD               actualLength,
                            WORD far *         exceptPtr);
```

The array of bytes pointed to by `dataPtr` is sent to the data mailbox, `mailbox`. The number of bytes sent is specified by `actualLength`, which is automatically limited to 128 if a greater value is specified. The data item is not interpreted as an iRMX string, so the first byte at `dataPtr` is not interpreted as the length of the item being sent. (It is not interpreted as a null-terminated C string either, for that matter.) Only the value of `actualLength` determines how many bytes are sent to the mailbox. No provision exists for a `response` mechanism for data mailboxes.

```
extern WORD
rqreceivedata (             TOKEN              mailbox,
                            BYTE far *         dataPtr,
                            WORD               timeLimit,
                            WORD far *         exceptPtr);
```

The preceding call retrieves a data item from the data mailbox, `mailbox`. The message is copied into the array of bytes pointed to by `dataPtr`, and the actual number of bytes copied is returned as the value of the function. `timeLimit` specifies the amount of time the receiving task is willing to wait for data in 0.01-second units, with 0xFFFF signifying no limit.

Note: This call fails if less than 128 bytes of memory can be accessed starting at `dataPtr`, so you should reserve 128 bytes for received data even if you know that all calls to *rqsenddata()* will send fewer bytes.

7.2.3 Regions

The iRMX region object type deals with a problem associated with semaphores, called *priority inversion*, that can occur when semaphores are used to implement mutual-exclusion algorithms.

To see the problem, consider three tasks with relatively low, medium, and high priorities. Assume that the tasks called *low* and *high* both need to manipulate a shared resource of some type, and use a semaphore to ensure mutually exclusive access to it. If *low* receives a unit from a binary semaphore, *high* will block if *high* now attempts to also receive a unit. This situation is normal; *high* should not access the resource protected by the semaphore until *low* finishes the operation it has begun on the resource and sends a unit back to the semaphore. The inversion occurs if *medium* becomes ready while *low* is running and *high* is blocked. *Medium* will preempt *low* and *high* will now be waiting for *medium* to execute even though *medium* has nothing to do with *high*'s access to the shared resource.

An iRMX priority region object can be used instead of a semaphore to solve this problem. The priority of a task that controls access to a resource protected by a priority region object automatically has its priority raised to match that of any higher-priority task that tries to obtain control of the region at the same time. Using this mechanism in the previous example, task *medium* would be prevented from preempting *low* because *low*'s priority would have been raised to match *high*'s. When *low* relinquishes control of the region, *low*'s priority reverts to its normal level, and *high* receives control of the region and executes without ever waiting for *medium.*

All regions have a property that makes it important to use them carefully: a task that holds a region cannot be suspended or deleted, which also means that the job that owns the task cannot be deleted. For this reason, regions can be particularly troublesome when debugging HI commands because an error could cause a task to take control of the region and fail to release it, making it impossible to terminate the command with the usual <^C> mechanism from the console. The entire system must be rebooted when this situation occurs. The following are the system calls for regions:

```
extern TOKEN
rqcreateregion (            WORD                regionFlags,
                            WORD far *          exceptPtr );
```

This call creates the equivalent of a binary semaphore, with an optional provision for avoiding priority inversion as just described. The `regionFlags` parameter is set to 0 if the queue of tasks waiting at the region uses FIFO order, or set to 1 if it uses priority-based order. Note that FIFO regions do not deal with the priority-inversion problem. Only priority regions can cause a shift in priority for the task that occupies the region.

```
extern void
rqreceivecontrol (          TOKEN               region,
                            WORD far *          exceptPtr);
```

If `region` is available, the calling task takes control of the region and proceeds to execute. If the region is not available, the calling task goes to

sleep until it is at the head of the region's queue and the region is released. Note that there is no time-limit parameter associated with this call. Tasks that try to enter a region using this call must be willing to sleep indefinitely. (If a task is not willing to wait, it might instead call *rqacceptcontrol()*:)

```
extern void
rqacceptcontrol (           TOKEN               region,
                            WORD far *          exceptPtr );
```

The preceding call is almost the same as *rqreceivecontrol()*, but the call returns immediately if the region is not free, rather than putting the calling task to sleep. The condition code is set to 0x0003 (E_BUSY) if the region is occupied when this call is made, rather than the usual 0x0001 (E_TIME) that is returned when a time limit of 0 results in the failure of a system call such as *rqreceiveunits()* or *rqreceivemessage()*.

```
extern void
rqsendcontrol (             TOKEN               region,
                            WORD far *          exceptPtr);
```

This call allows another task to obtain control of a region. It can only be made by the task currently in control of `region`. If the calling task's priority has been temporarily raised while it occupied the region, this is the time at which it resumes its normal value. If a task has entered two or more regions, this call causes it to exit the most-recently entered region. A task that has had its priority raised while occupying regions has its priority restored only when it has exited all of the regions it entered.

7.2.4 Deadlock

Deadlock, the situation in which tasks are permanently stopped from executing because of their interactions with other tasks, is a potential problem in any system that allows multiple threads of execution to compete for a common set of resources. The resources can be anything, such as files, memory segments, or whatever. The problem occurs when an exchange mechanism, such as a region or semaphore, is used to enforce mutually exclusive access to individual resources. In its simplest case, deadlock can occur if two tasks each need exclusive access to the same two resources at the same time. Call the tasks 1 and 2, and the resources A and B. Deadlock occurs if Task 1 acquires Resource A, is preempted by Task 2, which acquires Resource B. Now Task 2 cannot proceed because it cannot acquire Resource A, and Task 1 cannot proceed because it cannot acquire Resource B.

The use of any iRMX exchange object can result in deadlock, although deadlock can be broken manually for semaphores and mailboxes by deleting the tasks involved, or broken automatically if at least one of the tasks

uses a finite time limit for the call that tries to obtain a resource and the time limit expires. The problem is particularly pernicious when tasks use *rqreceivecontrol( )* to obtain control of regions that control access to resources because there is no way to break an ensuing deadlock. No time limit is associated with *rqreceivecontrol( )*, you cannot delete a task that owns a region, and no other task can call *rqsendcontrol( )* for any of the regions involved because that can only be performed by the tasks that obtained control of the regions themselves.

Deadlock need not be a concern because it is easily avoided. The procedure is to have all tasks that acquire mutually exclusive access to multiple resources do so in a fixed sequence when they obtain elements from the set of resources and to then use the reverse sequence when the tasks release the resources. For example, if a set of resources are protected by regions named with different letters of the alphabet, deadlock can be avoided if all tasks that access any subset of the resources always attempt to receive control from the regions in alphabetical order and release control of those regions in reverse alphabetical order. It does not matter how the letter names are assigned to the regions, as long as all tasks that access any of them use the same alphabetic assignment.

7.3 Buffer Pools

In the discussion of message mailboxes, it was noted that a considerable amount of overhead is associated with creating and deleting memory segments, especially for protected-mode versions of the operating system. The system calls to create and delete segments are simple enough to code, but the Free Space Manager (FSM) must be invoked to manage the calling job's memory pool, to borrow and return pieces of memory from ancestor jobs, and to manage the descriptor table slots for the segments. A common strategy for dealing with this overhead is for an application to create all the memory segments it needs when it first starts running. This strategy moves the overhead of creating segments into the initialization phase of the application and out of the event loop portion of the code, which must concern itself with real-time constraints.

A common problem with this strategy arises because memory segments are often used to pass information in a single direction, from a source task to a destination task. The destination task must be able to recycle segments when it finishes processing the information in them. If the destination task does not do anything with the segments it receives, the application soon runs out of memory. If the task deletes the segments, the source task must create more segments to replace them. A technique to do this recycling is to set up a mailbox that acts as a place for destination tasks to send tokens for the segments they are ready to recycle. Buffer pools provide another method to do the same thing, and offer the advantage of making it easy to work with segments of different sizes. Buffer pools, however, are

less efficient than segment recycling if all segments are the same size.[2] Although the name *buffer pool* implies that buffer pools were developed for the management of memory segments to be used as I/O buffers (they were), the mechanism is perfectly general and can be used effectively to manage any set of memory segments.

A task creates first a buffer pool object, and then a number of memory segments that it *releases* to the pool. Additional segments can be released to the pool at any time, but the usual practice is to give the pool most, if not all, of the segments it is to manage as the application initializes itself. One of the internal design features of the buffer pool type manager is visible to users of buffer pool objects: the segments released to a buffer pool are entered onto internal lists according to segment size, and no more than 16 different sizes of segments can be released to a single buffer pool object.

Data chaining is an option that can be enabled for a buffer pool when it is created. With this option, an application can request a buffer larger than any of the segments residing in the pool, and the buffer pool type manager satisfies the request by giving the task as many segments as it needs to satisfy the request. In this case, it will return a data structure called a *chain block* that contains a list of pointers and sizes of the segments returned. When this option is used, the application must acknowledge that a single buffer from the buffer pool does not necessarily occupy a single contiguous segment in memory.

The following is a description of the basic system calls for buffer-pool use. Additional calls for buffer pools exist but are not listed here. Those calls support the use of buffer pools for message passing in iRMX systems that run on Multibus II platforms. The Multibus II system bus provides a hardware mechanism with which multiple processors can send messages to each other very rapidly. That mechanism is fully supported by iRMX, and buffer pool management is closely integrated with that support, but our attention here is on the system calls for basic buffer-pool management.

```
extern TOKEN
rqcreatebufferpool (          WORD              maximumBuffers,
                              WORD              poolFlags,
                              WORD far *        exceptPtr);
```

`maximumBuffers` limits the number of memory segments that can be released to the buffer pool at one time. The upper limit for this parameter is 8,192, the maximum number of slots in a descriptor table.

`poolFlags` signifies whether or not to support data chaining for this buffer pool. A value of 0 means no, and a value of 2 means yes.

---

[2]The iRMK kernel, available with iRMX III and iRMX for Windows systems, provides another memory management system even more efficient than recycling segments.

```
extern void
rqreleasebuffer(        TOKEN           bufferPool
                        TOKEN           bufferSegment,
                        WORD            bufferFlags,
                        WORD far *      exceptPtr);
```

`bufferSegment` is added to the list of segments occupying `bufferPool` in this example. Problems that can occur include the buffer pool being full (the `maximumBuffers` specified in *rqcreatebufferpool( )* already reached), or the size of the segment that is released to the segment is not one of 16 different segment sizes already released to the pool.

`bufferFlags` contains two pieces of information: whether the `bufferSegment` being released is a single memory segment or a data chain and how to handle the buffer pool full condition for this system call. If the `bufferSegment` token is for a memory segment to be released, add 0 to the value of this parameter. If the `bufferSegment` token is for a chain block, add 1 to this parameter. If you want the segment being released to be deleted if the buffer pool is full when this call is made, add 0 to this parameter. If you want the segment to be retained in this situation and have the condition code set to 0x0004 (E_LIMIT), add 2 to the value of this parameter.

```
extern TOKEN
rqrequestbuffer(        TOKEN           bufferPool
                        DWORD           bufferSize,
                        WORD far *      exceptPtr);
```

`bufferSize` must have a value between 1 and 0xFFFFFFFE. If a single segment is available that can satisfy the request, its token is returned, and the condition code is set to E_OK. If data chaining is allowed for the buffer pool and the buffer pool manager is able to satisfy the request only by creating a data chain, the token returned is for a segment containing a chain block, and the condition code is set to 0x000D (E_DATACHAIN). If the buffer pool does not have a segment that can satisfy the request (and data chaining is disabled), or if the pool does not have the segments that could satisfy the request (even though data chaining is enabled), the call fails and the condition code is set to 0x0002 (E_MEM).

7.4 Job Management

One recurring theme in understanding the iRMX operating system is that there seems to be different types of jobs. We have already used terms like Nucleus Job, I/O Job, and HI Command Job. In reality, all jobs are equivalent in the sense that they are originally created by one of two Nucleus sys-

tem calls, *rqcreatejob( )* and *rqecreatejob( )*, that differ from each other only in the values that can be specified for the memory pool parameters.[3]

7.4.1 Creating a Nucleus Job

The following are function prototypes for creating a Nucleus job:

```
extern TOKEN
rqcreatejob (           WORD                        directorySize,
                        TOKEN                       paramObj,
                        NATIVE_WORD                 poolMin,
                        NATIVE_WORD                 poolMax,
                        WORD                        maxObjects,
                        WORD                        maxTasks,
                        BYTE                        maxPriority,
                        EXCEPTIONSTRUCT far *       exceptHandler,
                        WORD                        jobFlags,
                        BYTE                        taskPriority,
                        far *                       startAddress,
                        TOKEN                       dataSeg,
                        WORD far *                  stackPtr,
                        NATIVE_WORD                 stackSize,
                        WORD                        taskFlags,
                        WORD far *                  exceptPtr);

extern TOKEN
rqecreatejob (          WORD                        directorySize,
                        TOKEN                       paramObj,
                        DWORD                       poolMin,
                        DWORD                       poolMax,
                        WORD                        maxObjects,
                        WORD                        maxTasks,
                        BYTE                        maxPriority,
                        EXCEPTIONSTRUCT far *       exceptHandler,
                        WORD                        jobFlags,
                        BYTE                        taskPriority,
                        far *                       startAddress,
                        TOKEN                       dataSeg,
                        WORD far *                  stackPtr,
                        NATIVE_WORD                 stackSize,
                        WORD                        taskFlags,
                        WORD far *                  exceptPtr);
```

`directorySize` signifies the number of entries to reserve for the job's object directory. The value of this parameter determines in part how much memory will be used for the segment containing the job object itself. The object directory is implemented as a hash table within the job object.

`paramObject` is a token for any iRMX object to be passed to the job. The object is normally a memory segment that contains data specific to one particular job out of a set of jobs that are otherwise identical. Applica-

[3]All Nucleus system calls have names that start with *rq*, and those calls that take advantage of the extended features of the 80286 microprocessor or later start with *rqe*. Of course, the *rqe* calls are not available for iRMX I.

tions that do not use this feature set this parameter to the selector of a null pointer (i.e., 16 bits of zeros). Any task belonging to a job can retrieve a copy of this token by calling *rqgettasktokens( )*. I/O jobs and HI command jobs have a parameter object passed to them, which is the token for a segment returned to the parent job when the child job calls *rqexitiojob( )*[4]. This segment does not seem to contain any useful information when the job is created.

`poolMin` and `poolMax` were discussed in chapter 6. For *rqcreatejob( )*, these values cannot be greater than 1M, and for *rqecreatejob( )*, which is not available for iRMX I, the values cannot be greater than 16M for iRMX II or 4G for iRMX III. In all cases, the values specified are given in units of 16-byte paragraphs. Thus, if you want a maximum memory pool of 512,000 bytes for a job, specify a value of 32,000 for `poolMax`.[5]

`maxObjects` and `maxTasks` place a limit on the number of objects a job can own in general, and on the number of task objects in particular. Values of 0xFFFF indicate no limit. These limits cannot be set to zero because every job must own at least one object: its initial task. Values less than 0xFFFF are subtracted from the corresponding limits for the parent job immediately, even before the child job creates any new objects.

`maxPriority` is the numerically lowest scheduling priority (0 to 255) that any task belonging to the new job can take. If you specify a value of 0 for this parameter, the job's maximum priority is set equal to its parent's maximum priority. A child job cannot have a higher maximum priority than its parent's, unless it is explicitly changed using the *rqesetmaxpriority( )* system call after the job has been created.

`exceptHandler` is a pointer to the exception handler structure described in chapter 6. A null pointer gives the job the system default handler and mode.

`jobFlags` contains just one bit that might affect the job. If this parameter is set to 1, and neither the calling job nor any of its ancestor jobs has selected parameter checking, the system will not check the validity of parameter values passed to system calls by the tasks of the child job. A speed advantage can be obtained for time-critical applications by using this feature for perfectly debugged applications. However, iRMX for Windows, as well as most other configurations of iRMX, have this bit turned off for the root job, so the argument is almost always ignored.

The remaining parameters for these two calls are all concerned with the new job's initial task. Basically, they have the same values and interpreta-

---

[4]Phrases to the effect that "some job executes some code" is shorthand for saying "a task belonging to some job executes some code."

[5]52,000 is 0x7D000 and 32,000 is 0x7D00. The term *paragraph* is a holdover from real-mode addressing in which segments are always a multiple of 16 bytes in size because the 16-bit base address is shifted 4 bits left before adding the offset.

tions as the corresponding parameters to *rqcreatetask( )*, (section 7.2) but there are a few differences to consider.

`taskPriority` is the initial priority for the initial task. Its value is limited by the `maxPriority` parameter for the job being created, not the calling task's job.

`startAddress` is interpreted the same as *rqcreatetask( )*, but there is an important implication of the logic of this call: because this is a parameter being passed to the system subroutine that will create the memory pool for the new job, it is impossible for this pointer to point to an address within the new job's memory. That is, *the initial task for every iRMX job starts executing code that resides in some other job's memory* (or in system memory, which does not belong to any job).[6]

An example of how this process might work is an application that contains separate procedures to be executed by the initial tasks of the application's various child jobs. All the procedures would be loaded as part of the application, and then, rather than call the procedures as subroutines, child jobs are created with their initial tasks set to execute the various procedures. Another example is the Application Loader (AL), which creates jobs with initial tasks set to start executing a loader procedure that loads a program into the new job's memory from a disk file, and then branches to it.

The AL system calls are described in section 7.4.3.

If `stackPtr` is a null pointer, the Nucleus will use the `stackSize` parameter to create a segment for the task's stack, taking the memory for the stack from the new job's memory pool. If `stackPtr` is not null, the specified segment will be used instead (normally a segment belonging to the creating job, but not necessarily), and the child job's memory pool will not be reduced by the size of the stack.

7.4.2 Creating an I/O job

Any task that makes calls to the Extended I/O System (EIOS) must belong to an I/O job. The reason for this requirement is the way I/O processing is performed, especially the file-protection mechanism, discussed in chapter 8. Our focus at this point is to examine the *rqcreateiojob( )* and *rqecreateiojob( )* system calls provided by the EIOS to see how they relate to the corresponding Nucleus calls for creating jobs. The following are the function prototypes:

```
extern TOKEN
rqcreateiojob (          NATIVE_WORD                  poolMin,
                         NATIVE_WORD                  poolMax,
                         EXCEPTIONSTRUCT far *        exceptHandler,
                         WORD                         jobFlags,
```

[6]Unix aficionados might contrast this behavior to that system's *fork( )* call that copies or maps the parent process's code into the child process's memory.

```
                        BYTE                    taskPriority,
                        far *                   startAddress,
                        TOKEN                   dataSeg,
                        WORD far *              stackPtr,
                        NATIVE_WORD             stackSize,
                        WORD                    taskFlags,
                        TOKEN                   msgMbox,
                        WORD far *              exceptPtr);

extern TOKEN
rqecreateiojob (        DWORD                   poolMin,
                        DWORD                   poolMax,
                        EXCEPTIONSTRUCT far *   exceptHandler,
                        WORD                    jobFlags,
                        BYTE                    taskPriority,
                        far *                   startAddress,
                        TOKEN                   dataSeg,
                        WORD far *              stackPtr,
                        NATIVE_WORD             stackSize,
                        WORD                    taskFlags,
                        TOKEN                   msgMbox,
                        WORD far *              exceptPtr);
```

Most of the arguments to these two calls are exactly the same as the arguments to *rqcreatejob( )* and *rqecreatejob( )*, with the same interpretations for all values. The exception is the `taskFlags` parameter, which includes a bit to indicate if the initial task of the I/O job is to start executing immediately or not. If not, the task is suspended until some other task calls *rqstartiojob( )* with a token for the new job as its argument. This feature is used, for example, by the HI to allow it to alter the new job's object directory before letting it start executing, as described in section 7.4.4.

One additional parameter to these calls compared to the Nucleus calls is the `msgMbox` token (located in the second from the last position). This token is for a mailbox object to which a message will be sent when the new job terminates. (The use of iRMX mailbox objects was described earlier in this chapter.) The message sent to this mailbox is a segment that contains the following:

- The token for the terminating job (useful if a parent job wants to monitor the completion of several child jobs by using a single mailbox for all termination messages).
- A fault code equal to the first parameter of the terminating job's call to *rqexitiojob( )*.
- A message string (up to 89 bytes long) supplied as the second parameter of the terminating job's call to *rqexitiojob( )*.
- A termination code that states whether the terminating job called *rqexitiojob( )* itself or was deleted for some other reason, such as by an exception handler. This same segment is used internally by the EIOS as it sets up the object directory for the child job, and is used as the job's parameter object for this reason.

I/O jobs provide a case history in the way iRMX can be extended to provide type managers for new object types. The rules for making EIOS system calls from I/O jobs are well specified in the iRMX documentation, and explanations of how I/O jobs differ from Nucleus jobs are provided. There is also an explanation to deal with the catch-22 of I/O jobs: I/O jobs can only be created by other I/O jobs. (One or more I/O jobs are "automatically" created when an iRMX system is initialized, provided the configuration includes the EIOS layer.)

But what is an I/O job, really? How does it come into existence? Why is it necessary to have this second type of job? The answers cannot be complete at this point because all the necessary concepts have not been covered yet, but following are some of the answers.

I/O jobs are composite objects. That is, they consist of other iRMX objects. This new object type is defined by the EIOS job when it starts running at system initialization time. The Nucleus calls involved with creating a new object type and managing individual objects of that type are covered in chapter 10. The two points to know now are that composite objects are implemented as lists of tokens for other objects, and that there is the provision for a type manager to supply a deletion mailbox for objects of the composite type. Composite objects can be deleted by calling a routine supplied by the type manager for the object type, or automatically by the Nucleus as it deletes all objects belonging to a terminating job. The deletion mailbox provides a mechanism for the type manager to find out if one of the composite objects it is managing is being deleted without a call to the manager's deletion routine.

In the case of I/O jobs, the deletion routine provided by the type manager is the *rqexitiojob( )* system call. By using a deletion mailbox, the EIOS can also be informed if an I/O job is deleted by some other means, such as by an exception handler (any job can be deleted by any task that knows the token for the job), or by the user typing <^C> if the job is being run as an HI command job. When a job calls *rqexitiojob( )*, or when a token for an I/O job arrives at the type manager's deletion mailbox, the type manager formats the job's termination message and sends it to the mailbox that was specified in the `msgMbox` parameter when the job was created.

How does the type manager know what mailbox to send the message to? The token from the `msgMbox` parameter was stored as one of the items in the list of objects for the I/O job object type. How does the Nucleus know enough to send the token for an I/O job to the type manager's deletion mailbox if the job is deleted by a call to *rqdeletejob( )* (by an exception handler, for example) instead of by a call to *rqexitiojob( )*? How does the I/O job type manager know which I/O job is calling its *rqexitiojob( )* routine?

The answer to the first question is that the Nucleus keeps a list of all objects owned by each job so that it can delete the proper objects when a job is deleted. When a job being deleted owns a composite object, *rqdeletejob( )* automatically sends the token for the composite object to the proper dele-

tion mailbox if there is one (and there *is* one for I/O jobs). That is, an I/O job is a composite object owned by a Nucleus job. When a Nucleus job is deleted, the composite object is sent to the deletion mailbox being monitored by the I/O job type manager in the EIOS. There is nothing circular nor even tricky happening here. Just a little complex.

The secret is to reconstruct the logic of the *rqcreateiojob( )* system call. When an application task calls *rqcreateiojob( )*, it executes code in the EIOS that calls *rqcreatejob( )* with the same parameters as were passed to itself except for two: the `msgMbox` is not passed because there is no way to do so, and the `startAddress` parameter is changed to point to a procedure within the EIOS. When the initial task of the new job starts executing this procedure, it executes the code to create an I/O job composite object. The tokens it places in this composite object are the following:

1. A token for its own job (available in shared memory with the rest of the EIOS or by calling *rqgettasktokens( )*).
2. The token for the `msgMbox` that was passed to *rqcreateiojob( )* (available in memory shared with the rest of the EIOS).
3. A token for the memory segment to hold the exit message sent to `msgMbox` when the job exits. (The segment is created by the task that called *rqcreateiojob( )* so that it belongs to the parent job and will not be deleted when the child job exits.)

The procedure being executed by the initial task then performs a bit more housekeeping and jumps to the code pointed to by the `startAddress` parameter of the *rqcreateiojob( )* system call. If the `taskFlags` parameter for the job specifies suspending the initial task until *rqstartiojob( )* is called, the task suspends itself before jumping to `startAddress`.

That "bit more housekeeping" that the new job's initial task does before jumping to the `startAddress` provides the answer to the second question, which can be made a bit more general: How does the EIOS know what I/O job owns the task that makes any of the EIOS system calls that can be executed only by I/O jobs and not by Nucleus jobs? The secret lies in the use of an I/O job's object directory. Before jumping to `startAddress`, the initial task executes code to catalog four items into its own job's object directory using the following names:

`R?IOJOB`.[7] This is the object directory name for the token for the I/O job composite object. The EIOS can call *rqlookupobject( )*, described in chapter 6, for a token with this name to find out if the calling job is an I/O job or

[7]The choice of object directory entry names that start with `RQ` and `R?` was probably made to avoid conflicts with names used by application developers. They have nothing to do with the *rq* prefix used for all system call names (also chosen to avoid conflicts with application function names) and nothing to do with the `r?` prefix used to name hidden files in the file system.

not.[8] The EIOS' *rqexitiojob( )* routine, in particular, looks up this object and extracts the token for the segment to hold the termination message and the token for the mailbox to which the message is sent.

R?IOUSER. The token cataloged with this directory name is for another composite object type called an I/O user object. It consists of a memory segment containing a list of 16-bit user ID numbers (0x0000 for the Super user, 0xFFFF for the World user, and other values for individual users or groups defined in the User Definition File). A token for one of these objects is one of the parameters for the BIOS calls that perform user access checking, notably *rqaopen( )*, described in chapter 8. When an application task calls the corresponding EIOS function *rqsopen( )*, the EIOS uses this token as one of the parameters when it makes the call to *rqaopen( )*.

$. This name is a token for an I/O connection to the application task's current working directory. I/O connections are another composite object type. Like I/O user objects, they are managed by type manager code in the BIOS. The EIOS needs this to pass on to the BIOS when the application attaches to a file and does not supply a full pathname. Again, chapter 8 provides more details about how I/O system calls work.

RQGLOBAL. The EIOS maintains the concept of a global job for I/O jobs. One specific example of a global job is the one created when a user logs on to an iRMX system, which becomes the global job for all jobs created by that user. (Every HI command executed by the user is run as a child of this global job, and any I/O jobs created by those HI command jobs have the same global job.) The EIOS uses the global job's object directory as one of the places it searches for logical names during *rqslookupconnection( )*. (Once again, refer to chapter 8 for more information about logical names and their associated system calls.)

The last three of the preceding items are copied into the new I/O job's object directory from its parent job's object directory. Without worrying about why that should be done for now, consider how it could be done from a job management point of view. The parent job is the one that owns the task that called *rqcreateiojob( )*, so the calling task could execute the calls to *rqlookupobject( )* to get these tokens from its own job's object directory. The values of these tokens should be stored in memory locations accessible by the procedure executed by the initial task of the child job. The new job's task then catalogs the tokens using the same names in its own job's object

[8] You already know that "the calling job" means the job that owns the task that made the call. Now, add the shorthand of "the EIOS calls *rqlookupobject( )*" to mean that the task making an EIOS system call enters a procedure in the EIOS' part of system memory which contains the code to call *rqlookupobject( )*. From this fact follows the important point that the object directory searched for the R?IOJOB entry is the one for the job that owns the task that makes the call, not the object directory for the EIOS job itself.

directory and then jumps to `startAddress`. Alternatively, the new job's task could look up the tokens in its parent job's object directory and then catalog them.

The net result of this exercise is that you can see that an I/O job really is just a Nucleus job after all, but that it has had a particular set of objects cataloged in its object directory — objects that make it possible for the EIOS to obtain information it needs when the job makes EIOS system calls. "I/O job" is actually something else as well: it is the name of a composite object type whose type manager is provided by the EIOS, and the token for an instance of this object type is one of the items cataloged in the object directory of every I/O job.

7.4.3 Using the AL

Many iRMX applications run in dedicated systems with both the OS and the application jobs loaded into memory when the system is initialized. Chapter 3 introduced the *sysload* command that can be used to load such resident application jobs for iRMX for Windows systems. Traditional iRMX systems would use the Interactive Configuration Utility, described briefly in chapters 9 and 10 to incorporate applications into the OS image. It is also possible to load application code into memory as the system is running by using the AL layer of the operating system[9]. This section describes the use of the AL to load an I/O job into memory from a disk file. The next section shows how this call can be used by the HI to create some of its offspring jobs. Sample code illustrating this technique is given in Figures 7.4 and 7.5. The output from running the first program might look like this:

```
This is the initial task: B9E8.
  I belong to job B8B0.
  My priority is 008E.
  My maximum priority is 008D.
  Now I will create a new I/O job.

This is the initial task again.
  I created I/O job BA80.
  Its exit code was 1234.
  Now I will print its message and exit.

Job: BA80.
  Task: BB58.
  Priority: 008D.
  Max priority: 008D.
  Exit.
```

[9]The *sysload* command uses the AL to load programs into memory.

The system call described here (*rqsloadiojob()*) is one of several calls provided by the AL for loading code into memory. For example, *rqaload()* loads a program into memory but does not create a new job for it. The segments for the program's code, data, and stack segments are taken from the calling job's memory pool, and no task is created to execute the code. The call returns selectors for all the segments, and a far pointer to the execution start address for the program. Another call, *rqsoverlay()*, is used to load different parts of a program into memory dynamically to reduce overall memory requirements for the program at the expense of a run-time delay as the overlay is read in from the disk. Since these two calls do not create an I/O job, they can be used in configurations of iRMX that do not include the EIOS layer of the operating system.

There is also an asynchronous version of *rqsloadiojob()* called *rqaloadiojob()* that allows the calling program to continue executing while the new job is being loaded from disk. The calling program must then check that the loading operation was successful using a technique analogous to that used for asynchronous I/O processing with the BIOS.

The following is the function prototype for *rqesloadiojob()*.

```
extern TOKEN
rqesloadiojob (          STRING far *                pathPtr,
                         DWORD                       poolMin,
                         DWORD                       poolMax,
                         EXCEPTIONSTRUCT far *       exceptHandler,
                         WORD                        jobFlags,
                         BYTE                        taskPriority,
                         WORD                        taskFlags,
                         TOKEN                       msgMbox,
                         WORD far *                  exceptPtr);
```

The first parameter is a pointer to an iRMX string (an array of bytes containing the length of the string in the first byte) that gives the iRMX pathname for the program to be loaded. Use the same rules for the pathname as when typing pathnames on a command line: if the pathname starts with /, :, or ^, it is a full pathname; otherwise, the first element must be the name of a file or directory in the current working directory.

The `poolMin` and `poolMax` parameters are normally coded as 0 for these calls, which means that the AL determines the appropriate values for these parameters before calling *rqcreateiojob()*. Recall from chapter 3 that an STL file begins with a header portion that tells the program's minimum and maximum memory pool requirements as specified on the `bndX86` command line, as well as the types and sizes of all the segments that make up the program. The AL reads in this header part of the file before it creates the I/O job and uses the information it finds there about the program's memory pool requirements to set those values for its call to *rqcreateiojob()*, unless the call to *rqesloadiojob()* provides nonzero values for these parameters, which would be used instead.

To solve the problem of loading a program into a job that has not yet been created, the AL has the new I/O job load itself into memory. You might be able to figure out how this is done by now. The AL sets the `startAddress` parameter of its own call to *rqcreateiojob()* to point to a procedure that it supplies for loading the program. That procedure ends with a call to create a new task that starts execution at the first instruction in the loaded program. The task that loaded the program (the real initial task of the child job) is used for loading overlays for the child job if it needs them, or deletes itself if the program does not contain overlays. As far as the application is concerned, the second task created for the job is its initial task. Since all tasks belonging to a job are equivalent siblings, there is no significance whether the real initial task or some other task executes the job's code.

The procedure that does the actual loading needs a certain minimum amount of memory for I/O buffers and its own housekeeping operations. If necessary, the `minPool` parameter for the call to *rqcreateiojob()* is adjusted to override the minimum memory pool specified in the loaded file's header to take this memory requirement into account. The segments created by this procedure are deleted when the loading operation finishes if the program does not contain overlays.

7.4.4 HI offspring jobs

The HI job is the first-level job created for the HI layer of the operating system when it initializes. The HI job creates a task for each terminal defined in `:config:terminals`, and each of these tasks displays a login prompt on its terminal's screen.[10] When a user logs in, this task reads information about the user from the `:config:udf` (User Definition File) and from a file that has the user's login name in the `:config:users` directory, and then creates a job that provides the environment for that user's work on the system. This job is referred to either as a *CLI job*, because it includes the task that executes the command line interpreter for the user, or as a *terminal job*, because it is associated with a particular login terminal. As mentioned earlier, this job is an I/O job that acts as the global job for all other I/O jobs spawned from the user's login session. The HI creates this job using *rqcreateiojob()* with a value for the `taskFlags` parameter that causes the initial task to wait before executing.[11] The HI then catalogs into the job's object directory tokens for those objects that it will need to perform its operations for the user. Some of these tokens replace

[10]The `:config:terminals` file might specify a static logon user for the terminal, in which case the task automatically performs the login process for the user on that terminal.

[11]Because the HI job is not an I/O job (there is no `R?IOJOB` entry in its object directory), there seems to be a bit of forgery going on in the operating system to circumvent the rule that only I/O jobs can create I/O jobs.

objects already cataloged in the job's object directory by the EIOS during the call to *rqcreateiojob()*. The following is a list of the objects cataloged in a CLI job's object directory by the HI.

R?IOJOB. This is the normal I/O job object cataloged by the EIOS and unchanged by the Human Interface.

R?IOUSER. Each user logged onto an iRMX system is checked against the entries in the `:config:udf` file, where group and individual user ID numbers are stored. These ID numbers are used to build a new user object for the individual who logs in. This user object replaces the one copied into the job's object directory from its parent, the HI job.

RQGLOBAL. This job is a global job itself, so the token cataloged under this name is changed from the copy inherited from the HI job into a token for itself.

HOME. The user's pathname to the home directory of the file system is found in `:config:users/<username>` when the user logs in, and a token for an I/O connection to this directory is cataloged using this name.

$. The I/O connection token inherited from the HI job for the current working directory in the file system is changed to match the token cataloged with the name HOME when the user logs in. No command changes the token cataloged with the name HOME, but $ changes when the user runs the *attachfile* command.

PROG. This directory entry is cataloged with the token for an I/O connection to the file system directory that has the pathname `:home:prog`. The CLI gets the `r?logon` and `r?logoff` files from this directory and submits them when the user logs on and off, respectively.

TERM, CI, CO. The same token is cataloged with these three different names. The token is for an I/O connection to the user's console input and output devices. The same connection is used for all three because the keyboard and CRT are parts of the same device, but the potential exists for separating CO and CI from TERM through command-line redirection using the > and < characters.

As seen in chapter 8, HOME, $, PROG, TERM, CI, and CO are all logical names for I/O connections. To steal the thunder from that chapter a bit, all logical names are implemented by cataloging a token for an I/O connection

object in the root, global, or local job's object directory. (A local job is simply whatever job owns the task that references the logical name.)

The CLI procedure adds more entries to the CLI job's object directory beyond those placed there by the HI. These objects are used internally by the CLI, and include the following.

R?CRT. A token for a segment that contains an iRMX string with the name for the type of terminal used for logging into the system. Applications that need to do full-screen operations can find this name in the `:config:termcap` file, along with the codes that a particular type of terminal recognizes for controlling the screen, and the keyboard codes the terminal generates for special keys such as the cursor arrow keys. The CLI, SoftScope, and *Aedit* use this information to adapt to different types of terminals.

R?ALIAS. A token for a segment that contains all the command aliases the user has defined.

R?BACKPOOL. A token for a segment that contains the default values for the minimum and maximum memory pool parameters that the CLI will use when creating a child job to run a *background* command.

R?ERROR. A token for a segment that contains the termination code for the most recent HI command run by the user. At least some versions of the CLI do not update the contents of this segment.

R?CURR$APP. A token for the HI command job that the user is running at the time. This token is not cataloged except when a command is actually running.

To create the actual jobs used for the commands, a user types at a terminal (HI command jobs), the CLI job reads the command line from the keyboard, and then sends the string typed by the user to the HI by calling *rqcsendcommand( )*. The HI parses the command line, and searches a particular set of directories in the file system for a file name that matches the beginning of the command line. It then passes the full pathname of this file as the first argument of a call to *rqsloadiojob( )*, with the job's `taskFlags` parameter set to suspend execution. The HI then updates the new job's object directory with the entries it will need if the job makes any system calls to the HI layer (system calls with names that begin with *rqc*, such as *rqcsendcoresponse( )* and *rqcsendcommand( )*). It then calls *rqstartiojob( )* and waits at the termination mailbox for the command job to exit. It then sets the condition code for the call to *rqcsendcommand( )* to indicate the exit status of the job. Of course, all the code executed in the HI by *rqcsend-*

command( ) is actually executed by a task in the CLI job, so the new HI command job is created as a child of the CLI job, and it is the CLI job task that actually waits at the termination mailbox for the command job to complete.

The CLI is not the only command line interpreter in iRMX systems. The HI commands *super*, *submit*, and *esubmit* all act as command line interpreters. A simple command line interpreter is relatively easy to construct, and one is included in appendix C.

Chapter

8

I/O Management

8.1 Overview

The Basic I/O System (BIOS), Extended I/O System (EIOS), Human Interface (HI), and User Development Interface (UDI) layers of an iRMX system provide system calls that application programs can use to perform input and output with peripheral devices connected to a computer system.

The terms *input* and *output* can have different meanings in different contexts. This chapter discusses I/O between the memory of a processor running iRMX and either the peripherals attached to that computer directly or the peripherals attached to a remote computer system running the networking software OpenNet.

Access to remote peripherals is possible if the local iRMX system is configured to include a job called *iRMX-Net*. iRMX-Net uses ISO-standard communication protocols provided by a software module called iNA-960 to communicate with complementary software running on remote computer systems. The messages that iRMX-Net exchanges with remote systems are in Microsoft's Server Message Block (SMB) format.

iRMX applications can also access any DOS device through the EDOS file driver described in this chapter. If the DOS side of an iRMX for Windows system has mapped remote devices to DOS drives using a Novell network, for example, those mapped devices are available to iRMX applications through EDOS. What unifies all these ways of accessing I/O devices is that the same iRMX system calls, the ones described in this chapter, are used in all cases.

Other types of information transfers are sometimes placed under the headings of input and output operations. For example, iRMX supports passing data from a task running on one computer to a task running on another. This form of I/O is called either *message passing* or *interprocess communication* (IPC). The latter is a term borrowed from Unix, where threads of execution are called processes. iRMX supports message passing between

computers running iRMX as well as other operating systems. The two techniques available for this are Nucleus Communications Service for communications between tasks (processes) running on two computers connected by a Multibus II system bus, and the ISO Transport Layer services provided by iNA-960. IPC mechanisms based on iNA-960 are covered in chapter 11.

This chapter is divided into three parts. The first part presents the system calls that the BIOS and EIOS provide for performing data operations, which involves developing a model for talking about I/O operations and an introduction to how files are maintained on an iRMX file system. The second part covers some of the other services provided by the BIOS, such as special device-dependent functions, user authentication, and time-of-day management. The third part of the chapter describes the disk structure of a native-mode iRMX file system and the utility program called *diskverify* that can examine and modify that structure.

8.2 Data Operations

One byword of the iRMX I/O system is *device independence*. No matter what type of actual device a program works with, whether a disk drive, printer, terminal, or robot, the program uses the same two system calls for data transfers, *rqaread( )* and *rqawrite( )*, which are provided by the BIOS layer. Other system calls for data transfers are provided by the EIOS, HI, and UDI layers, but all those calls ultimately interface to these two BIOS calls that actually do the work.

To accomplish this degree of device independence, the BIOS must be able to use mechanisms to transform a device-independent system call like *rqaread( )* into the very specific and device-dependent actions that must be invoked to perform a particular data transfer. Whether an application programmer is fully aware of these mechanisms or not, applications must perform a number of steps to prepare the BIOS for the device-independent calls to *rqaread( )* or *rqawrite( )*. Some of these calls are device dependent, but most are device independent. This section looks at those steps and explains how they relate to the structure of the BIOS.

8.2.1 An I/O model

One problem with learning any new system is mastering the terminology used to describe it. The iRMX I/O system uses consistent terminology for various concepts relating to data-transfer operations. The terms are based on a model situation in which a task does data transfers to and from named files located on a disk drive. The same terms are then expanded to encompass how to access generalized peripheral devices in a device-independent way. The model situation is described first.

Consider the structure of an actual iRMX file found on a disk device formatted to hold an iRMX file system. iRMX can work with disks that have been formatted with other file systems, notably the MS-DOS file system, but the iRMX file system provides a somewhat more general model than DOS. Details about the internal structure of an iRMX disk volume are presented in the last section of this chapter, while the DOS file system is described in a number of different sources, such as the *Disk Explorer* manual provided with the Norton Utilities software package for DOS. For now, some general characteristics of how files are stored and accessed are discussed.

A disk file is always organized as an unstructured sequence of bytes. The operating system does not add any control characters like record marks or end-of-file marks to the contents of a file. Instead, it maintains a separate data structure for each file that tells what disk blocks the file occupies, the total size of the file, and housekeeping information (discussed later). An iRMX disk block and a DOS file cluster are conceptually similar constructs. They refer to the smallest amount of space on a disk that can be allocated to a file, and they cannot be shared by more than one file. The size of disk blocks is fixed for a particular disk volume and is always an integral multiple of the sector size for the volume. The size requirement is because of the hardware restriction that one sector is the smallest amount of data that can be transferred to or from the disk at a time. A disk volume means one hard disk drive, one partition on a disk drive, or one diskette.

The housekeeping information that the I/O system maintains for disk files allows the disk files to occupy noncontiguous blocks of the disk. This fragmentation leads to efficiently using the space on a disk volume, but can result in performance problems because the disk heads can be forced to move to widely different locations on the disk to access different parts of a file. If you are familiar with the DOS file system, you probably already know that this same problem exists there too, and that DOS utility programs can reorganize a disk to make files contiguous. The problem is particularly serious for real-time systems which need to work with deterministic response times to meet their deadlines. The iRMX technique for dealing with the problem is given in the description of *rqacreatefile( )*, section 8.2.6.

An iRMX application must create an open connection to a file before it can read or write to it. Since many files can exist on a single volume, it follows that the system supports multiple simultaneous connections to files on a single disk device. In fact, the system allows multiple simultaneous connections even to a single file on the disk, provided that the different applications that have open connections to the file agree about how they will share the file with one another.

Sharing refers to the possible combinations of exclusive or shared access to a file for reading and/or writing. When an application opens its connection to a file, it indicates the operations it intends to perform (read and/or

write) and tells which operations it is willing to allow other applications to perform simultaneously. The I/O system checks the logical consistency between this application's request and all other open connections to the same file. An application closes its connection to the file when it no longer needs to access it, and the I/O system then updates its record of sharing constraints for the file.

Creating a connection to a file is a two-step process. First, the application must either create an I/O connection to the device that holds the file or reference an existing I/O connection to the device. It then creates its connection to the file based on the connection to the device. Only one connection can exist to a device at one time, so that connection is normally made readily available for sharing among the various applications that might want to use it as the basis for creating connections to files on the device. Creating an I/O connection to a device initializes a software module called a *device driver* that acts as the interface between the OS and the hardware device controller used to operate the device. (Device drivers are the topic of chapter 9.) Creating a connection to a device also associates another software module, a *file driver*, with the connection and all file connections based on it.

Following is a list of the steps that must be taken to perform data transfers on an iRMX system, along with the names of the system calls that might be used at each step. The calls that begin *rqa* are provided by the BIOS layer, the others are provided by the EIOS layer. Only steps 2 through 5 are normally performed from with an application program.

1. Connect to the device. (This step can also be done by the HI command, *attachdevice*.)
   rqaphysicalattachdevice( )
   rqlogicalattachdevice( )
2. Connect to the file. (This step can also be done by the HI command, *attachfile*.)
   rqaattachfile( )
   rqacreatefile( )
   rqacreatedirectory( )
   rqsattachfile( )
   rqscreatefile( )
   rqscreatedirectory( )
3. Open the file.
   rqaopen( )
   rqsopen( )
4. Read and/or write.
   rqaread( )
   rqawrite( )
   rqaseek( )

rqatruncate( )
rqsreadmove( )
rqswritemove( )
rqsseek( )
rqstruncatefile( )

5. Close the file.
   rqaclose( )
   rqsclose( )
6. Disconnect from the file. (This step can also be done by the HI command, *detachfile*.)
   rqadeleteconnection( )
   rqsdeleteconnection( )
7. Disconnect from the device. (This step can also be done by the HI command, *detachdevice*.)
   rqaphysicaldetachdevice( )
   rqlogicaldetachdevice( )
   rqhybriddetachdevice( )

The iRMX I/O system uses this same model and its terminology for all data transfer operations, not just disk file I/O. Even if an application is writing to a printer, it first obtains a connection to the device and then uses that connection as the basis for creating a connection to a file on the device, even though printers do not actually have files on (or in) them, according to the traditional use of the word *file*.

The device independence that arises from using the connect-to-a-file model for all I/O operations produces two "Big Wins," one for application developers and one for system programmers.

Application programs are easily written to be device independent. Because the same system call is used to write to a printer, a terminal screen, or a file, the actual device involved can be changed without changing the program. This, of course is the idea of command-line redirection using the '>' and '<' symbols, but redirection is used in less obvious situations as well. For example, a program that reads from the console input device normally receives its input from a keyboard, but the source is automatically changed to come from a disk file when the command is run from a submit file.

System programmers benefit from device independence. This independence is achieved by partitioning the I/O system into modules with well-defined interfaces between them. System developers can thus extend the functionality of the I/O system without interfering with existing applications and with minimal changes to existing portions of the I/O system itself. For example, support for network operations and the DOS file system have been added to iRMX without adding any new system calls or chang-

ing the logic of most existing I/O system calls. About the only change, aside from the added functionality, was the addition of new condition-code values returned by existing system calls. This modular organization with well-defined interfaces between modules also makes the addition of user-written device drivers to the system a relatively straightforward operation, as seen in chapter 9.

8.2.2 Sample I/O programs

Figures 8.1 and 8.2 are equivalent PLM and C programs that illustrate the steps a program takes to perform I/O transfers on an iRMX system. These programs are not typical device-independent applications. They are coded to illustrate all the steps in the model, rather than just usual program steps. Device-independent programs would not call *rqlogicalattachdevice( )* themselves. Rather, a user would make the connection to a particular device outside the program, such as by issuing the HI *attachdevice* command before running the program.

The programs read from the console input device device—independently (input can be redirected by using the '<' character on the command line), but write to a file on a particular device, called `b_dos`. The programs will run on an iRMX for Windows system running on an AT platform with a DOS-formatted diskette in drive `B:`, but if you do run them, be sure there is not a file named `typeon.txt` that you care about on the diskette you have in drive `B:`. The program will overwrite it.

The programs begin by creating a connection to the `b_dos` device by calling *rqlogicalattachdevice( )*. They then obtain connections to two files. For the output, a true named file on the diskette is used, but for the input, the connection is to the console keyboard device. The *rqscreatefile( )* system call is used to connect to the output file and create it if it does not yet exist. The *rqsattachfile( )* call is used to connect to the console input device as a file, using the logical name `:ci:` to identify the connection to the device that serves as the basis for the connection to the file.[1] This *rqsattachfile( )* system call can also be used to connect to disk files that already exist.

Once the connections to files have been created, they are opened, and the reading/writing loop proceeds until the user types a null line at the keyboard (By pressing <^Z> at the beginning of a line) or until the end of file is reached in the case where the input has been redirected to come from a file. The program calls for the use of an I/O system buffer for the output connection (the third parameter of the call to *rqsopen( )*), which means that output accumulates in the buffer until it fills, at which point it is written to the disk, and a new buffer is begun. If the buffer is partially full after the

[1]Actually, `:ci:` is already a connection to a file at this point. It was originally a connection to a device, and the I/O system builds on this device information to create a connection to a file.

Figure 8.1 Sample PLM program illustrating I/O using EIOS system calls; the program reads from the console keyboard and writes to a disk file named `typeon.txt` on the `B:` disk of an AT computer.

```
/***>  typeon.plm  <************************************************
 *
 *  Sample PLM program to illustrate EIOS I/O
 *  The program reads lines from the keyboard and writes them to a
 *  file named typeon.txt on the B: drive of a PC.
 *
 ************************************************************************/
$title ('Sample program to illustrate EIOS I/O')
typeon: DO;
$include (typeon.ext)

DECLARE
        EDOS          LITERALLY       '6',
        READALL       LITERALLY       '1',
        WRITENONE     LITERALLY       '5',

        (console, file)               TOKEN,
        (bytesRead, bytesWritten)     WORD_32,
        buffer (80)                   BYTE,
        Status                        WORD_16;

/*  Execution Starts Here
    =====================
*/
/*  Establish a connection to the b_dos device with the logical
    name :B:.  This is equivalent to the HI command,
        ATTACHDEVICE B_DOS AS B EDOS
    ------------------------------------------------------------
*/
    CALL rqlogicalattachdevice (@(1,'b'), @(5,'b_dos'), EDOS, @Status);

/*  Create connections to two files, the console input device and
    a file on the disk, and open them appropriately.
    ------------------------------------------------------------
*/
    console = rqsattachfile (@(4,':ci:'), @Status);
    file = rqscreatefile (@(13,':b:typeon.txt'), @Status);
    CALL rqsopen (console, READALL, 0, @Status);
    CALL rqsopen (file, WRITENONE, 1, @Status);

/*  Read from console, write to file -- until done
    ----------------------------------------------
*/
    bytesRead = rqsreadmove (console, @buffer, size(buffer), @Status);
    DO WHILE bytesRead <> 0;
      bytesWritten = rqswritemove (file, @buffer, bytesRead, @Status);
      bytesRead = rqsreadmove (console, @buffer, size(buffer), @Status);
    END;

/*  Close file, detach device, and exit
    -----------------------------------
*/
    CALL rqsclose (file, @Status);
    CALL rqlogicaldetachdevice (@(1,'b'), @Status);
    CALL rqexitiojob (0, NIL, @Status);

END typeon;
```

Figure 8.2 C program equivalent to Fig. 8.1.

```
/***>  typeon.c  <**************************************************
 *
 *  Sample C program to illustrate EIOS I/O
 *  The program reads lines from the keyboard and writes them to a
 *  file named typeon.txt on the B: drive of a PC.
 *
 ************************************************************************/
#include <rmxc.h>
#include <string.h>

#define ALWAYS      3
#define EDOS        6
#define READALL     1
#define WRITENONE   5

main (int argc, char *argv[]) {

EXCEPTIONSTRUCT ehStruct;
TOKEN           console, file;
DWORD           bytesRead, bytesWritten;
BYTE            buffer [80];
STRING          b[] = "b", b_dos[] = "b_dos", ci[] = ":ci:",
                pathName[] = ":b:typeon.txt";
WORD            Status;

/*  Convert C strings to iRMX strings
    --------------------------------
*/
  udistr (b, b);
  udistr (b_dos, b_dos);
  udistr (ci, ci);
  udistr (pathName, pathName);

/*  Let exception handler take care of errors
    -----------------------------------------
*/
  rqgetexceptionhandler (&ehStruct, &Status);
  ehStruct.exceptionmode = ALWAYS;
  rqsetexceptionhandler (&ehStruct, &Status);

/*  Establish a connection to the b_dos device with the logical
    name :B:.  This is equivalent to the HI command,
        ATTACHDEVICE B_DOS AS B EDOS
    ------------------------------------------------------------
*/
    rqlogicalattachdevice (b, b_dos, EDOS, &Status);

/*  Create connections to two files, the console input device and
    a file on the disk, and open them appropriately.
    ------------------------------------------------------------
*/
    console = rqsattachfile (ci, &Status);
    file = rqscreatefile (pathName, &Status);
    rqsopen (console, READALL, 0, &Status);
    rqsopen (file, WRITENONE, 1, &Status);

/*  Read from console, write to file -- until done.
    -----------------------------------------------
```

Figure 8.2 (*Continued*)

```
*/
    bytesRead = rqsreadmove (console, buffer, sizeof (buffer), &Status);
    while (bytesRead != 0) {
      bytesWritten = rqswritemove (file, buffer, bytesRead, &Status);
      bytesRead = rqsreadmove (console, buffer, sizeof (buffer),
&Status);
      }

/*  Close file, detach device, and exit.
    ------------------------------------
*/
    rqsclose (file, &Status);
    rqlogicaldetachdevice (b, &Status);
    rqexitiojob (0, NULL, &Status);
  }
```

last write operation to the file, the partially full buffer is written when the connection is closed. The file is closed automatically when the application exits, but the file must be closed explicitly for this program because the program deletes the connection to the device containing the file before exiting by calling *rqlogicaldetachdevice( )*. That system call does not flush the EIOS's buffers for file connections based on the device connection before deleting the device connection. The issue is not important for the input side because that connection is opened with no buffering and the connection to the device is not deleted by the program.[2]

8.2.3 Synchronous and asynchronous I/O operations

Most system calls for most operating systems are synchronous. A task makes a system call, and does not return to the calling program until all operations associated with the system call are complete. That is, the resumption of the calling program is automatically synchronized with the completion of the system call. For an I/O system call, this means that the calling task could be delayed for relatively long periods of time. Waiting several dozen milliseconds for a disk transfer to occur is a very long wait for an application that measures the time it takes to process a real-time event in microseconds. (Imagine reading from a terminal's keyboard when the operator decides it's time for a coffee break!)

If a task returns from a system call before the logic of the call completes, the system call is said to be *asynchronous*. Asynchronous system calls allow a task to be more productive; the task can perform other computations while another task executes the logic of the system call concurrently. A

[2]If there were input buffering, the application's call to *rqsreadmove( )* would not complete until the user typed 1,024 characters. This is not an issue for the sample programs, but it could wreak havoc with the "user-friendliness" of an interactive program!

mechanism must exist then by which a task can determine when the concurrent part of the call has actually completed. That is, there must be a way for the calling task and the task executing the system call to resynchronize.

A major distinction between the BIOS and EIOS layers of iRMX is that the BIOS layer supports asynchronous operation for most of the system calls it provides, but the EIOS supports only synchronous operations. The resynchronization mechanism is this: when the calling task makes an asynchronous system call, it supplies a token for a mailbox as one of the parameters. (This parameter is referred to as `responseMbx` in the system calls that follow.) The BIOS does some initial processing of the system call to ensure that the parameters for the system call make sense, and returns to the calling task. If the condition code returned for this synchronous part of the system call is 0 (`E_OK`), it means that a task in the BIOS has been dispatched to process the concurrent part of the call. When that task completes, it sends the token for a memory segment to `responseMbx`, having placed information in that segment to indicate whether the asynchronous part of the system call completed successfully or not. When the calling task is ready to resynchronize, it can use either of two system calls. The task can call *rqreceivemessage()* to obtain the token for the segment sent to the mailbox, in which case the task must examine the contents of the memory segment to determine the result of the call. Alternatively, the task can call a BIOS function, *rqwaitio()*, which receives the segment at the mailbox, examines the contents of the mailbox to determine how the asynchronous part of the call fared, and sets its own condition code to indicate the result. For both *rqreceivemessage()* and *rqwaitio()*, the normal rules for iRMX mailboxes apply. If a token is at the mailbox when the call is made, the call completes immediately; if a token is not at the mailbox, the calling task is put to sleep until either a token arrives or until a time limit, specified as the value of a parameter to the system call, expires.

Figure 8.3 is a PLM program that performs the same function as the programs in Figures 8.1 and 8.2, but uses BIOS calls instead of EIOS calls to demonstrate asynchronous coding. If nothing else, this program should make it clear to you how much easier it is to use EIOS calls! The C program to do the same thing has generously been left as an exercise for you to do.

The EIOS implements a second form of synchronization important to recognize. All I/O operations performed through the EIOS using an I/O connection are serialized. That is, once any task issues a call to *rqsreadmove()*, for example, using a particular I/O connection, the EIOS will not release to the BIOS any I/O requests made by other tasks that use the same connection object. If different tasks have different connections to the same file, the EIOS will allow them to access the file concurrently (provided the connections are opened for sharing), but tasks cannot perform concurrent I/O operations using a single connection. The BIOS does not implement any such serialization of I/O requests.

Figure 8.3 PLM program equivalent to Fig. 8.1, using BIOS (asynchronous) system calls.

```
/***>  atypeon.plm  <************************************************
 *
 *  PLM program to illustrate asynchronous (BIOS) I/O
 *  The program reads from the keyboard and writes to a disk file
 *  called typeon.txt on the B: drive of a PC.
 *
 ***********************************************************************/
$title ('Sample program to illustrate BIOS I/O')
atypeon: DO;
$include (atypeon.ext)

DECLARE
        EDOS        LITERALLY       '6',
        SEGMENT     LITERALLY       '6',
        READ        LITERALLY       '1',
        WRITE       LITERALLY       '2',
        SHARENONE   LITERALLY       '0',
        SHAREALL    LITERALLY       '3',
        HARD        LITERALLY       '0FFh',

        (console, file, inMbx, iorsTkn,
        outMbx, b_dos)              TOKEN,
        (bytesRead, bytesWritten)   DWORD,
        (ibuffer, obuffer)(80)      BYTE,
        (ioStatus, Status)          WORD_16,

        iors BASED iorsTkn STRUCTURE (
            status          WORD_16,
            unit$status     WORD_16,
$IF r_32
            actual          WORD_32,
$ELSE
            actual          WORD_16,
            actual$fill     WORD_16,
$ENDIF
            device          WORD_16,
            unit            BYTE,
            function        BYTE,
            sub$function    WORD_16,
            device$location WORD_32,
            buffer$p        POINTER,
$IF r_32
            count           WORD_32,
$ELSE
            count           WORD_16,
            count$fill      WORD_16,
$ENDIF
            auxilary$p      POINTER,
            link$for        POINTER,
            link$back       POINTER,
            resp$mbox       TOKEN,
            done            BYTE,
            iors$fill       BYTE,
            cancel$id       TOKEN,
            conn$t          TOKEN);
```

Figure 8.3 (*Continued*)

```
/*  Execution Starts Here
    ====================

    Set up response mailboxes for asynchronous operations
    Strategy will be to initiate operation on one connection before
    waiting for previous operation on other connection to complete
    ---------------------------------------------------------------
*/
    inMbx = rqcreatemailbox (0, @Status);
    outMbx = rqcreatemailbox (0, @Status);

/*  Establish a connection to the b_dos device with no logical name
    ---------------------------------------------------------------
*/
    CALL rqaphysicalattachdevice (@(5,'b_dos'), EDOS, outMbx, @Status);

/*  Create connection to the console input device as a file
    ------------------------------------------------------
*/
    CALL rqaattachfile (selectorof(NIL),
      rqslookupconnection (@(4,':CI:'), @Status),
      NIL, inMbx, @Status);

/*  Wait for output attachdevice to complete; create output file
    ------------------------------------------------------------
*/
    iorsTkn = rqreceivemessage (outMbx, 0FFFFh, NIL, @Status);
    IF rqgettype (iorsTkn, @Status) = SEGMENT THEN
        CALL rqexitiojob (iors.Status, @(20,'Attach Device Failed'),
@Status);
    b_dos = iorsTkn;
    CALL rqacreatefile (selector$of(NIL), b_dos, @(10,'typeon.txt'),
      1111b, 0, 0, 0, outMbx, @Status);

/*  Wait for input attachfile to complete; open input for reading,
    share with all
    ------------------------------------------------------------
*/
    iorsTkn = rqreceivemessage (inMbx, 0FFFFh, NIL, @Status);
    IF rqgettype (iorsTkn, @Status) = SEGMENT THEN
        CALL rqexitiojob (iors.Status, @(18,'Attach File Failed'),
@Status);
    console = iorsTkn;
    CALL rqaopen (console, READ, SHAREALL, inMbx, @Status);

/*  Wait for output createfile to complete; open output for writing,
    share with all
    ------------------------------------------------------------
*/
    iorsTkn = rqreceivemessage (outMbx, 0FFFFh, NIL, @Status);
    IF rqgettype (iorsTkn, @Status) = SEGMENT THEN

      CALL rqexitiojob (iors.Status, @(18, 'Create File Failed'),
@Status);
    file = iorsTkn;
    CALL rqaopen (file, WRITE, SHARENONE, outMbx, @Status);
```

Figure 8.3 (*Continued*)

```
/*  Set up for main loop: Be sure both connections opened all right,
    and initiate first read from console
    ------------------------------------------------------------------
*/
    iorsTkn = rqreceivemessage (inMbx, 0FFFFh, NIL, @Status);
    IF iors.Status <> 0 THEN
        CALL rqexitiojob (iors.Status, @(17,'Open Input Failed'),
@Status);
    CALL rqaread (console, @ibuffer, size(ibuffer), inMbx, @Status);
    iorsTkn = rqreceivemessage (outMbx, 0FFFFh, NIL, @Status);
    if iors.Status <> 0 THEN
      CALL rqexitiojob (iors.Status, @(20, 'Open Output Failed'),
@Status);

/*  Wait for first read to complete and initiate first write
    ------------------------------------------------------
*/
    bytesRead = rqwaitio (console, inMbx, 0FFFFh, @Status);
    if Status <> 0 THEN
      CALL rqexitiojob (Status, @(11, 'Read Failed'), @Status);
    CALL movb(@ibuffer, @obuffer, bytesRead);
    CALL rqawrite (file, @obuffer, bytesRead, outMbx, @Status);
    CALL rqaread (console, @ibuffer, size(ibuffer), inMbx, @Status);

/*  Read from console, write to file -- until done
    ----------------------------------------------
*/
    DO WHILE bytesRead <> 0;
      bytesRead = rqwaitio (console, inMbx, 0FFFFh, @Status);
      if Status <> 0 THEN
        CALL rqexitiojob (Status, @(11, 'Read Failed'), @Status);
      bytesWritten = rqwaitio (file, outMbx, 0FFFFh, @Status);
      if Status <> 0 THEN
        CALL rqexitiojob (Status, @(12, 'Write Failed'), @Status);
      CALL movb(@ibuffer, @obuffer, bytesRead);
      CALL rqawrite (file, @obuffer, bytesRead, outMbx, @Status);
      CALL rqaread (console, @ibuffer, size(ibuffer), inMbx, @Status);
    END;

/*  Detach device and exit
    ----------------------
*/
    CALL rqaphysicaldetachdevice (b_dos, HARD, selectorof(NIL),
@Status);
    CALL rqexitiojob (0, @(11,'Normal Exit'), @Status);

END atypeon;
```

Asynchronous system calls have names that begin with *rqa*, and equivalent synchronous call have names that begin with *rqs*. When a function that is available through both asynchronous and synchronous system calls is discussed, the name will start with *rq[as]* to indicate both calls.

8.2.4 IORSs and DUIBs

To describe the logic of certain I/O operations, you need to know the names of two data structures associated with the I/O system and what they are used for. The actual contents of each data structure are covered in some detail in chapter 9, where their roles in the operation of I/O device drivers are discussed.

The first data structure is called a *Device Unit Information Block*, or DUIB. This data structure contains the information the I/O system needs to work with a particular I/O device unit, such as a certain disk drive or terminal. A list of DUIBs is kept in memory at all times, and a unique ASCII name exists for each possible device to which an application could connect. These DUIB names are also known as *physical device names*, and they were mentioned in chapter 2 when the *attachdevice* HI command was introduced. You can use the *physnames* command to list the DUIBs on a system.

The DUIB named `b_dos` was used in the sample programs in Figures 8.1 and 8.2. If you know the name for a DUIB, you can see what the I/O system knows about it by using the system debugger (or SoftScope) *vb* command. The structure of a DUIB is not of concern here, but the fields are discussed in chapter 9, and a `typedef` for this data structure is presented.

The other data structure is called an *Input/Output Request/Result Segment*, or IORS. (At any moment it is either a request segment or a result segment, so it gets only one R in its acronym.) IORS is the segment that is sent to `responseMbx` when the asynchronous part of a BIOS system call completes. It is created when the synchronous part of the call is made, when it gets filled with information supplied by the parameters to the system call and then sent to the proper device driver to perform the work of the system call. When the device driver task completes its work for the operation, it puts a result code and other information, such as the actual number of bytes that were read or written, into other fields of the same IORS, and sends the token for the IORS to the response mailbox. The following is the typedef for an IORS.

```
#pragma noalign (iorsStruct)
typedef struct iorsStruct {
  WORD                                  status;
  WORD                                  unitStatus;
  NATIVE_WORD                           actual;
#if _ARCHITECTURE_ < 386
  WORD                                  actualfill;
#endif
  WORD                                  device;
  BYTE                                  unit;
  BYTE                                  funct;
  WORD                                  subfunct;
  DWORD                                 deviceloc;
  BYTE far *                            buff;
```

```
  NATIVE_WORD                          count;
#if _ARCHITECTURE_ < 386
  WORD                                 countfill;
#endif
  void far *                           aux;
  iorsStruct far
*                                      linkForward;
  iorsStruct far
*                                      linkBackward;
  TOKEN                                responseMbx;
  BYTE                                 done;
  BYTE                                 fill;
  TOKEN                                cancelID;
  TOKEN                                connection;
} IORSSTRUCT;
```

The first word, `status`, gives the concurrent condition code for asynchronous system calls, with `E_OK` (zero) signifying normal completion. Most of the other fields in this structure will make more sense as you read more of this chapter, so you might want to refer back to the structure from time to time.

8.2.5 I/O connection objects

Our model for iRMX I/O has an application connect to a device first and then to a file on the device. In keeping with the object-based nature of iRMX, the I/O system provides an object type, called an I/O connection, to manage the connections that applications make. The sample programs used variables named `console` and `file` to hold tokens for I/O connection objects. The structure of the information inside a connection object varies, depending on what type of device it connects to and whether it represents a connection to the device itself or a connection to a file on the device. You can talk about a device connection or a file connection, but both terms refer to the same object type, except with different internal data structures.

To make an application device independent, the application should include the code for connecting to a file, but not the code for connecting to a particular device. You can use logical names to accomplish this. To connect to a file, the application must specify an existing device-connection object. Tokens for a number of device-connection objects can be cataloged into the object directory of a well-known job (normally the root job of the system or the global job for the application) using well-known directory entry names. Applications then obtain device-connection tokens by referencing these well-known names rather than the actual device names. This way, the same application can work with different devices by transparently changing the device connection object that has been cataloged with the well-known name.

The well-known names in the object directories are called *logical names*. The EIOS conveniently provides system calls for setting up these logical names and searching the proper object directories for them for the application programs. The HI commands *attachdevice* and *attachfile* introduced in chapter 2 allow users to set up logical names from the command line.

Connections to devices can be shared across jobs, but connections to files cannot. Each job must obtain its own connection objects to files. It's not that two jobs cannot access the same file simultaneously, it's just that tasks in different jobs cannot use the same connection object to do so. The issue is one of system integrity. If two jobs share an open file connection and the job that owns the connection terminates, the other job is left with a token for a connection object that does not exist. This type of problem exists whenever jobs share objects, but for file connections, the memory protection faults and issues of file integrity that could ensue are too serious to allow the situation to occur at all. If a job can access a token for a file connection that belongs to another job, the BIOS system calls *rqaattachfile( )* or *rqacreatefile( )* can be used to copy the existing connection object, but this copy belongs to the calling job so that the job can have its own connection to the file.

8.2.6 System calls for managing connection objects

The BIOS and EIOS each supply a system call to create a connection object for a device. The following is the BIOS version.

```
extern void
rqaphysicalattachdevice (          STRING far *          deviceNamePtr,
                                   BYTE                  fileDriver,
                                   TOKEN                 responseMbx,
                                   WORD far *            exceptPtr);
```

`deviceNamePtr` is a pointer to an iRMX string (which consist of a byte containing the length of the string followed by the bytes that constitute the actual string) that names the DUIB for the device to be attached. The type of device is specified by the `fileDriver` parameter, with the following possible values.

Physical. The physical file driver is used for devices such as printers and terminals that do not support named file systems. It is also used for disks to be accessed on a block-by-block basis rather than by named files. The terms *physical file driver* and *physical attachdevice* have nothing to do with each other. Physical file driver identifies one of the five file drivers in this list. Physical attachdevice differentiates this system call from the EIOS system call *logical attachdevice*.

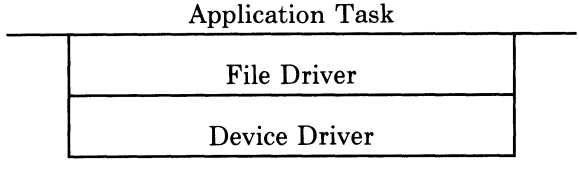
Stream. An iRMX stream is not really a peripheral device. Tasks can communicate with each other by writing to and reading from streams. The data that is read or written is transferred using memory buffers managed by the stream file driver.

Named. The named file driver is used for disks formatted to use the iRMX file system structure described later in this chapter. Name is the type of file system described in chapter 2.

Remote. The remote file driver is used for remote computer systems connected to the local computer by a network that supports Intel's OpenNet protocols. OpenNet uses ISO protocols to connect computers running iRMX, XENIX, UNIX System V, DOS, and VAX/VMS. The messages that OpenNet passes over the network adhere to Microsoft's Server Message Block (SMB) protocol.

EDOS. This acronym stands for *Encapsulated DOS*, and is used with disks formatted under Microsoft DOS. EDOS file systems are simpler but have fewer capabilities compared to iRMX Named file systems. This file driver allows access to DOS disks using the same syntax as for accessing iRMX files. For example, EDOS uses the forward slash character to separate components of a pathname rather than the backslash used by DOS.

Not all iRMX configurations support all five device types. As the name for this parameter implies, each type of device is supported by a software module called a *file driver*. *Device driver* might seem more appropriate, but file driver makes sense if you think of this module as the one that transforms file-oriented system calls into device-specific operations. Thus, application tasks, file drivers, and device drivers are related as shown, with the file driver layer named for the interface it presents to the application tasks above it.



Because *rqaphysicalattachdevice( )* is an asynchronous system call (note the *rqa* prefix to the system call name), the token for the new connection object is not returned immediately. Rather, the token is sent to the `responseMbx` mailbox after the asynchronous part of the call completes. This process is an exception to the general procedure for asynchronous BIOS calls. An IORS is sent to `responseMbx` only if this call fails, with the reason for failure given in a status field within the IORS.

This feature thus raises the question of how a task can know whether a token it receives at a message mailbox is for an IORS or an I/O connection object. The answer is that each iRMX object type has an associated type code, which can be determined by passing the token as a parameter to the system call *rqgettype( )*, which will return a word containing one of the following values:

```
0x0001                  Job
0x0002                  Task
0x0003                  Mailbox
0x0004                  Semaphore
0x0005                  Region
0x0006                  Segment
0x0007                  Extension
0x0009                  Multibus II Port
0x000A                  Buffer Pool
0x0100                  Composite ( I/O User )
0x0101                  Composite ( I/O Connection)
0x0300                  Composite ( I/O Job)
0x0301                  Composite (Logical Device)
0x8000-0xFFFF           User-Created Composites
```

Thus, the type code for the token that arrives at `responseMbx` is 0x0101 if the connection was created successfully, but 0x0006 if the call failed and an IORS is returned instead, as illustrated several times in Figure 8.3.

A critical concept here is that the I/O connection object created by *rqaphysicalattachdevice( )* belongs to the job that created it, and is automatically deleted when the job that created it terminates. Since device connections are normally shared across jobs, they should continue to exist after the job that creates them terminates. For example, the *attachdevice* HI command allows users to create a device connection and associated logical name that persist after the HI *attachdevice* command job exits. To achieve this permanence, the EIOS provides the *rqlogicalattachdevice( )* system call, which causes a task owned by the EIOS to call *rqaphysicalattachdevice( )*. This way, the device connection is owned by the EIOS job, and remains in place after the job for the *attachdevice* command exits.

Very few applications call *rqaphysicalattachdevice( )* to create connections that they use themselves. In addition to the problem of nonpermanent connection objects, applications that call *rqaphysicalattachdevice( )* or *rqlogicalattachdevice( )* lose their device independence. Rather, jobs usually catalog the token for the connection to a device in an object directory where other jobs can access it. This concept is the heart of the EIOS logical name construct.

The EIOS manages a composite object type called a logical connection (object type 0x0301) for this purpose. When an application calls *rqlogicalattachdevice( )*, the EIOS creates a type 0x0301 object and catalogs the token for it in the root job's object directory. Later, when the first file con-

nection based on this logical name is made, the EIOS calls *rqaphysicalattachdevice()* to create the type 0x0101 object, waits at `responseMbx` for the asynchronous part of that call to complete, and updates the type 0x0301 object to include the token for the type 0x0101 object. Thus, a logical name is the name of an object cataloged in the root job's object directory that can be used to find the token for a connection to a device. The EIOS handles logical names for connections to files a little differently, as you will soon see.

If you could enter an HI *logicalnames* command during the execution of the sample I/O programs, you would see the logical name `B` in the list displayed by the command. If you build the program and omit the call to *rqlogicaldetachdevice()*, the logical name would still be in place after the program stops running. If you look in the object directory of the root job while the program is running (using SoftScope), you will find a token for a type 0x0301 object cataloged with the name `B`. If you dump the memory segment for the type 0x0301 object before the call to *rqscreatefile()*, and then again after that call, you would find that a token for a type 0x0101 object appears in the segment as the result of the call. The following is the function prototype for *rqlogicalattachdevice()*.

```
extern void
rqlogicalattachdevice (          STRING far *          logicalNamePtr,
                                 STRING far *          deviceNamePtr,
                                 BYTE                  fileDriver,
                                 WORD far *            exceptPtr );
```

`logicalNamePtr` points to an iRMX string for the logical name to be created. The type 0x0301 object will be cataloged in the root job's object directory using this name. Colons around the name are optional, and letter case does not matter. The EIOS strips off any colons and converts all letters to uppercase before it calls *rqcatalogobject()*. EIOS also converts logical names into this canonical form before it calls *rqlookupobject()*, such as when an application task calls the EIOS system call *rqlookupconnection()*. The colons around a logical name are needed only when a pathname is supplied to the EIOS, so EIOS can recognize whether the pathname begins with a logical name or not.

`deviceNamePtr` and `fileDriver` are coded exactly the same as the corresponding parameters to *rqaphysicalattachdevice()*. The EIOS simply stores them in the new type 0x0301 object until the logical name is used for connecting to a file, at which time they are used as parameters in a call to *rqaphysicalattachdevice()*.

Once a device connection has been created, it can then be used to create a file connection using one of two BIOS calls. The first one is *rqaattachfile()*, which can be used regardless of which file driver was used when the I/O connection to the device was created.

```
extern void
rqaattachfile (          TOKEN                  user,
                         TOKEN                  prefix,
                         STRING far *           subpathPtr,
                         TOKEN                  responseMbx,
                         WORD far *             exceptPtr );
```

A token for either a new I/O connection object (type 0x0101) or an IORS, in case of an error, is returned to `responseMbx` when the asynchronous part of this call completes.

`user` is a token for an I/O user object (composite object type 0x0100) that consists of a list of user ID numbers. This user object is used only for files that reside on devices attached with the named or remote file driver. You can omit this parameter by coding a selector for a null pointer (a word of zeros). If you omit this parameter for a file that resides on a device attached with the named or remote file driver, the BIOS will use the default user object for the job, which is the one cataloged in the job's object directory under the name `R?IOUSER`. In general, the user object is not used when the file is attached, it is simply added to the I/O connection object at this time and used later, when the file is opened, to check for a user's access rights to the file. For remote files, the user object is used immediately, along with the user's name and encrypted password from the `:config:udf` file, to verify that the person who logged on to the local system is a legitimate user on the remote system, with the same password on both systems[3].

`prefix` and `subpathPtr` are used together to identify the file to be attached. Various combinations of these two parameters are possible, depending on the file driver being used. These combinations are:

File Driver	Prefix	Subpath
Physical or Stream	Either a token for a connection to the device, or a null selector.	Null pointer. Always ignored.
Named, Remote, or EDOS	Either a token for a connection to the device, or a null selector.	Either a pointer to a pathname string or a null pointer.

If `prefix` is a null selector (a 16-bit word of zeros), a default prefix must exist for the job, which is a token for a connection object cataloged in the job's object directory using the name `$`. The prefix itself can be a token for a connection to either a device or a file, because connections to files imply a

[3]iRMX is not a secure time-sharing system because any user who can execute a program can create a user object that includes the superuser's ID, giving the user super-user privileges for at least for the duration of the user's program. On the other hand, the operating system does not extend this *laissez-faire* attitude to computer systems with which it shares a network. iRMX systems do not allow users to break security mechanisms of remote systems through their own security loopholes.

unique device to the I/O system. In an apparent inversion of the normal relationship between the BIOS and the EIOS, you can specify a type 0x0301 connection object as the prefix (a token for a logical name created by the EIOS), and the BIOS will extract the embedded 0x0101 connection object from it automatically, provided the connection has been physically attached through a previous EIOS system call.

`subpathPtr` points to a string that actually names the file to be accessed. It is always a null pointer for physical and stream devices because those devices do not support named files. For the Named, Remote, and EDOS file drivers, the pathname string identifies a file or directory that can be located by starting at the file or directory specified by the prefix and then following <^> and </> characters in the string to determine what directory contains the file being attached. If the pathname string starts with </>, the search for the file always starts at the root directory of the device specified or implied by the prefix.

Although the rules about the `prefix` token and the `subpathPtr` string are complex, they need not be too confusing. One point to remember is that the prefix can specify a connection to either a device or a file. A second point is that the named, remote, and EDOS file drivers do not distinguish between files and directories at this point, so the token returned by the asynchronous part of the call can be for an I/O connection object to either a file or a directory. Finally, the relationships between the prefix and the subpath should become more intuitive when you look at the corresponding EIOS call, *rqsattachfile( )*, used in the sample I/O programs:

```
extern TOKEN
rqsattachfile (          STRING far *          pathPtr,
                         WORD far *            exceptPtr);
```

The token returned by this system call is for a type 0x0101 connection object, not for a type 0x0301 object. The latter are used only for logical device connections.

The EIOS code for this call generates a call to *rqaattachfile( )*. How does the EIOS have values for all the parameters for the call to *rqaattachfile( )* when it receives only two parameters for a call to *rqsattachfile( )*? First, the user parameter is always set to a null selector so that the BIOS will use the default user object, which is always cataloged in the object directory of an I/O job. Second, the EIOS has its own mailbox it uses for `responseMbx`, so that leaves only the `prefix` and `subpathPtr` parameters. Clearly, the EIOS creates the values for these two parameters from the `pathPtr` it receives.

The syntax rules for constructing the string pointed to by `pathPtr` are the same introduced in chapter 2 for typing pathnames in commands entered at the `iRMX>` prompt. The key to parsing a pathname string lies in the value of the first character in the string.

Rule	First Character at pathPtr	prefix	subpathPtr
1	$	Null selector.	Pointer to second character at pathPtr if there is one, otherwise null selector.
2	:	Look up the logical name. If the type 0x0301 object does not yet contain a type 0x0101 object, call *rqaphysicalattachdevice ( )* to get one. Use the token for the type 0x0101 object.	Pointer to the first character after the second : in the path if there is one, otherwise, null selector.
3	/	Null selector.	pathPtr.
4	^	Null selector	pathPtr.
5	Other character	Null selector	pathPtr.

Rule 1 includes four subcases:

1. The string following the $ character is empty. The application obtains a second connection to the same device or file that $ represents. Useful for an application that changes $ temporarily.
2. The character following the $ is a /. Same as Rule 3.
3. The character following the $ is a ^. Same as Rule 4.
4. Any other character follows the $. Same as Rule 5.

Rule 2 is used when the pathname starts with a logical name. The file or device represented by the logical name is the *prefix*, and anything that follows the logical name is the *subpath*. This use of a logical name to specify the prefix can save a lot of time compared to a full pathname starting at the root of the file system because the directories that lead from the root to the point of the logical name do not need to be searched.

In the case of Rule 3, a full pathname is specified from the root directory of the volume that contains the default prefix file.

Rule 4 is used for a pathname that is specified relative to the default prefix file, and Rule 5 is the same as Rule 4 as far as the EIOS is concerned. The BIOS, however, starts at different parts of the file system tree for Rules 4 and 5.

The other calls for creating connections to files are used only with devices that actually support named file systems (devices connected using the Named, Remote, or EDOS file drivers). These two calls, *rqacreatefile( )* and *rqscreatefile( )*, as their names imply, create a file on the disk if one does not already exist. If the file does already exist, the application can specify what to do with the current contents of the file for *rqacreatefile( )*, but the

contents are always discarded for *rqscreatefile( )*. The following is the more complicated BIOS version first:

```
extern void
rqacreatefile (          TOKEN                  user,
                         TOKEN                  prefix,
                         STRING far *           subpathPtr,
                         BYTE                   access,
                         WORD                   granularity,
                         DWORD                  size,
                         BYTE                   mustCreate,
                         TOKEN                  responseMbx,
                         WORD far *             exceptPtr );
```

The parameters with the same names as the corresponding *rqaattachfile( )* parameters have the same interpretations here as for that call.

`access` specifies the access rights with which the file is to be created. Four possible values tell whether the owner of the file can perform delete, read, append (write to the end) or update (write anywhere) operations on the file. The *rq[as]changeaccess( )* system calls can set the access rights for other users. The EDOS file driver creates all files with read access enabled for the World user because of the nature of the DOS file system. The Remote file driver must map these access rights to those of the remote operating system as best it can. The value for this parameter can be computed by adding the values one, two, four, and eight for delete, read, append, and update access rights, respectively.

The next three parameters, `granularity`, `size`, and `mustCreate`, work together to produce various effects. They control whether existing files are truncated or not, and they allow you to create three different types of new files: normal, real-time, and temporary.

Truncating existing files. Existing files are truncated when the `size` parameter has a value of 0. For new files, a size of 0 simply means that the file will not have any disk blocks allocated to it until an application opens and writes to the file. If you specify a value greater than 0 for `size`, enough disk blocks to accommodate the number of bytes specified will be allocated to the file, but not initialized with data in any way. If the file already exists, its current size will be either extended or shortened to match the size specified. Again, bytes added to a file this way are not initialized.

Normal iRMX file. A normal file is one that is allocated space on the disk volume on a demand basis. Normal files are likely to be fragmented because no particular constraint is placed on the locations of the disk blocks allocated to the file. A file's granularity is the number of disk blocks allocated to the file when it becomes too large for its current allocation, or freed when it becomes smaller. The expected size and growth dynamics of the new file determine the best value for this parameter. A large value can improve performance if a file is going to grow quickly, since the overhead of allocating new blocks to the file does not occur as often. On the other hand, a large file

granularity might result in wasted disk space. If a file needs just a few bytes beyond the end of its current last block, it is still allocated the number of bytes given by `granularity`. Most of those bytes are wasted until the file grows to occupy them. The `granularity` parameter is specified in bytes so that applications do not need to know what the block size of the disks being worked with are, but 1,024 is the most commonly used block size for iRMX disks. This parameter is automatically rounded up by the BIOS to be a multiple of the disk block size if necessary.

Real-time file. A real-time file is a contiguous file. To create a real-time file, you must specify the size of the file using the `size` parameter and use a value of 0xFFFF for the file granularity. The file is allocated a contiguous set of disk blocks if possible. The system allows a real-time file to be extended beyond its initial size, but any blocks added to the file after it is created are not necessarily contiguous. The EDOS file driver does not support real-time files.

Temporary file. A temporary file is an unnamed file automatically deleted when the connection to it is deleted, which is normally when the job that created it terminates. The BIOS actually deletes the file when the last connection to it is deleted in case the connection is shared across jobs. To create a temporary file, the prefix/subpath combination must identify an existing directory (not a file) and the `mustCreate` parameter is set to false (0). A connection to a named or EDOS disk device can be used as a connection to the root directory of that device, providing a convenient place to put temporary files.

A user can create a temporary file in any directory, whether that user has write access to the directory or not, because no directory entry is actually created for the temporary file. The exception to this case is for a temporary file created on a remote computer's disk. The temporary file is actually entered into the directory of the remote system, which requires the user to have write permission on the remote system.

The `mustCreate` parameter can also be used to ensure the file being created does not yet exist. If the parameter is true and the prefix/subpath does identify an existing file rather than a directory, the system call fails with a condition code of E_FEXIST (0x0020). Unix users might think this feature is important, as it is the basis for an important type of Unix IPC, file locks. File locks are not normally used on iRMX systems because of the more efficient mechanisms for IPC supplied by the Nucleus.

In contrast to the BIOS version, the *rqscreatefile( )* system call provided by the EIOS is simplicity itself. The price paid for this simplicity is loss of functionality, however. The following is the prototype.

```
extern TOKEN
rqscreatefile (          STRING far *          pathPtr,
                         WORD far *            exceptPtr);
```

You can create normal and temporary files with this system call, but not real-time (contiguous) files. If the file already exists, this call always tries

to truncate the file length to 0. The prefix/subpath parameters for the EIOS call to *rqacreatefile( )* are created using the same rules as for *rqsattachfile( )*.

The reasons for using *rqacreatefile( )* instead of the *rqsattachfile( )* call are for the following reasons:

- To obtain asynchronous processing of the system call. Several disk accesses are involved in each level of the file system tree that must be traversed to access a file or directory, so this call can take a relatively long time to execute.
- To create the file with the owner's initial access rights set to something other than delete, read, update and append.
- To pre-allocate disk blocks to a normal file or to set the size of an existing file to a value other than zero.
- To create a real-time (contiguous) file.

8.2.7 System calls for data transfers

Once a connection to a file has been created, whether the file is a disk file or simply a physical or stream device that is to be treated as a file, the connection must be opened before reading or writing with it. Three issues must be resolved when a connection is opened: sharing (all connections), access rights (connections based on the Named, Remote, and EDOS file drivers only), and buffering (*rqsopen( )* only). The following is the BIOS call for opening a connection.

```
extern void
rqaopen (               TOKEN               connection,
                        BYTE                mode,
                        BYTE                share,
                        TOKEN               responseMbx,
                        WORD far *          exceptPtr );
```

`connection` is a token for a connection to a file (not a device) that could have been created by a BIOS or EIOS *attachfile* or *createfile* system call. The connection object cannot be open at the time this call is made, although it can be closed and reopened any number of times.

`mode` is a value to indicate whether the connection will be used for reading (a value of 1), writing (a value of 2), or both (a value of 3). `share` is a value to indicate whether the application is willing to share the file with other readers (a value of 1), other writers (a value of 2), both readers and writers (a value of 3), or no other readers or writers (a value of 0). If a file is a directory, it must be opened for reading only, share with all. Special system calls exist for reading and writing directories. This call always returns an IORS to the response mailbox, and the application can use either the *rqwaitio( )* or *rqreceivemessage( )* system call to check for the result of the asynchronous part of this system call.

To take advantage of the automatic buffering facilities available with the EIOS, use *rqsopen( )* instead:

```
extern void
rqsopen (               TOKEN               connection,
                        BYTE                mode,
                        BYTE                numBuffers,
                        WORD far *          exceptPtr);
```

The value of `mode` is set to a single value that combines the `mode` and `share` parameters of *rqaopen( )*. The value is a number between 1 and 12 as follows:

	Share with None	Share with Readers	Share with Writers	Share with Both
Open for reading	4	7	10	1
Open for writing	5	8	11	2
Open for both	6	9	12	3

Mode number 1 is used for reading and share with all, which is the value that must be used for opening connections to directory files.

`numBuffers` is used to invoke automatic EIOS buffering of I/O data transfers, if desired. The EIOS maintains a pool of 1,024 byte buffers that it can use for managing data transfers. If you specify a value of 0 for this parameter, the EIOS will not buffer data for this connection. Whatever is written from an application program's buffer is transferred immediately to the device, and whatever is read from the device is copied immediately into a buffer supplied by the application. As mentioned previously, a value of 0 is particularly appropriate for terminals, for which buffering can lead to confusing interactions for a terminal user.

When the EIOS does buffering, it attempts to optimize data transfers to and from the device. If the program is reading from a file, the EIOS will start reading from the next sequential location in the file as soon as it has finished reading from the current location. It will continue reading ahead until it exhausts the number of buffers specified in the call to *rqsopen( )*. Likewise, when a program writes to a file using buffering, the EIOS does not actually write information to the file until a buffer is filled, which is then written to the device while the application proceeds to write more information into another EIOS buffer. If an application opens two files, one for reading and one for writing, each with two buffers, the EIOS allows the application to copy one file to the other with full overlap of reading and writing operations, but without the application handling asynchronous I/O at the BIOS level.

EIOS buffering is not always a good idea for disk file I/O. If an application uses *rqsseek( )* (described in section 8.2.8) to read from nonconsecutive locations within a file, the reading ahead that the EIOS does results in extra disk accesses that interfere with the actual disk operations the application needs to perform. Also, large transfers might be performed more efficiently by using larger buffers than the ones used by the EIOS. The EIOS uses a fixed size for all of its internal buffers, normally 1,024 bytes. This

buffer size can be set for the Named, EDOS, and Remote file drivers in the `rmx.ini` file or can be set using the ICU (discussed in chapter 9) for systems that do not support the ICU.

The BIOS calls for reading and writing are:

```
extern void
rqaread (                TOKEN              connection,
                         BYTE far *         bufferPtr,
                         DWORD              count,
                         TOKEN              responseMbx,
                         WORD far *         exceptPtr);
extern void
rqawrite (               TOKEN              connection,
                         BYTE far *         bufferPtr,
                         DWORD              count,
                         TOKEN              responseMbx,
                         WORD far *         exceptPtr);
```

For the EIOS, the calls are:

```
extern NATIVE_WORD
rqsreadmove (            TOKEN              connection,
                         BYTE far *         bufferPtr,
'                        DWORD              count,
                         WORD far *         exceptPtr);
extern NATIVE_WORD
rqswritemove (           TOKEN              connection,
                         BYTE far *         bufferPtr,
'                        DWORD              count,
                         WORD far *         exceptPtr);
```

For all of these calls, the number of `count` bytes are read or written using the open file connection object specified by `connection`. The bytes are written from or read into the memory location pointed to by `bufferPtr`. The actual number of bytes read or written is normally equal to the value of `count`, but will be less if an attempt is made to read past the end of the file, to write to a full disk, or if a memory protection violation occurs while accessing the buffer pointed to by `bufferPtr`. The actual number of bytes transferred is the returned value for the EIOS functions. For the BIOS calls, the actual number of bytes transferred is returned in the IORS when the asynchronous part of the call completes. The word *move* in the names of the EIOS functions refer to the fact that these functions move data between the application program's buffer and one of the EIOS's buffers, with the actual read or write operation occurring according to the buffering technique in place for the connection.

The BIOS rejects calls to *rqaread()* and *rqawrite()* for connections opened by the EIOS with a condition code value of E_BUFFEREDCONN (0x0036), so the EIOS record of what buffers contain what information for a file do not become invalid by circumventing the EIOS calls.

8.2.8 Seek and truncate operations

Two other system calls fall into the data operations category, *seeking* to a particular position in a file, and *truncating* a file. Both of these system calls

are executed in the BIOS by updating housekeeping information in memory, and they complete their execution very rapidly. Both operations might also initiate more time-consuming operations, however, that take place after the system call completes. The potentially time-consuming operations are discussed first, then the system calls that are used to initiate them.

Disk access. Disk access time is the time it takes to transfer data to or from a disk device. The four components to this time interval, are *seek*, *select*, *search*, and *transfer* times. To understand these components, the structure of a disk must first be reviewed.

Information is recorded on a disk by rotating a rigid or flexible surface under a read/write head. The read/write head assembly can be positioned at a number of discrete positions on the recording surface, defining a number of concentric circles where data can be stored, called *tracks*.

Each track is divided into a fixed number of segments called *sectors*, with one sector being the smallest amount of information that can be written to or read from a disk at one time. Typically, between 256 and 1,024 bytes are stored per sector, and between 9 and 63 or more sectors exist per track.

Flexible disks normally have two recording surfaces, one on each side of the rotating material, and two read/write heads linked together so that either of two tracks, one on either side of the diskette, can be accessed from one read/write head position. Hard disks often have several rotating platters linked together, with a linked set of read/write heads (two per platter) that can be positioned simultaneously. The tracks that can be accessed from a single position of the read/write heads are called a *cylinder*. For a floppy disk, there are two tracks per cylinder, and for a hard disk, there are two times the number of platters for each cylinder. Hard disks typically have 200 to 400 or more cylinders and anywhere from two to 50 platters.

Seek time is the time it takes to move the read/write heads from their current position to the cylinder that contains the next data to be read or written. The term *track-to-track positioning time* is generally used to refer to the time it takes to move the read/write heads from one cylinder to an adjacent cylinder, which is usually a few milliseconds for hard drives. Because of the inertia involved in starting and stopping head movements, less time is required to move the heads as a group across a set of adjacent cylinders than to move across the same set one at a time. For a truly random set of disk accesses, the average seek time should be one half the time to move the read/write heads across all the cylinders on the disk, or half a second for a 200 cylinder drive with a track-to-track positioning time of 5 msec.

Select time is the time it takes to select the read/write head that will be used for accessing one of the tracks in the cylinder. This operation is done electronically and is normally overlapped in time with one of the mechanical operations involved in accessing the disk.

Search time, sometimes called *rotational delay*, is the amount of time it takes for the desired sector to start passing under the read/write heads once the heads have been positioned on the proper track. Sometimes, the proper sector arrives almost immediately, other times the sector has just gone past the head. The average search time equals half the time it takes the disk to make one complete revolution. Hard disks typically rotate at 3,600 RPM, leading to average search times of 8 msec.

Data transfer time is the time it takes to copy information from the surface of the disk into the computer's memory, or vice versa, which usually expressed as its reciprocal, *data transfer rate*. More factors are involved here than just the time it takes for a sector to pass under the read/write head, including the speed of the bus connecting the disk to the system's memory. Values range from 1 KB to 8 MB per second.

One factor that can effect average search time for those cases in which several sectors are to be read or written sequentially is the order in which consecutively numbered sectors are stored on a track, which does not have to be in consecutive positions. The sectors on a track can be interleaved. For example, if nine sectors are stored in the sequence 1,6,2,7,3,8,4,9,5, the track is said to be 2-way interleaved because logically consecutive sectors are physically 2 sectors apart on the track. Interleaving can reduce average search time for computers that need to read consecutive sectors, but that cannot issue the commands to read successive sectors as fast as the sectors arrive at the read/write heads. Interleaving can be done easily because the logical number of each sector is stored on the disk at the beginning of the sector. The term *search time* refers to the fact that the disk controller searches a track for a sector with the logical number requested by the software driver, which makes the order of the logical sectors on the disk arbitrary, as far as the disk controller is concerned.

High-performance disks typically provide a cache on the disk controller capable of holding all the data for a complete track. By starting to read into this cache as soon as the heads reach the proper cylinder, the effects of rotational delay can be minimized and the need for interleaving eliminated. Three conclusions can be drawn from disk access time:

1. Disk access time is a composite value that depends on the physical characteristics of the disk, the organization of the information on the disk, and the pattern of accesses made to the data on the disk.
2. Seek time is just one component of the time it takes to read or write disk data.
3. Beware of anyone who tries to use the value of just one component of disk access time to tell you how fast a disk is, especially if that person is trying to sell you a disk!

The BIOS and EIOS system calls for controlling the position of disk accesses within a file are as follows:

```
extern void
rqaseek (               TOKEN               connection,
                        BYTE                mode,
                        DWORD               seekAmount,
                        TOKEN               responseMbx,
                        WORD far *          exceptPtr);
extern void
rqsseek (               TOKEN               connection,
                        BYTE                mode,
                        DWORD               seekAmount,
                        WORD far *          exceptPtr);
```

The BIOS maintains a DWORD in the data structure for an open connection to a disk file that signifies the next position in the file for reading or writing. This value is often called a *file pointer* but it is not a memory pointer (selector and offset), just an unsigned integer that starts at 0 when the file is opened and is incremented by the number of bytes actually transferred each time the file is read or written. For the Named file driver, the seek system calls simply assign a new value to this variable so that the next read or write operation occurs at the desired location within the file. In this instance, the system call executes very quickly, but it can affect the amount of time it takes to perform the next disk transfer for the file. For the physical file driver, *seek* calls are passed to the device driver immediately. Thus, the name *seek* for these calls is only loosely linked to the actual hardware seek operation performed during a disk access.

Seeking can affect the allocation of disk blocks to a file. If you seek to the end of a file and then write data to the file, the new data goes into whatever space is still available in the last data block allocated to the file, and causes a new block to be allocated if the last one fills up. If you seek beyond the current end of a file, additional disk blocks are allocated to the file for the space between the current end of file and the new end of file, but the bytes in that unused part of the file are not initialized.

These two system calls are interchangeable except the BIOS will not work with buffered EIOS connections `mode` tells one of four ways to interpret `seekAmount`:

- The new file position is to be the current file position minus `seekAmount (mode-1)`.
- The new file position is to be the value of `seekAmount (mode-2)`.
- The new file position is to be the current file position plus `seekAmount (mode-3)`.
- The new file position is to be the end of the file minus `seekAmount (mode-4)`.

Seek operations can be performed with connections built on the physical file driver as well as the Named, Remote, and EDOS if it makes sense to do so. You cannot seek with a terminal, but you can seek on a disk that was attached with the physical file driver. Such disks are just one big file as far as iRMX is concerned.

File truncation. File truncation is very similar to seeking. The following are the calls:

```
extern void
rqatruncate (               TOKEN               connection,
                            TOKEN               responseMbx,
                            WORD far *          exceptPtr);

extern void
rqstruncatefile (           TOKEN               connection,
                            WORD far *          exceptPtr);
```

These calls set the end of file to the current file position. Truncating is normally viewed as a way to remove all or part of a file, but by preceding a truncate call with a seek call beyond the end of the file, this call actually enlarges a file. Any disk blocks between the new end-of-file and the old end-of-file are released to the disk's free space immediately by this call if this call shrinks the file, or allocated immediately if the call enlarges the file. This call cannot be used with file connections based on the physical file driver because there is no known software technique for making a physical disk drive change its size.

8.3 Special Functions

Device-independent system calls are great, but they do not format disk drives or control character echoing on a terminal. Rather than add a new system call to the EIOS and BIOS for every device-dependent operation that an application might need to perform, one system call provides access to whatever special functions might be provided by a particular device driver. These system calls, appropriately enough, are called *rqaspecial()* and *rqsspecial()*. Each device driver in the system supports its own (possibly empty) set of special functions. The functions for formatting a disk drive and basic terminal operations are used as examples; If you are interested you can find out more from the documentation available for the various device drivers[4]. The following are the two system calls for special operations:

```
extern void
rqaspecial (            TOKEN                   connection,
                        WORD                    functionCode,
                        void far *              parameterPtr,
                        TOKEN                   responseMbx,
                        WORD far *              exceptPtr);
```

[4]The device drivers provided with the operating system are documented in the *iRMX Device Driver Programming Concepts* manual, volume 7 of the iRMX for Windows documentation set.

```
extern void
rqsspecial (          TOKEN                 connection,
                      WORD                  functionCode,
                      void far *            parameterPtr,
                      IORSSTRUCT far *      iorsPtr,
                      WORD far *            exceptPtr);
```

Here is a case where the EIOS user can work with the IORS that is normally returned to `responseMbx` for BIOS calls. Most EIOS users do not need this information, and code `iorsPtr` as a null pointer. If `iorsPtr` is not null, the EIOS receives the IORS at its own mailbox and copies part of it to the data structure the caller has reserved at the address pointed to by `iorsPtr`. The typedef `IORSSTRUCT` in `:include:rmxc.h` defines the actual fields returned for *rqsspecial()*.
`iorsPtr`. The typedef `IORSSTRUCT` in `:inlcude:rmxc.h` defines the actual fields returned for *rqsspecial()*.

The `functionCode` parameter specifies which of several special operations the device driver is to perform. Each device driver can interpret the function code differently, but the device drivers supplied with the operating system all interpret the values listed in Table 8.1 uniformly. The File Driver column in the figure lists the file drivers that can be used to connect to device drivers that support the functions listed. The functions are implemented by device drivers, not file drivers. If you add your own device driver and want it to support special functions other than those listed in the figure, use values from 0x8000 to 0xFFFF to ensure no conflict occurs with any additional standard codes that might be added to the system.

TABLE 8.1 Function Codes for *rqaspecial()* and *rqspecial()*.

Code	Function	File Driver
0	Format track	Physical
0	Query	Stream
1	Satisfy	Stream
2	Notify	Physical or Named
3	Get disk/tape data	Physical
4	Get terminal data	Physical
5	Set terminal data	Physical
6	Set signal	Physical
7	Rewind tape	Physical
8	Read tape file mark	Physical
9	Write tape file mark	Physical
10	Retension tape	Physical
11	Set character font	Physical
12	Set bad track/sector information	Physical
13	Get bad track/sector information	Physical
14, 15	Reserved	
16	Get terminal status	Physical
17	Cancel terminal I/O	Physical
18	Resume terminal I/O	Physical
19-0x7FFF	Reserved for other Intel drivers	
0x8000-0xFFF	Available for user-written drivers	

The `parameterPtr` argument is a pointer to one of several data structures. The specific data structure depends on the value of `functionCode`. Strict C pedants would argue that this argument be prototyped as a pointer to a union of the different data structures, but a pointer to void suffices.

All the special functions not discussed in the following subsections are fully covered in the documentation for the *rqaspecial( )* system call. The ones discussed in section 8.3.1 through 8.3.3 provide an idea of how the *rq[as]special( )* system calls work and introduce particularly useful terminal operations, such as setting up hot keys and controlling character echoing and line editing for the keyboard. Appendix C illustrates the use of two other *rq[as]special( )* functions, the query and satisfy functions for streams.

8.3.1 Format track

The first example is the function used to perform low-level formatting of a disk volume. The device drivers for some tape drives use this function to lay out tape blocks as well, but the disk operation is used as our model. As mentioned earlier, each track on a disk consists of a number of sectors, which might or might not be in consecutive order. Low-level formatting puts the binary framework on a track to set up sectors and identify each one with its logical position on the track. The HI *format* command does a low-level format of each track on the volume, and then creates a file system on the volume in a process called high-level formatting. High-level formatting initializes certain data blocks on the disk to act as the root directory and perform other housekeeping operations, such as setting up the fnodes for an iRMX volume or the File Allocation Table (FAT) for DOS volumes. The last section of this chapter describes the high-level format of an iRMX named disk. A disk volume accessed using the physical file driver is assumed only to have been low-level formatted. (It might have been high-level formatted by another OS, such as Unix.) The following is the data structure that `parameterPtr` points to for formatting a disk.

```
#pragma noalign (formatTrack)
struct formatTrack {
                                WORD            trackNumber;
                                WORD            interleave;
                                WORD            trackOffset;
                                WORD            fillCharacter;
}
```

`trackNumber` is the number of the track to be formatted. Rather than the application determining the physical organization of the disk, the device driver accepts a number between 0 and 1 less than the total number of tracks on the disk as the value of this field. The driver then translates this relative track number into the proper cylinder and read/write head number for the seek operation.

A value of 0 or 1 for `interleave` causes the track to be formatted with the sectors numbered sequentially on the track. Other values cause sector numbering to skip by the units specified, as described earlier. In the example given earlier, a sector sequence of 1,6,2,7,3,8,4,9,5 is obtained by specifying a value of 2 for this parameter. This parameter is ignored for devices that do not need to do interleaving.

An indicator marks the beginning of each track on a disk. If the mark goes past the read/write head twice without finding the sector number specified in a seek operation, the controller knows something is wrong with the track. Sector number 1 can be placed any number of actual sectors past the beginning-of-track mark by specifying a nonzero value for the `trackOffset` field.

When the controller formats the track, it fills all the data bytes in each sector with a single data value. Some controllers allow this character to be specified by the `fillCharacter` field, but others always use a built-in character.

8.3.2 Get/set terminal data

Treating a terminal as two sequential files, one for input and one for output, is good unless an application wants to treat a terminal as a terminal. Using the get and set terminal data functions of the *rq[as]special()* system calls, an application can control a large number of terminal operating characteristics. The idea behind this pair of functions is that the terminal device driver maintains a data structure for each I/O connection to a terminal. The data structure contains values specifying how line editing and character echoing are to be handled for the terminal. To change a setting in the data structure, an application first gets a copy of the current values for the data structure (using function 4), modifies the copy to make the desired changes, and then sends the copy back to the device driver (using function 5) to change the actual data structure maintained internally. The data structure used for these two functions looks like:

```
#pragma noalign (terminalAttributes)
struct terminalAttributes {
   WORD                          numWords;
   WORD                          numUsed;
   WORD                          connectionFlags;
   WORD                          terminalFlags;
   NATIVE_WORD                   inBaudRate;
   NATIVE_WORD                   outBaudRate;
   WORD                          scrollLines;
   WORD                          xySize;
   WORD                          xyOffset;
   WORD                          specialModes;
   WORD                          highWaterMark;
   WORD                          lowWaterMark;
   WORD                          fcOnChar;
```

```
WORD                    fcOffChar;
WORD                    linkParameter;
WORD                    spcHiWaterMark;
BYTE                    specialChar [4]
}
```

The distinction between the items defined in `connectionFlags` and those defined in `terminalFlags` is important. The items in `connectionFlags`, which include the echo-character function illustrated in Figures 8.4 and 8.5 apply only to the connection specified by the first parameter to *rqaspecial( )* or *rqsspecial( )*. Since I/O operations can only be performed using connections that belong to the calling job, it follows that one job cannot affect the `connectionFlags` options for another job. The sample programs and the CLI run as different jobs, so they have their own connections to the console device. When the sample programs exit, their connections are deleted, and their echo-suppression operation has no effect on the CLI's I/O to the same device. A corollary of this relationship with regard to console I/O, incidentally, is that a program cannot change the behavior of *rqcsendcoresponse( )* by setting `connectionFlags` because there is no way for an application program to modify the connections to `:CI:` and `:CO:` used for that system call.

As its name implies, the `terminalFlags` field is used to control features that affect all connections to a terminal, such as the use of modem control functions. A utility program called *term* can be used to change many of these functions from the command line.

The PLM program in Figure 8.4 illustrates control of character echoing. It uses the standard EIOS system calls seen earlier in this chapter to create a file connection to the terminal based on the connections already established for the job with the logical names `:CI:` and `:CO:`. The calls to *rqsattachfile( )* could have been replaced with calls to *rqlookupconnection( )* with the same effect. The connections are opened, and the connection to `:CI:` is then modified by turning bit number 2 of the `connectionFlags` word for the console connection on to suppress character echoing. Once the user's password has been carefully read in without echoing to the screen, it is returned to the main program, which promptly displays it for all to see.

The C program, Figure 8.5, shows that you can modify `connectionFlags` for the I/O connections that the C run-time library uses for standard I/O. To do so, you need to know which iRMX connection object is being used by the run-time library for the particular I/O stream to be modified. The iCx86 run-time libraries provide a function, *_get_rmx_conn( )* which returns a token for the proper iRMX connection object, given the file descriptor for the I/O stream. In the figure, this token is the file descriptor for the standard input device. The sample program uses the standard library function *fileno( )* to convert the `FILE` pointer `stdin` to a file descriptor, although the value of 0 for the file descriptor could have been

Figure 8.4 PLM program to read a user's password from the console input device (:CI:) without displaying it on the screen.

```
/***>  passwd.plm  <*************************************************
 *
 *  PLM Program to read from :CI: without echo
 *  The program prompts for a password, which does not echo as the
 *  user types it.
 *
 ************************************************************************/
passwd: DO;
$include (passwd.ext)

/*  Procedure to prompt for password and read it without echo
    ========================================================
*/
getpassword: PROCEDURE (replyPtr);
DECLARE
        replyPtr                    POINTER,
        password BASED replyPtr (1) BYTE,
        (ciToken, coToken)          TOKEN,
        terminalData                STRUCTURE (
            numWords            WORD_16,
            numUsed             WORD_16,
            connectionFlags     WORD_16),
        (BytesRead, BytesWritten)   WORD_32,
        Status                      WORD_16;

/*  Create and open connections
    ---------------------------
*/
    ciToken = rqsattachfile (@(4,':CI:'), @Status);
    CALL rqsopen (ciToken, 1, 0, @Status);
    coToken = rqsattachfile (@(4,':CO:'), @Status);
    CALL rqsopen (coToken, 2, 0, @Status);

/*  Suppress echo for :CI: connection
    ---------------------------------
*/
    terminalData.numWords = 1;
    terminalData.numUsed = 1;
    CALL rqsspecial (ciToken, 4, @terminalData, NIL, @Status);
    terminalData.connectionFlags = terminalData.connectionFlags OR 4;
    CALL rqsspecial (ciToken, 5, @terminalData, NIL, @Status);

/*  Read the password and return
    ----------------------------
*/
    bytesWritten = rqswritemove (coToken, @('Enter password: '), 16,
    @Status);
    bytesRead = rqsreadmove (ciToken, @password(1), 16, @Status);
    password(0) = BYTE(bytesRead);
    RETURN;
  END getpassword;

/*  Execution Starts Here
    =====================
*/
DECLARE
```

Figure 8.4 (*Continued*)

```
        newline     LITERALLY    '0Dh,0Ah',
        Message (*)              BYTE
            INITIAL (19, newline, 'Your password is ***************'),
        Reply (81)               BYTE,
        Status                   WORD_16;

    CALL getpassword (@Reply);
    CALL movb (@Reply(1), @Message(20), Reply(0));  /* strcat in PLM */
    Message(0) = 19 + Reply(0);
    CALL rqcsendcoresponse (NIL, 0, @Message, @Status);
    CALL rqexitiojob (0, NIL, @Status);

END passwd;
```

Figure 8.5 C program illustrating the use of *rqsspecial()* to suppress character echoing for the standard C function, *gets()*.

```
/***>  passwd.c  <*****************************************************
 *
 *  C program to illustrate reading from stdin without echo
 *  The program prompts the user to enter a password, which is read from
 *  the standard input device without echoing the characters typed.
 *  The program determines the iRMX connection being used for stdin and
 *  modifies the character echo attribute of the connection.
 *
 ************************************************************************/
#include <stdio.h>
#include <rmxc.h>

#define getTerminalData 4
#define setTerminalData 5

/*  Prompt for password, and read it without echoing to screen
    ==========================================================
*/
void
getpasswd (char *reply) {
#pragma noalign (terminalAttributes)
struct        terminalAttributes {
  WORD          numWords;
  WORD          numUsed;
  WORD          connectionFlags;
  WORD          terminalFlags;
  } stdinAttributes;
TOKEN         stdinConnection;
WORD          Status;

    stdinConnection = _get_rmx_conn (fileno(stdin));
    stdinAttributes.numWords = 1;
    stdinAttributes.numUsed = 1;
    rqsspecial (stdinConnection, getTerminalData, &stdinAttributes,
NULL, &Status);
    stdinAttributes.connectionFlags |= 4;
    rqsspecial (stdinConnection, setTerminalData, &stdinAttributes,
NULL, &Status);
```

Figure 8.5 (*Continued*)

```
    printf ("Enter password: ");
    gets (reply);
    return;
    }

/*  main(): get password and display it.
    ====================================
*/
int
main (int argc, char *argv[]) {
char    password[16];

  getpasswd (password);
  printf ("\nYour password is %s\n", password);
  }
```

hard-coded[5]. Character echoing is suppressed by setting bit 2 of the `connectionFlags` for this connection to 1.

One final word. Note that the iRMX Terminal Support Code (TSC, a software module available to all terminal device drivers) allows terminal attributes to be set by embedding escape sequences in the character stream being written to or read from a terminal. For example, Appendix B shows how the TSC can be used to get the iRMX for Windows console driver to respond to ANSI X3.64 escape sequences, the same function provided by the ANSI.SYS device driver for DOS.

8.3.3 Set signal character

The last special function to be discussed is a function that allows an application to set up the equivalent of hot keys. The difference between these iRMX signal characters and true hot keys is that signal characters operate only in the context of a single application, whereas true hot keys, such as the <alt-SysRq> key used by iRMX for Windows, operate no matter which job is running.

The idea behind signal characters is very straightforward. A call to *rqsspecial()* or *rqaspecial()* with function code 6 is used to tell the terminal device driver which character code is to be treated as a signal character and to associate a semaphore with that character. From then until the connection is deleted (or the signal character is reset by another *special* call), the device driver examines each incoming character to determine if it is a signal character. If the character is a signal character, the driver sends a unit to the corresponding semaphore and discards the character. Otherwise, the

[5]The standard input, standard output, and standard error streams always correspond to file descriptors 0, 1, and 2 respectively.

driver simply passes the character on to the application. Up to 12 different signal characters can exist for a connection at one time.

A problem can arise when establishing signal characters for terminals connected to the system through buffered device controllers. In this situation, a user might type a signal character, but the device driver will not see it until the controller's buffer fills with other characters typed by the user after the signal character. This delayed effect of signal characters can be overcome for up to four special characters. The preceding terminal-attributes data structure includes an array (`specialChar[4]`) for characters that are to be forwarded to the device controller as soon as they are typed. (The `spcHiWaterMark` field can be used to cause a specific number of special characters, more than one, to be typed before being sent to the device driver.)

All HI command jobs have a signal character automatically set up for <^C>, along with a task that waits at the <^C> semaphore, ready to abort the job if the character is pressed. The HI layer of the OS provides a system call for changing the semaphore associated with this one signal character (*rqcsetcontrolc()*), but an application can also do so itself by calling *rq[as]special()*. The UDI layer also provides a system call, *dqtrapcc()*, that can be used to cause a user-written procedure to be called when <^C>is typed.

The signal characters, in general, must be control characters (<^A> through <^Z> have ASCII codes 0x01 through 0x1A), but <rub> (0x7F) as well as all characters with ASCII codes between 0x00 and 0x1F are valid. The device driver discards any characters that have been typed ahead (entered by the user but not yet read by an application) if 0x20 is added to the value of the character code given for this call.

The data structure pointed to by `parameterPtr` for setting a signal character is the token for the semaphore and the code for the character:

```
#pragma noalign (signalPair)
typedef struct signalPair {
   TOKEN                          Semaphore;
   BYTE                           Character;
   } SIGNALPAIR;
```

8.4 File System Structure and Management

This section introduces the data structures the BIOS maintains on a disk volume formatted with an iRMX Named file system. Note the distinction between the *Named file system*, which is the iRMX native-mode organization for supplying a tree-structured volume of named files and directories, and the generic uncapitalized term, *named file system*. The latter could be an iRMX, DOS, Unix, or VAX/VMS file system, all of which support tree-structured volumes with named files and directories. A Named volume

connected to the local computer is accessed using the Named file driver; a DOS volume connected to the local computer is accessed using the EDOS file driver. All four types of named volumes attached to remote computers can be accessed using the Remote file driver as long as OpenNet software is running on both the local and remote systems.

An HI command called *diskverify* can be used to examine and modify the data structures described in this section. In addition to the documentation on the command-line options for *diskverify* in the first part of the *iRMX Command Reference* (volume 10 of the iRMX for Windows documentation set), an appendix in that manual explains how to use *diskverify* interactively to examine and modify the data structures that the iRMX BIOS maintains on a disk volume. Another appendix provides complete information about the data structures introduced in this section. *Diskverify* is to iRMX Named file systems as *Norton Utilities* or *PC Tools* are to DOS file systems. *Diskverify* does not work with DOS file systems, but the system calls mentioned in this section work with any named file system.

8.4.1 Files and directories

Each file or directory on an iRMX Named volume consists of a set of volume blocks on the disk. A volume block is simply the smallest amount of disk space that can be allocated to a file, which is always a multiple of the disk's sector size. Volume blocks are numbered sequentially from zero to one less than the maximum number of blocks on the disk, analogous to how tracks are numbered when they are formatted with *rq[as]special()*. Files can contain any type of information using any structure appropriate to the application; to the OS a file consists of an ordered sequence of bytes.

Certain special files exist on a Named volume, however. One-of-a-kind housekeeping files are created when the volume is formatted (see section 8.4.3), and a distinction exists between normal files and directories. Directories are stored the same as normal files on the disk, but the BIOS imposes restrictions on how their contents can be accessed, and enforces an internal structure on their contents. The distinction between normal files, sometimes called *data files*, and directory files is not always important, and the generic term *file* refers to both types of files. The structure imposed on the contents of a directory is a sequence of 16-byte entries. Each entry has the following format:

```
#pragma noalign (directoryEntry)
struct directoryEntry {
   WORD                          fnode;
   BYTE                          pathComponent[14];
   }
```

The bytes in `pathComponent` (the name of a file) are arbitrary characters, not necessarily printable, that can include spaces and punctuation marks

(except : or $ at the beginning or :, $, /, or ^ anywhere, including the beginning), as well as normal ASCII characters. Uppercase and lowercase letters are stored as specified in *rq[as]createfile( )*, but there is no case sensitivity for any system call that references files by name. No intrinsic significance is attributed to periods in a file name, although iRMX development tools impose a name-and-extension structure on file names by using periods. The `pathComponent` field is padded on the right with 0x00 for file names less than 14 characters long.

Directory entries do not contain any information about the file other than its name, not even an indication of whether the entry is for a data file or another type of directory. That information, along with just about everything else the BIOS knows about the file or directory, is kept in a data structure called an *fnode*. The `fnode` field of a directory entry is a 16-bit value that uniquely identifies the fnode for the entry.

Fnodes are discussed in the next section, but first, the system calls that can be used to create and access directories are presented. The following are the BIOS and EIOS calls to create a directory:

```
extern void
rqacreatedirectory (          TOKEN              user,
                              TOKEN              prefix,
                              STRING far *       subpathPtr,
                              BYTE               access,
                              TOKEN              responseMbx,
                              WORD far *         exceptPtr);
```

The parameters to this call are the same as those to *rqacreatefile( )*, with the omission of that call's `granularity`, `mustCreate`, and `size` parameters. Directories are always created with size of 0, granularity of 1 (they expand by one volume block each time they expand beyond their current allocation). Also, a file or directory cannot already exist with the same name as specified by this call's `prefix` and `subpathPtr`. The EIOS call to create a directory is *rqscreatedirectory( )*:

```
extern TOKEN
rqscreatedirectory (          STRING far *       pathPtr,
                              WORD far *         exceptPtr);
```

You can write to a directory only by creating a file in the directory using *rq[as]createfile( )* or *rq[as]createdirectory( )*. You cannot write directly to a directory. You can, however, examine the contents of a directory either by reading it as a normal data file (it must be opened for reading and share with all), or with the *rq[as]getdirectoryentry( )* system calls:

```
extern void
rqagetdirectoryentry (        TOKEN              connection,
                              WORD               entryNumber,
                              TOKEN              responseMbx,
                              WORD far *         exceptPtr);
```

connection is a token for a connection to the directory, normally obtained by *rq[as]attachfile()*. The connection does not need to be open. The entries in a directory are numbered sequentially, and entryNumber identifies which entry in the directory the caller is interested in. When an application wants to search a directory, it normally makes the call from within a loop, with the value of this parameter starting at 0 and incrementing by 1 each time through the loop. This call does not return an IORS to responseMbx; it returns a segment containing a data structure that looks deceptively similar to the actual data structure of a directory entry on the disk:

```
#pragma noalign (directoryEntryInfo)
struct directoryEntryInfo {
   WORD      Status;
   BYTE      Name[14];
   };
```

Two differences exist between this structure and the structure of the directory entries on the disk. The more obvious difference is that the first word of this structure is a condition code value, not the entry's fnode number. The other difference is that the Name field is padded on the right with blanks (0x20) in this structure, rather than with 0x00s as on the disk.

Status has three interesting values. As usual, a value of 0 indicates normal completion of the system call, with a valid file or directory name in Name. A value of 0x0024 (E_EMPTYENTRY) indicates that the directory entry specified by the entryNumber parameter once held a valid entry, but does not now. This situation is normal, and occurs whenever a file or directory is deleted from the system.

Rather than rearrange the entire directory file from which the deletion is being made, the BIOS simply changes the fnode field of the entry to 0. No data file or directory has an fnode number of 0, so this value always marks empty entries, which are reused the next time a new file or directory is created in this directory.

Changing the fnode field of the entry to 0 is like the 0xE5 at the beginning of a file name that DOS uses to mark deleted directory entries. The difference between the DOS and iRMX file systems is that iRMX cannot recover deleted files because the fnode number, rather than part of the file name, is lost. A program searching a directory simply increments the entryNumber and repeats the system call if this value is returned for Status.

The other interesting value that might be returned in the Status field is 0x0025 (E_DIREND) which, as its name implies, indicates that entryNumber is beyond the end of the directory.

```
extern void (
rqsgetdirectoryentry (          STRING far *          dirNamePtr,
                                WORD                  entryNumber,
```

```
                                STRING far *            entryNamePtr,
                                WORD far *              exceptPtr );
```

As usual, the EIOS version of the call is a bit simpler to use. In this case, the user supplies a buffer to hold the entry name, passing a pointer to this buffer as `entryNamePtr`. Otherwise, this version operates the same as the asynchronous version.

8.4.2 Fnodes

An iRMX file system uses a data structure called an *fnode* (file node) to hold the information it needs for each file on a volume. Fnodes are stored on the disk in a special file named `R?FNODE`. Don't bother to try to read (or write!) this file yourself; it can only be accessed by the BIOS. You can, however, see much of the information in an fnode by using the extended option on the *dir* command. From the listing that command produces, you can see that the system knows the name of the file (obtained from the directory entry for the file), your currently allowed access rights for the file (computed from your default user object and the accessor list for the file), and the following information taken from the fnode:

- Whether it is a file or directory.
- The size of the file in bytes.
- The number of volume blocks it occupies.
- The times at which the file was created, last modified, and last accessed.
- The complete accessor list for the file.

You might notice that the *dir* command takes considerably longer to execute, especially over a network, if you specify the long, short, or extended options rather than the fast (default) option. The reason for this difference in execution times is that the fast option lists just the names of the files in the directory, which can be obtained by simply reading the directory itself or calling *rq[as]getdirectoryentry( )*. Generating the other listings requires additional disk accesses to obtain the information contained in the fnode for each entry.

An application can call *rq[as]getfilestatus( )* to obtain a copy of most of the information about a file's fnode, along with some additional information maintained in memory by the Named file driver:

```
extern void
rqagetfilestatus (          TOKEN                       connection,
                            TOKEN                       responseMbx
                            WORD far *                  exceptPtr);

extern void
rqsgetfilestatus (          STRING far *                pathPtr,
                            SFILESTATUSSTRUCT far *     fileStatusPtr,
                            WORD far *                  exceptPtr);
```

These calls can be used to obtain information from the EDOS, Remote, and Physical file drivers as well as the Named file driver. In the case of the Physical file driver, only the first part of the following data structure is returned to the program.

```
#pragma noalign (sfilestatusstruct)
typedef struct sfilestatusstruct {
  WORD    deviceshare;
  WORD    numberconnections;
  WORD    numberreaders;
  WORD    numberwriters;
  BYTE    share;
  BYTE    namedfile;
  BYTE    devicename[14];
  WORD    filedrivers;
  BYTE    functions;
  BYTE    flags;
  WORD    devicegranularity;
  DWORD   devicesize;
  WORD    deviceconnections;
/*    The remainder of this structure is returned only for files accessed using
      the Named, EDOS, and Remote file drivers. */
  WORD    fileid;
/* fnode number */
  BYTE    filetype;
/ * See text */
  BYTE    filegranularity;
  WORD    ownerid;
  DWORD   creationtime;
  DWORD   accesstime;
  DWORD   modifytime;
  DWORD   filesize;
  DWORD   fileblocks;
  BYTE    volumename[6];
  WORD    volumegranularity;
  DWORD   volumesize;
  WORD    accessorcount;
  BYTE    owneraccess;
/* The following fields are returned only for the asynchronous version of the
     call, and are not declared in rmxc.h. */
  WORD    ownerid;
  BYTE    secondaccess;
  WORD    secondid;
  BYTE    thirdaccess;
  WORD    thirdid;
  BYTE    volumeFlags;
/* See text */
} SFILESTATUSSTRUCT;
```

The information in the second part of this data structure is largely what you find in the fnode of an iRMX named file, modified as necessary to accommodate the different information available to different file systems. For example, the DOS file system has no concept of different users, so the EDOS file driver makes the owner of all DOS files the iRMX World user. Furthermore, DOS files are either read-only or totally accessible, so the EDOS file driver does not differentiate among delete, append, and update access rights. (If you have one right, you get all three.)

Clearly, some of the information in the second part of this structure applies to the disk volume containing the file and not to the file itself (volume name, granularity, size, and Flags). The name of the volume is similar to the label of a DOS disk, but the EDOS file driver can fit only the first 6 of an 11-character DOS label into the `volumename` field. Volume granularity was mentioned earlier in defining a volume block, and volume size is the unformatted storage capacity of the volume. The `volumeFlags` field, which is not available for iRMX I disks, tells whether the volume was properly shut down the last time it was used (value 0) or not (value 1). The `filetype` field tells whether the file is a directory (value 6), a data file (value 8), or one of the housekeeping files for the volume, which are described next.

8.4.3 Housekeeping files

Formatting an iRMX Named volume consists of performing a low-level format operation on every track of the volume, followed by writing housekeeping information onto the disk in the form of a set of files. Although these housekeeping files might appear to be normal disk files in the sense that their names are visible in the root directory of the disk, these files can be read and written only by low-level routines within the BIOS. Each of these files has a unique file type, which is the same as the number of the fnode used for the file itself. This section describes each of these files in file-type order.

File type 0. The file with fnode number 0 is the file that contains the fnodes themselves. When the volume is formatted, the *format* command takes a command line argument, `files = n`, to determine how many fnodes the fnode file is to hold. If the argument is not given, *format* uses a default value for n based on the capacity of the volume. Each fnode occupies a fixed number of bytes in the fnode file, so the file can be viewed as an array of fnode structures. The fnode numbers are used as indices into this array. To make accesses to this important file as efficient as possible, the file is created as contiguous, as are the other housekeeping files. Unlike normal files, the fnode file can never expand after it has been created. Thus, the file can never become noncontiguous, but the maximum number of files and directories on an iRMX volume is fixed at the time the volume is formatted, and can never be changed. It is perfectly possible to use up all the fnodes on a volume without exhausting the supply of disk blocks that can be allocated to files.

To understand fnodes better, the following is a display of fnode number 0 (the fnode for the fnode file) of a 1.2 MB diskette, as generated by the *diskverify* utility.

```
Fnode number = 0
                    flags : 0005 => short file
                     type : 00 => fnode file
```

```
      file gran/vol gran : 01
                   owner : 0000
create,access,mod times : 1A791F88, 1A791F88, 1A791F88
 total size,total blocks : 000048C6, 00000025
       block pointer (1) : 0025, 00049F
       block pointer (2) : 0000, 000000
       block pointer (3) : 0000, 000000
       block pointer (4) : 0000, 000000
       block pointer (5) : 0000, 000000
       block pointer (6) : 0000, 000000
       block pointer (7) : 0000, 000000
       block pointer (8) : 0000, 000000
              this size : 00004A00
               id count : 0001
           accessor (1) : 00, 0000
           accessor (2) : 00, 0000
           accessor (3) : 00, 0000
        parent, checksum : 0006, 0000
                aux (*) : 000000
```

The fields in this structure, from top to bottom, are:

`flags`. The bits in this field describe the status of the fnode itself as well as the file it describes. Bit position 0 tells whether this fnode is actually in use or not. (Remember, the entire fnode file is created when the disk is formatted.) When a file is added to the volume, an unused fnode must be allocated for it from the pool of unused fnodes in the fnode file. Bit 0 tells if the fnode is available or not. Bit 1 tells if the file is long or short. (See the description of the block pointers that follows for more information.) Bit 2 is always set to one, bit 5 identifies if the file has been modified or not, and bit 6 tells if the file is marked for deletion. All other bits are zero. The flags for this fnode have bits 0 and 2 on (0x0005), so you can conclude that this fnode is allocated for a short file not marked for deletion that has not been modified.

`type`. This byte will have a value of 6 for a directory, 8 for a normal file, and a value equal to the fnode of the file for housekeeping files. Fnode number 0 is the fnode in the example, which is the fnode for the fnode file itself, so its type is 0.

`file gran/vol gran`. This field is the file granularity specified when the file was created. The notation here is a reminder that this value is specified as a multiple of the volume granularity.

`owner`. This field is the ID of the owner of the file, which in this case is the Super user. You might think that the Super user could do something with the fnode file, but it's not true. The normal rules for file access do not apply to housekeeping files.

`create, access, mod times`. These three 32-bit words give the number of seconds since January 1, 1978, when the file was created, last accessed,

and last modified. All three of these files are the same for the fnode file, because you cannot access or modify this file in the normal sense of the terms.

`total size, total blocks`. Total size is the size of the file in bytes, and total blocks is the number of volume blocks allocated to the file. For the fnode file, these values never change. (See `this size`.)

`block pointer (1)` **through** `block pointer (8)`. These eight fields tell where the file is actually stored on the disk. A file can occupy up to eight extents on a disk volume. The blocks within an extent are contiguous, but the extents may be noncontiguous with respect to one another. Block pointer number 1 shows that this fnode file occupies 0x25 blocks, starting at volume block 0x49F (i.e., volume blocks 0x49F through 0x4C3). Because the fnode file is always contiguous, it never occupies more than one extent, so block pointers 2 through 8 are always unused for the fnode file. If a file needs more than eight extents, the file becomes a long file by treating each of the block pointers in the fnode as indirect references to data blocks that contain the actual block pointers for the file.

This change from short to long happens automatically when the need arises, and is reflected in bit 1 of the flags word for the fnode described earlier. Long files can become extremely large (each indirect block can hold 100 to 200 block pointers, depending on the volume granularity, and there can be 64K block pointers per extent of a long file), but performance degrades sharply as extra disk accesses are required just to locate parts of the file on the disk.

Long and short do not necessarily indicate relative sizes of disk files. A long file could consist of as few as nine volume blocks if they are all noncontiguous, and a short file could consist of 512K volume blocks made up of eight extents of 64K blocks each. At 1,024 bytes per volume block, that is a half-gigabyte short file and a 9,216 byte long file. The terms *short* and *long* really refer to the degree of fragmentation of the file, rather than its size.

`this size`. `this size` is obtained by multiplying the number of blocks allocated to the file by the volume granularity. The difference between `this size` and `total size` represents space that has been allocated to the file but is not in use. A few more fnodes could have been allocated for this diskette when initially formatted without making the fnode file any larger.

`id count` **and** `accessor (1)` **through** `accessor (3)`. These fields are the accessor list for the file. The `id count` tells how many of the three entries are actually used, with the first entry always used for the file's owner. Each entry consists of a byte specifying the access rights to the file (only the low-order four bits of the byte are used) and the user's access rights. For

this file, there is only one accessor, the Super user (an ID of 0), who has no access rights to the file.

parent, checksum. The `parent` is the fnode of the directory that contains this file. As you will soon see, fnode 6 is always used for the root directory of a volume. The fnode file does not actually appear in any directory, but if it did, it would be in the root directory. Remember, fnode number 0 in a directory entry means that the entry is empty. The `checksum` is a checksum on the fnode itself, used to help detect file system corruption. Apparently, it is not computed for the fnode file, but is updated for other fnodes every time they are modified.

aux(*). These three bytes are called the extension data for the fnode, and can be used for any purpose. The format command adds a three-byte extension to every fnode by default (making the total fnode size 88 bytes), but the number of bytes can be any number from 3 to 255. Anyone can examine or modify these bytes for a file's fnode by using the *rqagetextensiondata( )* and *rqasetextensiondata( )* system calls. The idea is that you could add your own layer to the iRMX operating system (perhaps replacing the HI or EIOS layer, for example), and use this field to store whatever information you want with each file of your customized operating system.

File type 1. The file with fnode 1 is the volume free-space map and appears in the root directory with the name `R?SPACEMAP`. Remember, you must use the invisible option on the *dir* command to see files that have names beginning with `R?` or `r?`. This file is owned by the World user, who has read access to it. The contents of the file is a bitmap, with each bit representing one volume block on the disk. A bit equal to 1 means the corresponding block is free to be allocated to a file; a bit equal to 0 means the corresponding block has been allocated.

File type 2. The file with fnode 2 is the free fnode bitmap file. If a bit in this map is 1, the corresponding fnode is free to be used for a new file or directory. If a bit is 0, the corresponding fnode is already in use. The redundancy between the bits in this file and bit number 0 in each fnode's `flags` field provides a basis for one of the consistency checks that *diskverify* can do for a disk volume. This file is in the root directory of the volume with the name `R?FNODEMAP` and can be read by anyone.

File type 3. fnode number 3 is used for an empty file that is called the Accounting File. The file is not presently used, cannot be accessed by any users, and does not appear in any directory. An *accounting* command can be used to cause the HI to keep a record of every login to an iRMX system, but that command does not use this file.

File type 4. fnode number 4 is used for the volume's bad-blocks bitmap. A 1 bit means that the corresponding volume block has a defect and cannot be used. This bitmap is initialized by *format* from the defect list supplied by the drive manufacturer if possible. The defect list can be supplied to *format* from a file if desired, and the map can be updated using *diskverify* interactively if new bad blocks are encountered as the file system is used. This file can be read by anyone; its name is R?BADBLOCKMAP in the root directory.

A distinction exists between volume blocks and physical blocks. Most controllers can make a disk appear to be perfect by using spare tracks as replacements for physically bad blocks. Physically bad blocks do not show up in the bad blocks bitmap unless the controller is unable to substitute a spare or the user forces blocks to be marked bad using *diskverify*.

File type 5. fnode 5 is used for the volume-label file, which always occupies the first 3,328 bytes of any iRMX Named volume. It can be read by anyone, using the file R?VOLUMELABEL in the root directory. The volume label consist of six parts:

- Space reserved for a bootstrap loader.
- An iRMX volume label.
- A bootloader location table.
- An ISO volume label.
- Reserved space for future ISO standardization.
- More reserved space for bootstrap loaders.

ISO standardization provides the potential for automatic recognition of floppy disk characteristics despite differences in their physical and logical structure on the rest of the disk. The iRMX volume label holds such information as the following:

- Volume name.
- Volume granularity.
- Where the fnode file starts on the disk (fnode 0 tells where the fnode file is too, but that does not help if you cannot find the fnode file to read fnode 0 from it).
- How large each fnode is (which depends on the size of the extension data field in each fnode).
- The volume flags byte, which tells whether the volume was shut down properly or not.

The bootstrap loader part of the volume label contains the code executed to start the bootstrap-loading process for the operating system. The boot-

loader location table is specifically used by Multibus II systems to find the special bootstrap-loader code used with that platform.

File type 6. All directories on the disk have a type field of 6 in their fnodes. The root directory for the volume actually uses fnode number 6. This directory is created by the *format* command and initially contains entries for some of the other housekeeping files. Except for being set up by *format* rather than by a call to *rq[as]createdirectory()* and the fact that the directory cannot be deleted, this directory is just like any other directory on the volume. The owner of the root directory is the user who formatted the volume, not necessarily the super user.

File type 8. All other files on the disk have a file type of 8, meaning a normal file (not a directory). Two housekeeping files might also be written by *format* that are created as type 8 files. If they are present, they use fnode numbers 7 and 8. One file is the second stage of the Multibus II bootrap loader, which appears in the root directory as R?SECONDSTAGE. The other file is a copy of the fnode file, which appears in the root directory as R?SAVE. The file is created if you specify the reserve option on the *format* command, and is updated from the fnode file if you specify the backup option on the *shutdown* command. If the master fnode file is damaged, you can replace it with the copy in R?SAVE by using the *diskverify* command interactively and giving its *backupodes* subcommand.

The data recovery mechanisms available for iRMX Named volumes are not as slick as the utilities available for DOS disks, but they are very powerful and easy to use once you understand the structure of a disk. Experimenting with *diskverify* and a spare floppy disk can be good preparation for dealing with a hard disk disaster.

Applications that demand highly reliable disk systems should consider using a technique called *disk mirroring*, which operates two disk drives in parallel for automatic redundancy and recovery in case one drive fails. The HI *mirror* command is not currently available for iRMX for Windows, but it is available for iRMX III systems, which suggests that it will become available for iRMX for Windows also.

8.5 Time-of-Day Management

The BIOS layer of iRMX supplies routines to track the date and time. Time-of-day management refers to maintaining a record of the current time and date with one-second granularity. When an iRMX system initializes, the BIOS obtains the current time and date from either a battery-backed clock, if one is available, or from the time at which the system disk (the one the OS was bootstrap loaded from) was last shut down. This time and date value is converted into a count of the number of seconds that have elapsed since midnight, January 1, 1978. A BIOS task repeatedly sleeps for

one second, adds one to the current time counter, and goes back to sleep again. At any time, an application can determine the current value of this counter or set it to a new value using the system calls *rqgettime( )* and *rqsettime( )*. The HI commands *time* and *date* allow users to look at or change the setting of the counter using meaningful character strings to represent times and dates.

The time maintained by the BIOS can easily be inaccurate, either because the initial setting of the time and date when the system initializes was inaccurate or because the BIOS task that updates the time and date at one-second intervals might be preempted by other tasks during, for example, periods of intense I/O activity. If inaccuracy is a problem, a system could introduce a task that periodically obtains the correct time and date from an external source (or from an on-board battery-backed clock) and updates the time and date counter from that source.

This problem of time-of-day drift raises the matter of real-time accuracy of iRMX, but that is a totally separate issue. The Nucleus manages the real-time clock interrupts that occur at 0.01-second intervals on most iRMX systems. iRMX for Windows reprograms the host system's Programmable Interrupt Timer to generate interrupts at this higher rate, and then emulates the lower clock rate normally generated on AT platforms for DOS. These interrupts are normally connected to the highest-priority interrupt request lines of the system's master Programmable Interrupt Controller (See chapter 5).

C programmers might be aware that the ANSI standard specifies a different starting date for the time and date functions in the standard C runtime library, namely midnight January 1, 1970. This difference in starting time (called the *epoch* in ANSI parlance) between the ANSI standard and the iRMX value is handled automatically by the Intel version of the C runtime library. The following are the function prototypes for the two BIOS calls to manage the time of day.

```
extern WORD
rqgettime (       WORD *      excer);


extern void
rqsettime (       DWORD       dateTime,
            WORD *excer);
```

The DWORDs in these two function prototypes are declared as `time_t` in `:INCLUDE:rmxc.h`, which is a `typedef` for an unsigned long (the same as a DWORD) in the `time.h` header file that `rmxc.h` includes. If a battery-backed clock is available on the system, its time and date can be set or examined using the following two system calls.

```
extern SETTIMESTRUCT
rqgetglobaltime (   WORD *                          exceptpr);
```

```
extern void
rqsetglobaltime (   SETTIMESTRUCT *          dateTimeptr,
                    WORD *                   exceptPr);
```

The typedef for SETTIMESTRUCT is:

```
#pragma noalign (settimestruct)
typedef struct settimestruct {
   BYTE   seconds;
   BYTE   minutes;
   BYTE   hours;
   BYTE   days;
   BYTE   months;
   WORD   years;
   } SETTIMESTRUCT;
```

8.6 Logical Name Reprise

Now that you have seen how an iRMX volume is organized on the disk, consider the operations to read from a file on a Named volume. You will see some of the logic that must be performed by the file driver along the way, but the real idea here is to look at the disk accesses involved in doing read operations. You will see how logical names for directories, or even files, can impact real-time perfoance by encapsulating as many time-consuming disk accesses as possible in the initialization phase of an application.

To make things concrete, assume that a floppy disk has the file structure shown in Figure 2.1, and that an application is going to read from `filel` on the diskette. The complete pathname would be `/dl/d2/filel`. For this example, assume the diskette is a 3.5″ floppy mounted in the `A:` drive of an AT platform system running iRMX for Windows. The logic for the example would be the same for a hard disk, except for the issue of the partition table on the hard drive of an AT platforms that would have to be negotiated to get to the iRMX file system. Here is the scenario:

1. The user attaches the device using the command
   `iRMX>attachdevice amh as a named` [1]
2. The user makes `d2` the current directory by using the command
   `iRMX>attachfile :a:dl/d2 as $` [2]
3. The application starts running and makes the following system calls:

```
fileConn = rqsattachfile ( ''filel'', &Status);
rqsopen (fileConn, 1, 0, &Status);
for (;;) {
rqsreadmove (fileConn, buffer, size (buffer), &Status);
. . .
}
```

The following is a list of the number of the disk accesses involved here, and when they occur relative to the loop, which you can assume is where the bulk of the work done by the application occurs. When the device is attached (command [1]), the Named file driver must invoke the following disk accesses:

1. Read the volume label to find the location of the fnode file and the size of the fnodes.
2. Read fnode 0 from the fnode file to find its extent on the disk.

At this point the connection object for the device can be built and its token cataloged in the root job's object directory using the logical name `:A:`. When the *attachfile* command is run (command [2]), the following additional disk accesses must take place:

3. Read fnode 6, the fnode for the root directory, from the fnode file. This fnode can be kept in memory, and the `:A:` connection object is updated to incorporate the information from the fnode. (An fnode memory pointer appears in the debugger's *vt* display for the connection.)
4. Use the block pointers in the root directory's fnode to find the first disk block that contains directory entries of the root directory and search for the first element of the pathname, `d1`. Since the sample file system has only two entries in the root directory (`d1` and `d3`) plus entries for four or five housekeeping files, only one disk block must be read. For a very large directory, several additional disk accesses would be required to read in additional blocks of the directory and search them.
5. From the directory entry for `d1`, determine its fnode number. Compute the offset of the fnode into the fnode file (fnode number multiplied by fnode size), call *rqaseek()* to that position in the fnode file (no disk access required), and call *rqaread()* to read the fnode for `d1`. If the fnode crosses a disk block boundary, two disk accesses are needed to read in the fnode. People who worry about this extra disk access sometimes format their disks with additional extension data to make the size of fnodes a multiple or divisor of the volume's block size. (Make the extension size 43 instead of the default of three, and each fnode will occupy 128 bytes).
6. Read directory entry blocks from `d1` until an entry for `d2` is found. For the sample file system, the entry for `d2` will be found in the first block of the directory.
7. Read the fnode for `d2` into memory and create a new connection object that includes a pointer to this fnode as part of its internal structure. The copy of the fnode for `d1` in memory can be discarded at this time. The new connection object is cataloged in the global job's object directory using the logical name `:$:`, and becomes the default prefix for subsequent calls to *rqsattachfile()*.

Now the application program starts running and makes its call to *rqsattachfile()*. The EIOS recognizes from the syntax of the pathname (`file1` does not start with /, ^, or :) that it is to use the default prefix to locate the file.

8. The fnode for the default prefix tells what disk blocks to read in the search for a directory entry named `file1`. Again, the sample file system will require reading just one block to find the entry, but a larger directory could require additional accesses at this point.
9. Read the fnode for `file1` into memory, and create a new connection object that includes a pointer to the fnode.

   The call to *rqsopen()* requires no disk accesses. The accessor list from the fnode is already in memory and the default user object was cataloged in the application job's object directory when the job was created, so access rights and sharing can be validated using information already available without going to the disk. If the application had specified EIOS buffering, reading the first blocks of data from the file would be initiated at this time, but the example specified zero EIOS buffers.

   Finally, the program enters its main processing loop.
10. Each call to *rqsreadmove()*, depending on the number of bytes being read and their position in the file, might require a disk access. Even without EIOS buffering, the BIOS must read an entire block into memory at a time, so it is possible that read requests will be satisfied from data already in memory. On the other hand, a single call to *rqsread()* can request bytes from the file not aligned with disk blocks, requiring multiple disk accesses so that the BIOS can perform de-blocking at either end of the user's buffer. Our tenth disk access simply represents a typical case.

Consider now the following two alternative scenarios, which summarize how logical names can be used to control when disk accesses occur.

The first alternative is if the application had used a full pathname, `:A:dl/d2/file1`. In this case, the EIOS would use the token for `:A:` cataloged in the root job's object directory as the prefix, and `dl/d2/file1` as the subpath. The BIOS would have to repeat disk accesses 4 through 7 to get ready to search for `file1` Disk access 3 would still have been done for the *attachfile()*, so the fnode for the root directory would already be available.) The application would have incurred four additional disk accesses during its initialization phase.

For the second alternative, assume that the user had given the following command before entering the program, and used the pathname `:x:` in the call to *rqsattachfile()*:

```
iRMX>attachfile :a:dl/d2/file1 as x                    [3]
```

In this case, the connection to the file would have been established before the application started running, with the logical name `:x:` representing the token for this connection object cataloged in the application's global job. In this case, all of disk accesses 1 through 9 would have occurred before the application started running, and the only accesses the application would encounter would be those involved in actually reading from the file.

Chapter

9

Extending iRMX: Adding Device Drivers

9.1 Overview

Although iRMX is a proprietary operating system with regard to the architecture of the processor on which it runs, it has always been very much an open system with regard to its support for peripheral devices. Intel's operating system has always supported the hardware devices that Intel manufactures, of course, and it has also always supported commonly used devices not manufactured by Intel. But what is truly open about iRMX is the provision it makes for anyone to add their own support for nonstandard devices, and to incorporate that user-written code into the operating system on an equal footing with the code supplied by Intel.

This chapter looks at the structure of iRMX device drivers. Device drivers are especially interesting because they must interface with all of the following:

- The hardware of a device controller attached to the computer's system bus.
- The microprocessor's interrupt logic.
- The software of the operating system.
- The code of the application program that invokes I/O operations through system calls (indirectly).

It is not surprising, perhaps, that this chapter builds on concepts introduced in chapters 5 through 8. The primary source of information for this chapter is the *iRMX Device Driver Programming Concepts* manual, volume 7 of the iRMX for Windows documentation set. The manual includes how to use the device drivers supplied with the operating system, as well as how to write new device drivers. Only the latter topic is covered in this chapter.

9.2 I/O Terminology

A mini-glossary is necessary before starting to discuss device driver concepts. The following three terms are used when speaking about the iRMX I/O system:

device unit The iRMX name for an actual I/O device, such as a terminal, printer, or disk drive.

device controller The hardware interface that connects a device unit to the microprocessor's I/O bus. (Device controllers on PCs are called *adapters*.) More than one device unit can be connected to a single device controller. For example, a single device controller for disks might have several drives attached to it.

device driver The software that actually interacts with a device controller. Device drivers are considered part of the BIOS itself. A single device driver would be used for all the device units attached to one device controller, and it is even possible for one device driver to manage several different device controllers at the same time.

Readers familiar with device drivers for DOS will find that the iRMX use of the term *device driver* is much more restricted than DOS's. An iRMX device driver interacts directly with the iRMX BIOS using well-defined interfaces described in this chapter. Application tasks can interact with an iRMX device driver only through the standard I/O system calls described in chapter 8. DOS uses the term much more loosely to describe almost any program that remains resident in memory while other applications run. Unlike iRMX device drivers, DOS device drivers often provide their own system calls that applications can call. In this regard, DOS device drivers are more akin to iRMX operating system extensions, described in chapter 10.

Four types of device drivers are supported by the BIOS. The simplest type to understand is called a *custom driver*, used in this chapter in the presentation of device driver concepts. The other three types of device drivers are actually custom drivers for which iRMX provides boilerplate code to permit developers to concentrate on the special characteristics of their own device controllers without writing a considerable amount of standard code over again. After the structure of a custom driver is discussed, the features of the other three types are examined as well. The names for these other three are *common*, *random*, and *terminal*. These names are quite descriptive: common drivers are used for relatively simple devices, such as a printer or tape drive; random drivers are used for random-access devices such as disks; and terminal drivers are used for terminals.

9.3 Logical Structure of a Device Driver

A device driver consists of three major components, which are related to each other as shown in Figure 9.1. Figure 9.1 also shows the relationships between the device driver components, a device controller, the BIOS, and an application task. The system bus in Figure 9.1 divides the hardware and software sides of the I/O system.

The module labeled Interface with the Device Controller is responsible for managing most of the device driver hardware interactions, which are interrupt handling and the actual transfer of data between the device controller and the processor's memory. The Interface with BIOS module consists of a set of procedures called by one of the file drivers within the BIOS when it needs to interact with the device driver. The driver task module is the code that acts as the intermediary between the other two modules. It receives IORSs from the BIOS, processes the functions requested in the IORSs by preparing data operations for the hardware interface, and returns each IORS to its `responseMbx` when the function is complete.

9.3.1 Interface with the device controller

Chapter 5 introduced the hardware side of the x86 microprocessor's I/O interface. To review, data, control, and status information are written to and read from a device controller by writing and reading I/O port addresses in much the same way as memory locations are written and read using normal memory addresses. The device controller signals the completion of data transfers between itself and a device unit by generating an Interrupt Request (INTR) signal to the microprocessor through a Programmable Interrupt Controller (PIC) that manages simultaneous interrupt requests from multiple device controllers. When the microprocessor acknowledges an INTR, the PIC presents it with an 8-bit interrupt-level number that indicates the source of the interrupt request. The CPU uses the interrupt-level number to index into a table of descriptors for software routines called *interrupt handlers* (or *interrupt service routines* in DOS), and forces the equivalent of a *call* instruction to the appropriate handler. When the handler starts executing, further interrupts are blocked from occurring in two ways.

First, the PIC will not generate any more interrupt requests until it receives a command byte, called an end-of-interrupt (EOI) command, from the processor. One of the responsibilities of an interrupt handler is to output the EOI byte to the PIC's command register either directly or by using a system call.

Second, the interrupt enable (IE) bit in the processor's flags register is turned off when the handler is called, preventing the processor from recognizing interrupt requests even if the `INTR` signal line is asserted by the PIC again. As the CPU forces the *call* to an interrupt handler, it pushes the cur-

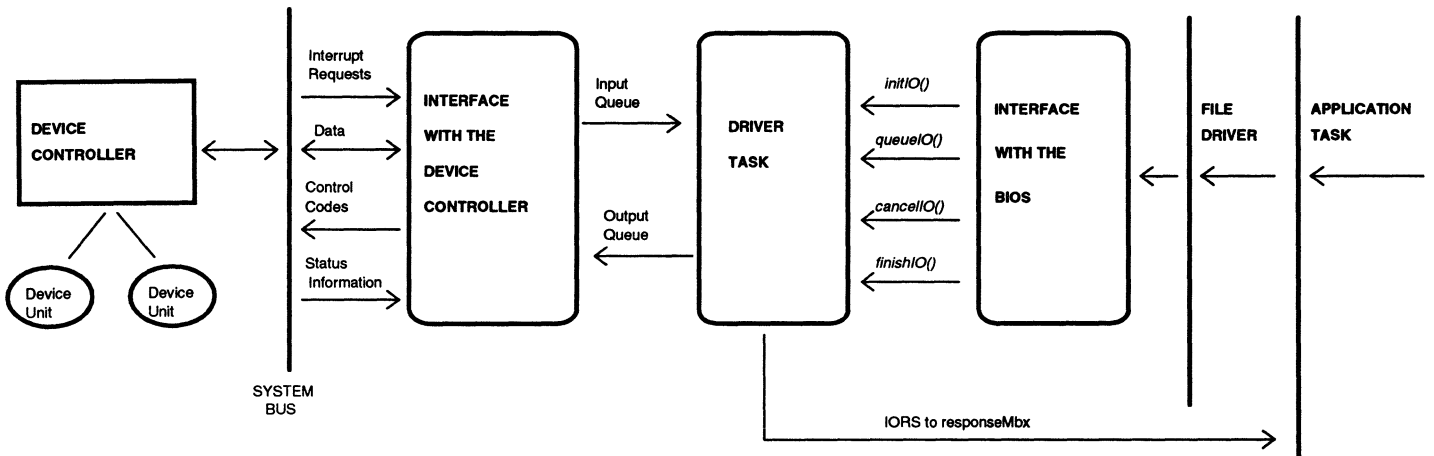


Figure 9.1 Three software units that constitute a custom device driver.

rent state of the flags register (which must have the IE bit on to recognize the interrupt) onto the stack, along with the return address for the interrupted task. The special return instruction executed by interrupt handlers (*iret*) pops the flags register and the return address when the handler exits, thus turning the IE bit back on.

iRMX places severe limits on the processing interrupt handlers can do because they execute in the context of whatever iRMX task happens to be running at the time of the interrupt. The interrupt handler performs a small amount of processing in response to the interrupt and calls the Nucleus to schedule execution for a task associated with the given interrupt level.

Each iRMX task priority level has a set of interrupt levels associated with it that are prevented from occurring when a task of that level is running. For tasks with priorities 129 through 255, the set of interrupt levels is the empty set, but for higher priority tasks, the Nucleus programs the PIC to prevent certain interrupt levels from occurring as part of the process of scheduling a task for execution. An interrupt task for a given interrupt level automatically has a priority assigned to it that prevents all interrupt levels lower than its own from interrupting the CPU while the task is running. The task priorities and associated set of disabled interrupt levels are given in a table in the manual *iRMX Nucleus Programming Concepts*, volume 3 of the iRMX for Windows documentation set.

Assuming a device controller that performs both input and output data transfers, our Interface with Device Controller module consists of two interrupt handlers and two interrupt tasks. The first priority is to see how to code an interrupt handler procedure, then how to create an interrupt task, and finally how to get the two to communicate with one another.

An interrupt handler is a normal procedure (a void function) declared as an interrupt handler when it is compiled. The following fragments show how to do this for a handler named *myhandler()* coded in either C:

```
#pragma interrupt (myhandler)
void myhandler( );
```

or PLM:

```
myhandler: PROCEDURE PUBLIC INTERRUPT;
```

The compilers generate special code for interrupt procedures. The prologue code for interrupt handlers includes machine instructions to save a copy of all the processor's registers on the stack. The epilogue code includes the instructions to restore the registers, and the procedure ends with the special *iret* instruction that restores the processor's flags register and return address from the stack. In short, the code is generated so that the procedure can be executed in the context of any thread of execution without interfering with the interrupted thread.

The next two steps are to enter the address of the interrupt handler into the proper slot of the processor's Interrupt Descriptor Table (IDT) and establish an interrupt task for the interrupt level. It is possible to have an interrupt handler without having an interrupt task, but assume the need for both for now. Both steps are accomplished with a single system call, *rqsetinterrupt( )*.

```
extern void
rqsetinterrupt (        WORD            level,
                        BYTE            interruptTaskFlag,
                        void far *      interruptHandler,
                        TOKEN           interruptHandlerDS,
                        WORD far *      exceptPtr);
```

`level` is an encoded value that identifies which interrupt level is being configured. Rather than enter a number between 0 and 255 that would serve as the index into the Interrupt Descriptor Table (IDT), iRMX works with encoded levels that identify interrupt sources by their connections to PICs. One PIC, called the *master*, is always connected to the microprocessor and up to seven slave PICs can then be connected to the master, as shown in Figure 5.6. The value of `level` takes the form 0x00MS, where M is the number of the interrupt request line for the master PIC that generated the request, and S is the interrupt request line on the slave PIC, which is 8 if the interrupt request line is connected directly to the master PIC. For example, a handler for interrupt requests from a device controller connected to interrupt request line 3 of the master PIC has a value of 0x0038 for `level`, and a handler for interrupt requests from a device controller connected to interrupt request line 4 of the slave PIC, which in turn is connected to interrupt request line 7 of the master PIC, has an encoded level of 0x0074.

Encoded values of `level` are mapped to two different numbers by the Nucleus. One mapping is to the 8-bit index into the IDT for the level. IDT slots 56 through 127 are used for interrupt handlers, with slots 56 through 63 used for master levels, and 64 through 127 used for slaves. This mapping is important to know about if you want to invoke an interrupt handler by means of a software interrupt instruction, such as when developing a device driver before an actual device controller is available.

The second mapping is to the priority of the interrupt task for the level mentioned previously. Priorities increase in a monotonic sequence from 4 to 128 as `level` varies from 0x00 to 0x77. iRMX application tasks always run with priorities numerically greater than 128 so that they do not interfere with the interrupt response mechanism of the system.

`interruptTaskFlag` indicates whether there is to be an interrupt task for the interrupt level or not. If the value is 0, there is to be no interrupt task, and all interrupt processing for the level must be done by the inter-

rupt handler. The focus here is on the situation that does involve an interrupt task.

If `interruptTaskFlag` is not zero, *the calling task immediately becomes the interrupt task for the level.* No separate system call creates an interrupt task, only this call, which transforms a normal task into an interrupt task and associates it with a particular level. If the calling task is to become an interrupt task, it immediately assumes the priority associated with `level`. Thus, the maximum priority for the job that owns the calling task must be high enough to accommodate this priority shift. This requirement can be a problem for loadable device drivers, and the solution to this problem is to use the *rqesetmaxpriority( )* system call to raise the maximum priority for the job. This parameter is more than just a Boolean flag. It also indicates how many times the interrupt handler can be activated without the interrupt task indicating readiness to process new interrupts. (See the next subsection on Handler-Task synchronization.) If the limit indicated by the value of `interruptTaskFlag` is reached, the interrupt level is automatically disabled.

`interruptHandler` is a pointer to the interrupt handler procedure. The procedure must have been coded as an interrupt procedure, cannot receive any arguments nor return any value, must handle exceptions in-line, and it must remain in memory as long as the interrupt level is set. Remaining in memory is of particular concern for loadable device drivers because there is the possibility that the job owning the handler's code and/or memory data segments might be deleted, and its memory segments returned to the Free Space Manager (FSM). This problem does not exist for device drivers loaded by *sysload* because they are owned by the Human Interface (HI) job, which will not terminate until the system is reset. The dynamic device driver mechanism described later in this chapter tries to minimize this concern, but cannot eliminate it entirely.

`interruptHandlerDS` is a selector for a memory segment to be used as the data segment when the interrupt handler executes. The interrupt handler can have this selector loaded into its `ds` register when it executes by calling *rqenterinterrupt( )*. This feature could be used if the handler and the task wanted to use the same data segment but normally run with different data segments. This situation would only occur if the handler and the task were compiled in separate modules using the large segmentation model. Since this condition is uncommon, this parameter is normally coded as a null selector, and the handler's prologue code loads the `ds` automatically.

Handler-task synchronization. The CPU disables all interrupts and generates a call to the interrupt handler each time it acknowledges an interrupt for the appropriate level. The PIC does not make any additional interrupt

requests to the processor until it receives an EOI command byte. When the handler is called, it performs whatever processing is appropriate (as little as none), and then makes the Nucleus call *rqsignalinterrupt()*.[1] The Nucleus does three things when *rqsignalinterrupt()* is called:

1. Sends the EOI byte to the PIC.
2. Moves the interrupt task for the level to the ready queue. By definition, the interrupt task has higher priority than the task running at the time of the interrupt and will preempt it.
3. As part of the scheduling process for the interrupt task, the Nucleus programs the PIC to ignore interrupt requests for lower interrupt levels, and turns on the CPU's interrupt enable bit.

Interrupt handlers that do not call *rqsignalinterrupt()* to signal an interrupt task must either call *rqexitinterrupt()* to get the EOI command sent, or output the EOI byte to the PIC's control port directly, which is more efficient than making the system call. By calling *rqsignalinterrupt()*, the handler is preempted and does not run again until the interrupt task relinquishes the CPU.[2] When the handler runs again, it executes its epilogue code, which restores all registers to the values they had when the handler was called and executes the special *iret* instruction that restores the CPU flags register from the stack. Interrupt handlers, then, are procedures that act like normal procedures in that they return after they are called. In contrast, the procedure executed by any task, including interrupt tasks, typically contains an endless loop that never returns.

The procedure executed by an interrupt task follows the normal model for event-processing tasks. After doing some initialization, including the call to *rqsetinterrupt()* that makes the task an interrupt task, the code consists of an endless loop that begins with a call to either *rqwaitinterrupt()* or *rqetimedinterrupt()*. These calls enable interrupts for the particular level if they are disabled, and cause the interrupt task to sleep until the interrupt handler calls *rqsignalinterrupt()*.[3] The difference between *rqwaitinterrupt()* and *rqetimedinterrupt()* is that *rqetimedinterrupt()* includes a standard iRMX time-limit parameter, a WORD that tells the maximum number of real-time clock ticks (normally in 0.01-second units) that the task is willing to wait for the handler to call *rqsignalinterrupt()*. Both calls include an encoded `level` parameter, which allows several interrupt tasks to use a single procedure for their code.

---

[1]The interrupt level is the only parameter for this call other than `exceptPtr`.

[2]Higher-priority interrupts can occur while the interrupt task is running but their nested processing is invisible to the process being described here.

[3]If a handler has called *rqsignalinterrupt()* more times than the value of interruptTaskFlag in the *rqsetinterrupt()* system call without the interrupt task calling *rqwaitinterrupt()* or *rqetimedinterrupt()*, the Nucleus programs the PIC to disable interrupts for that level.

Interrupt handlers run with all interrupts disabled because the processor's interrupt mechanism disables the interrupts automatically. Because interrupts are disabled when interrupt handlers execute, good system design dictates that they should perform their work as quickly as possible. Also, because the handler is running in the context of an arbitrary iRMX task, the Nucleus restricts interrupt handlers to making only those system calls directly concerning interrupt processing.

An interrupt task, on the other hand, runs with interrupts disabled only for lower priority levels (because the Nucleus programs the PIC to do this when it schedules the task) and operates in its own context. As a result, much less concern exists about reducing the computation time for interrupt tasks and no constraints exist on which system calls an interrupt task can make. If the response to an interrupt requires a significant amount of processing time, however, the interrupt task should pass control to an application task (one that runs at a priority that does not disable any interrupt levels) using a semaphore or mailbox. The application task can then complete its processing without interfering with the system's response to any interrupt levels.

9.3.2 Interface with the BIOS

The interface between a device driver and the BIOS consists of two data structures and four procedures that the device driver provides. The two data structures are the IORS, introduced in chapter 8, and the DUIB (device unit information block), mentioned briefly before and shown here:

```
#pragma noalign (duibStruct)
typedef struct duibStruct {
  BYTE          name[14];
  WORD          fileDrivers;
  BYTE          functions;
  BYTE          flags;
  WORD          deviceGranularity;
  DWORD         deviceSize;
  BYTE          device;
  BYTE          unit;
  WORD          deviceUnit;
  void          (near * initializeIO) ();
  void          (near * finishIO) ();
  void          (near * queueIO) ();
  void          (near * cancelIO) ();
  void far *    deviceInfoPtr;
  void far *    unitInfoPtr;
  WORD          updateTimeout;
  WORD          numBuffers;
  BYTE          priority;
  BYTE          fixedUpdate;
  BYTE          maxBuffers;
  BYTE          reserved;
}
```

Most DUIBs for an iRMX system are loaded into system memory at the time the OS is initialized, but the *rqinstallduibs( )* system call can be used to load additional DUIBs at run-time. The ability to add DUIBs to the system at run-time is a relatively recent addition to iRMX, and is particularly valuable for developing loadable device drivers, a technique described later in this chapter.

At this point, just a few fields in the DUIB data structure need to be mentioned. The `name` field is a blank-padded array of bytes containing the device name for the DUIB. When a task calls *rqphysicalattachdevice( )*, the BIOS searches through all the DUIBs in memory to find a match between this field and the string pointed to by the `devNamePtr` parameter of the call. There must be at least one blank at the end of this field, effectively limiting the name to 13 characters.

The `fileDrivers` field is a bit array that identifies which file drivers can be used when performing an attach-device operation with this DUIB (Physical, Named, Stream, Remote, or EDOS). `functions` is another bit array and identifies which functions the device driver will support for the device unit referenced by the DUIB. These functions are covered in the Driver Task section that follows. The first four of the pointers in the middle of the structure point to four procedures, described next, that the BIOS will call to communicate with the device driver. The other two pointers are to data structures used by the common, random, and terminal device drivers. Custom drivers can use these two pointers, for anything they like, or code them as null pointers.

Just four procedures serve as the interface between the BIOS and a device driver. In the case of common, random access, and terminal drivers, the BIOS provides other routines called by these four procedures, but for custom drivers, these routines are the complete procedural interface to a driver. Although the BIOS calls the procedures by referencing their pointers rather than their names, it is convenient to give generic names to the procedures, *initializeIO( )*, *finishIO( )*, *queueIO( )*, and *cancelIO( )*.

```
void
initializeIO (          DUIBSTRUCT far *        duibPtr,
                        TOKEN far *             deviceDataPtr,
                        WORD far *              exceptPtr);
```

As its name implies, *initializeIO( )* is called to initialize the device driver to work with a particular device controller. If a device controller is connected to more than one device unit, this function is called only if no other device units connected to the controller are already attached. The function is called when an application task calls *rqaphysicalattachdevice( )*, such as through an EIOS system call, but, like the other three procedures in this set, is actually called by a task that belongs to the BIOS job rather than directly by the application task. The implication of this indirect call is that the objects that this procedure creates belong to the BIOS job, not to the

application task that initiated the operation. Typically, this procedure creates a task for the device controller being attached (the driver task described in the next section) and sets up the queue for passing IORSs to the driver task.

The `duibPtr` parameter points to the DUIB that contains the pointer to this procedure. This pointer is passed to the driver's *initializeIO( )* procedure to accommodate the case of several DUIBs containing pointers to the same driver. A single, generalized device driver can serve a number of different device controllers and units by having the `deviceInfoPtr` and `unitInfoPtr` fields of the DUIBs point to data structures that describe the particular controller and unit to be used.

A device driver might want to have a memory segment to hold information about the device units connected to a particular controller. To do so, it can create such a segment when *initializeIO( )* is called, and place a token for the segment in the variable pointed to by `deviceDataPtr`. This token will then be passed back to the device driver as one of the parameters to each of the other three procedures that compose the driver's procedural interface to the BIOS (*queueIO( )*, *cancelIO( )*, and *finishIO( )*). This mechanism allows a single device driver's code to be used to service multiple device controllers. If a driver does not need such a data segment, it should place a null selector in the variable pointed to by this parameter.

Although this procedure is not a system call, the last parameter looks strikingly familiar: a pointer to a word containing a condition code. In this case, the BIOS task that calls this procedure supplies the pointer and the word to which it points. It is the responsibility of the *initializeIO( )* procedure to store an appropriate condition code value in the word, which is returned as the condition code for the application's call to *rqphysicalattachdevice( )*. It is important to set the value of this variable properly.

```
void
finishIO (   DUIBSTRUCT far *                duibPtr,
             TOKEN                           deviceDataTkn);
```

The *finishIO( )* procedure is called when a device unit is detached and no other device units are still attached for the device controller. Its job is to delete all the resources created when the device was first attached, such as the driver task, interrupt tasks, and IORS queue. If this housekeeping were not performed, repeatedly attaching and detaching a device would lead to unused objects accumulating in the system, thus resulting in wasted memory. This routine can also perform any final processing necessary to allow the device controller to be attached again at a later time. The two parameters are a pointer to the DUIB for the device unit being detached and a token for the data segment created for this device when *initializeIO( )* was called the first time the device was attached. Deleting this segment is one of the housekeeping operations this function performs.

```
void
queueIO (          TOKEN                    iorsTkn,
                   DUIBSTRUCT far *         duibPtr,
                   TOKEN                    deviceDataTkn);
```

The *queueIO()* procedure is the workhorse of the interface between the BIOS and a device driver. It is called for every I/O operation invoked by every application task that uses the driver. Its function is actually quite simple: it places the token for an IORS, its first parameter, onto the queue of IORSs maintained for work to be performed by the driver task. A device driver that supports multiple device controllers might have multiple driver tasks and IORS queues, in which case the `deviceDataTkn` is normally the selector for the memory segment holding the head of the IORS queue for all the device units attached to a particular controller. This segment is the one that would have been created when *initializeIO()* was called at the time the first device unit was attached. Each IORS includes fields to set up a doubly linked list of IORSs on a queue that can be used by *queueIO()*, but a rudimentary device driver could use as simple a mechanism as an object mailbox to implement its IORS queue. The only requirement is that this procedure act as a producer of IORSs that can be consumed by a driver task.

queueIO() performs the synchronous portion of a BIOS system call. Once the IORS has been queued, this procedure returns to the file driver that made the *queueIO()* call, which in turn allows the application task that made the system call to continue execution. The driver task then processes the IORS asynchronously, ultimately returning it to `responseMbx` when processing is complete.

```
void
cancelIO (   WORD                     cancelID,
             DUIBSTRUCT far *         duibPtr,
             TOKEN                    deviceDataTkn);
```

cancelIO() is a housekeeping procedure that the BIOS calls to remove IORSs from a queue. It undoes the effects of previous calls to *queueIO()*. Thus, this procedure removes those IORSs whose `cancelID` fields match the `cancelID` parameter of this call. This procedure is called, for example, when a connection is closed to ensure that no I/O operations are performed using a closed connection. Device drivers that process all IORSs sequentially can often skip the functionality of this procedure, at the risk of performing some I/O operations queued by a job that terminates abnormally. Note that proper implementation of this procedure precludes use of the simple mailbox mechanism for the IORS queue mentioned previously, since a mailbox queue of IORS tokens cannot be searched like a linked list of IORSs themselves.

9.3.3 The Driver Task

In our model of a device driver, the Driver Task is the heart of a device driver; witness its central position among the software units in Figure 9.1. The task is created by *initializeIO( )*, and begins its processing by completing any initialization not already performed. Typically, this initialization includes programming the device controller as necessary (to tell it to start generating interrupts and to set the baud rate, for example), and creating the interrupt tasks and handlers for the controller. The driver task then enters an endless loop in which it receives IORSs from its IORS queue and manages whatever work must be done to process each I/O request. This model is particularly convenient for drivers that must service separate interrupt levels for input and output with a device controller. This model is not the only one that will work, however. For example, the drivers supplied by Intel accomplish the work the example assigns to the driver task either in the BIOS task that calls *queueIO( )* or the driver's interrupt task. The following is the structure declaration for an IORS, previously given in chapter 8, for easy reference.

```
#pragma noalign (iorsStruct)
typedef struct iorsStruct {
  WORD                                  status;
  WORD                                  unitStatus;
  NATIVE_WORD                           actual;
#if _ARCHITECTURE_ < 386
  WORD                                  actualfill;
#endif
  WORD                                  device;
  BYTE                                  unit;
  BYTE                                  funct;
  WORD                                  subfunct;
  DWORD                                 deviceloc;
  BYTE far *                            buff;
  NATIVE_WORD                           count;
#if _ARCHITECTURE_ < 386
  WORD                                  countfill;
#endif
  void far *                            aux;
  iorsStruct far *                      linkForward;
  iorsStruct far *                      linkBackward;
  TOKEN                                 responseMbx;
  BYTE                                  done;
  BYTE                                  fill;
  TOKEN                                 cancelID;
  TOKEN                                 connection;
} IORSSTRUCT;
```

The `funct` field of an IORS is a function code. Values for `funct` symbolic names and a summary of driver task processing are:

0 `fRead` — The driver task reads the number of `count` bytes from the device controller into the application's buffer, pointed to by `buff`. If appropriate for the device, `deviceloc` tells from where on the device the data are to be read, such as a sector number on a disk.

1	`fWrite`	The driver writes the number of `count` bytes from the application's buffer, pointed to by `buff`, to the device controller. If appropriate for the device, `deviceloc` tells where on the device the data are to be written. (Reading and writing are described in more detail in the following section.)
2	`fSeek`	For disks, the device driver positions the read/write head where specified by the `deviceloc` field.
3	`fSpecial`	This function is used when a task calls *rq[as]special()*. The `subfunct` field of the IORS contains the value of the `functionCode` parameter to those calls, so the device driver will know whether to format a track, set a signal character, or whatever device-specific operation is needed by the application. The `aux` pointer of the IORS is the same as the `parameterPtr` argument of the call to *rq[as]special()*. That is, it points to whatever data structure is appropriate for the particular operation to be performed. For example, for the signal character function described in chapter 8, `aux` points to a structure that contains a byte code and a semaphore token.
4	`fAttachDevice`	The device driver performs whatever initialization necessary for attaching a particular device unit. *initializeIO()* is called when a device is attached only if no other device units for the device are already attached. An `fAttachDevice` IORS, however, is sent to *queueIO()* for every device unit attached. A device driver that supports multiple device units connected to a single controller might be structured with a main driver task, created by *initializeIO()*, which receives all IORSs. When this main driver task receives an `fAttachDevice` IORS, it could create a new driver task for the device unit being attached and forward all future IORSs for that device unit to its proper task, based on the `device` and `unit` fields of the IORSs (which match the corresponding fields of the DUIB used to attach the device unit).
5	`fDetachDevice`	An `fDetachDevice` IORS is sent by the BIOS for each device unit to be detached. The device driver deletes whatever resources were allocated to the device unit and does whatever is necessary to shut the unit down so that it can be attached again at a later time. If no other device units for this device controller are attached at the time, the BIOS calls the driver's

finishIO() routine immediately after queuing this IORS.

6	`fOpen`	Most of the logic associated with opening a connection to a file is performed by the file driver in the BIOS, including checks for access rights, sharing mode consistency, and connection data structure updating to show that the connection is indeed open. An `fOpen` IORS notifies a device driver that a connection is open. A driver for a local device normally does very little, if any, processing for this function, but it is very important for network drivers, which use this function to trigger checking of remote access rights.
7	`fClose`	Like `fOpen`, this function is mostly a courtesy call for drivers, managing local devices, but is used by network drivers to get the remote server to close its connection to the file.

After processing an IORS, the driver task must update the `status` and `unitStatus` fields of the IORS to indicate the result of the operation. (At this point the meaning of IORS shifts from I/O Request Segment to I/O Result Segment.) A value of 0 for `status` means the operation completed normally, and a value of 0x2B (E_IO) signifies a typical I/O error. If the driver sets status to E_IO, it should normally set `unitStatus` to one of the standard values listed:

0	`ioUnclass`	An error other than one of the codes listed here occurred.
1	`ioSoft`	An error occurred that might correct itself by simply retrying the operation.
2	`ioHard`	Unrecoverable I/O error. Retry is useless.
3	`ioOprint`	Operator intervention required. (No paper in the printer, etc.)
4	`ioWrprot`	Cannot write to a write-protected volume.
5	`ioNoData`	For magnetic tape, no data exists in the next tape record.
6	`ioMode`	A read or write was attempted before the previous read or write for the same device unit completed. (This condition can occur for tape drive operations.)
7	`ioNoSpares`	A disk needs an alternate track or sector to replace a defective one, but no spares are available to make the assignment.
8	`ioAlt-Assigned`	An alternate track or sector had to be assigned.

In addition to setting the `status` and `unitStatus` fields, the driver must also fill in the `actual` field to indicate the actual number of bytes transferred for `fRead` and `fWrite` functions. The value is normally equal to the `count` field, unless an event occurs, such as a disk filling up on output or an end-of-file being encountered on input.

Finally, the device driver sends the token for the IORS to the mailbox specified in the `responseMbx` field to indicate it has completed the operation.

9.3.4 Driver task and interrupt task interactions

When a device driver receives an IORS for a data transfer, it follows one of several models described as follows for synchronizing device controller operations with the CPU.

Polling. The driver task repeatedly reads the device controller's status register to determine when an I/O transfer is complete. This technique does not use the interrupt mechanism of the processor at all, and is seldom used by iRMX device drivers. If the task that polls the controller has a very high priority, it can starve other time-critical tasks in the system. If the driver task has a low priority, there is little or no advantage over using the interrupt system.

No interrupts. This technique is used when the driver task calls another procedure that handles the interrupt(s) for the data transfer. A primary example of this technique is used by some iRMX for Windows drivers running on an AT platform that make ROM-BIOS procedure calls to perform data transfers. The driver task calls the ROM-BIOS procedure, and does not run again until that procedure completes the I/O transfer, including all interrupt processing associated with the transfer. Since the driver polls for a completion flag to be set by a ROM-BIOS interrupt handler, this technique is really no better than straight polling.

One interrupt per IORS. DMA device controllers can perform an entire data transfer given the information from the `buff` and `count` fields of an IORS. The driver task sends this information to the device controller, along with whatever control codes are needed to indicate whether the operation is a read or a write. When the entire data transfer is complete (either successfully or not), the device controller signals the processor with an interrupt request. The driver task then reads status information from the controller to test whether the operation completed normally or not. Once the interrupt is processed, the driver task can process the next IORS when it arrives. If the device driver processes I/O requests one at a time using this technique, the driver task can serve as the interrupt task for the driver.

Multiple interrupts per IORS. This type of operation is used for device controllers that cannot complete an entire data transfer in a single step. The situation arises in the case of an unbuffered serial port that generates an interrupt for each byte of data it transmits or receives, or in the case of a disk controller, that cannot perform data transfers that cross sector or cylinder boundaries without additional seek operations.

A device driver for a Universal Asynchronous Receiver-Transmitter (UART) provides a good example of a driver that must process multiple interrupts per IORS, and the following discussion of this driver includes concurrent processing of input and output operations as well. A UART is a single integrated circuit that operates as an almost complete device controller for an asynchronous serial communication line, such as an RS-232 or RS-422 serial port connected to a device unit such as a modem, terminal, or another computer. All the UART needs to complete the controller is an external clock circuit to set the baud rate and some buffers to provide and buffer the proper transmission voltages.

A simplified diagram of a UART is shown in Figure 9.2. The UART provides two on-chip data buffers, one for transmitting and one for receiving characters. These buffers typically hold as little as a single byte of data each. There are UARTs with large data buffers on chip, in which case they are considered to be *buffered controllers*, and are typically handled using the one-interrupt-per-IORS model.

For this discussion, the input buffer consists of a one-byte receiver register that receives serial data (one bit at a time) from the device unit and a one-byte input data buffer that can be read in parallel (*n* bits at a time, with *n* usually being 8) by the processor. The arrival of a character from the device unit begins with a voltage transition on the input data wire called a *start bit*. Using the baud rate clock for timing, the UART then shifts successive bits into the receiver register until the register has received the proper number of data and parity bits. At that point, the receiver register must receive a pulse opposite in polarity to the start bit, called a *stop bit*.

If the number of data bits and the value of the parity bit match the UART's expectations, which are determined by a set of control codes sent before transmission began, the UART dumps the receiver register into the data buffer register and generates an interrupt request signal. This signal goes to the processor through the mediation of a PIC as described in chapters 5 and 8. If an error occurs, such as an invalid parity bit value (a parity error), failure to receive a stop bit at the expected time (a framing error), or arrival of a character before the previous character has been read from the input data buffer by the processor (a data overrun error), the UART sets an appropriate error code value in an on-chip status register before generating its interrupt request.

The output buffer is just the opposite of the input buffer. It consists of a one-byte output data buffer that receives parallel data from the processor and a transmitter register that sends serial data to the terminal, modem, or

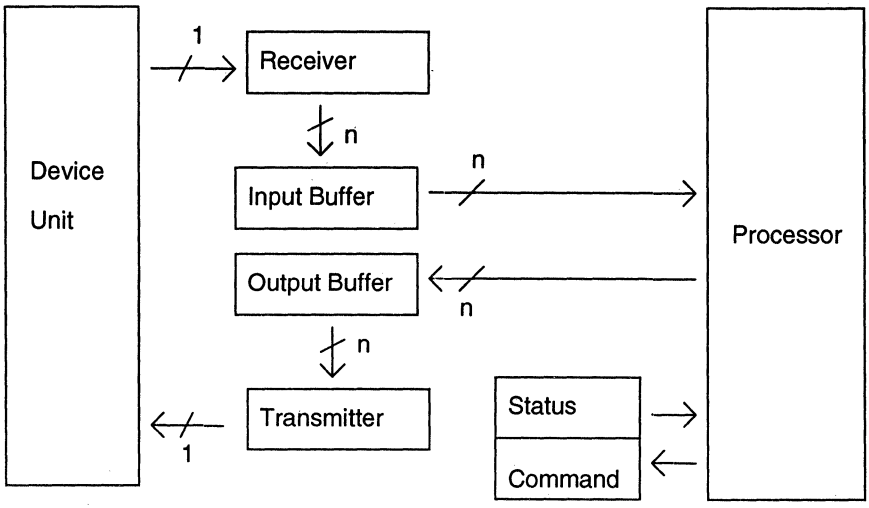


Figure 9.2 UART registers.

remote computer. The transmitting side of the UART generates an interrupt request every time the UART is ready to accept a new byte of data from the processor. The first data byte it receives is immediately loaded into the transmitter shift register so that the output data buffer can immediately receive another byte of output. The processor can monitor the status of the output data buffer and the transmitter by examining the status register of the UART, but the interrupt mechanism typically takes care of this automatically.

The two sides of the UART can operate concurrently, and the device can generate separate interrupt requests for the two sides independently. It is electrically possible to merge the two interrupt request signals into one, but the two interrupt requests are generally separate, so there are two interrupt tasks. (A single task cannot act as an interrupt task for two interrupt levels.)

A device driver for this type of device controller, then, requires two interrupt tasks, and can profitably be constructed with a third driver task for managing IORSs. Figure 9.3 shows the relationships among the tasks for such a device driver. The structure in Figure 9.3 could be established by *initializeIO()* as follows:

Step 1. Create the input and output buffer queues. The input buffer holds input data that arrives from the UART before the driver task processes a corresponding *read* IORS. For a terminal connected to a serial port, this buffer is often called a *typeahead buffer* because it holds the characters that

a user types before they are read by a program.[4] The output buffer holds outgoing data produced by the driver task in response to *write* IORSs, but which have not yet been output to the UART by the output task. Logically, this buffer could be eliminated if the driver task served as the output interrupt task, but such an arrangement would limit the amount of concurrent input and output that could occur.

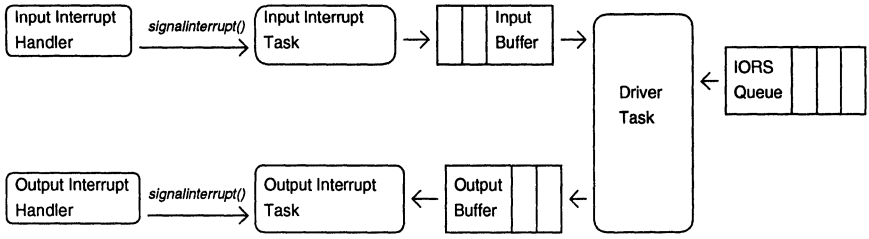


Figure 9.3 Device driver tasks for concurrent input and output.

Not shown in Figure 9.3 is the possibility that the output buffer queue can be accessed by the input interrupt task for sending flow control characters to the device unit. For example, if the input interrupt task finds that the input buffer queue is filling up too rapidly, it could send an <xoff> character to the device unit by inserting the character at the head of the output buffer queue. The <xoff> character would tell the device unit to stop sending data to the UART until the device receives an <xon> character, which the input interrupt task would place in the output buffer queue when the input buffer queue has sufficiently emptied. (The ASCII <xoff> and <xon> characters are <^S> and <^Q>, respectively.) The point at which the number of characters in the input queue triggers an <xoff> character is called the *high-water mark*, and the point at which the number of characters is low enough to generate an <xon> is called the *low-water mark*.

Step 2. Once the input and output buffer queues have been initialized, the two interrupt tasks can be created. The algorithms for these two tasks are relatively straightforward. After calling *rqsetinterrupt()*, the input task enters an endless loop in which it first calls *rqwaitinterrupt()*. The task will block until the input interrupt handler calls *rqsignalinterrupt()*, which might be the only thing that procedure does when it is activated in response

[4]Device drivers for terminals are not normally coded as custom drivers on iRMX. Terminal drivers, introduced in section 9.4.4, handle the typeahead buffer somewhat differently.

to an interrupt request from the UART. The interrupt task then reads the status byte from the UART to ensure that no error has occurred, and then reads the contents of the UART's input data buffer, which also clears any error conditions that have occurred. The interrupt task then adds the character just read to the input buffer queue and returns to the top of its processing loop, where it calls *rqwaitinterrupt()* again.

A couple of details should be noted here. First, the interrupt task must verify that space is available in the input buffer queue before it adds data to it, and block and/or invoke a flow control mechanism if the queue is full. Second, if an *rq[as]special()* system call has established signal characters for the device, the interrupt task should recognize signal characters when they arrive and process them by sending a unit to the signal character semaphore and possibly flushing the input buffer queue instead of adding them to the queue.

The output interrupt task really is simple: it waits for a character to arrive in the output buffer queue, writes it to the UART's output data buffer, calls *rqwaitinterrupt()* to wait until the UART is ready to receive another byte of data, and loops back to the top of its processing loop to await the arrival of more data in the output buffer queue.

Step 3. After initiating steps 1 and 2, *initializeIO()* can now create the IORS queue for the driver and the driver task itself. *InitializeIO()* sets the status word passed to it to 0 and returns to the file driver that called it, which will proceed to add IORSs to the IORS queue in response to system calls by application tasks. The driver task now has the responsibility for processing IORSs as they arrive at the queue. Essentially, this consists of copying data from the application's buffer to the output buffer queue for *write* IORSs, or copying data from the input buffer queue to the application's buffer for *read* IORSs.

iRMX device drivers for serial device controllers provide a number of ancillary functions that can be added to the operations performed for *read* IORS processing. Most of these operations are normally performed by routines in a software module called the Terminal Support Code (TSC), described in the section on Terminal Drivers. A few of these operations are described here to illustrate some of the interactions that could take place in a three-task device driver, such as the one being considered here.

The first operation to be considered is *character echoing*. For computer-to-computer communication, the input and output data streams are normally independent of one another, but for communication between a terminal and a computer, the characters that a user types at the keyboard must be echoed to the terminal's display device to be visible. The echoing can also be done by the terminal itself, as is normal in half-duplex operation, where the terminal and computer cannot both transmit data to each other at the same time. But for a full-duplex connection, the computer echoes the characters it receives to make them visible.

The echoing can be done by either the input interrupt task or the driver task. The latter is preferable because the characters do not appear on the user's screen until they are actually read by an application task. The user thus receives positive acknowledgment that what was typed was actually read by a program, and can tell exactly which program read them by where they appear on the screen relative to output prompt messages. Echoing, like the other operations discussed here, is controlled for iRMX terminal drivers by a *setterminalattributes* call to *rq[as]special()*. If echoing is enabled for the three-task driver, characters taken from the input buffer queue are simply copied to the output buffer queue as well as to the application's buffer.

A second operation is end-of-input recognition. For normal terminal operation, iRMX drivers recognize <cr>, <lf>, and <sub> (0x1A, obtained by typing <^z>) as end-of-message characters. Receiving one of these characters from the input buffer queue completes processing of a *read* IORS, even if the number of characters provided in `iors.count` have not yet been transferred to the application's buffer. The driver task sets `iors.actual` to indicate the number of characters actually transferred. Most software treats an `iors.actual` value of 0, obtained when a user types <^z> at the beginning of an input line, as an end-of-file condition. Without end-of-message recognition, the driver task cannot complete processing of a *read* IORS until it can obtain `iors.count` characters from the input buffer queue. The three characters are treated differently:

- <^z> is not echoed to the user's screen and is not placed in the application's buffer. `iors.actual` is not incremented.
- <cr> is placed in both the output buffer queue for echoing to the screen and in the application's buffer. It also causes the driver to add a linefeed character to both the application's buffer and the output buffer queue. `iors.actual` is incremented by 2.
- <lf> is simply put in the output buffer queue and the application's buffer. `iors.actual` is incremented by 1.

Terminal drivers for buffered controllers must also manage special characters that trigger end-of-input recognition when they are typed. Special characters are established by a *setterminalattributes* call to *rq[as]special()*.

The third operation considered is line editing. Application tasks can specify one of three line editing modes by calling *rq[as]special()* with function code 5 (which is set terminal attributes). The line-editing mode is specified by a two-bit value as follows:

0	Not allowed	
1	Transparent mode	Signal characters received by the interrupt handler are processed normally, but all other characters are placed in the application's buffer

unchanged. A read operation completes only when the number of characters in `iors.count` are obtained from the input buffer queue. That is, <cr>, <lf>, and <sub> have no significance and are placed in the application's buffer unchanged.

2 Normal mode — In addition to the message-ending significance of the three characters mentioned previously, a <rub> character typed by the user can be used to delete characters from a line being typed. If any characters have been placed in the application's buffer when this character is detected, the driver task decrements `iors.actual` by one, decrements its pointer into the application buffer (if it is not using `iors.actual` as an index into the buffer), and places a character sequence in the output buffer queue to indicate that a character has been erased. For a CRT, the sequence is <bs><sp><bs>, which moves the cursor left one position, writes a blank character, and moves the cursor left again.

3 Flush mode — This mode is the same as transparent mode, except that the read operation completes immediately with however many bytes are available in the input buffer queue at the time the driver task starts processing the *read* IORS.

For all three modes, it is possible that the input buffer queue will contain more bytes than requested by the *read* IORS. In this case, the extra characters simply accumulate in the queue until another read request arrives.

There are three more functions that a terminal device driver should support, which are invoked by application calls to *rq[as]special()* with function codes 16, 17, and 18. Code 16 is used by applications to obtain information about the state of a connection, such as the number of characters present in the typeahead buffer. Code 17 is used to cancel outstanding operations on a connection (the BIOS calls the driver's *cancelIO()* routine for this function rather than *queueIO()*). Code 18 is used to resume I/O for a terminal that has been blocked because the user entered a control character, such as <^S>.

As mentioned earlier, this discussion has not shown how Intel's iRMX terminal drivers actually work, and it has not shown many of the control functions that can be used with iRMX terminal drivers. Some of that material is presented in Section 9.4.4, and a full explanation of all the features supplied to terminal drivers by the Terminal Support Code is given in the manual *iRMX Device Drivers Programming Concepts*. What this section

has tried to do is show how a custom device driver *could* be designed to implement the functions described.

Bounded buffer implementation. The three tasks in the program provide a classic example of the producer-consumer relationships common to concurrent processing problems found in systems programming. Before moving on to a discussion of other types of device drivers, this relationship is examined, along with how standard iRMX programming techniques are used to implement an efficient solution.

One example of a producer-consumer relationship is provided by the input interrupt task and the driver task. The input interrupt task produces information by placing data bytes in the input buffer queue, and the driver task consumes the information by removing the data bytes from the same queue at some later time. The queue is known as a *bounded buffer* simply because its capacity is bounded by the amount of storage allocated to it when it is created. (An *unbounded buffer* would be allowed to grow by allocating more memory to it if it fills up.) Each task must obey a simple constraint: the consumer cannot remove data from the buffer when it is empty, and the producer cannot add data to the buffer when it is full. A task that is blocked by its constraint condition must wait until the condition is lifted before proceeding to carry out its operation. An iRMX solution to this problem can be implemented using the following data structure.

```
typedef struct boundedBuffer {
  TOKEN   critical;
  TOKEN   occupied;
  TOKEN   free;
  WORD    nextPut;
  WORD    nextGet;
  BYTE    bufferBytes [bufSize];
  }
```

To create a bounded buffer using this `typedef`, a task could perform the following sequence of steps:

1. Create a memory segment the size of this structure to hold the buffer and its associated variables. The size would be 10 or 14 bytes, depending on the size of a pointer for the processor's architecture, plus the length of the `bufferBytes` array.
2. Set the values of `nextPut` and `nextGet` to 0. These variables will be used as indices into the `bufferBytes` array by the producer and consumer tasks, respectively.
3. Create a counting semaphore with an initial value of 0 and a maximum value equal to `bufSize`. Place the token in `occupied`.
4. Create another counting semaphore with both its initial and maximum values equal to `bufSize`. Place its token in `free`.

5. If there are to be multiple producer tasks or multiple consumer tasks, create a region and place its token in `critical`. The region is used to control access to `nextPut` and `nextGet`, but is not needed if each variable is manipulated by only one task. The two variables could be protected by two different regions, but accesses to these variables are so brief a second region is probably not worth the overhead of creating it.

The producer uses the following algorithm to add a data byte to the bounded buffer:

1. Receive a unit from the `free` semaphore. If no free space exists in the buffer, the task will sleep until the consumer removes a data item and signals that it has done so by sending a unit to this semaphore.
2. If multiple producer tasks might access the buffer, receive control of the `critical` region.
3. Store a data byte at `bufferBytes[nextPut]`. Add one, modulo `bufSize`, to `nextPut`. (The array acts as a circular buffer.)
4. Send control of `critical` if there are multiple producer tasks.
5. Send a unit to the `occupied` semaphore to indicate the availability of the data.

Meanwhile, consumer tasks can execute the following algorithm:

1. Receive a unit from the `occupied` semaphore. If none is available, the task sleeps until a producer task enters data into the buffer and sends a unit to the semaphore indicating its availability.
2. If multiple consumer tasks might access the buffer, receive control of the `critical` region.
3. Do whatever is desired with `bufferBytes[nextGet]`, such as copy it into a private variable. Add one, modulo `bufSize` to `nextGet`.
4. Send control of the region if there are multiple consumer tasks.
5. Send a unit to the `free` semaphore to indicate the availability of an unused slot in the array.

In reviewing the logic of the three-task driver presented in this section, it should be clear that the automatic synchronization provided by this mechanism was implicitly relied upon when the algorithms were described, followed by all three tasks. A very attractive feature of iRMX is the relatively low amount of overhead this mechanism places on the processor compared to other operating systems or programming languages that incorporate synchronization primitives in their semantics.

9.4 Common, Random, and Terminal Drivers

If a device controller operates with a single interrupt level, even if it generates multiple interrupts per I/O transfer, a new device driver can usually be developed for it without coding the procedures and tasks previously outlined for a custom driver. The iRMX BIOS includes code for a standard interrupt task and standard versions of *initializeIO( ), finishIO( ), queueIO( ),* and *cancelIO( )* that work for a wide variety of device controllers.

In the discussion that follows, terms such as *system initializeIO( )* refer to the versions of these procedures supplied with the operating system. Three types of device drivers exist with built-in BIOS support, Common, Random, and Terminal. The term *user driver* refers to a Common, Random, or Terminal driver developed by a user using system-supplied services. The drivers supplied with iRMX are also Common, Random, and Terminal drivers that use these same system-supplied services. They are usually called *Intel drivers.*

To take advantage of these system-supplied services, the developer needs only to supply procedures and data structures and link them into the BIOS. Many techniques are available for adding such code to the operating system, including dynamically adding them at run time. The remainder of this chapter introduces the concepts involved in developing a user device driver and adding it to the BIOS. For full details, consult the *iRMX Device Drivers User's Guide.*

All user drivers rely on a data structure called a *Device Information Table* (called a DIT here, but not in the iRMX documentation), that is pointed to by the `deviceInfoPtr` field of a DUIB. The format of a DIT varies for Common, Random, and Terminal drivers, but in all cases the DIT contains pointers to user-written procedures called by the system-supplied code to perform device-specific operations.

A DIT contains information specific to a single device controller. As Figure 9.4 shows, this information includes pointers to several user-supplied routines. For Common and Random drivers, there are five of these pointers, and for Terminal drivers, there are seven.[5] These procedures are called by the system-supplied parts of the driver whenever it is time to perform a device-specific operation. In addition, the system supplies utility and housekeeping procedures that the user-supplied procedures can (utilities) or must (housekeeping) call at certain times.

---

[5]iRMX II and III, when running on a Multibus II platform, support a form of interaction between a device controller and a processor known as message passing. The format of a DIT and much of the material in the remainder of this chapter applies only to interrupt-driven I/O, not to message passing. Message passing drivers are described in the iRMX Device Driver Programming Concepts.

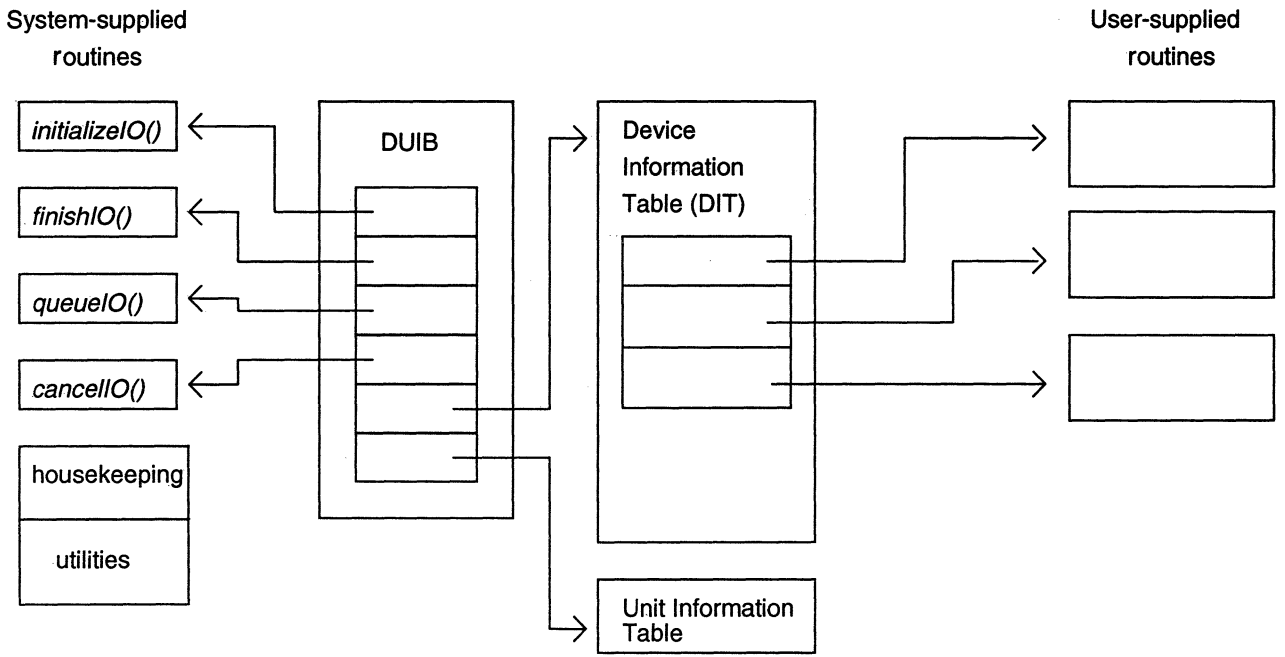


Figure 9.4 User-driver data structures.

The following are descriptions of two system-supplied utility procedures that can be called by any device driver. Other procedures are listed later for the specific types of drivers that call them.

biosgetaddress() takes a far pointer as an argument and returns the corresponding physical memory address in a doubleword. A second argument is a pointer to a condition-code word set to a nonzero value if the pointer contains either an invalid selector or offset. This routine performs the same function as the *rqegetaddress()* system call, but executes faster because it is a local procedure and does not involve the overhead of the system call mechanism. Any type of device driver can use this procedure.

gdelay() takes two WORD parameters. The first is the number of 10-microsecond intervals the driver would like to delay its execution, and the second is a delay factor that depends on the type of CPU and its clock rate. Device drivers sometimes need to invoke small time delays between device controller operations, and this routine incurs less overhead and allows finer time resolution than possible with *rqsleep()*. The delay factor for a particular processor is found in the memory segment that is cataloged in the root job's object directory using the name `RQSYSINFO`; the structure of this segment, including the `delay_const` field, is given in `/rmx386/inc/sysinfo.lit`. By including the delay factor as a parameter, this procedure can work accurately in different processing environments, unlike the similar built-in procedure, *time()*, available in PLM.

Figure 9.4 shows the relationships among the data structures involved in developing a user driver. It does not show the logical relationships among the procedures involved, which are shown in Figure 9.5. The sections that follow provide a more detailed view of a user driver than the generic model presented in these two figures.

9.4.1 Common drivers

Common drivers and random drivers are very similar. This section describes common drivers, and the next section tells how random drivers differ from them.

The procedures that must be supplied by a common driver are called *deviceInit()*, *deviceFinish()*, *deviceStart()*, *deviceStop()*, and *deviceInterrupt()*. These are generic names for the procedures. The actual access to them is through pointers in the DIT, and are called as follows:

```
void
deviceInit (          DUIBSTRUCT far *          duibPtr,
                      void far *                deviceDataPtr,
                      WORD far *                exceptPtr);
```

This procedure takes exactly the same arguments as *initializeIO()* described earlier, and is in fact called directly from the system version of that

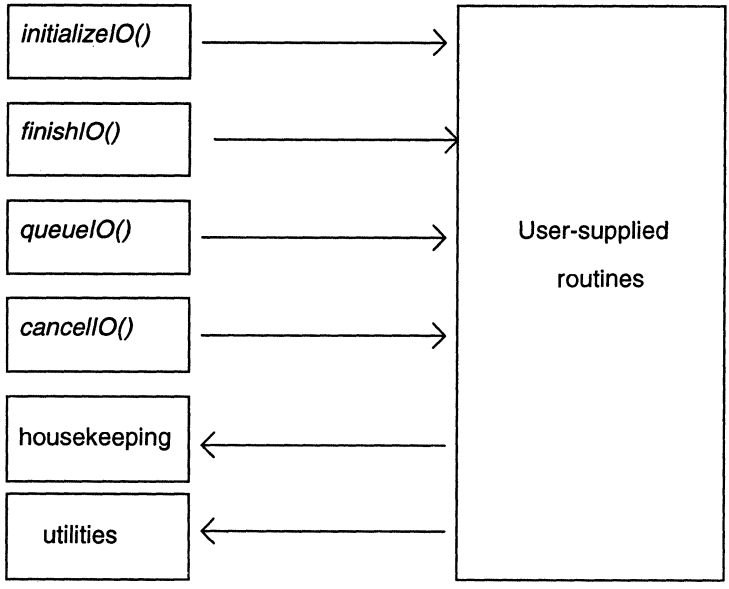


Figure 9.5 User-driver logical structure.

procedure with the corresponding parameters passed to this routine. This routine, however, does not have to create the IORS queue or an interrupt task for the driver; those operations are done automatically by the system version of *initializeIO( )*.

The `deviceDataPtr` parameter points to an area of memory that can be used in any way the driver wishes. The same pointer will be passed to each of the other procedures in this set when they are called. The size of this data area is specified in the DIT. When the system *initializeIO( )* procedure is called, it creates a data segment large enough to hold the information it needs to support the driver (the head of the IORS queue and other housekeeping information) plus the size of the device storage area specified in the DIT. The pointer that *deviceInit( )* receives points to the first location in this segment after the housekeeping information, as shown in Figure 9.6. This procedure only performs necessary initial programming of the device controller when the controller is first attached.

```
void
deviceFinish (          DUIBSTRUCT *          duibPtr,
                        void far *            deviceDataPtr);
```

The preceding procedure is called from the system *finishIO( )* procedure. This procedure performs any device controller programming needed to allow the controller to be attached again in the future. All other housekeep-

ing operations associated with detaching the device are handled automatically by the system *finishIO()* procedure.

```
void
deviceStart (          IORSSTRUCT far *          iorsPtr,
                       DUIBSTRUCT far *          duibPtr,
                       void far *                deviceDataPtr);
```

The *deviceStart()* procedure is called when a new IORS must be processed. Two different conditions can cause this procedure to be called. One is when the system *queueIO()* procedure is called, and the IORS queue for the driver is empty. The other is when the system interrupt task finishes processing one IORS and there is already another IORS on the IORS queue for this driver. In either event, the purpose of this routine is to start, and possibly complete, processing an IORS. For data transfer operations (reading, writing, and certain special functions), this routine normally starts the operation and returns to its caller. Completion of these operations is signaled by an interrupt from the device controller. For other operations, this routine completes processing of the IORS immediately and returns to its caller.

Before returning to the caller, this procedure must set the `done` field of the IORS to true (0xFF) or false (any even-numbered value) so the caller knows whether to leave the IORS on the IORS queue or not.[6] If the operation is complete (`done` is true), this routine must also set the `status` field of the IORS to indicate whether the operation completed normally or not. In this case, the caller removes the IORS from the IORS queue and returns it to the application's `responseMbx`.

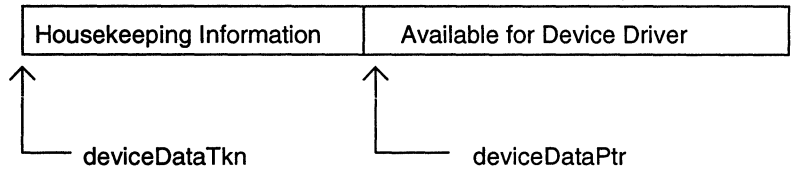


Figure 9.6 Relationship between the device data token and the device data pointer for common and random drivers.

[6]These conventions for true & false are based on the semantics of the PLM language. C programmers must note the differences from that language's definition of true (any nonzero value) and false (zero).

```
void
deviceStop (          IORSSTRUCT far *          iorsPtr,
                      DUIBSTRUCT far *          duibPtr,
                      void far *                deviceDataPtr);
```

This procedure is called from the system *cancelIO()* function. Its job is to do whatever is necessary to stop the device controller from completing an operation that has been started but has not yet completed. This routine is important for device controllers that might take a long time to complete processing of a single IORS function, but is not as important for device controllers that always operate quickly, in which case the procedure might simply do nothing and return. Of course, "long time" and "quickly" are relative terms, so the developer must decide whether this routine must actually do anything or not.

```
void
deviceInterrupt (          IORSSTRUCT far *          iorsPtr,
                           DUIBSTRUCT far *          duibPtr,
                           void far *                deviceDataPtr);
```

This procedure is called from the interrupt task created for the driver by the system *initializeIO()* procedure. It is called every time an interrupt from the device controller is recognized by the processor. That is, the system-supplied interrupt handler calls *rqsignalinterrupt()* every time it is activated, and the system-supplied interrupt task calls this procedure every time its own call to *rqwaitinterrupt()* completes.

In the case of multiple interrupts per IORS, this procedure sets the `done` field of the IORS to false, and does whatever necessary to handle the present interrupt and prepare itself and the device controller for the next one. For example, this process could involve reading a byte of data from the controller, putting it in the application's input buffer, and updating the `actual` field of the IORS. If the interrupt marks the completion of the work for an IORS, because `actual` reaches `count` in the IORS, for example, this routine sets the `done` field of the IORS to true and sets the `status` field of the IORS to the proper completion code (0 means no error). In this case, the interrupt task will de-queue the IORS and return it to the application's `responseMbx` when this procedure returns.

It is possible that a device controller will generate a spurious interrupt when there are no IORSs on the IORS queue. In this case, this routine will receive a null pointer for `iorsPtr`, and it can simply return without doing anything.

The housekeeping and utility functions that the system supplies for a Common driver to call are described after Random drivers, since some of these functions are designed specifically for use by those drivers.

9.4.2 Random drivers

Random drivers and Common drivers are implemented using exactly the same system-supplied routines. They share the same copy of *initializeIO()*, *finishIO()*, *queueIO()*, and *cancelIO()*. Their interrupt tasks and handlers execute the same procedures, and the algorithms for calling user-supplied procedures are also the same. Only two differences exist between the two types of drivers, and they are discussed next.

First, common drivers have a value of 0 in the `numBuffers` field of their DUIB, but Random drivers have a nonzero value. The file driver creates sector-sized buffers (as many as this field specifies) when the device is attached so it can read and write entire disk sectors to service data transfer requests not aligned with sectors on the disk. If an application task reads or writes data that completely spans one or more disk sectors, each complete sector of data is read or written directly to or from the application's buffer without using the buffers reserved with the `numBuffers` field. Any data transfer that does not begin and end on a sector boundary requires the BIOS to read an entire sector into one of these buffers, transfer the proper portion of the sector to or from the application's buffer, and, in the case of a write operation, transfer the sector back to the disk drive. The file driver layer of the BIOS manages the use of these buffers for this purpose automatically.

Second, random drivers must supply a valid pointer in the `unitInfoPtr` field of the DUIB. This field must point to a data structure that specifies the track and cylinder sizes of the disk, as well as a count of the number of times an I/O request should be retried before returning an IORS with an `ioSoft` code in the `unitStatus` field. The system-supplied *queueIO()* and interrupt task handler retries automatically. The cylinder size tells how many sectors there are per cylinder on the disk. If this value is greater than zero, it tells the file driver when to initiate seek operations on the disk drive, that is when a read or write is to be performed at a new position of the read/write heads.

9.4.3 Housekeeping and utility routines for common and random drivers

The following is a list of the housekeeping and utility procedures provided by the BIOS for use by Common and Random drivers.

notify(). This procedure must be called when a driver finds that a disk device unit is off line. For example, the *deviceInterrrupt()* procedure for a floppy disk drive would call *notify()* if it received an interrupt from a drive because the door to the drive is open. When this happens, the files on the

disk can no longer be accessed, and the file driver rejects I/O requests for the device unit until the door is closed again. Very few (but some) of the 3.5″ and 5.25″ floppy drives found on PCs generate the door-open signal, which is why it is necessary to detach and reattach diskette drives manually each time a diskette is changed on most PC platforms. The issue does not arise for the EDOS file driver because it uses the processor's ROM-BIOS to access floppies instead of an iRMX Random or Common driver. The ROM-BIOS assumes the diskette has been changed every time a drive is accessed and does not buffer any information from the diskette between accesses.

seekcomplete(). When the file driver encounters a data transfer that requires moving the disk's read/write heads, the file driver first sends an IORS to the device driver for the seek operation. Once the seek operation has successfully started, the driver returns the IORS with its `done` field set to true and then proceeds to allow I/O operations for other device units attached to the controller. When the device driver determines that the seek operation has completed, it calls *seekcomplete()*, and the file driver then generates the IORS for the actual data transfer operation. This optimization is important when multiple disk drives are attached to a single controller because it allows concurrent disk operations during seeks, which may take large fractions of a second to complete.

On the other hand, this optimization is not important when only one disk unit is connected to a controller because the seek and data transfer must be performed sequentially on a per-disk unit basis. The two arguments passed to this procedure are a byte identifying the unit number of the drive and a pointer to the user portion of the device's data storage area (the `deviceDataPtr` passed to the *deviceInterrupt()* procedure).

The seek operation being considered here refers to the physical movement of the device unit's read/write head assembly. The *rq[as]seek()* system calls do not invoke this sort of seek operation, they simply adjust the byte offset into a file where the next read or write operation will occur. Only when *rqaread()* or *rqawrite()* is actually called will the file driver determine if physical movement of the heads is necessary. If multiple drives are attached to a single controller and an application issues concurrent read or write operations for the different drives, the file driver issues all the physical seeks concurrently and initiates the data transfers in the order in which the seeks complete. Applications that wish to decouple seeks from data transfers explicitly must use the `fSeek` function code in a call to *rq[as]special()* for the device unit.

beginlongtermop() and endlongtermop(). These two procedures, which both take a unit number and a `deviceDataPtr` pointer as arguments, are used for overlapping long-term operations other than seeks across device units connected to a single device controller. When a device driver deter-

mines that it is to begin a long operation that will not interfere with access to other device units through the same controller, such as rewinding a tape drive, it marks the IORS done field to true and calls *beginlongtermop( )*. When the operation completes, the driver calls *endlongtermop( )*, just as it would call *seekcomplete( )* when a seek operation completes. These calls are necessary because the file driver cannot automatically determine when long-term operations other than seeks are being invoked.

getiors( ). When an interrupt marking the end of a long-term operation occurs, the *deviceInterrupt( )* procedure is called with a null pointer for the first parameter passed to it instead of a pointer to the IORS. If the driver must access the IORS, to indicate an error condition for example, or perhaps to modify the IORS to indicate a linked operation that is to be performed (reading the beginning-of-tape mark after rewinding a tape is the standard example), the driver can call *getiors( )* to get the token for the IORS being processed. This call also takes a unit number and `deviceDataPtr` as its arguments.

9.4.4 Terminal drivers

The general model for a Common or Random driver shown in Figures 9.4 and 9.5 holds for Terminal drivers in general too. The DUIB contains a pointer to a DIT that points to user-supplied routines called by system-supplied versions of *initializeIO( )*, *finishIO( )*, *queueIO( )*, and *cancelIO( )*. The user-supplied routines in turn call system-supplied housekeeping routines as they execute.

The first difference between Terminal drivers and Common or Random drivers is that the system-supplied routines pointed to by the DUIB are different, and have the names *tsinitilizeIO( )*, *tsfinishIO( )*, *tsqueueIO( )*, and *tscancelIO( )*. The *ts* prefix stands for *terminal support*. The second difference is that the DIT for a terminal driver includes pointers to eight different user-supplied procedures. The unit information table pointed to by the DUIB for a Terminal driver also has a different structure from the unit information table pointed to by the DUIB for a Random driver. It contains the connection flags, terminal flags, baud rate information, number of lines on the screen, and any additional static information the driver might want to maintain for an individual terminal device unit.

Before looking at the user-supplied routines for a Terminal driver, it is important to understand the steps that terminal data takes as it moves between the terminal device unit and the application task's buffer. Two buffers are involved in addition to the application's buffer. Starting at the device unit, the first step is for input characters to be entered into a raw input buffer as they are typed by the operator. This buffer can reside in the device controller itself or, using the Custom driver developed earlier as a model, can be implemented as a bounded buffer for which the input inter-

rupt task acts as the producer and the driver task acts as consumer.

The second buffer is the Terminal Support Code (TSC) buffer. The TSC is the system-supplied code that performs line editing and character echoing on terminal input. For both input and output operations, the TSC can also recognize escape sequences embedded in the stream of characters passing through its buffers and act on those sequences, either by modifying the stream of characters or by setting terminal parameters such as the connection flags or terminal flags.

Appendix B is an example of using TSC escape sequences so that the TSC can translate ANSI escape sequences into control codes that a non-ANSI terminal will recognize. The ability to use one set of escape sequences to get the TSC to recognize and act upon another set of escape sequences illustrates the power of the TSC code, although it is a bit difficult to master its use. For the Custom driver developed earlier, the buffering, line editing, and escape sequence processing done by the TSC would all have to be performed by logic in the driver task of that driver. The set-terminal-attributes function of the *rq[as]special()* system call can be used to control or circumvent the TSC functions dynamically for Terminal drivers.

Another item to clarify about Terminal drivers is the difference between buffered and nonbuffered device controllers. Multibus II-hosted systems can also use message-passing controllers that operate as buffered device controllers. The UART presented in the development of a Custom driver was an example of a nonbuffered controller. It generates a separate interrupt for every character read from or written to the controller, and the characters are written one at a time. It is possible to have a nonbuffered terminal controller with multiple UARTs (multiple device units for one device controller). In fact, there are single integrated circuits with multiple UARTs on the chip.

A buffered controller includes both a UART and on-board memory for holding input and output characters. A processor on the controller board reads characters into on-board memory and generates an interrupt request for the host processor when the buffer fills or when one of a set of special characters is received. Likewise, the controller's processor manages the character-by-character transmission from an on-board buffer through the UART when writing and generates an interrupt request for the host when it is ready to accept more data for output. The power of a buffered device controller is the use of dual-ported memory for the controller's buffers. The host processor reads and writes the controller's buffer memory as if it were its own memory, normally using block move instructions that require no program loops to move data between a controller's buffer and a driver's or application's buffer. Message processing controllers simply use the Multibus II message-passing mechanism to access the controller's buffer memory rather than dual-ported memory. Buffered terminal device controllers almost always support multiple device units per controller.

The following is an overview of the eight user-supplied functions for a Terminal driver.

```
void
terminit ( TSCDATASTRUCT          far *          tscDataPtr);
```

This procedure is called from *tsinitializeIO( )*, but the parameters passed to that routine are not passed on to this routine as was the case for a Common or Random drivers' *deviceInit( )* procedure when it was called from *initializeIO( )*. Rather, this procedure is passed a pointer to a data structure that the TSC maintains for the device connection. It is actually a pointer to a device data segment created by *tsinitializeIO( )* and then returned to the file driver through the `deviceDataPtr` parameter passed to that routine. The format of a `TSCDATASTRUCT` consists of a header used for all device units connected to a single device controller and a set of separate data structures maintained for each of the device units. See the *iRMX Device Drivers User's Guide*, or look in `/rmx86/inc/xtsdtn.lit` for the structure definition.

For nonbuffered device controllers, this routine must create a memory segment for the raw input buffer for the device unit and initialize the equivalent of `nextGet` and `nextPut` from our bounded buffer example to zero. The routine must put the token for the segment and the indexes into fields of the unit-specific part of the TSC data structure. Before returning to *tsinitializeIO( )*, the routine must set a status word in the header of the TSC data structure to indicate its completion status, with 0 signifying no error.

```
void
termfinish (          TSCDATASTRUCT far *          tscDataPtr);
```

This routine is called from *tsfinishIO( )* to allow the driver to do any processing necessary when the last device unit on the controller is detached to prepare the unit for later reattachment.

```
void
terminalsetup (              TSCUNITSTRUCT far *              unitDataPtr);
```

This procedure is called when a terminal is attached and again if the baud rate or parity checking values for the terminal change because of a call to *rq[as]special( )*. For buffered device controllers only, the procedure is called again when the terminal is detached. The routine, as its name implies, is used to set up the terminal for operation: it sets the baud rate, sets parity checking, asserts Data Terminal Ready if there is a modem, and enables reading and writing through the controller. The `unitDataPtr` points to the unit-specific information within the TSC data segment for the particular terminal unit being attached.

If the baud rate is not specified in the unit information table pointed to

by unitDataPtr, this routine must initiate a baud-rate scan by trying to read a character from the terminal. The user must type in an uppercase letter U (ASCII code 0x55). The driver can then determine the baud rate at which the terminal is operating by reading at a high baud rate and counting the number of 1s in the received character.

One piece of housekeeping must be done for nonbuffered device controllers. Since no data has been written yet to the device controller, and since an output interrupt is generated only when the controller completes transmission of an output character, the TSC must be told that the terminal unit is ready to receive its first output byte whenever an application tries to write to it. The housekeeping procedure this routine must call is named *xtssetoutputwaiting( )*, which takes a copy of the unitDataPtr as its single argument.

```
void
termcheck (          TSCDATASTRUCT far *          tscDataPtr);
```

This routine is called from the interrupt task for a device every time the controller generates an interrupt. Interrupts can signal the arrival of new data at the controller, readiness to accept new output data, arrival of a special character at a buffered controller (the application can designate up to four characters as special by calling *rq[as]special( )*), or a change in modem status.

If the terminal setup function initiated a baud rate scan, this routine should read the character the user typed in and appropriately set the baud rate in the unit-specific part of the TSC data segment. In all cases, this routine must update a field in the TSC data segment to indicate the type of interrupt that occurred, move data into the raw input buffer for the unit if the interrupt type indicates data is available from a nonbuffered controller, and return to the interrupt task. What happens next significantly depends on what type of interrupt this routine indicates occurred.

```
void
termout (          UNITDATASTRUCT far *          unitDataPtr,
  char                                           outputChar);
```

The TSC calls this routine to output a character to an nonbuffered device controller. This routine must set the parity bit of outputChar if necessary before writing it to the device controller. This routine will not be called for buffered controllers.

```
void
termutility (              UNITDATASTRUCT far *              unitDataPtr);
```

This procedure must be able to perform about a dozen different operations for a buffered terminal controller. When the TSC calls this procedure, it

first sets a function code field in the unit's data structure to indicate which operation the routine is to perform. Functions include moving data from the application task's output buffer to the controller's output buffer for the proper terminal unit, responding to changes in the unit's modem status and terminal attributes (such as specifying a new set of special characters to be recognized by the controller or changes in flow control parameters), and handling canceled input or output operations.

```
void
terminalanswer (        UNITDATASTRUCT far *        unitDataPtr);

void
terminalhangup (        UNITDATASTRUCT far *        unitDataPtr);
```

These two procedures are used only if the device unit is connected to a modem. The TSC calls *terminalanswer( )* when the *terminalcheck( )* procedure sets the interrupt-type code to indicate that the modem reported a telephone ring indication, and it calls the *terminalhangup( )* procedure when *terminalcheck( )* sets the interrupt-type code to indicate that the modem reported carrier loss. Escape sequences can also be embedded in either the input or output data stream for the terminal to tell the TSC to call these procedures. In any event, the jobs of these two procedures are to set or reset the DTR signal for the modem. If there is no modem, these procedures do nothing but return to the TSC.

9.5 Adding a Device Driver to the System

Several options are available for adding device drivers to an iRMX system. The technique used with iRMX for Windows is to develop the driver as loadable. To use this technique, the developer must write a front-end routine for each device driver to install it while the system is running. This technique installs the driver on a near-equal footing with the drivers supplied with the system. In fact, many of the device drivers supplied with iRMX for Windows are installed this way.

For other versions of iRMX, there is a tool called the *Interactive Configuration Utility* (ICU) that is used to build a new copy of the operating system tailored to a particular set of requirements. The ICU provides two ways of incorporating user-written device drivers into a system, one of which is the same technique used for building the iRMX-supplied drivers themselves. A characteristic of both of these techniques is that once installed into the system, a device driver cannot be removed or replaced, as would be desirable during development of a new device driver.[7]

---

[7]The -u flag for the *sysload* command, introduced with iRMX for Windows 2.0c allows *sysloaded* jobs to be unloaded.

The third technique for adding drivers to a system described here allows drivers to be installed and removed dynamically. This technique is referred to as the dynamic device driver mechanism to distinguish it from loadable device drivers. Support for dynamic device drivers must be added to an iRMX system using one of the previous two techniques.

For iRMX III and iRMX for Windows systems, SoftScope III provides support for debugging user-developed device drivers (loadable or resident). For other versions of iRMX, only the dynamic driver mechanism allows symbolic debugging with SoftScope. The other techniques require the use of machine-language debugging tools, such as the HI *debug* command.

9.5.1 Loadable device drivers

To build a loadable device driver, the programmer must create an STL file (an executable program) that includes the DUIBs, DITs, and UITs the driver needs, along with the code for all functions the driver uses. All loadable drivers must supply DITs, even Custom drivers for which DITs are normally optional. The pointer fields in each DUIB used for *initializeIO( )*, *finishIO( )*, *queueIO( )*, and *cancelIO( )* are all filled in with a constant value indicating the type of driver being installed (0xFFFFFFFF = Custom, 0xFFFFFFFE = Common, 0xFFFFFFFD = Random, 0xFFFFFFFC = Terminal, and 0xFFFFFFFB = Message Passing), and the system fills in the addresses of the actual system supplied routines for each driver type when the DUIB is installed.

For Custom drivers, the user must supply a DIT that contains far pointers to the four driver procedures for *initializeIO( ), et al.* This step is necessary for custom drivers because the user-supplied routines are in a different code segment from the rest of the BIOS, but the DUIB provides room only for near (offset-only) pointers.

For non-Custom drivers, the DIT and UIT have the normal formats required for each type of driver, except that each pointer in the tables must be a far pointer rather than a near pointer. The DIT and UIT must be initialized with the proper pointers and data values before a DUIB is installed. This difference between near and far pointers is one item that was alluded to previously when it was said that loadable device drivers are installed on only a "near-equal footing" with drivers configured into the system.

A second difference between loadable device drivers and resident drivers is that the binder cannot link either the user's code to the system-supplied utility or housekeeping routines for Common, Random, and Terminal drivers. Instead, iRMX supplies versions of these routines in a library, `/rmx386/lib/ldd.lib`, bound with the loadable driver's code.

A loadable driver, like normal HI commands, has an initial task that must initialize any necessary data structures, such as fields in the DIT, that install the DUIB(s) by calling *rqeinstallduibs( )*, and then suspending or deleting itself. The initial task must not call *rqexitiojob( )* because that

would delete the job and return its memory to the free space manager. The initial task for loadable device drivers supplied with iRMX all write a log file in the directory from which they were run to indicate whether they were loaded successfully or not, and several of them accept command-line arguments that can specify driver parameters to be initialized before the DUIBs are installed. The function prototype for *rqeinstallduibs()* looks like:

```
void
rqeinstallduibs (       WORD                        numDUIBs,
                        DUIBTABLESTRUCT far *       duibsPtr,
                        void far *                  auxPtr,
                        WORD far *                  exceptPtr);
```

A `duibstablestruct` is simply a contiguous array of DUIBs, with `numDUIBs` giving the number of elements in the array. The `auxPtr` is not used in present versions of iRMX, and should be coded as a null pointer.

A loadable driver can be loaded two possible ways. The first would be to run it as a normal HI command. Since the terminal from which the command is issued becomes unusable (the program never exits), the command must be run as a background command. This approach is really unsatisfactory, though, because of the possibility that the job will terminate (because the user logs off or kills the job from the console) and leave pointers to interrupt handlers no longer resident in the system, as well as DUIBs containing DIT pointers that no longer point to valid DITs. Rather, loadable drivers are run using the *sysload* command. When a command is run from *sysload*, it is created as a child of the HI job rather than a child of the user's terminal job, and never risks being deleted.

9.5.2 Using the interactive configuration utility

The interactive configuration utility (ICU) is really an editing and file generating program that allows a user to edit a special-format file called a *system definition file* that specifies what features and parameter values are needed for a customized copy of the operating system. Once the definition file has been constructed with the ICU editor (*icu86*, *icu286*, or *icu386*), the file generator segment of the ICU produces a set of files — assembly language and PLM code, binder commands, build file information, and submit files — that will assemble, compile, bind, and build a new copy of iRMX. The editor works by presenting the user with a sequence of menu screens, and the user enters values for the various menu items on each screen.

The ICU can include a user driver in an iRMX configuration in two ways. One way is to provide the ICU with the pathnames to already-compiled code of user-written driver procedures and to assembly language source code for DUIBs, DITs, and UITs used by the driver. The ICU in-

serts the source code into the assembly language that it generates for all the data tables of the BIOS and includes the object code in the bind of the BIOS layer. This technique is fairly easy to use, but it does require the user to adjust the values coded into the `device` and `deviceUnit` fields of his or her DUIBs after examining the source code in the DUIB file that the ICU generates. These values must then be recoded every time the system is reconfigured if the I/O configuration changes.

A more general way to incorporate a user-written driver is to have the ICU include the driver in its menu screens, and have it generate the DUIBs, DITs, and UITs automatically. This is done in two steps. First you prepare a text file that describes the driver. The file includes a version number, a driver name, an abbreviation for the driver (used by the ICU to identify menu screens for this driver), and a driver type identifier. Driver types allowed are Terminal drivers with 0, 1, or 2 interrupt levels; Common or Random drivers; Message Passing drivers; and a General (i.e., Custom) driver type. The file also includes descriptions of any additional DIT or UIT values the driver requires beyond those always included for the particular driver type being developed.

A program named *uds* is then run, which generates two new files based on the user's text file, called a *screen-master file* and a *template file*. A listing file is also generated by *uds* so the user can verify that the screen master and template files were generated correctly. When they are correct, they are merged into a new copy of the *icu* program by running the *icumrg* utility. When this version of *icu* is run, the operator can add instances of the user-written driver using the same type of screen menus as are used for all the device drivers supplied with iRMX. The only difference is that a user running the customized version of *icu* must bring up the UDS Device Drivers Module screen menu and give the pathname of the file containing the object code for the user-written device driver procedures.

Device drivers added to iRMX using either of the ICU techniques described here operate on exactly the same footing as iRMX-supplied resident device drivers. They are bound with the BIOS layer of the system, loaded with the rest of the operating system, and are in every way indistinguishable from device drivers supplied by Intel. The fact is, Intel uses exactly the same technique to incorporate the drivers it provides with iRMX.

9.5.3 Dynamic device drivers

The third method for adding device drivers to an iRMX system is somewhat of a hybrid between the loadable device drivers technique and the ICU technique. Its advantage is that it allows the user to load a device driver, test it, remove it, and reload an improved version without reconfiguring or even rebooting the system. This mechanism is not supplied by Intel. Rather, it is described in a series of articles in the iRUG newsletter

(Vickery, 1991), and the source code to provide support for Custom drivers using this technique is available from the iRUG library or the author. The description that follows is for this Custom driver version; if you follow its logic, you will realize that developing Common, Random, or Terminal versions requires linking dynamic drivers with the `ldd.lib` library mentioned previously to provide access to housekeeping and utility procedures.

A DUIB is added to the system either as a loadable driver or by using the ICU. This DUIB, called `dyndrvcu`, remains resident in the system at all times. This driver provides small procedures for *initializeIO( )*, *finishIO( )*, *queueIO( )*, and *cancelIO( )* that detect the presence or absence of a dynamic driver when they are called. If no dynamic driver is installed when one of these procedures is called, the procedure completes the call with an appropriate error indication, generally simulating reference to a nonexistent DUIB. If, however, a valid dynamic driver is installed, these routines call the corresponding device driver functions, passing their input parameters on unchanged. When the dynamic driver routines return to `dyndrvcu`'s routines, they return control to the BIOS.

The secret to the success of `dyndrvcu` is its ability to handle unexpected termination of a dynamic driver's job. This is accomplished by creating an operating system extension that instantiates a type manager for a new object type called a *driver interface object*. The technique for creating this extension is described in chapter 10. For a dynamic driver to install itself, it must first make a system call to create an object of type driver interface. The procedure is *qccreatedynamicdriver( )*:

```
TOKEN
qccreatedynamicdriver (       WORD                driverType,
                              WORD                fileDriver,
                              WORD                duibFunctions,
                              PROCSTRUCT far *    procArray,
                              INTSSTRUCT far *    intsArray,
                              WORD far *          exceptPtr);
```

The system call takes one code value that signifies the type of the dynamic driver (Custom, Common, Random, or Terminal), two values for modifying fields in the DUIB that might be examined by the BIOS (a code telling which file drivers the device driver supports and a code telling which IORS functions the driver supports). The call also takes a pointer to an array of pointers to the dynamic driver's *queueIO( ), et al.* procedures (`procArray`), plus a pointer to a data structure that tells how many interrupts the driver will use and their levels (`intsArray`). The system call returns a token for a driver interface object owned by the dynamic driver job, so it will be deleted, along with all the other resources belonging to that job when it terminates, either normally or abnormally.

There is also a *qcdeletedynamicdriver( )* system call, but dynamic drivers do not really need to use it. The Nucleus provides a facility, called a dele-

tion mailbox, for informing type managers when objects are deleted. Dyndrvcu includes a task that waits at the deletion mailbox for driver interface objects and performs the critical operation of calling *rqresetinterrupt()* for each of the interrupt levels indicated by the dynamic driver when it created its driver interface object. Thus, even if a dynamic driver job is terminated while the device is attached, dyndrvcu will know about it and can recover completely. Of course, it is still possible for a dynamic driver to crash the system, by failing to return a critical IORS for example, but that is true of any driver not fully debugged.

9.5.4 Debugging strategies for device drivers

Speaking of debugging, the following are two strategies for debugging device drivers. Debugging these device drivers poses several challenges for the developer:

- Device drivers execute in the context of the BIOS job of the operating system. It can be tricky to get output debugging information from them because they do not have access to a user's logon terminal as an output medium, as normal HI commands do.
- Device drivers configured into the OS are loaded into memory at the time the system is initialized, so symbolic information about the driver is not normally available to SoftScope. It is possible to include the symbolic information that SoftScope needs for iRMX III systems by editing the *submit* file generated by the ICU, but it is not possible to provide this information under iRMX for Windows.
- Device driver execution is inherently asynchronous with respect to other tasks within the system. Drivers that incorporate tasks in their design pose the problem of tracking the execution of these tasks within the driver itself as well.

The first strategy is to use some of the special features available with SoftScope III, and thus this strategy is available only for iRMX III systems, including iRMX for Windows. The second strategy involves adding code to the driver so that it outputs its own debugging information. This technique requires rebuilding the driver each time a bug is encountered for which there is not yet suitable output information. The one advantage of getting a driver to output its own debugging information is that the technique can be used with either loadable drivers (iRMX III and iRMX for Windows) or dynamic drivers (any version of iRMX).

SoftScope III provides full support for debugging both loadable and configured-in device drivers. While the normal process for using the ICU to configure a version of the OS eliminates all debugging information from the file that is bootstrap loaded, the *bld386* command generated by the ICU

to build the system can easily be edited manually to change the `notype` and `nodebug` controls to `type` and `debug`, respectively. Using the SoftScope *load* command to load the operating system image file that has been bootstrap loaded and is already running causes SoftScope to extract the debugging information from the file. All normal debugging techniques are then available, such as setting breakpoints within the driver, examining its variables and data structures, *etc.* SoftScope does not actually try to load the operating system itself again, it just gathers the symbolic debugging information. Since the system image file retains the information about the modules that were combined to build the operating system, SoftScope can recognize module names and locate the listing file for the driver thus displaying source statements when execution enters the driver's code.

The procedure for debugging a driver with SoftScope includes developing a normal HI command that exercises the driver's functions. In a single SoftScope session, the user loads this program in the usual way and loads the debugging information for the driver. The SoftScope *task* command can then be used to display the status of all tasks being debugged, both the driver and the test program's tasks. The *task* command also tells SoftScope which listing file to work with, the driver's, or the test program's. (SoftScope can have just one listing file open at a time.)

SoftScope III can also be used to debug loadable device drivers. In this case, the driver is loaded just like any other HI command to be debugged. The driver code can be debugged just like any HI command, even though some of the code in the module is executed by the HI command and some is executed by tasks belonging to the BIOS. The problem with debugging a loadable device driver from SoftScope is that there is no way to unload a device driver when it is time to exit SoftScope. The driver job will be a child of the SoftScope session job and will be deleted when SoftScope terminates, requiring a reboot of the system when the debugging session is complete. The only reason you need SoftScope III rather than an earlier version of SoftScope to debug loadable drivers is that loadable drivers are supported only for iRMX III, and only SoftScope III runs under iRMX III.

For all drivers, dynamic, loadable, and configured-in, care must be taken when setting breakpoints within interrupt handlers and tasks. To determine when an interrupt handler has been executed, it is necessary to add code to the handler so that it modifies a static variable (such as an interrupt counter) that can be examined by SoftScope rather than to try to set a breakpoint in the handler. Remember, all interrupts are disabled while an interrupt handler is executing, so SoftScope will not be able to read or write the operator's console from a breakpoint set within an interrupt handler.

Because loadable and dynamic drivers are installed by the initial task of an HI command, which normally suspends or deletes itself after it has installed the driver, another possibility exists for debugging these drivers. This technique is to have the initial task create a mailbox, leaving the

token for it in a variable accessible from any procedure within the module. After installing the driver, the initial task waits at the mailbox for messages. Debugging code is added to the driver procedures to send messages to this mailbox, which you can do even though the tasks executing the procedures belong to the BIOS job. The HI command's initial task then displays these messages on its standard output device as they arrive.

Chapter

10

Extending iRMX: Adding System Calls and Type Managers

10.1 Overview

iRMX is a layered operating system. A kernel layer is implemented by the iRMK kernel on iRMX III and iRMX for Windows systems, which provides primitive task scheduling and communication facilities. The iRMX Nucleus is built on top of the kernel to provide a robust multitasking operating system, including memory, task, and interrupt management facilities described in earlier chapters. All layers of iRMX above the Nucleus are implemented using the primitives supplied by the Nucleus to add new system calls to the operating system and to implement type managers for new object types to the system.

This chapter examines the resources provided by the Nucleus for adding new system calls and type managers to iRMX. Understanding the material in this chapter can be important in several ways:

- An accurate model of how the operating system is built helps programmers develop more robust and efficient programs to run on the system.
- The design considerations used to developing a new layer or system call illuminate issues that real-time and systems programmers face in general, especially object and concurrency management.
- Developers may wish to add their own layer(s) to iRMX.

Sometimes, a new layer added by a developer is specific to the needs of a particular application, but more often a new layer is added to provide some set of utility functions that can be used by any number of applications. The dynamic device driver mechanism described in chapter 9 is one example of such a utility. Another example is the implementation of the Unix *socket()*

mechanism for interprocess communication over a network, described in Vickery (1990).

The tools and techniques used by developers to add new system calls and layers to an iRMX system are exactly the same as those used by the iRMX developers at Intel to add the optional layers, such as the BIOS, to iRMX itself. Two separate issues are involved in adding a layer to the system, new system calls and new object types. The iRMX documentation uses somewhat different terminology from that used in this chapter, which will be explained as they are used. First, take a look at the following mini-glossary for some terms used in this chapter:

System call. For this chapter, a system call is a procedure called through a call gate. Thus, our discussion of adding new system calls to iRMX is focused on iRMX II and iRMX III systems, since iRMX I does not use call gates. iRMX I system calls are installed as interrupt handlers and invoked by *int* machine language instructions. The x86 call gate mechanism was introduced in Section 5.4.

Operating system extension. (OSE or extension) An operating system extension is any iRMX object type that is not one of the primitive object types defined as part of the Nucleus. Most of the primitive Nucleus object types were discussed in chapter 6. They are jobs, tasks, mailboxes, semaphores, regions, memory segments, and buffer pools. OSE is also the name of another primitive object type provided by the Nucleus, which is used to implement new extensions.

Several iRMX layers of the operating system provide system calls but not OSEs. These include the Application Loader (AL), the Human Interface (HI), and the Universal Development Interface (UDI). Thus, system calls and OSEs are independent entities.

Type manager. A type manager is a combination of an OSE and a set of system calls for creating, manipulating, and deleting instances of an OSE object type. Although it is common to add system calls to iRMX without adding an OSE, the reverse is much less common. For this chapter, assume that every OSE is accompanied by at least two system calls, one to create instances of the new object type and another to delete them.

Composite object. A composite object is an instance of an object type. To illustrate, "I/O Job" is the name of an OSE managed by the BIOS, but a particular I/O job created to run a single HI command, for example, is a composite object — an instance of that object type. Here, the word *composite* indicates that all objects of user-defined types are composed of other objects.

Component object. Each composite object consists of a set (possibly the empty set) of other objects. All of these other objects are called *component objects*, which can be either primitive objects or other composite objects.

10.2 A Sample Type Manager

To provide a framework for this chapter, sample code is presented that implements part of a type manager for a new object type, the bounded buffer introduced in section 9.3.4. The iRMX documentation also presents a type manager for this object type (called a *ring buffer*) in the *Nucleus User's Guide or iRMX Nucleus Programming Concepts* manual (volume 3 in the iRMX for Windows documentation set), depending on the OS version.

The material presented here omits some of the system calls the type manager would supply (the ones that actually add bytes to the buffer and remove them), but includes more detailed descriptions of the assembly-language interfaces that must be used to implement system calls. Bounded buffers could be called the *koan* of systems programming. Like the *hello world* program that Kernighan and Ritchie (1978) used to introduce the C language, successful implementation demonstrates basic mastery of the situation.

The job that owns an OSE object and the memory segments containing the procedures that implement system calls must exist as long as there are applications that own composite objects of the OSE type or that might invoke the system calls. This means that a user-supplied layer is normally either loaded by *sysload* when the system is initialized or is configured into the system using the ICU. The Nucleus enforces the rule that a job that owns an OSE cannot be deleted, and an OSE cannot be deleted as long as there are composite objects of that OSE's type in existence. The Nucleus cannot prevent a system call's memory from being released to the free space manager, however.[1] Thus, it is not wise to use HI commands to run jobs that create OSEs or system calls because such jobs could either become impossible to delete if there is a problem deleting the extension or a composite object or be deleted accidentally, leaving behind call gates that point to free memory rather than system call procedures.

---

[1]The issue is efficiency. The Nucleus could keep a list of all the selectors that appear in interrupt or call gates and refuse to delete segments that appear on this list. The security that would accrue to systems being used to develop user-written device drivers or system calls is outweighed by the unnecessary overhead that each call to *rqdeletesegment( )* would incur for the vast majority of code for which the issue is not a problem. In contrast, the system must maintain a separate list of composite objects for each OSE type for normal Nucleus operations anyway, so checking if this list is empty or not during *rqdeletejob( )* processing is trivial.

Of course, these two conditions are most likely to occur when a new OSE or system call is being debugged, which is exactly the time when it would be most convenient to load and test the code as an HI command. Currently you cannot debug a program symbolically if it is loaded with *sysload*, and developing either configured-in code or *sysload*-ed code requires rebooting the operating system every time a change is made to the code. Accordingly, in the spirit of the dynamic device driver mechanism discussed in section 9.5.3 to deal with similar considerations for user-written device drivers, this chapter demonstrates the structure of an HI command that could be used to develop an OSE and some system calls. Changes that would be made when the code is ready to be configured into the system or loaded by *sysload* are also presented.

The sample code consists of four source modules bound into one HI command. The first module (`bbmanage`), which is given in equivalent and interchangeable PLM and C versions in Figures 10.1 and 10.3, can be compiled to run any one of three ways. The PLM compiler control set or the C compiler control `define` can be used to set one of the symbols `first-level`, `sysload`, or `HIcmd` to select the environment for which the module is to be compiled. Exactly one of these three symbols should be defined when this module is compiled.

Figure 10.1 PLM code for an HI command to install and test two user-written system calls, *qccreateboundedbuffer()* and *qcdeleteboundedbuffer()*.

```
/**>  BBMANAGE.PLM  <************************************************
 *  Bounded Buffer Type Manager
 *  This module includes a main program that creates an OS extension
 *  for Bounded Buffer objects and establishes call gates 440 and 441
 *  as the slots for calling qccreateboundedbuffer() and
 *  qcdeleteboundedbuffer().
 ************************************************************************/

$compact (exports bbcreate, bbdelete, bbdeletetask)
bbmanage: DO;
$include (bbmanage.ext)

DECLARE BB_TYPE     LITERALLY   '8000h';
DECLARE createGate  LITERALLY   '440';
DECLARE deleteGate  LITERALLY   '441';

/*
 *  The system call procedures
 */
bbCreate: PROCEDURE (bbSize, exceptPtr,
          application_eip, application_ebp) EXTERNAL;
DECLARE bbSize          WORD_32,
        exceptPtr       POINTER,
        application_eip WORD_32,
        application_ebp WORD_32;
END bbCreate;

bbDelete: PROCEDURE (thisBB, exceptPtr,
          application_eip, application_ebp) EXTERNAL;
```

Figure 10.1 (*Continued*)

```
DECLARE     thisBB          TOKEN,
            exceptPtr       POINTER,
            application_eip WORD_32,
            application_ebp WORD_32;
END bbDelete;

/*
 *  The interface procedures
 */
qccreateboundedbuffer: PROCEDURE (bbSize, exceptPtr) TOKEN EXTERNAL;
DECLARE
        bbSize      WORD_32,
        exceptPtr   POINTER;
END qccreateboundedbuffer;
qcdeleteboundedbuffer: PROCEDURE (bbTkn, exceptPtr) EXTERNAL;
DECLARE
        bbTkn       TOKEN,
        exceptPtr   POINTER;
END qcdeleteboundedbuffer;

/*
 *  Static Variables and Constants
 */
DECLARE
        hexTab(16)  BYTE DATA ('0123456789ABCDEF'),
        coConn      TOKEN,
        bbDelTsk    TOKEN,

        bbDelMbx    TOKEN PUBLIC,   /*  Global -- used by bbDelete()
*/
        bbOSE       TOKEN PUBLIC;   /*  Global -- used by system calls
*/

$if HIcmd OR sysload
/*************************************************************************
 *
 *  Utility Procedure to Convert a Hexadecimal Word
 *  to 4 ASCII Characters
 *
 *************************************************************************/
word2hex: PROCEDURE (value, where);
DECLARE
        value       WORD_16,
        i           INTEGER,
        where       POINTER,
        xxxx BASED where (1) BYTE;

  DO i = 3 TO 0 BY -1;
    xxxx(i) = hextab(value AND 0Fh);
    value = shr (value, 4);
  END;
END word2hex;

/*************************************************************************
 *
 *  Check Status utility
 *
 *************************************************************************/
```

Figure 10.1 (*Continued*)

```
checkStatus: PROCEDURE (a_token, status_in, msgPtr);
DECLARE
        a_token                 TOKEN,
        status_in               WORD_16,
        msgPtr                  POINTER,
        message based msgPtr(1) BYTE,
        valMess(6)              BYTE,
        actual                  WORD_32,
        Status                  WORD_16;

  valMess(4) = 0Dh;
  valMess(5) = 0Ah;
  actual = rqswritemove (coConn, @message(1), message(0), @Status);
  IF (status_in = 0) THEN DO;
    actual = rqswritemove (coConn, @(' succeeded.  Token = '), 21,
@Status);
    CALL word2hex (WORD (a_token), @valMess);
    END;
  ELSE DO;
    actual = rqswritemove (coConn, @(' failed.  Status = '), 19,
@Status);
    CALL word2hex (status_in, @valMess);
    END;
  actual = rqswritemove (coConn, @valMess, 6, @Status);

  RETURN;
  END checkStatus;
$endif

/*****************************************************************************
 *
 *  Procedure Executed by the Deletion Task
 *
 *****************************************************************************/
bbDeleteTask: PROCEDURE PUBLIC;

DECLARE
        thisBB          TOKEN,
        tokenList STRUCTURE (
            numSlots    WORD_16,
            numUsed     WORD_16,
            tokens(5)   TOKEN),

        Status          WORD_16;

/*  Handle exceptions in-line
 *  -------------------------
 */
ehStruct.handler = NIL;
ehStruct.mode = 0;
CALL rqsetexceptionhandler (@ehStruct, @Status);

/*  Wait at the deletion mailbox for BB objects to delete;
 *  delete the composite and, if possible, its components.
 *  ------------------------------------------------------
 */
  DO WHILE 1;
    thisBB = rqreceivemessage (bbDelMbx, 0FFFFh, NIL, @Status);
```

Figure 10.1 (*Continued*)

```
/*  Get the tokens for the component objects
 *  ----------------------------------------
 */
  tokenList.numSlots = 5;
  CALL rqinspectcomposite (bbOSE, thisBB, @tokenList, @Status);

/*  Delete the composite to make the BB unavailable to applications
 *  and to unblock any task that called rqdeletejob() or
 *  rqdeleteextension().
 *  ------------------------------------------------------------------
 */
  CALL rqdeletecomposite (bbOSE, thisBB, @Status);

/*  Delete the region first.  If there is a task using this BB, this


    *  call will automatically block until the region is released.
    *  ------------------------------------------------------------------
    */
     CALL rqdeleteregion (tokenList.tokens(4), @Status);

   /*  Now delete the other component objects.
    *  Any or all of these calls may fail with no consequence.
    *  ------------------------------------------------------
    */
     CALL rqdeletesegment (tokenList.tokens(0), @Status);
     CALL rqdeletesemaphore (tokenList.tokens(1), @Status);
     CALL rqdeletesemaphore (tokenList.tokens(2), @Status);
     CALL rqdeletesegment (tokenList.tokens(3), @Status);

     END;

  END bbDeleteTask;

/*************************************************************************
 *
 *  Initial Task
 *
*************************************************************************/

DECLARE
        ehStruct STRUCTURE (
            handler POINTER,
            mode    BYTE),

        thisJob     TOKEN,
        a_Buf       TOKEN,
        b_Buf       TOKEN,
        actual      WORD_32,
        Status      WORD_16;

/*  Handle exceptions in-line
 *  -------------------------
 */
ehStruct.handler = NIL;
ehStruct.mode = 0;
CALL rqsetexceptionhandler (@ehStruct, @Status);
```

Figure 10.1 (*Continued*)

```
/*  Create connection for writing messages
 *  --------------------------------------
 */
$if HIcmd
    coConn = rqsattachfile (@(4,':CI:'), @Status);
$endif
$if sysload
    coConn = rqscreatefile (@(6,'bb.log'), @Status);
$endif
$if sysload or HIcmd
    CALL rqsopen (coConn, 3, 0, @Status);
$endif
$if sysload
    thisJob = rqgettasktokens (1, @Status);
    CALL checkStatus (thisJob, Status, @(17,'Start sysload job'));
$endif

/*  Create the OS Extension.
 *  Spawn a task to monitor the deletion mailbox.
 *  ---------------------------------------------
 */
  bbDelMbx = rqcreatemailbox (0, @Status);

$if HIcmd
  CALL checkStatus (bbDelMbx, Status, @(23, 'Create deletion mailbox'));
$endif

  bbOSE = rqcreateextension (BB_TYPE, bbDelMbx, @Status);

$if HIcmd OR sysload
  CALL checkStatus (bbOSE, Status, @(16,'Create extension'));
$endif
$if HIcmd
  bbDelTsk = rqcreatetask (0, @bbDeleteTask, selectorof (@hexTab),
             NIL, 8192, 0, @Status);
  CALL checkStatus (bbDelTsk, Status, @(20,'Create deletion task'));
$endif

/*  Install the type manager procedures.
 *  Make sure slots are free first.
 *  -----------------------------------
 */
  CALL rqesetosextension (createGate, NIL, @Status);
  CALL rqesetosextension (deleteGate, NIL, @Status);
  CALL rqesetosextension (createGate, @bbCreate, @Status);
  CALL rqesetosextension (deleteGate, @bbDelete, @Status);

/*  Type Manager is set up.
 *  -----------------------
 */

$if firstlevel
/*  Signal the Nucleus and become the deletion task
 *  for the type manager.
 *  -------------------------------------------
 */
    CALL rqendinittask;
    CALL bbDeleteTask;
```

Figure 10.1 (*Continued*)

```
$endif

$if sysload OR HIcmd
/*  Issue set up message.
 *  --------------------
 */
    actual = rqswritemove (coConn,
        @('Bounded Buffer Manager is Set Up', 0Dh, 0Ah), 34, @Status);
$endif

$if sysload
/*  Close the log file and become the deletion task
 *  for the type manager.
 *  ---------------------------------------------
 */
    CALL rqsdeleteconnection (coConn, @Status);
    CALL bbDeleteTask;
$endif

$if HIcmd
/*  Test Type Manager functionality.
 *  Create two bounded buffers, delete one,
 *  and clean everything up so the HI command can exit.
 *  --------------------------------------------------
 */

  a_Buf = qccreateboundedbuffer (5280, @Status);
  CALL checkStatus (a_Buf, Status, @(23,'Create a bounded buffer'));
  b_Buf = qccreateboundedbuffer (5280, @Status);
  CALL checkStatus (b_Buf, Status, @(23,'Create a bounded buffer'));

  CALL qcdeleteboundedbuffer (b_buf, @Status);
  CALL checkStatus (b_Buf, Status, @(23,'Delete a bounded buffer'));
  CALL rqdeleteextension (bbOSE, @Status);
  CALL checkStatus (bbOSE, Status, @(23,'Delete the OS extension'));

  CALL rqesetosextension (createGate, NIL, @Status);
  CALL rqesetosextension (deleteGate, NIL, @Status);
  CALL rqexitiojob (0, NIL, @Status);
$endif

END bbmanage;
```

If `firstlevel` is defined, the module produces code for a first-level job, suitable for inclusion in a configuration of iRMX generated by the ICU, introduced in section 9.5.2. Jobs configured into the system using the ICU can be run either as first-level jobs (immediate children of the root job) or as child jobs of the EIOS. The difference between the two is that first-level jobs are not I/O jobs, so they cannot make EIOS system calls, but child jobs of the EIOS are I/O jobs, although they do not have a default prefix (default directory, `:$:`).

If `firstlevel` is defined, no I/O code is compiled, and the program calls *rqendinittask()* after it has set up the bounded buffer type manager.

This system call is used during system initialization to signal the Nucleus each time a first-level job finishes initialization so that the next first-level job in sequence can be created and initialized. Note that no error exists in the code in Figures 10.1 and 10.3; *rqendinittask()* is the one iRMX system call that does not take any arguments.

If `sysload` is defined, the code will be suitable for running from a *sysload* command. In this case, the job is run as a child of the HI and is allowed to perform I/O. The convention is that jobs run from *sysload* create a log file to indicate they have been loaded and initialized successfully, so code to do so is compiled if this option is chosen. The default directory for jobs run from *sysload* is `:config:`, so the log file, `bb.log`, appears in that directory.[2]

If `HIcmd` is defined, the program can run as an HI command from the command line as a background job or under SoftScope. The code outputs status messages as it initializes, and exercises the OSE by creating a bounded buffer and then deleting it. Finally, it deletes the OSE and the system calls so the HI command can terminate.

An alternative design for the program is to make the code that exercises the OSE a separate command. In this case, it is necessary to implement a mechanism for removing the OSE cleanly so it could be re-run after any changes are added during development. The present design was chosen because the OSE is automatically deleted as soon as testing completes.

If none of these three symbols is defined when the code is compiled, you will get a program suitable for running from *sysload*, but which generates no log file.

As a type manager, the initial task for this program creates the OSE for bounded buffer objects and installs two new system calls for creating and deleting bounded buffers. Looking at the code in Figures 10.1 and 10.3, you will see that if `HIcmd` is defined, the initial task also creates a second task that executes a procedure to monitor a mailbox, called the *deletion mailbox*. If `HIcmd` is not defined, the initial task executes this procedure itself. The role of this procedure is discussed in the section on deletion mailboxes, section 10.4.3. At this point, however, note that this procedure is normally executed by the job's initial task, but a separate task is needed to execute this procedure when the initial task is being used to exercise the new system calls.

The second source module (`bbsysca1`) is also shown in equivalent and interchangeable PLM and C versions in Figures 10.2 and 10.4, respectively. This module contains the actual system call procedures that create and delete bounded buffer objects. Procedures for the other system calls the type manager is to supply (to put bytes into a bounded buffer and to get bytes out) would go in this module as well. Considerable care must be used

[2]Most jobs loaded by *sysload* specify an explicit pathname for the log file in case the default directory changes in the future.

Figure 10.2 PLM procedures to implement the *qccreateboundedbuffer()* and *qcdeleteboundedbuffer()* system calls.

```
/**>  BBSYSCAL.PLM  <*************************************************
 *  Bounded Buffer System Call Procedures
 *  This module contains the procedures that implement type manager
 *  functions for the Bounded Buffer object type.
 *  ---------------------------------------------------------------
 *  A bounded buffer object is constructed from the following objects:
 *
 *  0 - A segment containing the BB structure:
 *          bufSize DWORD
 *          nextGet DWORD
 *          nextPut DWORD
 *  1 - A counting semaphore containing one unit for each occupied slot
 *  2 - A counting semaphore containing one unit for each free slot
 *  3 - A segment to hold the buffer itself.
 *  4 - A region for controlling concurrent accesses to the BB
 *
 *  Tokens for these objects occupy tokenList.tokens positions zero
 *  through four, in sequence.
 ********************************************************************/

$compact (exports bbcreate, bbdelete)

bbsyscal: DO;

$include (bbsyscal.ext)
$include (:inc:error.lit)

DECLARE     bbOSE           TOKEN EXTERNAL,     /*  Extension Object  */
            bbDelMbx        TOKEN EXTERNAL;     /*  Deletion Mailbox  */

/*
 *  System Call exit procedures
 */
sys_exit_n: PROCEDURE EXTERNAL;         /*  No return value  */
END sys_exit_n;

sys_exit_v: PROCEDURE (value) EXTERNAL; /*  Return 1-4 bytes  */
DECLARE value   WORD_16;
END sys_exit_v;

sys_exit_e: PROCEDURE (code, parameterNum) EXTERNAL; /*  Error return
*/
DECLARE (code, parameterNum) WORD_16;
END sys_exit_e;

/********************************************************************
 *
 *  System call procedure for qccreatedboundedbuffer()
 *  Create a Bounded Buffer object
 *
 ********************************************************************/
bbCreate: PROCEDURE (bbSize, exceptPtr,
          application_eip, application_ebp) REENTRANT PUBLIC;
DECLARE bbSize          WORD_32,

        exceptPtr       POINTER,
        application_eip WORD_32,
```

Figure 10.2 (*Continued*)

```
        application_ebp WORD_32;

DECLARE tokenList STRUCTURE (
            numSlots    WORD_16,
            numUsed     WORD_16,
            tokens(5)   TOKEN);

DECLARE
        bbStructTok     TOKEN,
        BBstruct based bbStructTok STRUCTURE (
                bufSize WORD_32,
                nextGet WORD_32,
                nextPut WORD_32),

        callersEH   STRUCTURE (
            handler     POINTER,
            mode        BYTE),
        savedMode       BYTE,
        thisBB          TOKEN,

        Status          WORD_16,
        Status0         WORD_16,
        Status1         WORD_16,
        Status2         WORD_16,
        Status3         WORD_16,
        Status4         WORD_16,
        Status5         WORD_16;

/*  Get the caller's exception handling mode and change to in-line
 *  ---------------------------------------------------------------
 */
 CALL rqgetexceptionhandler (@callersEH, @Status);
 savedMode = callersEH.mode;
 callersEH.mode = 0;
 CALL rqsetexceptionhandler (@callersEH, @Status);

/*  Create the objects that will make up the BB object
 *  --------------------------------------------------
 */
  tokenList.numSlots = 5;
  tokenList.numUsed = 5;
  tokenList.tokens(0) = rqcreatesegment (size(BBstruct), @Status0);
  tokenList.tokens(1) = rqcreatesemaphore (0, bbSize, 0, @Status1);
  tokenList.tokens(2) = rqcreatesemaphore (bbSize, bbSize, 0, @Status2);
  tokenList.tokens(3) = rqcreatesegment (bbSize, @Status3);
  tokenList.tokens(4) = rqcreateregion (0, @Status4);
  thisBB = rqcreatecomposite (bbOSE, @tokenList, @Status5);

/*  If any part of the initialization failed, delete all objects,
 *  set the appropriate condition code, and return to the caller.
 *  -------------------------------------------------------------
 */
IF (Status0 OR Status1 OR Status2 OR Status3 OR Status4
OR Status5) <> E$OK THEN DO;
  CALL rqdeletesegment (tokenList.tokens(0), @Status);
  CALL rqdeletesemaphore (tokenList.tokens(1), @Status);
  CALL rqdeletesemaphore (tokenList.tokens(2), @Status);
  CALL rqdeletesegment (tokenList.tokens(3), @Status);
```

Figure 10.2 (*Continued*)

```
    CALL rqdeleteregion (tokenList.tokens(4), @Status);
    CALL rqdeletecomposite (bbOSE, thisBB, @Status);

    /*  Restore calling task's exception handling mode and return
     *  ---------------------------------------------------------
     */
    callersEH.mode = savedMode;
    CALL rqsetexceptionhandler (@callersEH, @Status);

    IF (Status0 = E$MEM) OR (Status3 = E$MEM) THEN
      CALL sys_exit_e (E$MEM, 1);
    IF (Status0 = E$LIMIT) OR (Status1 = E$LIMIT) OR
       (Status2 = E$LIMIT) OR (Status3 = E$LIMIT) OR
       (Status4 = E$LIMIT) OR (Status5 = E$LIMIT)
    THEN CALL sys_exit_e (E$LIMIT, 0);
    ELSE CALL sys_exit_e (E$CONTEXT, 0); /* Default exception code */

    END;    /*  if failure  */

/*  Set initial values for the housekeeping variables.
 *  -------------------------------------------------
 */
  bbStructTok = tokenList.tokens(0);
  BBstruct.bufSize = bbSize;
  BBstruct.nextGet = 0;
  BBstruct.nextPut = 0;

/*  Restore calling task's exception handling mode
 *  and return the composite
 *  ---------------------------------------------
 */
  callersEH.mode = savedMode;
  CALL rqsetexceptionhandler (@callersEH, @Status);

  CALL sys_exit_v (WORD(thisBB));
  END bbCreate;

/*************************************************************************
 *
 *  System call procedure for qcdeleteboundedbuffer()
 *  Delete a Bounded Buffer object.
 *
 *************************************************************************/

bbDelete:   PROCEDURE (thisBB, exceptPtr,
            application_eip, application_ebp) REENTRANT PUBLIC;

DECLARE

            thisBB              TOKEN,
            exceptPtr           POINTER,
            application_eip     WORD_32,
            application_ebp     WORD_32,

            callersEH           STRUCTURE (
                handler     POINTER,
                mode        BYTE),
            savedMode           BYTE,
```

Figure 10.2 (*Continued*)

```
            (Status, Status1)   WORD_16;

/*  This call sends the token for the buffer to be deleted to the
 *  deletion mailbox.
 *  ----------------------------------------------------------------
 */
/*  Get the caller's exception handling mode and change to in-line
 *  ----------------------------------------------------------------
 */
  CALL rqgetexceptionhandler (@callersEH, @Status);
  savedMode = callersEH.mode;
  callersEH.mode = 0;
  CALL rqsetexceptionhandler (@callersEH, @Status);

  CALL rqsendmessage (bbDelMbx, thisBB, selectorof(NIL), @Status1);

/*  Restore calling task's exception handling mode and return
 *  ----------------------------------------------------------
 */
  callersEH.mode = savedMode;
  CALL rqsetexceptionhandler (@callersEH, @Status);

  IF Status1 = E$OK THEN CALL sys_exit_n;
  CALL sys_exit_e (E$CONTEXT, 0);

  END bbDelete;

END bbsyscal;
```

Figure 10.3 C code equivalent to Fig. 10.1.

```
/**>  BBMANAGE.C  <*********************************************************
 *  Bounded Buffer Type Manager
 *  This module includes a main program that creates an OS extension
 *  for Bounded Buffer objects and establishes call gates 440 and 441
 *  as the slots for calling qccreateboundedbuffer() and
 *  qcdeleteboundedbuffer().
 ***************************************************************************/

#include <stdio.h>
#include <fcntl.h>
#include <rmxc.h>
#include <common.h>
#include "boundbuf.h"

#pragma noalign (bbTOKENLISTSTRUCT)
typedef struct {
    WORD    numSlots, numUsed;
    TOKEN   tokens[5];
    } bbTOKENLISTSTRUCT;

#define BB_TYPE  0x8000
#define createGate  440
#define deleteGate  441

extern void far         /*  The system call procedures  */
```

Figure 10.3 (*Continued*)

```
bbCreate ();
extern void far
bbDelete ();

static TOKEN            bbDelTsk;
static FILE             *msgOut;

TOKEN                   bbDelMbx; /*  Global -- used by bbDelete()  */
TOKEN                   bbOSE;    /*  Global -- used by system calls  */

#if HIcmd || sysload
/*************************************************************************
 *
 *  Check Status utility
 *
 *************************************************************************/
void
checkStatus (TOKEN a_token, WORD status, char *message) {
  fprintf (msgOut, "%s", message);
  if (status == E_OK)
       fprintf (msgOut, " succeeded.  Token = %4X\n", (WORD) a_token);
  else fprintf (msgOut, " failed.  Status = %4X\n", status);
  }
#endif

/*************************************************************************
 *
 *  Procedure Executed by the Deletion Task
 *
 *************************************************************************/
void far                /*  Task to monitor deletion mailbox  */
bbDeleteTask (void) {

TOKEN               thisBB;
bbTOKENLISTSTRUCT   tokenList;
EXCEPTIONSTRUCT     ehStruct;
WORD                Status;

/*  Handle exceptions in-line
 *  -------------------------
 */
  ehStruct.offset = 0;
  ehStruct.base = (selector) NULL;
  ehStruct.exceptionmode = 0;
  rqsetexceptionhandler (&ehStruct, &Status);

/*  Wait at the deletion mailbox for BB objects to delete;
 *  delete the composite and, if possible, its components.
 *  ------------------------------------------------------
 */
  for (;;) {
    thisBB = rqreceivemessage (bbDelMbx, 0xFFFF, NULL, &Status);

  /*  Get the tokens for the component objects
   *  ----------------------------------------
   */
    tokenList.numSlots = 5;
```

Figure 10.3 (*Continued*)

```
  rqinspectcomposite (bbOSE, thisBB,
                      (TOKENLISTSTRUCT far *) &tokenList, &Status);

/*  Delete the composite to make the BB unavailable to applications
 *  and to unblock any task that called rqdeletejob() or
 *  rqdeleteextension().
 *  ------------------------------------------------------------------
 */
  rqdeletecomposite (bbOSE, thisBB, &Status);

/*  Delete the region first.  If there is a task using this BB, this
 *  call will automatically block until the region is released.
 *  ------------------------------------------------------------------
 */
  rqdeleteregion (tokenList.tokens[4], &Status);

/*  Now delete the other component objects.
 *  Any or all of these calls may fail with no consequence.
 *  ------------------------------------------------------
 */
  rqdeletesegment (tokenList.tokens[0], &Status);
  rqdeletesemaphore (tokenList.tokens[1], &Status);
  rqdeletesemaphore (tokenList.tokens[2], &Status);
  rqdeletesegment (tokenList.tokens[3], &Status);
  }
}

/*****************************************************************************
 *
 *  Initial Task
 *
 *****************************************************************************/
int
main (int argc, char * argv[]) {

EXCEPTIONSTRUCT ehStruct;
TOKEN           a_Buf, b_Buf;
WORD            Status;

/*  Handle exceptions in-line
 *  -------------------------
 */
  ehStruct.offset = 0;
  ehStruct.base = (selector) NULL;
  ehStruct.exceptionmode = 0;
  rqsetexceptionhandler (&ehStruct, &Status);

#ifdef sysload
/*  Redirect standard output for sysload-ed jobs
 *  --------------------------------------------
 */
    msgOut = fopen ("bb.log", "w");
    fprintf (msgOut, "This is job %4X\n", rqgettasktokens (1, &Status));
#elif HIcmd
    msgOut = stdout;
#endif

/*  Create the OS Extension.
```

Figure 10.3 (*Continued*)

```
 *  Spawn a task to monitor the deletion mailbox if necessary.
 *  ----------------------------------------------------------
 */
  bbDelMbx = rqcreatemailbox (FIFO_QUEUING, &Status);

#ifdef HIcmd
  checkStatus (bbDelMbx, Status, "Create deletion mailbox");
#endif

  bbOSE = rqcreateextension (BB_TYPE, bbDelMbx, &Status);

#if HIcmd || sysload
  checkStatus (bbOSE, Status, "Create extension");
#endif
#ifdef HIcmd
  bbDelTsk = rqcreatetask (0, &bbDeleteTask, (selector) &bbDelMbx,
             NULL, 8192, 0, &Status);
  checkStatus (bbDelTsk, Status, "Create deletion task");
#endif

/*  Install the type manager procedures.
 *  Make sure slots are free first.
 *  -----------------------------------
 */
  rqesetosextension (createGate, NULL, &Status);
  rqesetosextension (deleteGate, NULL, &Status);
  rqesetosextension (createGate, &bbCreate, &Status);
  rqesetosextension (deleteGate, &bbDelete, &Status);

/*  Type Manager is set up.
 *  -----------------------
 */
#if sysload || HIcmd
        fprintf(msgOut,"Bounded Buffer Manager is Set Up\n");
#elif firstlevel
/*  First level jobs signal the Nucleus and become the deletion task
 *  ----------------------------------------------------------------
 */
  rqendinittask();
  bbDeleteTask();
#endif

#if sysload
/*  Sysloaded commands must delete the log file connection,
 *  then become the deletion task
 *  -----------------------------------------------------
 */
  fclose (msgOut);
  bbDeleteTask();
#endif

#if HIcmd
/*  Test Type Manager functionality.
 *  Create two bounded buffers, delete one,
 *  and clean everything up so the HI command can exit.
 *  ---------------------------------------------------
 */
  a_Buf = qccreateboundedbuffer (5280, &Status);
```

Figure 10.3 (*Continued*)

```
  checkStatus (a_Buf, Status, "Create a bounded buffer");
  b_Buf = qccreateboundedbuffer (5280, &Status);
  checkStatus (b_Buf, Status, "Create a bounded buffer");

  qcdeleteboundedbuffer (b_Buf, &Status);
  checkStatus (b_Buf, Status, "Delete a bounded buffer");
  rqdeleteextension (bbOSE, &Status);
  checkStatus (bbOSE, Status, "Delete the OS extension");

  rqesetosextension (createGate, NULL, &Status);
  rqesetosextension (deleteGate, NULL, &Status);
  rqexitiojob (0, NULL, &Status);
#endif

  bbDeleteTask();

  }
```

Figure 10.4 C functions equivalent to Fig. 10.2.

```
/**>  BBSYSCAL.C  <*****************************************************
 *  Bounded Buffer System Call Procedures
 *  This module contains the procedures that implement type manager
 *  functions for the Bounded Buffer object type.
 *  ---------------------------------------------------------------------
 *  A bounded buffer object is constructed from the following objects:
 *
 *  0 - A segment containing the BB structure:
 *          bufSize DWORD
 *          nextGet DWORD
 *          nextPut DWORD
 *  1 - A counting semaphore containing one unit for each occupied slot
 *  2 - A counting semaphore containing one unit for each free slot
 *  3 - A segment to hold the buffer itself.
 *  4 - A region for controlling concurrent accesses to the BB
 *
 *  Tokens for these objects occupy tokenList.tokens positions zero
 *  through four, in sequence.
 ************************************************************************/

#include <rmxc.h>
#include <common.h>
#include <error.h>  /*  /rmx386/inc16/error.h with '$' changed to '_' */

#pragma noalign (bbTOKENLISTSTRUCT)
typedef struct {
    WORD    numSlots, numUsed;
    TOKEN   tokens[5];
    } bbTOKENLISTSTRUCT;

#pragma noalign (bbStruct)
typedef struct bbStruct {
  DWORD    bufSize, nextGet, nextPut;
  } BBSTRUCT;

extern TOKEN    bbDelMbx;    /* Declared and initialized in bbmanage.c */
extern TOKEN    bbOSE;       /* Declared and initialized in bbmanage.c */
```

Figure 10.4 (*Continued*)

```
/*
 *  System Call exit procedures
 */
void near
sys_exit_n (void);          /*  No return value  */

void near
sys_exit_v (WORD);          /*  Return 1-4 bytes  */

void near
sys_exit_e (WORD, WORD);    /*  Error return  */

/*************************************************************************
 *
 *  System call procedure for qccreatedboundedbuffer()
 *  Create a Bounded Buffer object
 *
 *************************************************************************/
void far
bbCreate (DWORD bbSize, WORD far *exceptPtr,
          DWORD application_eip, DWORD application_ebp) {

  bbTOKENLISTSTRUCT tokenList;
  EXCEPTIONSTRUCT   callersEH;
  BYTE              savedMode;
  BBSTRUCT          *thisBBstruct;
  TOKEN             thisBB, bbStructSeg, occSem, freeSem, bbBufferSeg;
  WORD              Status0, Status1, Status2, Status3, Status4,
Status5,
                    Status;

/*  Get the caller's exception handling mode and change to in-line
 *  ---------------------------------------------------------------
 */
  rqgetexceptionhandler (&callersEH, &Status);
  savedMode = callersEH.exceptionmode;
  callersEH.exceptionmode = 0;
  rqsetexceptionhandler (&callersEH, &Status);

/*  Create the objects that will make up the BB object
 *  --------------------------------------------------
 */
  tokenList.numSlots = 5;
  tokenList.numUsed = 5;

  tokenList.tokens[0] = rqcreatesegment (sizeof (BBSTRUCT), &Status0);
  tokenList.tokens[1] =
    rqcreatesemaphore (0, bbSize, FIFO_QUEUING, &Status1);
  tokenList.tokens[2] =
    rqcreatesemaphore (bbSize, bbSize, FIFO_QUEUING, &Status2);
  tokenList.tokens[3] = rqcreatesegment (bbSize, &Status3);
  tokenList.tokens[4] = rqcreateregion (FIFO_QUEUING, &Status4);
  thisBB =
    rqcreatecomposite (bbOSE, (TOKENLISTSTRUCT *) &tokenList, &Status5);

/*  If any part of the initialization failed, delete all objects,
 *  set the appropriate condition code, and return to the caller.
 *  -------------------------------------------------------------
```

Figure 10.4 (*Continued*)

```
*/
 if (Status0 || Status1 || Status2 || Status3 || Status4 || Status5) {
   rqdeletesegment (tokenList.tokens[0], &Status);
   rqdeletesemaphore (tokenList.tokens[1], &Status);
   rqdeletesemaphore (tokenList.tokens[2], &Status);
   rqdeletesegment (tokenList.tokens[3], &Status);
   rqdeleteregion (tokenList.tokens[4], &Status);
   rqdeletecomposite (bbOSE, thisBB, &Status);

   /*  Restore calling task's exception handling mode and return
    *  ---------------------------------------------------------
    */
    callersEH.exceptionmode = savedMode;
    rqsetexceptionhandler (&callersEH, &Status);

    if ((Status0 == E_MEM) || (Status3 == E_MEM)) sys_exit_e (E_MEM, 1);
    if ((Status0 == E_LIMIT) || (Status1 == E_LIMIT) ||
       (Status2 == E_LIMIT) ||  (Status3 == E_LIMIT) ||
       (Status4 == E_LIMIT) || (Status5 == E_LIMIT))
         sys_exit_e (E_LIMIT, 0);
    sys_exit_e (E_CONTEXT, 0);       /*  Default exception code  */
    }

  thisBBstruct = buildptr (tokenList.tokens[0], 0);
  thisBBstruct -> bufSize = bbSize;
  thisBBstruct -> nextGet = 0;
  thisBBstruct -> nextPut = 0;

/*  Restore calling task's exception handling mode
 *  and return the composite
 *  -----------------------------------------
 */
  callersEH.exceptionmode = savedMode;
  rqsetexceptionhandler (&callersEH, &Status);

  sys_exit_v ((WORD) thisBB);
  }

/*********************************************************************
 *
 *  System call procedure for qcdeleteboundedbuffer()
 *  Delete a Bounded Buffer object
 *
 *********************************************************************/

void far
bbDelete (TOKEN thisBB, WORD far *exceptPtr,
          DWORD application_eip, DWORD application_ebp) {

EXCEPTIONSTRUCT     callersEH;
BYTE                savedMode;
WORD                Status, Status1;

/*  This call sends the token for the buffer to be deleted to the
 *  deletion mailbox.
 *  ------------------------------------------------------------
 */
/*  Get the caller's exception handling mode and change to in-line
```

Figure 10.4 (*Continued*)

```
*  ------------------------------------------------------------
*/
 rqgetexceptionhandler (&callersEH, &Status);
 savedMode = callersEH.exceptionmode;
 callersEH.exceptionmode = 0;
 rqsetexceptionhandler (&callersEH, &Status);

 rqsendmessage (bbDelMbx, thisBB, (selector) NULL, &Status1);

/*  Restore calling task's exception handling mode and return
 *  ---------------------------------------------------------
 */
callersEH.exceptionmode = savedMode;
rqsetexceptionhandler (&callersEH, &Status);

if (Status1 == E_OK) sys_exit_n();
sys_exit_e (E_CONTEXT, 0);
}
```

in the design of procedures that will be system calls in a multitasking system such as iRMX. These procedures are discussed more fully in the next section.

The third module (`sys_exit`) is coded in assembly language, given in Figure 10.5. The procedures in this module are general-purpose routines that could be used by any system call to handle the conventions for returning condition code and function values to an application through the processor's registers[3]. This module, however, will work only for 32-bit applications. Designing a comparable module that could accommodate both 16-bit and 32-bit applications is left as an exercise for the reader!

Figure 10.5 Assembly language exit procedures for 32-bit system calls.

```
name sys_exit
;
;   System Call Exit Routines
;
;       Bind these routines to a system call procedure to enable it to
;       return to an interface procedure properly.  These routines are
;       for 32-bit code only.
;

code    segment er public

        public  sys_exit_n  ; normal exit, no return value
        public  sys_exit_v  ; normal exit, return 8, 16, or 32-bit value
        public  sys_exit_d  ; normal exit, return 64-bit value
```

[3]The procedures for *sys_exit_d()* and *sys_exit_p()* are not actually used by the sample system calls but are included for completeness.

Figure 10.5 (*Continued*)

```
        public  sys_exit_p  ; normal exit, return far (48-bit) pointer
        public  sys_exit_e  ; error exit

;   The routines in this module must make FAR returns to the
;   interface procedure because they must complement the interface
;   procedure's call to a call gate, which is also a FAR call.

exits   proc    far

; - - - - - - - - - Procedure Exit - No Return Value - - - - - - - - -
;   Calling sequence:
;      CALL sys_exit_n();

sys_exit_n  label   near

        add     sp,4                    ;Drop sys call's near return
address
        xor     cx,cx                   ;Make exception code E_OK
        leave                           ;Drop sys call's local variables
        pop     ds                      ;Restore application's DS
        ret                             ;Far return to interface proc

; - - - - - - - - - Function Exit - Return Value up to 32 Bits - - - - -
;   Calling sequence:
;      CALL sys_exit_v (return_value);

sys_exit_v  label   near

        add     sp,4                    ;Drop sys call's near return
address
        pop     eax                     ;Get return value
        xor     cx,cx                   ;Make exception code E_OK
        leave                           ;Drop sys call's local variables
        pop     ds                      ;Restore application's DS
        ret                             ;Far return to interface proc

; - - - - - - - - - Function Exit - Return doubleword or far pointer
value
;   Calling sequence:
;      CALL sys_exit_d (return_value);
;      CALL sys_exit_p (return value);

sys_exit_d  label   near
sys_exit_p  label   near

        add     sp,4                    ;Drop sys call's near return
address
        pop     eax                     ;Get low-order part of return
value
        pop     edx                     ;Get high-order part of return
value
        xor     cx,cx                   ;Make exception code E_OK
        leave                           ;Drop sys call's local variables
        pop     ds                      ;Restore application's DS
        ret                             ;Far return to interface proc

; - - - - - - - - - Error Exit - Return Code & Parameter # - - - - - - -
;   Calling sequence:
```

Figure 10.5 (*Continued*)

```
;       CALL sys_exit_e (error_code, parameter_number);
;
sys_exit_e  label   near

        add     sp,4                    ;Drop sys call's near return
address
        pop     edx                     ;Get parameter number
        pop     ecx                     ;Get exception code
        mov     eax,0FFFFFFFFh          ;Forced value on error
        leave                           ;Drop sys call's local variables
        pop     ds                      ;Restore application's DS
        ret                             ;Far return to interface proc

exits endp
code  ends
end
```

These first three modules are bound together to form the bounded buffer type manager. If `bbmanage` is compiled for use as a first-level job, the *bnd386* command would omit the `rc(dm(. . .))` and `ss (stack(. . .))` controls, and the resulting linkable file would normally be called `boundbuf.lnk` using the binder's `oj(boundbuf.lnk)` control. The ICU screen for adding first-level jobs to an iRMX configuration, the `USERJ` screen, obtains this information by asking for the values of the job's memory pool and stack segment size.

The `USERM` screen is used to tell the ICU the pathname to `boundbuf.lnk`. If the type manager is to be run as an HI command or by *sysload*, the three modules would be bound just like any other loadable module, with the compiler controls for the `bbmanage` module determining whether the program will be configured to run from *sysload* or not. If the command is configured to run as an HI command, the interface procedures for *qccreateboundedbuffer( )* and *qcdeleteboundedbuffer( )* must also be bound to the program, which brings us to the fourth module.[4]

The fourth module (`bbifc32`) is the assembly language code for the interface procedures for these two system calls. The design of `bbifc32` is described in section 10.3.2. This module must be bound to every application program that uses the bounded buffer type manager. It is bound to the program when configured to run as an HI command because the HI command includes code to call the interface procedures after adding the system calls to the system. Module `bbifc` would not be bound to the program if it is loaded using *sysload* or configured as a first-level job because those versions of the program do not invoke either of the new system calls themselves.

---

[4]If a separate HI command were written to exercise the OSE, the interface procedures for *qccreateboundedbuffer( )* and *qcdeleteboundedbuffer( )* would be bound only to this exerciser command, not to the code that installs the OSE.

If there were several type managers or if the interface procedures for each system call were to be assembled separately, various object modules for interface procedures should be placed into a library file using *lib386*. This step would be unnecessary overhead in the present case, where there is only one object module that contains both of the interface procedures. The code in this interface procedure module is discussed in the next section. For now, like the `sys_exit` module, the code in this module works only for 32-bit applications. The exercise mentioned previously to develop a `sys_exit` module that works for both 16-bit and 32-bit applications includes the exercise to develop a set of interface procedures to match!

10.3 Adding a System Call to iRMX

The iRMX system call mechanism was introduced in section 6.8, and the material presented here is a continuation of that section. Five issues are involved in adding a system call to an iRMX system:

1. Adding the call gate for the system call to the iRMX Global Descriptor Table (GDT).
2. Coding the assembly language interface procedure that applications call to gain access to the system call.
3. Passing parameters from the application task to the system call procedure.
4. Coding the system call procedure to work correctly in a multitasking environment.
5. Returning values and error codes from the system call to an application task.

10.3.1 Installing the call gate

A procedure can be connected to a call gate in two ways: by using the ICU to add the procedure when the system is configured or by calling *rqesetosextension( )*. The choice of name for this system call is unfortunate. The call is used to add a system call to the operating system and its use might or might not have anything to do with implementing an operating system extension. As indicated in the preceding mini-glossary, it is quite common to add system calls to iRMX without adding an operating system extension. The name should be something like *rqesetsystemcall( )* or *rqesetcallgate( )*. The prefix *rqe* would indicate that the system call is available only for iRMX II and III, because, as you recall, iRMX I does not use call gates.

Before adding a system call to the operating system, you must select which call gate the system call will use. A cluster of call gates starting at GDT slot number 440 (in decimal) is available for user-defined system

calls. iRMX for Windows systems determine the size of this cluster from the OSX parameter of the :config:rmx.ini file that is consulted when the system is initialized.[5] For systems configured using the ICU, the size of the cluster is determined by the number of OSEXT screens the user incorporates in the definition file. Note that both OSX and OSEXT signify Operating System Extension, and perpetuate the failure to distinguish between system calls and extension objects in the iRMX documentation.

A potential problem exists in managing the call gate numbers used by user-supplied system calls. Interface procedures must be hard-coded with the proper call-gate numbers, so any conflict between call-gate numbers or any change to the call-gate number allocated to a particular system call must be resolved by reassembling all the interface procedures involved and rebinding all applications that use the call to the new interface procedures. If this is a problem for developers who want to distribute their system calls to a broad range of iRMX users, it is necessary to establish some sort of registry for new system calls and their call gate numbers. Presently, the problem has not extended beyond the scope of single development sites, and local management of call-gate slot numbers has been satisfactory.

Adding system calls with the ICU. Any iRMX II or III system that supports the ICU must have a call gate cluster reserved when the system is configured if it is to allow user-written system calls. The user can link any subset of the gates in the cluster (possibly none) to actual procedures when configuring the system. The relevant menu screens are Subsystems, OSEXT, and USERM. The Subsystems screen includes an OS Extension option that must be set to yes to establish a GDT cluster. Once this option has been selected, the ICU presents OSEXT screens when the user steps to the proper place in the definition file. For each GDT slot to be reserved, the user enters a slot number (starting at 440) and an optional public procedure name to be linked to the slot. No procedure name needs to be specified because the linkage can be made at run-time by calling *rqesetosextension( )*. If any public procedures are specified on the OSEXT screens, the user must supply the ICU with the pathname to the file containing the object modules on the USERM screen.

Adding system calls by calling *rqesetosextension( )*. Whether the GDT cluster for user-written system calls is reserved at configuration time using the ICU or at system initialization time using the OSX parameter of the rmx.ini file, it is always possible to set up or to change the association between a GDT slot and a system call procedure by calling *rqesetosextension( )*.

[5]The default value is 20 at the time of publication.

```
void
rqesetosextension (          WORD              callGate,
                             void far *        systemCallPtr,
                             WORD far *        exceptPtr);
```

The `callGate` parameter specifies the number of the GDT slot to be used, and the `systemCallPtr` is a pointer to the user-written procedure to be bound to the gate. If `systemCallPtr` is a null pointer, the current binding for the GDT slot is cleared. You might recall from section 5.4 that a call gate is a descriptor that contains a far pointer to a procedure. When a task makes a far call (supplying both the selector and offset as the address of a procedure) and the selector provided with the call is found to reference a GDT slot containing a call gate, the call instruction's offset value is ignored, and the complete far pointer to the procedure is taken from the call gate. Clearly, the two parameters to this call are exactly what the operating system needs to know to install a call gate into a particular slot of the GDT. Figures 10.1 and 10.3 demonstrate the use of this call, first to clear any call gates already using slots 440 and 441, and then to associate those gates with the system call procedures shown in either Figure 10.2 or 10.4.

10.3.2 The interface procedure

As indicated in section 6.8, an assembly language interface procedure must exist for every system call added to the operating system. This procedure must:

1. Ensure that the stack frame is in the proper format for passing parameters to the system call, considering the caller's model of compilation;
2. Make the far call to the proper GDT slot;
3. Check for exceptions; and
4. Pass the system call's return value and condition code back to the caller as appropriate. The interface procedures for *qccreateboundedbuffer()* and *qcdeleteboundedbuffer()* are in module `bbifc32`, Figure 10.6.

The module name follows the iRMX library module naming convention: `-ifc-` stands for interface for the compact model, and `32` indicates that the procedures are specific to 32-bit applications. iRMX supplies three interface procedure libraries for system calls supplied with the operating system: `rmxifc32.lib` for 32-bit applications (iRMX III only), `rmxifc.lib` for 16-bit compact model applications (iRMX I, II, or III), and `rmxifl.lib` for 16-bit large model applications (iRMX I, II, or III). The versions of `rmxifc.lib` and `rmxifl.lib` for iRMX I must be different from the versions for iRMX II and III because iRMX I runs in real mode and cannot use call gates, but they use the same file names. The

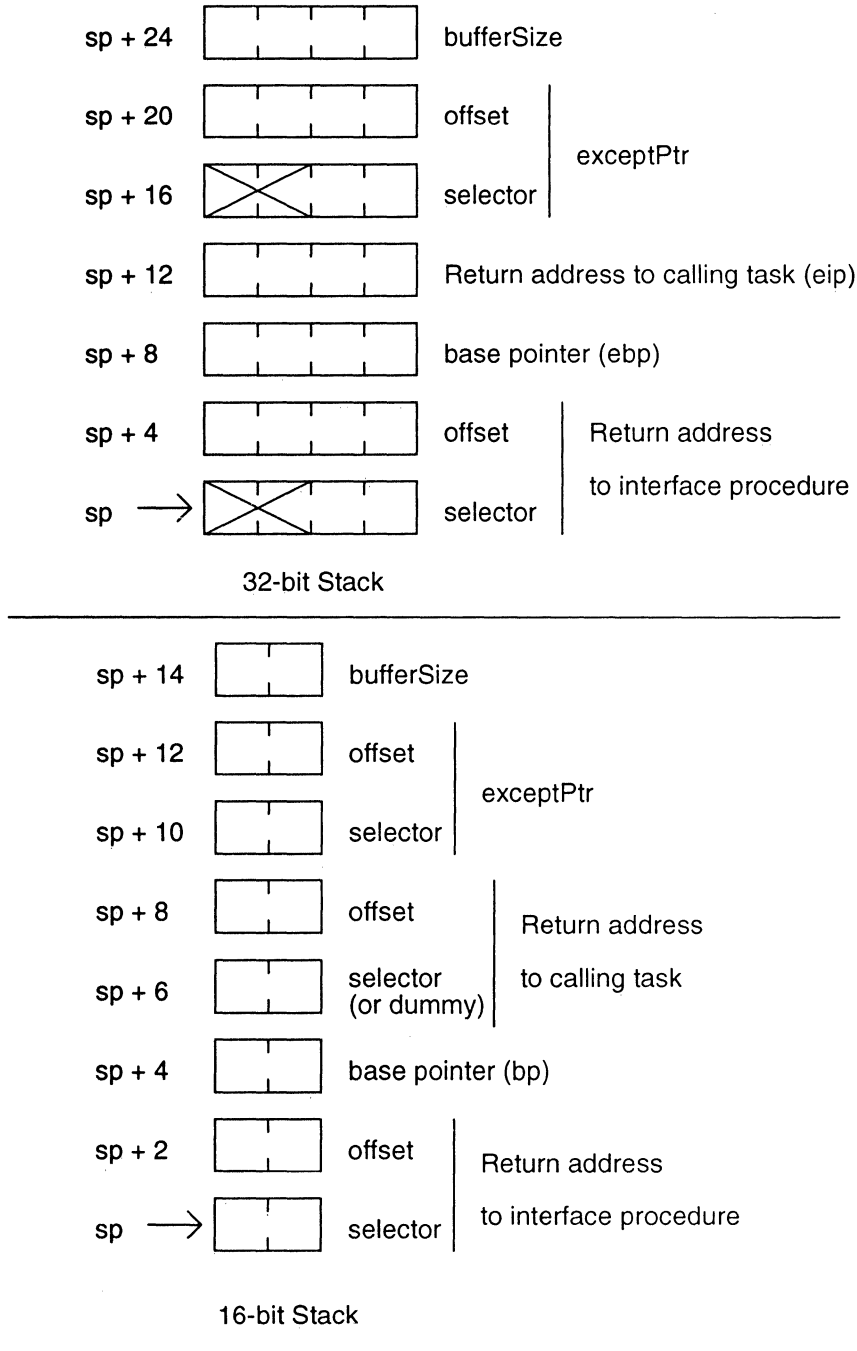


Figure 10.6 Stack frames for 32-bit and 16-bit applications upon entry to the system call *createBB()* procedure.

iRMX II and iRMX III versions of `rmxifc.lib` are identical to each other, as are the iRMX II and iRMX III versions of `rmxifl.lib`.

The names that applications use to access the example system calls are determined by the names given to the procedures in this module, not by the names of the procedures that actually implement the system calls. Thus, the name of the new system call to create a bounded buffer is *qccreateboundedbuffer()*, as defined in `bbifc32.asm` (Figure 10.6), rather than *bbCreate()*, as defined in `bbsyscal.plm` or `bbsyscal.c` (Figure 10.2 or 10.4). iRMX-supplied system call names start with *rq*, often followed by a letter indicating the layer of the operating system that supports the call. *qc* was chosen to start the example system call names just to differentiate them from the iRMX-supplied calls.

The first issue an interface procedure must deal with is to ensure the stack frame appears identical, whether the application made a near call or a far call to the system call. This issue is not applicable for 32-bit code because the interface procedure is always reached from a near call. (Remember that 32-bit compilers treat the compact and large models identically.) For 16-bit code, the interface procedure for the compact model must push a dummy 16-bit word onto the stack to take the place of the `cs` register value that a far call would have placed there. Once this has been done, the caller's parameters are at a fixed offset into the stack (four bytes), regardless of the model of compilation. The procedure prologue generated for 16-bit code would then push the current stack frame pointer register (`bp`) onto the stack and load it with the current top of stack pointer (`sp`). For the interface procedures presented here, the `ebp` and `esp` registers are the corresponding 32-bit registers. A system call procedure that accepts calls from either 16- or 32-bit applications must deal with the difference in the number of bytes pushed onto the stack by the prologue for the two different types of code, as discussed shortly.

The actual call to the system subroutine is then made using the appropriate call gate number as the selector part of a far `call` instruction. The instruction to call a call gate is the same as the instruction to call any far procedure, except that the value of the offset part of the subroutine address is irrelevant because the actual pointer to the subroutine's code is taken from the call gate itself. The interface procedures in `bbifc32` use a macro named *call_g()* to generate a far call with an offset of 0 and a selector formed by multiplying the gate number by 8.

Recall from chapter 5 that a selector has bits 3 through 15 (the left-most 13 bits) set to the index of the descriptor table slot to be accessed, bit 2 selects the GDT or the LDT, and bits 0 through 1 (the right-most two bits) specify the requested privilege level for the gate access. Multiplying the gate number by eight creates a selector in this format for the GDT with privilege level 0.

When the system call returns to the interface procedure, it must deal with the condition code value returned. By convention, all iRMX system

calls return their condition code value in the cx register. If that register is nonzero when the call returns to the interface procedure, it calls the procedure *rqerror()*, which makes another system call, *rqsignalexception()*, which in turn either calls the exception handler for the task or returns to the interface procedure if the task is handling exceptions in-line. The *rqerror()* procedure is in the interface libraries supplied with iRMX, although users could substitute their own code for the system-supplied version simply by listing an appropriate object module before the interface library on the binder's input file list.

If the exception handler does not delete the job (because there was no error and *rqerror()* was not called, or because *rqerror()* was called but exceptions are being handled in-line, or because the application has provided an exception handler that does not delete the job) the interface procedure stores the condition code value from cx into the caller's condition code parameter, cleans up the stack frame (16-bit, compact-model interface procedures must drop the dummy cs register that was pushed onto the stack before the prologue), and returns to the caller. The iRMX system call conventions also specify that nonzero condition code values are accompanied by an index of the parameter causing the problem in register dx if possible. Parameters are numbered from left to right starting at 1 for this value. If you decide to write your own exception handler procedure, it should be coded to receive these two values, plus two more words, as arguments. This information is also useful when debugging an application program. If you single step into an interface procedure for a system call that is causing a problem, you can examine the registers on the next instruction after the call to the call gate to see the value in the Dx register.

If the value in the cx register is 0 (E_OK), the value is simply stored in the word pointed to by the caller's last argument. The *les* instruction in the sample interface procedures loads the selector portion of the caller's exceptPtr argument into the es register and the offset portion into the ebx register, and the following instruction stores the condition code value. Unlike their 16-bit counterparts, PLM-386 and iC-386 compilers assume that the es and ds registers always contain the same values, and the sample interface procedures make sure that this is the case before returning to the caller.

Calculating the number of bytes to drop from the caller's stack on the *ret* instruction must also be done differently for 16-bit and 32-bit interface procedures. The calculation requires full understanding of how the compilers use the task's stack for passing parameters, which is covered in the next section. You might refer back to the *ret* instructions in the sample interface procedures after reading the following material to verify that the code matches your understanding of how the stack is used.

System call procedures that return function values will do so in the processor's registers, and interface procedures do not need to be concerned with this issue other than to avoid destroying the value accidentally. For

32-bit code, a value is returned in the eax register and possibly the edx. For 16-bit code, the ax, bx, and dx registers can be used for return values. Setting these registers is the responsibility of the exit procedures for the system call, and is discussed further in section 10.3.5.

10.3.3 Receiving parameters in the system call procedure

Although system call procedures, especially for user-written system calls, are normally written in PLM or C, the system call programmer must have a good understanding of the conventions used by the compilers for passing arguments on the stack. First, the compilers enforce a notion of a 16-bit or 32-bit stack different from the microprocessor's rules for operating on the stack. For a 16-bit stack, the microprocessor always pushes or pops 16-bit values. You cannot push or pop a single byte, and pushing or popping a pointer or a 32-bit value involves pushing or popping two 16-bit values. The 16-bit compilers use this same model for passing and receiving parameters, so this issue is minor.

For a 32-bit stack, however, the microprocessor and the compilers use slightly different models. The microprocessor still works with 16-bit units. You still cannot push or pop a single byte, and pushing a 16-bit value, a 32-bit value, or a 64-bit value pushes 2, 4, or 8 bytes, respectively. If you push a segment register, the microprocessor pads it from 2 bytes to 4. The compilers use a 32-bit modulus for all values passed as parameters on a 32-bit stack. Whether you pass a 1-, 2-, or 4-byte value as an argument, it occupies 4 bytes on the stack, and the compiled code in a subroutine automatically ignores the unused bytes of any value passed. Far pointers are passed as a four-byte offset and a four-byte value containing the two-byte selector and two unused bytes.

Figure 10.7 shows the structure of a 16-bit stack and a 32-bit stack when the interface procedure has executed its far call to *createBB()* through call gate 440. As the figure shows, the caller's parameters, the stack frame pointer pushed by the interface procedure, and the return address to the interface procedure all occupy different amounts of stack space for the two types of stack. The 16-bit stack is the same for both the compact and large models of compilation, but is significantly different from the 32-bit stack. Here, full credit is given to the Intel engineers who developed iRMX III system call procedures that accept either 16-bit or 32-bit stacks, and proceed to the simpler problem of developing system call procedures that work only for 32-bit stacks.

The two procedures in this module are coded as far procedures even though they are compiled using the compact memory segmentation model. They are compiled using the compact model to not incur unnecessary code segment changes when calling other procedures with which they are bound. The exit procedures are such procedures in the example. The procedures must be coded as far procedures so the compiler will generate code

Figure 10.7 Assembly language interface procedures for 32-bit implementation of *qccreateboundedbuffer()* and *qcdeleteboundedbuffer()* called through call gates 440 and 441.

```
name    bbifc32    ; 32-bit interface procedures for Bounded Buffer
;
;   This is the interface module for bounded buffer system calls.
;   The following system calls are supported:
;
;       TOKEN
;       qccreateboundedbuffer (DWORD bbSize, WORD far *exceptPtr);
;
;       void
;       qcdeleteboundedbuffer (TOKEN bbOSE, WORD far *exceptPtr);
;
;   ------------------------------------------------------------------
;
;   First define a macro to generate call instructions that reference
;   call gates.  The assembler does not provide a codemacro for this
;   instruction.
;
$genonly
%*define(call_g(arg))
(
            db      9Ah     ; op code for far call
            dd      0       ; offset is ignored for call gates
            dw      %arg*8) ; for GDT, Privilege Level 0

;
;   Set up the code segment
;
code    segment     er public

        extrn       rqerror: near


; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;   qccreateboundedbuffer interface procedure
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

qccreateboundedbuffer   proc near
            public      qccreateboundedbuffer

;   Procedure prologue and call to the system call procedure
;   -------------------------------------------------------
;
            push    ebp
            mov     ebp,esp

            %call_g(440)

;   Return here from sys_exit procedure - check for errors
;   ------------------------------------------------------
;
            and     cx,cx               ;Test condition code
            jz      to_app_c            ;Zero is E_OK

            call    rqerror             ;Possible call to task's EH
;   If rqerror returns, it means that the application is handling errors
;   in-line, so we return to it after storing the error code.
```

Figure 10.7 (*Continued*)

```
;   ------------------------------------------------------------------
;
to_app_c:
            les     ebx,[ebp+08h]           ;Get app's last parameter
            mov     es: word ptr [ebx],cx   ;Store condition code
            mov     di,ds                   ;Be sure es == ds
            mov     es,di                   ; for 32-bit code
            pop     ebp                 ;Epilogue code
            ret     12                  ;Drop one dword and one pointer

qccreateboundedbuffer   endp

; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;   deleteboundedbuffer interface procedure
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

qcdeleteboundedbuffer   proc near
            public      qcdeleteboundedbuffer

;   Procedure prologue and call to the system call procedure
;   -------------------------------------------------------
;
            push    ebp
            mov     ebp,esp

            %call_g(441)

;   Return here from sys_exit procedure - check for errors
;   ------------------------------------------------------
;
            and     cx,cx               ;Test error code
            jz      to_app_d            ;Zero is E$OK

            call    rqerror             ;Possible call to task's EH

;   If rqerror returns, it means that the application is handling errors
;   in-line, so return to it after storing the error code.
;   ------------------------------------------------------------------
;
to_app_d:
            les     ebx,[ebp+08h]           ;Get app's last parameter
            mov     es: word ptr [ebx],cx   ;Store condition code
            mov     di,ds                   ;Be sure es == ds
            mov     es,di                   ; for 32-bit code
            pop     ebp                 ;Epilogue code
            ret     12                  ;Drop one token and one pointer

qcdeleteboundedbuffer   endp

code        ends
end
```

to load the procedures' ds register during the procedure prologue. In general, these procedures are not bound to the application task's code, so they must use their own code and data segments. For the PLM version, the procedures are made far by coding the $compact (exports createBB deleteBB) compiler control before the first DO block. For C, it is done simply by declaring the procedures to be far.

To refer to the parameters being passed to it by the application task, the procedures must be coded to account for the values pushed onto the stack by the interface procedure. The compiler automatically knows about the far return address and the changes it makes to the stack during the procedure prologue, so it is just a matter of accounting for the base pointer value and application task's return address, as shown by the two parameters, application_eip and application_ebp, declared after the parameters of interest in the two procedures. With these declarations in place, the compiler generates the proper offsets into the stack to access the application's parameters.

To put some perspective on the sample code, the following is a review of how the different versions of the interface procedures available for iRMX-supplied system calls work to provide a consistent interface to the system call procedures.

32-bit code (rmxifc32.lib). The interface procedure pushes a 32-bit flag value (-1) onto the stack to signal a call from a 32-bit application before pushing ebp and making the system call. The system call procedure contains an assembly language prologue that tests for the flag value on the stack and branches to the code that references parameters from 32-bit applications.

16-bit compact (rmxifc.lib). The interface procedure creates a 16-bit dummy offset value for the return address on the stack by pushing the bp register, updates the stack frame pointer by pushing the bp register again, and makes the system call. The system call's prologue finds two copies of the bp register, which never have the value -1, in place of the flag value, and knows to treat the application's part of the stack as 16-bit values. The system call code itself is 32-bit code, so the stack is assumed to contain 32-bit values, but this assumption can be overridden by referencing 16-bit registers and operands in the code. The system call, however, must use the proper offsets into the stack to obtain the operands.

16-bit large (rmxifl.lib). The caller's return address on the stack already contains a 16-bit offset value, so the interface procedure just updates the stack frame pointer and makes the system call. The system call procedure does not need to know whether the application is using the compact or large model, only that it is passing 16-bit values on the stack rather than 32-bit values.

Note that the interface procedures for 16-bit code are exactly the same for iRMX II and III. Only the interface procedures for 32-bit applications and the system call procedures themselves are different for iRMX III.

If you develop a system call to be used with 32-bit applications only (which implies running on iRMX III or iRMX for Windows only), the sample code provides a slightly more efficient protocol for passing parameters to system call procedures. The sample code does not bother with the flag value on the stack and thus eliminates the need for an assembly language prologue for the system call procedure to test the flag value and determine the word size of the caller's stack.

10.3.4 Design of a system call procedure

A procedure that will operate as a system call must deal with the following issues:

- Operation in a multitasking environment.
- Memory and object management.
- Condition code and exception management.

All these concerns are examined in the sample code in the *createBB()* and *deleteBB()* procedures of Figures 10.2 and 10.4, although memory and object management are covered in section 10.4.2. Looking at the structure of these procedures, it is important to remember that they are executed by an application task, so they must take care to maintain that task's environment and work properly, even if the calling task is preempted at any point during the execution of the system call procedure.

Thus, the first consideration for a multitasking environment is to ensure that the procedure maintains separate copies of each caller's local variables so that different tasks can call the system call procedures concurrently. For C programs, this is done automatically because local variables are allocated on the caller's stack by default (unless declared `static`). For PLM programs, however, it is necessary to declare the procedures `reentrant` to achieve the same effect. Remember, each iRMX task has its own stack segment.

A second consideration for operating in a multitasking environment is not illustrated in the code being examined in this chapter. The issue is managing global state information needed by the OSE. In this case, there is no such information; each bounded buffer is created and operated independently of all others. Another OSE might need global state information, however. For example, the BIOS allows only a single I/O connection to be made to a device at a time, so it must maintain knowledge about what devices have I/O connections globally — across all the I/O connection objects that it manages. In a multitasking operating system such as iRMX, more

than one task could make calls to an OSE that involve examining or modifying such global state information concurrently. It is crucial for this information to be protected through standard mutual exclusion mechanisms, such as semaphores and regions, to ensure correct operation of the OSE.

The other matter to address is how to handle exceptions that occur during execution of the system call procedures. One strategy is to not change the calling task's exception handling mode within the system call. Leaving exception handling unchanged, the system call procedure is aborted if an exception is encountered and the application has elected to use an exception handler that deletes the task or job when an exception occurs. However, the system call must be coded to test the condition code after each system call in case the application has elected the in-line mode for handling exceptions. In this case, each system call made from within a system call procedure would be followed by a test for nonzero `Status`, and a return of that value to the caller through *sys_exit_e()*.

The strategy adopted in the sample code for *bbCreate()* is to force the application to do in-line exception handling while it is in the system call, restoring the original exception handling mode of the calling task before exiting. This way, our system call procedure can complete successfully even if an iRMX system call that is not essential to the operation of our code fails. Our *bbDelete()* procedure, on the other hand, makes only one iRMX system call (*rqsendmessage()* to send the token for the composite object to the deletion mailbox), which can fail only if the type manager is not properly installed. So *bbDelete()* does not bother to check for an exception when making this iRMX call; it just passes the status from *rqsendmessage()* back to the caller as its own condition code value.

Forcing in-line exception handling also allows system call procedures to recover cleanly if they encounter an iRMX exception in the middle of processing. For example, *bbCreate()* deletes all component objects if it is unable to create any one of them successfully. This technique also allows the system call to pass back condition code values that might be more meaningful to an application than whatever condition code happens to be returned by a nested system call. For example, *qccreateboundedbuffer()* is guaranteed to return one of only four condition code values, E_OK, E_MEM if not enough memory exists to create the buffer object, E_LIMIT if the calling task's job has reached its object limit, or E_CONTEXT if any other error caused the call to fail.

10.3.5 Exit procedures

Once a system call procedure completes its work, it must return its condition code to the application and, if it is a function, a return value as well. The calling conventions for iRMX require that the condition code and pa-

rameter number causing an error, if any, are returned in registers cx and dx, and that function values from 8 to 32 bits long are returned in register eax.

If a function returns a pointer or a 64-bit doubleword, the selector part of the pointer or the high-order half of the doubleword is returned in register edx, and the offset or low-order half is returned in eax. The PLM-386 compiler returns function values using this convention automatically, as does the C-386 compiler for functions declared with the `fixedparams` pragma, as all iRMX system call interface procedures are. Setting the cx and dx registers, however, must be done in assembly language, and cannot be done in PLM or C. Instead of returning directly to the interface procedure, system calls return to the interface procedure indirectly through one of four exit procedures given in Figure 10.5. These exit procedures are:

- *sys_exit_n()*
- *sys_exit_v()*
- *sys_exit_d()*
- *sys_exit_e()*

sys_exit_n() is called for a normal exit by a system call that returns no value and completes normally. This procedure sets register cx to zero, the E_OK condition code. This procedure is called from *bbDelete()* when no error occurs.

sys_exit_v() is called for a normal exit from a system call that returns a value in register eax. This procedure loads the return value, which the system call passes to it on the stack, into register eax, and sets cx to zero. This procedure is called from *bbCreate()* to return the token for a new bounded buffer composite object.

sys_exit_d() is called for a normal exit from a system call that returns a far pointer or a doubleword in registers edx and eax. The same code is used for returning *far* pointers as for returning doublewords, so the alias *sys_exit_p()* is provided for this function. Neither procedure is actually used by the code in this chapter.

sys_exit_e() is called to return a nonzero condition code value and parameter number for system calls that fail. It is called from various places in both system calls.

Exit procedures do not return to the system call procedures that call them. After setting the processor's registers, exit procedures return to the interface procedure that called the system call procedure. Thus, the exit procedures first add 4 to the stack pointer to eliminate the near return address used to return to the system call procedure. They then pop the ds register to restore the interface procedures' data segment register, and then use the *leave* instruction to drop any local variables the system call procedure left on the stack and to pop the ebp register value that was pushed by the compiler-generated prologue code for the system call procedure. The

form of this code is predicated on knowing what code compilers insert as the prologue for any *far* procedure. The various exit procedures then make *far* returns to the interface procedure.

Exit procedures for 16-bit applications must be coded differently to reflect the differences in calling conventions and stack usage compared to 32-bit code. A system call that serves both 16- and 32-bit applications must use two different sets of exit procedures to accommodate these differences. The register conventions for 16-bit and 32-bit applications are fairly easy to summarize:

First, 32-bit functions return all 1-, 2-, or 4-byte values in register `eax`. 64-bit values are returned with the low-order half in `eax` and the high-order half in register `edx`. Pointers are returned with the offset part in `eax` and the selector part in register `dx`.

Secondly, 16-bit functions return 8- and 16-bit values in register `ax`, and use `dx` for the high-order half of 32 bit values. However, they return the offset part of a pointer in `bx` instead of `dx` and return the selector part of a far pointer in `es` instead of `dx`.

10.4 Adding a Type Manager

The good news about adding a type manager to an iRMX system is that it doesn't involve any assembly language coding! Five system calls are involved in setting up a type manager and using it, and these are described in this section. In chapter 6 you saw that one of the primitive object types supported by the Nucleus is called an operating system extension, or OSE object. An OSE object is a meta-object: it defines a new object type for the system. Two of the five system calls described here are used to create and delete an OSE object. The other three calls are used to create, modify, and delete objects of the new OSE type, *i.e.*, composite objects.

10.4.1 Creating an extension

Before looking at the system calls for creating and deleting new object types, object type codes must be considered. Every type of iRMX object has a unique type code. For example, the following object type codes are used for the primitive objects managed by the Nucleus:

Type code	Primitive object type
1	Job
2	Task
3	Mailbox
4	Semaphore
5	Region
6	Memory Segment
7	OS Extension
8	Composite Object
10	Buffer Pool

These type codes are used by the Nucleus to verify that objects of the correct types are passed to system calls. This object type checking adds greatly to the robustness of the system and helps detect programming errors early in the development process. The system simply will not let you pass a token for a mailbox to *rqsendunits( )*, for example. Object types 7 and 8 are of particular concern here. For every type of object beyond those listed previously, a type manager must call *rqcreateextension( )* to define the new object type to the system. The following is the function prototype for that call.

```
extern TOKEN
rqcreateextension (          WORD              typeCode,
                             TOKEN             deletionMbx,
                             WORD far *        exceptPtr);
```

The first parameter to this call is a user-selected type code for the new type being created. Intel reserves type codes 0 through 0x7FFF for the extensions they add to iRMX, and users can use any type codes between 0x8000 and 0xFFFF. Similar to the allocation of call gates to user-written system calls, a problem can arise in deciding which OSEs are to use which type codes. Again, some sort of managed registry must be set up if this becomes a problem. The `deletionMbx` parameter is discussed in section 10.4.3. For now, the concern is with type codes.

iRMX has a set of OSEs already defined by various layers of the system. The ones defined at the time of this writing use the following type codes:

Type Code	Extension Object Type
0x0009	Message Port
0x0100	I/O User
0x0101	I/O Connection
0x0300	I/O Job
0x0301	Logical Device

These five object types are managed by different layers of the operating system: Message Port is defined as an extension type by the Nucleus itself (message ports are used for Multibus II message passing), I/O Users and I/O Connections are defined by the BIOS, while I/O Jobs and Logical Devices are defined by the EIOS.

If you pass a valid token for any iRMX object to the *rqgettype( )* system call, you will get back the type code for either one of the primitive object types, one of the extension object types listed previously, or the value of the `typeCode` parameter of the *rqcreateextension( )* system call that defined a user-developed extension type. Objects created by a type manager for an OSE actually have two type codes: the extension type code for the particular extension and the primitive type code 8, indicating that the object is a composite object. The Nucleus keeps the primitive type code for each ob-

ject in a hidden data structure known as the *canonical* part of the object. When the Nucleus encounters a composite object (type code 8), it knows to look for the actual type code in a data structure called the *base segment* for the object. As shown in Figure 10.8, this base segment contains the extension type code for the object and a list of tokens for the component objects that constitute the composite object. The component objects that constitute a composite object can be either Nucleus objects or other composite objects.

The token returned by *rqcreateextension( )* is called a *license* for the new object type in the iRMX documentation. The idea is that the value of this token, `bbOSE` in the sample code, is made available to the routines that compose the type manager, but is not available to other code. The iRMX system calls used to create and manipulate composite objects all require this token as one of their parameters. This strategy helps encapsulate composite objects by making it impossible for a task to use system calls to examine or modify a composite object without a copy of the license to do so.[6]

10.4.2 Managing composite objects

Once an extension has been created, applications can create composite objects that are instances of the new type by calling *rqcreatecomposite( )*. This function is called from within the *create* system call provided by the type

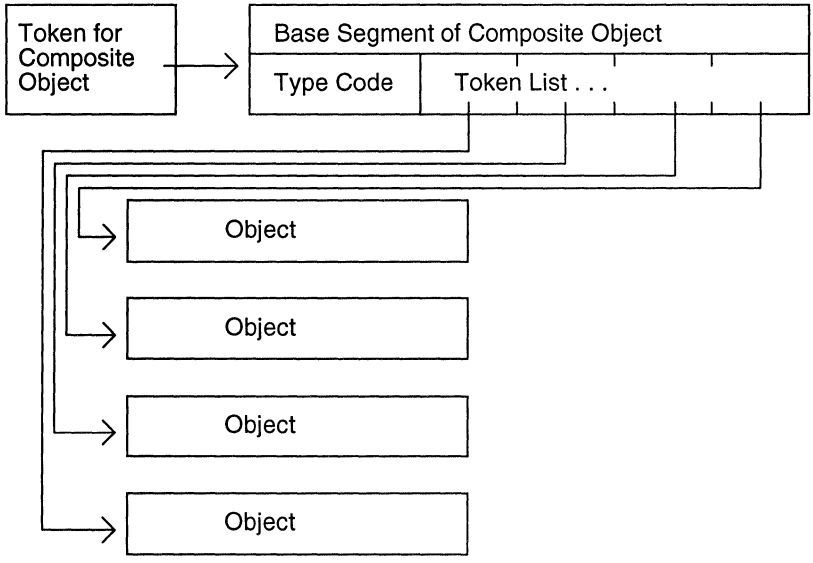


Figure 10.8 Structure of a composite object.

[6]A task could cheat and examine or modify the contents of a composite's base segment directly, but it could not use *rqinspectcomposite( )* or *rqaltercomposite( )* to do so.

manager. The sample code shows a call to this function in the system call procedure for *qccreateboundedbuffer( )*, procedure *bbCreate( )* in Figure 10.2 or 10.4. The function prototype is the following:

```
extern TOKEN
rqcreatecomposite (       TOKEN                       extensionTkn,
                          TOKENLISTSTRUCT far *       tokenListPtr,
                          WORD far *                  exceptPtr);
```

The first parameter is the token returned by *rqcreateextension( )* to identify the object type of the composite object to be created. `TOKENLIST-STRUCT` consists of a count of the number of objects that the composite can contain, followed by a counted list of tokens for objects that are to constitute the initial contents of the object. That is, it starts with a word telling the maximum number of tokens the object can contain, followed by a count of how many tokens are actually being initialized as the object is created. The sample *bbCreate( )* function creates composites that can contain five objects and initializes all five of them: a segment for housekeeping information, a segment for the buffer itself, plus two semaphores and a region for controlling access to the object. It is acceptable to initialize slots in `tokenList` with null selectors to act as place holders, although this is not shown in the sample code.

Once a composite object has been created, type manager functions can change the object's token list by calling *rqaltercomposite( )*:

```
extern void
rqaltercomposite (        TOKEN               extensionTkn,
                          TOKEN               compositeTkn,
                          WORD                componentIndex,
                          TOKEN               replacingObject,
                          WORD far *          exceptionPtr);
```

In principle, the value of the `extensionTkn` parameter is implied by the `compositeTkn` parameter to this call. Requiring both tokens helps enforce object encapsulation, as described previously. The other two parameters for this system call are the index into the `tokenList` for the token to be changed, with one being the first element of the list, and the new token to be stored in `tokenList` for the object. The value for `replacing-Object` can be a null selector if you need to delete a component object from a composite.

When a composite has been created, applications can call type manager procedures with the token for a composite as one of the parameters, normally the first. The type manager procedure gets a copy of the `tokenList` for the composite to be manipulated by calling *rqinspectcomposite( )*:

```
extern TOKEN
rqinspectcomposite (      TOKEN                       extensionTkn,
                          TOKEN                       compositeTkn,
                          TOKENLISTSTRUCT far *       tokenList,
                          WORD far *                  exceptPtr);
```

As with *rqaltercomposite()*, the caller must supply both the license for the extension type and the token for the particular composite to be inspected. For this call, the first word of the `tokenList` is a count of the number of tokens the caller is willing to receive. You can elect to look at just the first few tokens of a large composite object by controlling the value of this word.

The sample code calls *rqinspectcomposite()* from within the deletion task's procedure in the `bbmanage` module. The code illustrates how to access component objects that are part of a composite, in this case to delete them. The system call to add bytes to a bounded buffer would use the following algorithm:

1. Inspect the composite.
2. Receive a unit from `tokenList[2]`, the free space counting semaphore.
3. Receive control from `tokenList[4]`, the critical region for the buffer.
4. Store the byte in the `tokenList[3]` segment, indexed by the nextPut value in the `tokenList[0]` segment.
5. Update `nextPut` by one, modulo `bufSize`.
6. Release the region.
7. Send a unit to `tokenList[1]`, the occupied space counting semaphore.

Similarly, the following algorithm would be used by the system call that removes bytes from a bounded buffer:

1. Inspect the composite.
2. Receive a unit from `tokenList[1]`, the occupied space counting semaphore.
3. Receive control from `tokenList[4]`, the critical region for the buffer.
4. Copy the byte in the `tokenList[3]` segment, indexed by the nextGet value in the `tokenList[0]` segment.
5. Update `nextGet` by one, modulo `bufSize`.
6. Release the region.
7. Send a unit to `tokenList[2]`, the free space counting semaphore.

These system calls modify the components of the composite object, but do not modify a composite object itself. That is, they would not call *rqaltercomposite()*.

10.4.3 Deleting composites and extensions

There are three distinct situations in which a composite object is deleted, as shown in Figure 10.9.

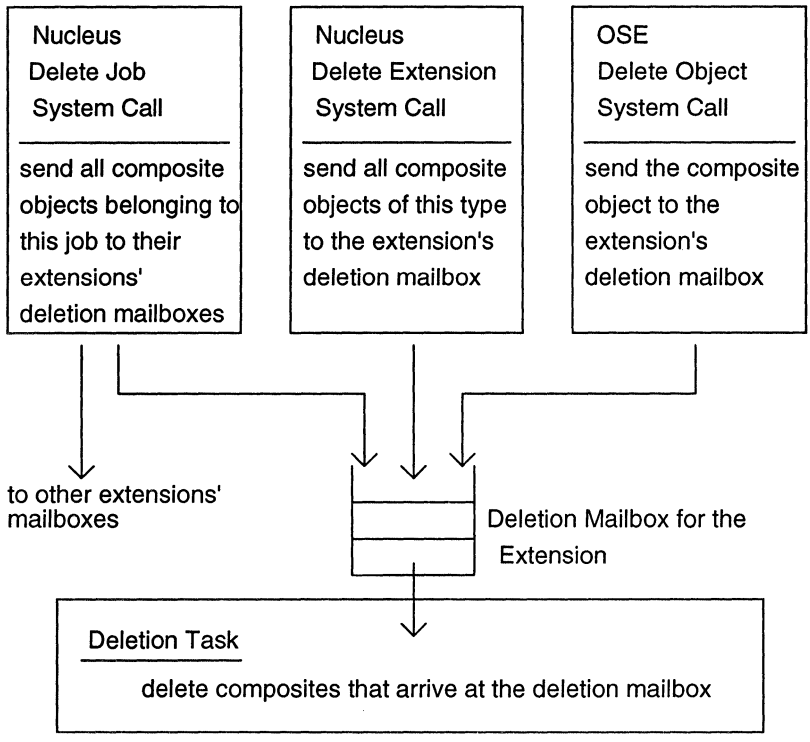


Figure 10.9 Three ways to delete a composite.

The first situation is if a job that owns a composite object terminates. If the type manager has established a deletion mailbox for the extension, the Nucleus automatically sends tokens for such objects to the deletion mailbox as part of its processing of the *rqdeletejob( )* system call. If no deletion mailbox exists, the Nucleus deletes the object itself.

The second situation is if the type manager calls *rqdeleteextension( )* to delete an OSE. The Nucleus deletes all composites of the OSE type before deleting the OSE. If there is a deletion mailbox, the composites are sent there for deletion. Otherwise, they are simply deleted.

Third, the type manager for the extension can provide a system call for deleting composites, and an application task makes this system call.

The second situation is uncommon. Normally, a type manager is installed when the operating system initializes, and never deletes the OSE object for the type it is managing. This case is important, however, when a type manager is being developed and debugged as an HI command. For example, the sample type manager, when the `bbmanage` module is compiled with the `HIcmd` symbol set, creates two bounded buffer composite objects, calls *qcdeleteboundedbuffer( )* to delete one of them, then calls *rqdeleteextension( )*, and exits the job. When the task calls *rqdeleteextension( )*, tokens

for all existing bounded buffer composites are automatically sent to the deletion mailbox. One such object still has not yet been deleted, so that object is sent to the deletion mailbox at this time.

If the sample program had not called *rqdeleteextension( )*, the same token would have been sent to the deletion mailbox as part of the processing of the *rqexitiojob( )* system call, which calls *rqdeletejob( )*. This situation would have been an example of the second situation. The call to *rqexitiojob( )* would have failed in this case, however, because the job still owned an extension object and thus could not be deleted.

When a task calls *rqdeletejob( )* or *rqdeleteextension( )*, it deletes composite objects sequentially. That is, instead of just sending all the tokens for all the composite objects to be deleted to their respective deletion mailboxes and letting them queue up for the deletion tasks to process at their leisure, these routines wait for positive acknowledgment that each composite object has actually been deleted before proceeding to delete the next object belonging to the job or the extension. This strategy allows type managers to delete composite objects that contain other composites. The Nucleus guarantees that composites will be deleted in a most-recently created sequence so that a deletion task can count on being able to access component objects that constitute a composite even if those component objects are slated for deletion during processing of the same *delete* system call.

The preceding rule deals with two different issues. First, the deletion tasks for different type managers could be put into a race condition resulting in indeterminate system behavior if two composite objects of different types, each containing a component of the other type, are sent to their deletion mailboxes at the same time by *rqdeletejob( )*. The rule guarantees that a deletion task will be able to access component objects of any type provided they were created before the composite object itself. By extension, a deletion task can safely access component objects of the same type as the composite itself, provided such components are created before their containing composite.

The code for *rqdeletejob( )* also follows the rule of deleting all primitive objects only after deleting all composite objects, regardless of when the primitive objects were created. The considerations concerning which component objects can be accessed by a deletion task really do apply only to composite components, not primitive components.

What should the deletion task do when it receives a token for an object to be deleted? One essential piece of business is to call *rqdeletecomposite( )*:

```
extern void
rqdeletecomposite (          TOKEN              extensionTkn,
                             TOKEN              compositeTkn,
                             WORD far *         exceptionPtr);
```

This call is essential if `compositeTkn` was sent to the deletion mailbox by *rqdeletejob( )* or *rqdeleteextension( )* because this call is what provides the

positive acknowledgment that allows the Nucleus to continue processing the call to *rqdeletejob()* or *rqdeleteextension()*.

But what if the `compositeTkn` was sent by the type manager's *delete()* system call? Aside from the fact that there is no reason that a deletion task would want to avoid making this call, there are three reasons for an application to delete an object: to make the object unavailable for access, to reclaim the memory used by the object, and to free the GDT slot used by the composite. Calling *rqdeletecomposite()* satisfies all three requirements.

Deleting a composite object does not delete the component objects that constitute the object. Doing so, which the sample deletion task does, is generally important, but not universally so. Deleting the composite objects prevents what are known as *memory leaks*, which can be a significant issue for long-running applications that create and delete objects dynamically during their lifetimes. For example, a network server job might create and delete composite objects as client programs establish and break communication channels. If the composites for each communication channel are not deleted, the server job gradually consumes more and more of its memory pool (and GDT slots) until it can no longer handle new client requests.

Sometimes, it is actually inappropriate to delete component objects, namely when the same component object is contained in multiple composite objects. The deletion task would need to be coded to handle such situations appropriately, taking care to manage the global state information implied by this situation properly. (See section 10.3.4.)

The Nucleus enforces its own rule in this regard: no object can be deleted if it belongs to a job for which *rqdeletejob()* is being processed. Our bounded buffer type manager would encounter this situation if it is compiled with `HIcmd` defined and if the call to *rqdeleteextension()* were omitted. In that case, `a_buf` would be sent to the deletion mailbox when the initial task called *rqexitiojob()*, and all the *delete()* system calls executed by the deletion task would fail. The example deletion task simply ignores any such errors. There is no problem because the component objects are deleted automatically as the *rqdeletejob()* system call proceeds.

The bounded buffer deletion task illustrates one more important concept in the design of robust system calls, which is the consideration of the interactions among multiple tasks that might access a single composite concurrently. It is possible that a token will arrive at the deletion mailbox for a composite object already in use by some other task than the one that called *qcdeleteboundedbuffer()*. No problem exists if another task has a copy of the token for the object being deleted because, once the deletion task calls *rqdeletecomposite()*, any task that presents the token for that composite to a system call will fail with an E_EXIST condition code.

The problem is more subtle: if a task uses the token for a bounded buffer, to get a byte from the buffer, for example, then enters the region for the buffer, and is at that point preempted by another task that deletes the

buffer, what happens to the task trying to get a byte from the buffer? The answer is that the task completes its call successfully because the deletion task deletes the region for the buffer before deleting any of the other components of the buffer. Since *rqdeleteregion( )* automatically waits for any task that has entered the region to leave it (the task that is getting a byte from the buffer in this case), the situation is handled correctly.

A decision was made in the design of the example type manager that should be made explicit before leaving this topic. There seems to be two equally reasonable ways to handle object deletion. The way the example handled it was to have the *deleteBB( )* system call send objects to the deletion task and have the deletion task perform the actual deletion process. Another design might have been to have *deleteBB( )* perform the deletion, and have the deletion task call *qcdeleteboundedbuffer( )* for each object that arrives at the deletion mailbox. The difference between the two is that *deleteBB( )* would have to be designed to handle concurrent calls from different tasks, whereas the deletion task is a single thread of execution. A call to *rqdeleteregion( )* causes deadlock if it is called by two tasks for the same region at the same time, so *deleteBB( )* would have had to be coded to serialize calls to *rqdeleteregion( )* in the alternate design, probably by adding a binary semaphore to the bounded buffer composite structure. This overhead is circumvented by using the deletion task which, as a single thread of execution, automatically serializes all delete operations.

Chapter

11

iRMX Network Programming

11.1 Overview

Networking support is an important feature of the iRMX operating system. Networking was introduced in chapter 2 where the system's support for networked access to remote files was discussed from a user's perspective. In this chapter, the nature of iRMX networking is covered in more detail so you can develop programs that use the network directly. You will also see how the BIOS' remote file driver uses the network to implement access to networked file systems.

Networking is a key element of the iRMX for Windows support for the Windows Dynamic Data Exchange (DDE) mechanism. The DDE is what allows Windows applications to exchange data with one another, such as a field in a word processing document containing a value extracted from a cell of a spreadsheet. iRMX for Windows allows iRMX applications to use the DDE mechanism to share data with Windows applications and, more significantly, to extend the DDE to operate transparently over a network. With the iRMX for Windows networking software in place on computers running Windows, any mixture of iRMX and Windows applications running on different computers can exchange data the same way that two conventional Windows applications use the DDE without iRMX. The DDE mechanism is discussed in more detail in chapter 12.

11.2 A Network Model

No discussion of networking can begin without mention of the familiar seven-layer reference model for networking known as the Open Systems Interconnection (OSI) Reference Model and promulgated by the International Standards Organization (ISO).[1]

[1]There are those who believe that the ISO purposely chose the name OSI for their model to confuse people with two acronyms that are anagrams of each other.

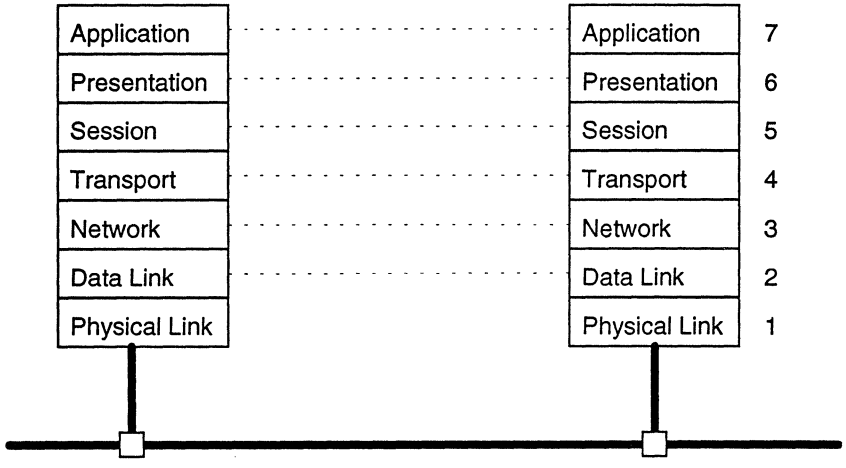


Figure 11.1 ISO seven-layer reference model.

Figure 11.1 shows two systems linked by a network in terms of the OSI model. The physical layer is the medium that connects systems on the network, such as an Ethernet cable. The data link layer is very similar to the hardware device controller that connects a computer to the physical medium, and the network layer can be thought of as the device driver software that acts as the interface between the operating system and the device controller. This characterization of the lower three layers as structures should already be familiar to you from the discussion of device drivers in chapter 9, but is only approximately accurate. Nonetheless, it provides a useful starting point.

Above the network layer, software modules at the corresponding layers on different systems can communicate with each other by passing requests up and down the stack of layers on the local system. For example, an application on one computer that wants to communicate with an application on another computer can do so by using the Presentation layer on the local computer. The Presentation layer passes each request from the Application layer to the Session layer, and so on, until the Physical layer actually transfers the request over the network to the remote system. At the remote computer, the Physical layer receives the request, and passes it up to the Data Link layer, which passes it on up the stack of layers, which repeats until the request is delivered to the appropriate application.

The OSI model allows for a many-to-one relationship between software modules at one layer and the next lower layer. For example, several applications can simultaneously share the same Presentation layer software. At the other end of the stack, a single Ethernet cable might carry packets for Novell, Transmission Control Protocol/Internet Protocol (TCP/IP), and ISO networks. They would all be processed equivalently by the Physical layer, but the Data Link layer manages the differentiation among packet

types to support the three different Network layer modules concurrently. The ISO has published standards for the interfaces among the OSI reference model layers, but networks such as Novell and TCP/IP do not use these standards. The implementations of the lowest layers, however, do support the sharing of common Physical and Data Link hardware and software among these three as well as other types of networks.

Here, by the way, is an example of where the analogy between the lower three layers and devices, device controllers, and device drivers breaks down. One mechanism for supporting multiple networking protocols is called a *packet driver*, which is a device driver that resides between the data link and the network layer in the OSI model. When a Novell packet arrives at the device controller, the packet driver passes it on to the Novell device driver. When an Internet Protocol (IP) packet arrives, the packet driver passes it on to the IP device driver. When an ISO packet arrives, the packet driver passes it on to the ISO device driver. Another mechanism is for the ISO device driver to receive all packets and pass the IP packets on to the IP driver. Both mechanisms are used on iRMX systems, and neither fits perfectly into the ISO model. The packet driver mechanism is used on PC platforms, and the ISO-receives-all mechanism is used in a TCP/IP implementation for iRMX currently under development at Intel. The difference between the two approaches is not terribly significant beyond noting that it is possible for the Data Link layer to support higher-level protocols that do not adhere to ISO standards.[2] This topic is further discussed at the end of this chapter when the Data Link layer services available to iRMX programs are detailed.

A natural break exists in the OSI stack between the Transport layer and the layers above it because the Transport layer is the lowest layer that provides for the reliable exchange of packets between processes running on separate computers. Applications such as file transfer, remote procedure calls (RPC), and E-mail can be built directly on top of a Transport layer implementation. The Internet uses the Transmission Control Protocol (TCP) and IP to implement the layers at the Transport layer and below, and applications such as FTP for file transfer and Telnet for remote login are implemented on top of TCP/IP. Rose (1990) argues that the TCP/IP protocols are more efficient than the corresponding ISO layers, and he has implemented the ISO Development Environment (ISODE) for implementing upper-layer ISO modules on top of the TCP/IP Transport layer interface. iRMX provides a Transport layer that is compatible with the ISO standards, and there is an experimental implementation of TCP/IP available as well, with a complete TCP/IP package, including FTP and Telnet applications, expected to be available from Intel in 1993.

---

[2]To provide a consistent implementation of TCP/IP across various computing platforms, the TCP/IP layer performs all its network processing by making calls to the ISO Data Link layer. On the PC platform, the Data Link layer uses the Packet Driver to perform the actual network data transfers.

11.3 THE iRMX Networking Context

iRMX provides networking support in two software modules. The Transport layer and below are encapsulated in one module, called iNA-960.[3]

Remote file access (the Session layer and above) is provided by a second module, called iRMX-Net. In keeping with the OSI many-to-one relationship among layers, it is possible for other applications besides file access to be built on top of the Transport layer, and the latter part of this chapter is devoted to the concepts involved in developing such applications on top of the Transport layer.

iNA-960 runs either as an iRMX job using the processor running the iRMX operating system itself, or is downloaded to a network controller board that supplies its own processor and memory. The version of iNA-960 that runs in protected mode as an iRMX II or iRMX III job is called iTP-4. (TP-0 and TP-4 are the names of the most commonly used ISO standards for the Transport layer; the digit in the names refers to increasing levels of reliability.) The version of iNA-960 that is downloaded to a network controller board runs in real mode as standalone code and is called iNA-961. (The name iNA will refer to either implementation.)

To provide a consistent interface to applications, all access to iNA is achieved through software that implements the Message Interprocessing Protocol (MIP). If iNA is running as an iRMX job, the MIP software simply passes messages back and forth between the application and iNA using internal procedure calls. If, however, iNA has been downloaded to a separate network controller processor, a version of MIP is used that uses the system bus to pass messages. Depending on the nature of the system bus, this message passing involves using either the message passing facility of the bus (Multibus II) or shared dual-ported memory (Multibus I and AT bus). Although MIP is an interface to the Transport layer, it does not fit into the OSI reference model as a Session layer implementation. Rather than acting as one of several users of the Transport layer, the MIP is simply the unique interface to the Transport layer for all upper-layer software that works, regardless of the implementation of the Transport layer in a particular configuration.

iRMX-Net uses iNA to provide interoperability between iRMX, DOS, Unix, and VAX/VMS file systems. Basically, interoperability means that computers running any of these operating systems can share files and printers, provided they run ISO Transport layer software and Intel's OpenNet software. OpenNet, in turn, acts as an interface between the local computer's I/O system and the network. There are different versions of OpenNet for different operating systems. OpenNet for iRMX is called iRMX-Net, OpenNet for DOS is called MS-Net, OpenNet for VMS is

[3]iNA stands for Intel Network Architecture. The number 960 is an arbitrary part of the name and has nothing to do with the i960 microprocessor.

called VMS-Net, OpenNet for Unix System V is called SV-Net, and OpenNet for Xenix is called Xenix-Net. You will see how iRMX-Net operates in this chapter, which will provide some background for using iNA directly, and you will also see how MS-Net operates in anticipation of the networking discussion under iRMX for Windows.

Before discussing iRMX-Net and MS-Net, you need to be aware of a fundamental difference between typical PC networking systems and iRMX or Unix networking. Because DOS is inherently single-threaded, most PC networks run network servers on one computer and network clients on separate computers. The server computer runs a special operating system dedicated to satisfying network requests made by remote clients for file or device access. When a user operating a client machine needs access to a remote file or printer, his or her local computer communicates with the server computer to send or retrieve the proper data. While the transfer is taking place, the client computer (running DOS) is dedicated to processing the transfer, just as it is normally dedicated to servicing local disk accesses when they occur. The server computer, on the other hand, can normally process several client operations concurrently because of the design of its networking operating system.

Unix and iRMX, however, are inherently multithreaded operating systems. There is no problem for the same computer to simultaneously act as both a server and a client for the network, except perhaps for the additional computing load placed on the user's processor. (This overhead is less when iNA is running on a separate processor located on the network controller board.) Rather than dedicating an entire computer to server operations, all computers on a Unix or iRMX network can concurrently operate as both servers and clients in what is known as a *peer-to-peer* networking relationship. The OSI reference model fits a peer-to-peer network structure well, but does not require that structure. Of course, some computers in a peer-to-peer network might be thought of primarily as servers because of the particular peripherals attached to them, such as printers or large disks, which these computers make available to the other computers on the network. Still, such servers are peers in the sense that they can also act as clients in principle, if not in practice. Peer-to-peer networks such as OpenNet commonly allow systems running different operating systems to work with each other, but they do not require a special operating system just for server operations.

11.3.1 iRMX-Net

Figure 11.2 shows the structure of the networking components of an iRMX-Net system. The iRMX-Net job is a resident program that acts as the intermediary between the iRMX BIOS and the Transport layer interface to the network. When an application program uses an I/O connection that was attached using the Remote file driver, it acts as a file consumer. In

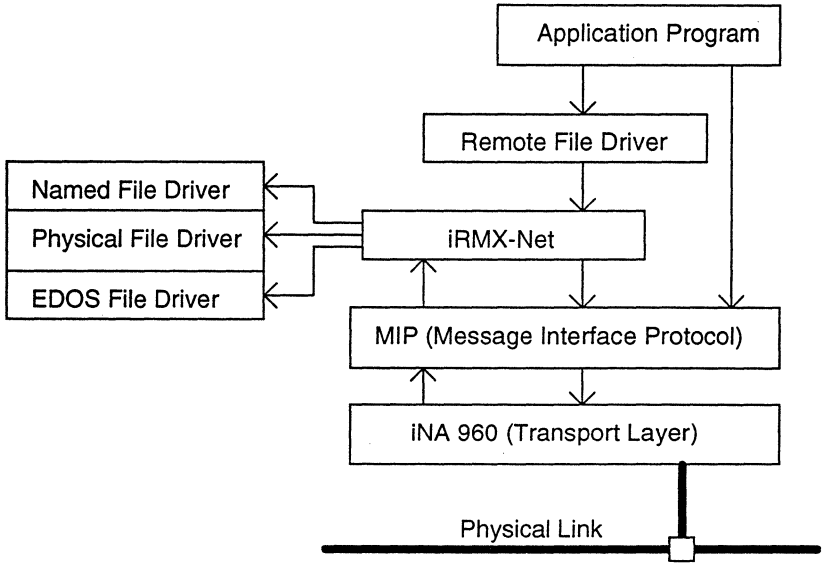


Figure 11.2 Relationships among the networking components of an iRMX system.

this case, the iRMX-Net job acts as the device driver for all I/O operations. It translates each I/O request into an appropriate network request, which it sends to the transport layer software, iNA, using the MIP layer as the interface to iNA. The iRMX-Net (or VMS-Net, or whichever) job running on the remote computer receives the request over the network and acts as the file server for the request by making I/O calls on the remote computer to perform the requested operations on behalf of the client. Figure 11.2 represents this server feature of iRMX-Net by showing connections from iRMX-Net to the other BIOS file drivers of an iRMX system — Physical, Named, and EDOS. Figure 11.2 also shows that iRMX applications can access iNA services directly to implement other data-link and transport-level operations besides file transfers. The direct interface between iRMX applications and iNA is the focus of most of this chapter.

Implicit in the notion of an iRMX-Net file server or file consumer are the issues of user security and network addressing. iRMX-Net builds on the User Definition File (UDF)-based security mechanisms inherent in the BIOS. (The UDF was introduced in the discussion of I/O user objects in chapters 7 and 8.) Because iRMX itself is not designed as a secure operating system, iRMX-Net can use either of two techniques to validate a user's network requests.

The first technique is *client-based protection*, which can be used when the security of an iRMX system can be trusted. In this case, the server assumes that the client machine has verified the authenticity of its users and accepts all requests from a client machine, provided only that the client is listed in the server's `:config:cdf` file, the Consumer Definition File

(CDF) for the server. The command *modcdf* can be used to add the names and passwords of client machines to the server's CDF. The CDF is similar in function to the `hosts.equiv` file on Berkeley Unix systems.

Client-based protection uses the actual user name supplied when an individual logged into a system rather than a user's current ID at the time of a remote file access to reduce the likelihood of problems due to forged user objects. For example, this feature prevents a user from forging Super-user status on an iRMX system and then using the forged identity to obtain unwarranted permissions for a remote system.

The second technique is *server-based protection*, which is used to provide a more secure networking environment. In this scheme, there is no entry in the server's CDF for a client. Every network access initiated by a client (such as issuing an *attachdevice* to a remote system) is accompanied by the name and password supplied when the user logged into the client system, which must match the name and password for the user in the server system's UDF. (The user and group ID numbers do not need to match, but the names and passwords must.)

Network addresses are rather complicated data structures that are discussed later in this chapter. iRMX-Net provides a distributed database that allows programs to obtain network addresses (and other information) using simple names. The database is accessed through a software module called the Name Server (NS). Although the NS is part of iRMX-Net, it can optionally be run either on the iRMX processor or downloaded to the LAN board and accessed independently of the file services provided by iRMX-Net.

Chapter 2 described how to access a file on a remote computer from the command line. That process began with an *attachdevice* command, such as the following:

```
iRMX>attachdevice system1 as 1 remote                    [1]
```

This command establishes `:1:` as the logical name for the virtual root directory on a computer system called `system1`. When this command is issued (or the equivalent system call is executed), the remote file driver routes the operation to the iRMX-Net job to act as the device driver for operations that involve the connection created with the logical name `:1:`. The iRMX-Net job uses the NS to query the network's database to determine the network address of `system1`, and stores that address (or a pointer to it) as part of the connection's internal data structure.

For this operation to succeed, some computer on the network must previously have loaded the network address of `system1` into its portion of the distributed database. For iRMX systems, each computer typically loads its own name and network address into its own part of the database when the operating system initializes. However, other OpenNet systems do not include the NS provided by iRMX-Net. For those systems, one of the iRMX computers can act as the spokesperson for the other computer. For exam-

ple, `systeml`'s portion of the database can include entries both for itself and another machine called `systemu` that is running SV-Net (Unix System V). In this case, `systeml` is said to act as the spokesperson for `systemu`.

11.3.2 MS-Net

From a user's point of view, the most significant difference between MS-Net and the other members of the OpenNet family is that MS-Net does not support peer-to-peer networking. A computer running MS-Net can operate as either a client or a server, but not as both simultaneously. When MS-Net is running as a client, a user can issue DOS commands to access remote disks and printers using the same command line syntax as that used to access local devices. When MS-Net is running as a server, DOS continues to run on the computer, but the local user is prevented from entering any DOS commands except for those that interact with the server itself, such as to shut it down. The reason for this characteristic of MS-Net, as mentioned earlier, is the single-threaded nature of the underlying DOS operating system.

Two features of the implementation of MS-Net are important to mention here: NetBIOS and the Redirector. NetBIOS is a standard interface for DOS software that performs network operations. The NetBIOS Application Programming Interface (API) is accessed by constructing a data structure known as a Network Control Block (NCB) that contains information such as code for the network operation to be performed and values for the parameters needed to perform the operation. A pointer to the NCB is loaded into a register pair (`es:bx`), and the application issues a software *int 5C* instruction to call the NetBIOS software. There is no single NetBIOS software module for DOS. Rather, a user loads a Terminate and Stay Resident (TSR) program that will implement the NetBIOS API as appropriate for the particular network being accessed. For example, *netbios.exe* is the name of a program supplied as part of MS-Net that installs itself as the interrupt handler for level 5C, translating the NetBIOS NCBs into requests to be carried out by iNA software running on the Network Interface Adapter (NIA) installed in the user's PC.

The Redirector is built on the capability available in DOS to work with what could be called virtual device names. For example, even without networking software installed, a DOS user can make disk-drive letters, such as `A:` or `B:`, represent something other than actual physical disk drives. The DOS *assign* command can be used to tell DOS that all I/O operations that name one disk are actually to be performed using a different disk, and the DOS *subst* command lets the user reference arbitrary directories using drive letters as well.

There are at least three ways to implement I/O redirection in DOS. The first way DOS supports network redirection is through a set of system calls

(*int 21*, functions 0x5F02, 0x5F03, and 0x5F04) that allow device names for printers and disk drives to be redirected to the network. Function code 0x5F03 is used to establish such a redirection mapping, 0x5F02 is used to find out what mappings are in effect, and 0x5F04 is used to cancel a redirection mapping.

The second method is for software to chain to the DOS *int 21* vector. In real mode, any program can access the interrupt vector in low memory. A program can therefore install itself as the interrupt handler for interrupt level 21, which is used for DOS system calls. Such a chained software module would examine each I/O request made to DOS to determine whether the request references a device for which the program is providing redirection. If such a request is detected, the program would service it itself; if not, the program would pass the request back to DOS's original *int 21* handler. Novell's network redirector uses this technique to intercept I/O operations that must be processed by its IPX network driver.

The third method is to use the internal DOS interrupt, level 2A, to which a network redirector can chain instead of level 21. MS-Net chains to this level, generating NetBIOS requests on level 5C when it detects network accesses.

You will see in chapter 12 how the standard NetBIOS interface provides support for the DDE mechanism in iRMX for Windows.[4]

You will also see how the notion of a network redirector is added to the iRMX side of the network to provide network access for the Windows DDE mechanism. At this point, the discussion focuses on programming iRMX applications that interact directly with the iNA implementation of the ISO Transport layer.

11.4 Network Mechanisms

Two processes communicate over a network by sending messages to each other. Each message includes the data to be exchanged, along with addressing information that tells each layer of the network stack where the message is to be delivered. This addressing information is prepended to each message by each layer of software, so that a message sent over the network might ultimately look like Figure 11.3. Each layer in the protocol stack prepends its own addressing information as the message is sent down the stack, and, at the other end, each layer of the stack uses the addressing at the head of the message to direct it to the next layer above, removing its own addressing information in the process.

---

[4]When running iNA on an iRMX for Windows system, a DOS program called *pcnet.exe* intercepts NetBIOS requests on level 5C, and generates equivalent requests on level 5B. These requests are received by an iRMX job called *netrdr.job*, which turns these requests into iNA request blocks.

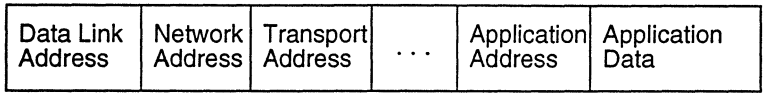


Figure 11.3 Structure of a network message.

The ellipsis (...) in Figure 11.3 is more than a convenience to make the figure more compact. Rather, it indicates that some layers of the protocol stack might not actually be present. For example, you will see how applications can communicate with each other through direct interaction with the Transport layer without going through any Session or Presentation layer at all. Also, note that there is no Physical layer address in the figure. Each message is broadcast to all computers connected to the network, and logic at the Data Link layer (the device controller) recognizes which messages are to be accepted by that computer. The Data Link layer simply ignores all other messages on the network.

iNA supports two types of network addresses, Null2 and ES-IS. The difference is whether the Network layer is to be included in the system or not. If all the computers that need to communicate are on the same subnetwork (connected to the same Ethernet cable, for example), Network layer addressing is not needed, and iNA can be set up to use Null2 addresses, which omits the logic associated with the Network layer. On the other hand, if two computers that need to communicate are on different subnetworks, they can only communicate with the help of a third system connected to both subnetworks. The two computers that need to communicate with each other are called *End Systems* (ES), and the third computer is called an *Intermediate System* (IS), providing the name for the ES-IS address format. The IS accepts packets from all subnetworks and sends the packets out on the subnetwork so they will be delivered to the proper ES. Another name for an IS is an *internetwork router.*

11.4.1 Packets and messages

Figure 11.3 is an abstraction of a network message form, and the actual data structures used by the programmer to build a network message are discussed shortly. First, two related concepts must be covered: packets and connections. The terms *packet* and *message* have been used somewhat loosely and interchangeably throughout this book. All information sent over an Ethernet cable is sent a packet at a time, and the Ethernet specification (IEEE standard number 802.3) specifies the maximum number of bytes (called *octets* by parts of the networking community) a packet can accommodate. If a single message is too large to fit in one packet, it must be divided into smaller chunks that will fit into the packets. The packets are transmitted separately over the network and reassembled into a single

message again at the other end of the network connection for delivery to the receiving application.

The terms *Transport Protocol Data Unit* (TPDU) and *Transport Service Data Unit* (TSDU) are used to distinguish between packets and messages. A TSDU is a network message, and can be essentially any size, including the size of an entire disk file. A TPDU is the part of a message sent in a single packet. Each TPDU is small enough to fit into a single Ethernet packet along with the addressing information used by the lower network layers. A TPDU also includes sequencing information so that the Transport layer software at the receiving end can assemble the entire message in the correct order, even if the individual packets are delivered across the network out of sequence. Out-of-sequence delivery is a distinct possibility when alternate networking routes are available between a sending ES and a receiving ES.

Applications do not need to be concerned with TPDUs, but they do deal with TSDU (message) boundaries. If a fixed-length entity such as a file is being sent as a message, an application could request that the Transport layer send the entire message, or send the file in chunks whose sizes depend on the buffer sizes the application uses to read the file from disk before writing it to the network. The application uses two different commands for writing the data, `send_data` and `send_eom_data`. The former is used for sending all but the last chunk, and the latter (`eom` stands for end of message) is used to send the last chunk in the message. The receiving application can read the message in whatever chunk sizes are convenient for itself, independent of the chunk sizes used to send the message. The receiver receives a special condition code when it reads the end of the message. The Transport layer software apportions the chunks being sent into TPDUs at the sending end, divides TPDUs into the chunks requested by the receiving end, and handles the end-of-message condition at both ends, automatically.

This ability to send and receive data over the network independent of message boundaries is crucial for applications that work with *stream data*, defined as a sequence of data that must be consumed by a receiver as it is produced by a sender, without waiting for an end-of-message indicator. An example would be an interactive remote login server, which must read and echo each keystroke as it is typed by a remote user without waiting for the user to type an end-of-line character.

Stream data is characterized by having fixed-length data units with internal codes that indicate message boundaries. For the terminal server example, each octet is a data unit, and an ASCII `<cr>` could indicate the end of a message. Another example is found in the X Window System protocol in which network requests consist of request blocks containing various numbers of bytes. The first part of each request block is a fixed-length header that always includes a field identifying the total size of the request block. The receiving application reads the header and then reads as many additional bytes as necessary to read the rest of the block.

11.4.2 Network connections

The ISO standards provide for two types of communication over a network, *connectionless* and *connection-oriented*. Connectionless communication, which is also called *datagram service*, treats each packet sent over the network as a separate message, called a *datagram*. The network plays a passive role for datagram service. Once the local machine's lower networking layers verify that a datagram has been sent out successfully, the local machine's networking software is no longer involved with that datagram.

Because a network packet is received by any and all systems on the network that recognize the address in the packet's header, it is possible that no remote systems (or, perhaps, several) will accept a datagram sent out by a local system. In an ES-IS environment, a datagram might go through several retransmissions before reaching its destination, leading to a degree of uncertainty on the part of a sending program about whether a datagram has actually been delivered successfully or not. One solution to this uncertainty problem is to have applications that want to communicate with each other using datagrams establish their own protocol for reliable communication. A datagram-based server, for example, could be programmed to send an acknowledgment packet to the sender of every packet it receives. If the sender does not receive an acknowledgment in a reasonable amount of time, it retransmits the datagram. Individual datagrams could be given unique ID numbers so that a server would ignore duplicate datagrams sent because an acknowledgment datagram got lost or took too long to be delivered.

The need for this type of reliable delivery service is so common that it has been built into the Transport layer itself. The mechanism involves setting up a connection called a *virtual circuit* (VC) between the two programs that want to communicate. The term *connection* in this context has nothing to do with the iRMX BIOS's connection object type. You can think of a VC as an object type defined by and managed by iNA software that is independent of the iRMX operating system.[5] Connection-oriented communication goes through several distinct phases, as described as follows:

Connection establishment. This phase sets up a virtual circuit, which can involve a negotiation process between the Transport layer software on both computers. When this process is complete, the programs at each end

[5]iRMX-NET creates an operating system extension (OSE) for VC objects, largely so that iRMX-NET can be informed when a job that has created a VC terminates without deleting the VC itself. (See the discussion of deletion mailboxes in Chapter 10 for more information on the rationale behind this use of OSEs.) When iNA is implemented on a controller board with its own processor and memory, the data structure associated with a VC is allocated from the controller board's memory, and the OSE provides a mechanism to help manage the memory on the controller board.

both have a token used to identify the VC to their local Transport layer software.

Data transfer. Either program can send messages of any length to the program at the other end. The Transport layer divides the messages into TPDUs as necessary, verifies that individual packets are received by the Transport layer software at the other end, and assembles arriving TPDUs into messages for delivery to the local user of the VC on demand.

Connection termination This phase is supposed to provide for an orderly shutdown of communication between programs at the two ends of the VC. In practice, closing a VC is not like closing a BIOS or EIOS I/O connection. In particular, any data queued for transmission but not yet sent is lost when a VC is closed. Applications need to establish their own protocols to tell each other when it is safe to close a VC between them.

11.5 Transport Address Buffers

Probably the most difficult part of network programming is the management of network addresses. Transport layer datagrams and virtual circuits use a data structure called a `ta_buffer`, shown in Figure 11.4.

The buffer consists of four parts: the local Network Service Access Point (NSAP), the local Transport Service Access Point (TSAP), the remote NSAP, and the remote TSAP. The idea of a service-access point such as a TSAP or NSAP is to support the many-to-one relationship between one protocol layer and the software running one layer above it. For example, each application that uses the Transport layer must specify a unique identification number for itself (its TSAP) so the Transport layer can distinguish one application from another. Likewise, an NSAP allows the Network layer to tell which of possibly several Transport layer implementations it is communicating with. Since iNA (the Transport layer) and the

Figure 11.4 Structure of a transport address buffer.

```
struct ta_buffer {
  BYTE      local_nsap_selector_length;
  BYTE      local_nsap_selector[LOCAL_NSAP_SELECTOR_LENGTH];
  BYTE      local_tsap_selector_length;
  BYTE      local_tsap_selector[LOCAL_TSAP_SELECTOR_LENGTH];
  BYTE      remote_address_length;
  BYTE      remote_address[REMOTE_ADDRESS_LENGTH];
  BYTE      remote_tsap_selector_length;
  BYTE      remote_tsap_selector[REMOTE_TSAP_SELECTOR_LENGTH];
  }
```

Network layer are implemented as a unit on iRMX systems, it should be clear that there is no real need for a local NSAP for iRMX networking, and the ISO standards allow a value of zero for LOCAL_NSAP_ADDRESS_LENGTH.[6]

Selector (like connection) is another term that has different meanings in iRMX and networking contexts. For networking, it is simply a name for an identification number and has nothing to do with microprocessor protected-mode memory addressing as described in chapter 5. Indeed, other commonly used names for NSAP selectors (found in a remote_nsap_address, described later) and TSAP selectors are NSAP IDs and TSAP IDs. If two programs will communicate with each other using the Transport layer, they must specify matching TSAP ID values for the ta_buffer used when connecting (or for each datagram for connectionless operations). This requirement for matching TSAP IDs can be handled in one of three ways:

1. The applications that want to communicate select any convenient values for the TSAP IDs. This technique works provided two different applications do not decide to use the same values for their TSAP IDs on the same network.
2. The name *server*, described below, can be used by applications to publicize the TSAP IDs they are using. For example, a server could determine a TSAP ID that is not being used and enter that value into the name server database using some well-known name. Clients could query the database by name to determine the TSAP ID value to use for connecting with the server.
3. A central authority could be set up to assign TSAP IDs to application protocols. For example, an organization could decide that a certain file exchange protocol will have all file servers use a TSAP ID value of 0x1000 and all file consumers use a TSAP ID value of 0x1100.

Readers familiar with the TCP/IP protocols used on the Internet might recognize that TSAP IDs play the same role as port addresses in those protocols. For example, all Internet File Transfer Protocol (FTP) servers accept requests for service in the well-known TCP/IP port number 21. The Internet Protocols, including port numbers used, are published in documents called Request for Comments (RFCs), available from the Network Information Center (NIC), which can be accessed using FTP at the Inter-

[6]The structure shown in Figure 11.4 should be considered pseudo-code because a value of zero leads to a syntax error if you try to compile it; In particular, the local_nsap_selector_length field is usually zero, so the local_nsap_selector array must be omitted from the code for the structure.

net address *nic.ddn.mil.* Thus, the NIC could be thought of as the central authority for Internet Protocol port numbers.[7]

For another example of choosing TSAP IDs, the values provided in item (3) are the actual port numbers used by OpenNet file servers and consumers. They are entered in the name server's database using the names FSTSAP and FCTSAP, implying that it would be possible to develop OpenNet file servers and consumers that use different TSAP values. That might actually be true, but there is no reason to do so, and it is not clear that existing OpenNet software can adapt to different TSAP IDs in a meaningful way. Rather, it is better to think of these TSAP IDs as being assigned by another central authority, namely the OpenNet developers at Intel.

The REMOTE_NSAP_ADDRESS field of a ta_buffer provides the information that the network layers below the Transport layer will need to connect to a remote system. There are three forms this field may take, *Null2*, *Static Internetwork*, and *ES-IS Network* formats. These forms are discussed in the following subsections.

11.5.1 Null2 network addresses

Null2 addresses can be used if all the computers communicating with each other are connected to the same network medium, such as a single Ethernet cable. In networking parlance, all the computers are said to be on the same *subnetwork.* Thus, the Null2 format can be used when there is no need for an IS to route packets from one subnetwork to another. The name *Null2* refers to the idea that Null2 addresses are used when iNA is configured with no (or null) network routing capabilities.

iRMX for Windows supports Transport layer communication using Null2 addressing even when no network device controller is installed in the computer. In this case, two applications running on a single computer can use networking protocols to exchange data with each other whether they are both Windows applications, both iRMX applications, or a combination of the two. Furthermore, the applications can be ported transparently to operate over a real network, as demonstrated in the discussion of the iRMX for Windows DDE mechanism in chapter 12.

A C structure for a Null2 address would be the following:

```
struct Null2 {
  BYTE        AFI;
  WORD        subnet;
  BYTE        host_id[6];
  BYTE        lsap_selector;
  BYTE        nsap_selector;
  }
```

[7]Comer (1988) presents a good summary of the IP addressing mechanism and provides a list of many of the well-known port addresses used on the Internet. Stevens (1990) provides an excellent guide to this and many other networking topics relevant to this chapter.

The AFI field is the Authority and Format Identifier. All iNA network addresses use a value of 0x49 for this byte. If another ISO Network layer were to use the same subnetwork as iNA, it could use a different value for this byte, and the format of the remainder of the network address could be decoded according to the different AFI values encoded here. This feature is useful in concept only, however, as the iNA implementation of the Network layer recognizes only this one value.

The subnet field is used for internetwork routing. For Null2 addresses it is always set to a constant value, 0x0001.

The host_id field, for Ethernet connections, is a unique identifier associated with the network device controller. Every company in the world that manufactures Ethernet device controllers is assigned a range of these Ethernet addresses by a central authority, and every controller board manufactured by a company is built with a different address within the company's assigned range configured into it. The idea is to guarantee that addressing conflicts do not exist among computers connected to an Ethernet, regardless of from where the Ethernet device controllers came. The Ethernet address is also known as the Media Access Control (MAC) address for a computer.

The lsap_selector field is a Link Service Access Point identifier, serving an analogous role for the Data Link layer to the TSAP and NSAP IDs for the Transport and Network Layers. Since the Data Link layer is implemented within iNA, this selector is always coded with the same value, 0xFE, for all iNA applications.

Finally, the nsap_selector field is an ID number used for network routing. Since the Null2 address format does not support network routing, this byte is always coded as 0x00. Technically, this byte can be omitted for Null2 addresses, with the remote_nsap_address_length field of the ta_buffer indicating whether it is present (length equals 11) or absent (length equals 10). Standard practice is to include it.

How to fill in this data structure is discussed later in this chapter when name server operations are covered in section 11.9 and when a small datagram application is presented in section 11.7.

11.5.2 Static and dynamic internetwork addresses

Network routers (the ISs in our terminology) accomplish their job by maintaining a set of tables to identify how to direct packets from one subnetwork to another. These tables are managed by iNA for OpenNet networks, and can either be set up when the network is configured, or constructed and modified as the network is running. The former is less flexible, but makes fewer demands on the network itself. The flexibility of dynamic router tables also requires additional software to create and

maintain them. Static internetwork addresses look almost exactly like Null2 addresses:

```
struct Static {
  BYTE          AFI;
  BYTE          area_id[5]
  BYTE          subnet;
  BYTE          host_id[6];
  BYTE          lsap_selector;
  BYTE          nsap_selector;
}
```

The difference is that the 2-byte `subnet` field has been replaced by 6 bytes that incorporate an `area_id` and a `subnet` number. The other difference is that iNA interprets an address as a Null2 address if the `nsap_selector` field is 0, even if an `area_id` is included. The values for `AFI`, `host_id`, and `lsap_selector` have the same interpretations as the corresponding fields of a Null2 address.

The format of internetwork addresses for networks that support dynamic routing tables is basically the same as for Null2 and static routing (`AFI`, `subnet`, `host_id`, `lsap_selector`, and `nsap_selector` fields). However, the interpretation of the `subnet` field depends on how the routing tables are configured for the network, which is beyond the scope of this chapter. Interested readers should consult the *iNA 960 Programmer's Reference* manual (Intel, 1991a).[8]

11.6 The Request Block Interface to iNA

To communicate with the network, an iRMX application must communicate with iNA. To the application, communicating with iNA is similar to the IORS interface to the BIOS, introduced in chapter 8. In networking, the application creates a Request Block (RB), places a token for a response mailbox in the RB, sends the RB to iNA, perhaps performs other processing while iNA processes the RB, and then waits at the response mailbox, where the RB is returned with a completion status and other information.

The differences between iNA RB processing and BIOS IORS processing are that the application constructs RBs itself, but the BIOS constructs IORSs for applications automatically, and that the communication of IORSs between the BIOS, the device driver, and the task that calls the

[8]This manual is being replaced by the *Network Programmer's Reference* manual. The *Network Programmer's Reference* manual and the *iRMX Network Concepts* manual, which replaces the one written in 1991 (Intel 1991b), are bound into a single volume and included with the iRMX III documentation set.

BIOS is all managed directly by iRMX, whereas RBs are passed between iRMX and iNA using the MIP protocol mentioned earlier. Using the MIP often involves communication between two different microprocessors (often including the processor on the device controller that is not running iRMX) that communicate through shared memory or the message passing features of a hardware bus.

Each RB includes a function code, room for status information, the token for a response mailbox, and pointers to the buffers needed for the particular function being requested. The exact nature of this information is discussed shortly, but first, programmers must be aware of a basic problem that must be dealt with, at least in most iNA implementations: pointers that make sense for one microprocessor, such as the one running iRMX, do not automatically make sense for another microprocessor, such as the one running iNA, even if the pointers refer to the same shared physical memory addresses.

For example, a protected-mode version of iRMX would use the `selector:offset` form for pointers that refers to the microprocessor's private Local Descriptor Table (LDT) or Global Descriptor Table (GDT) descriptor tables (see chapter 5), whereas iNA typically runs on a processor that runs in real mode with `base:offset` pointers. For the MIP to accomplish the necessary pointer conversions in the various configurations in which it must operate, all pointers in RBs are passed and returned as 32-bit values put into the proper form for a particular environment using the function *cqcommptrtodword( )*. The matching function, *cqcommdwordtoptr( )*, turns a 32-bit address back into a protected-mode pointer. For *cqcommdwordtoptr( )* to function correctly in all situations, all of the buffers referenced by an RB must be in the first 64 kilobytes (KB) of the same segment as the RB. In addition, the selector (token) for the segment containing the RB must be stored in the header portion of the RB itself.

At this point, you can see the general algorithm that tasks use for all interactions with iNA:

1. Obtain a segment to hold the RB data structure and all buffers that the RB references.
2. Fill in the RB data structure, including the opcode, the token for a response mailbox, and whatever other information might be required.
3. If the RB includes pointers to buffers, use *cqcommptrtodword( )* to convert those pointers to a form recognized by iNA. Put the token for the segment containing the buffers in the proper field of the RB.
4. Use the function *cqcommrb( )* to send the RB to iNA for processing, a process called *posting an RB*. The RB is not necessarily moved anywhere, but iNA receives access to it. The application must not alter the contents of the RB until iNA finishes processing it.

5. Wait at the response mailbox for the RB token to be returned. A task can post multiple RBs using the same response mailbox, if desired.
6. If necessary, the task can use *cqcommdwordtoptr( )* to convert any physical addresses in the RB back to protected mode pointers after it receives the RB back from iNA.

11.6.1 The request block header

Every RB starts with the same data structure, called a Request Block Header, shown in Figure 11.5. The fields are summarized as:

`reserved`. iNA uses this field internally to build a doubly linked list of all the RBs being processed.

`length`. This field is the total length of the RB in bytes. This value includes the length of the RB header, plus the length of any arguments that follow the header. It does not include the lengths of any buffers pointed to by the arguments. An incorrect value in this field can lead to errors that are difficult to trace.

`user_id`. Before sending any RBs to iNA, an application must obtain a `user_id` value by calling *cqcreatecommuser( )*, and it must place this value in the `user_id` field of every RB the application uses. (Function prototypes for all the *cq* system calls are given in the next section.)

`response_port`. This field is always coded as the constant 0xFF.

`response_mailbox`. A token for an object mailbox is placed in this field. The token for the RB segment is returned to this mailbox when iNA has finished processing it.

`segmentTkn`. The token for the segment that contains the buffers pointed to by the arguments in this RB goes in this field. The value is used only by *cqcommdwordtoptr( )*, as described previously.

`subsystem`. This code identifies the part of iNA that processes the RB. The first nibble (high-order four bits) tells the OSI layer, using the numbers in Figure 11.1. The Transport layer subsystems are 0x40 for virtual circuits and 0x41 for datagrams. Non-OSI operations, such as the Network Management Facility introduced in Section 11.10, use a value of 8 for the first nibble.

Figure 11.5 Structure of a Request Block (RB) header.

```
struct rbHeader {
  WORD      reserved[2];
  BYTE      length;
  WORD      user_id;
  BYTE      response_port;
  TOKEN     response_mailbox;
  TOKEN     segmentTkn;
  BYTE      subsystem;
  BYTE      opcode;
  WORD      response_code;
  }
```

opcode. This field is the operation code for the specific function to be performed.

response_code. Response code is a condition code value filled in when the RB is returned to the response_mailbox if an error occurred. The application should set this field to 0 before posting the RB. This field can be set by either the MIP, if the problem occurred delivering the RB to iNA, or by iNA to indicate the result of processing the RB. MIP response code values are in the range 0xFF00 to 0xFFFF. Note that the standard iNA "OK" response code is 0x0001, rather than the normal iRMX value of 0x0000. Programs that use literal names for response code values are more reliable than those that use numeric constants; the values sometimes change. Literal names for iNA response codes are given in the various :include:cq*.h and :inc:cq*.lit files for C and PLM programs, respectively.

OK_RESPONSE is the standard name for the iNA response code 0x0001. Note also that the iRMX exception handling mechanism described in chapter 6 does not apply to iNA or MIP exceptions.

The user_id field offers an interesting example of the use of an Operating System Extension (OSE), described in chapter 10. A bit of poking around with the System Debugger (SDB) or SoftScope indicates that the value returned by *cqcreatecommuser()* is a token for an iRMX composite object with type code 0x0165. The OSE for this object type is owned by the iRMX-Net job, and the extension does have a deletion mailbox associated with it. The idea is that if a job that owns a user_id object terminates without first calling *cqdeletecommuser()*, a copy of the user_id token is sent to the deletion mailbox as the job is deleted, and the task that monitors that mailbox can notify iNA to free its resources that have been allocated to that user. If this mechanism were not in place, iNA's memory resources could easily become depleted over time, requiring an eventual system reset to continue processing.

11.6.2 Function prototypes for RB operations

Only five functions exist in the API for the RB interface to iNA.[9]

The interface procedures are in library files called cqc.lib, cql.lib, and cqc32.lib for 16-bit compact, 16-bit large, and 32-bit applications, respectively. These libraries are found in :SD:rmx386/rmxnet for iRMX for Windows; their directories vary for other versions of iRMX. The following are their function prototypes.

```
extern TOKEN
cqcreatecommuser (          WORD far *          exceptPtr);
```

[9]The API for iNA is simply a programming interface to the network services supplied by iNA. The API is independent of any network protocols used to exchange information over a network.

The value returned is a `user_id`, as described previously. The exception code returned is a normal iRMX exception code; any nonzero value means the iRMX-Net software could not create the communication user object.

```
extern void
cqcommrb (                    TOKEN           rbTkn,
                              WORD far *      exceptPtr);
```

This function is used to post an RB to iNA. The application should not modify the memory used by the RB or its buffers until the token is returned to the `response_mailbox`.

```
extern DWORD
cqcommptrtodword (            void far *          Ptr,
                              WORD far *          exceptPtr);
```

The value of `Ptr` is converted into a 32-bit form understood by iNA. If iNA runs on a separate processor, the form can be a 32-bit physical memory address. If iNA runs on the same processor as the application, the form can be a 16-bit selector and 16-bit offset. For 32-bit code, the latter form restricts the addresses referenced by RBs to the first 64 KB of a segment.

```
extern void far *
cqcommdwordtoptr (            DWORD               dw,
                              WORD far *          exceptPtr);
```

The 32-bit address in `dw` is converted into a protected-mode pointer. For this call to work properly, it requires `dw` to be an address within a segment that begins with a valid RB header, as described previously.

11.6.3 Alternative interfaces to iNA

As the examples which follow show, programming the RB interface to iNA can be a tedious process. Each field of each RB must be initialized with a separate assignment statement, and a lot of repetitive work must occur to create RBs, sending them to iNA, get them back, check for exceptions, and the like. There are several ways to improve on the situation.

First, write a personal set of utility functions to perform common operations such as creating an RB segment, filling in a `ta_buffer`, and the like. The *rprint* command for network printing, available from the iRUG users group, includes such a set of functions developed by Fred Richter of Intel's Islandia, New York, office.

Second, use a programmatic interface to iNA instead of the RB interface. An excellent way to do this would be to implement a new type manager and set of system calls for network operations using the techniques described in chapter 10.Finally, Vickery (1990) describes an interface that makes iNA virtual circuits appear to implement the socket mechanism

available for BSD Unix systems. An alternate interface currently under development at Intel is based on the Unix Transport Layer Interface (TLI).

Readers interested in developing networking applications should consult the *iNA 960 Programmer's Reference* manual (Intel, 1991a) for detailed information on the RB interface to iNA.

11.7 A Datagram Example

At this point, you are ready to look at a sample networking application to see the RB processing operation in action. The example is a time-of-day server that could be used on a network in which some of the computers do not have a battery backed-up time-of-day clock. A computer that does maintain the correct time-of-day information is designated as a time-of-day server, and always runs a job, called *timesrv*, that accepts a datagram from any other computer that wants to obtain the current time of day. When such a request arrives, the server sends back another datagram containing the current time, using the standard iRMX representation for time and date: a 32-bit integer giving the number of seconds since midnight January 1, 1978.

Any computer on the network (including the one running *timesrv*) can run the second program, *gettime*. This program sends a datagram to the time server requesting the current time of day, receives a datagram containing the time in return, and uses the value in the returned datagram to set the local computer's time of day. The datagram sent to the server normally contains just one 32-bit word with a value of 0. If the user is the Super user and specifies `set` on the *gettime* command line, the value sent in the datagram is the local system's time of day, which the server uses to set its own time-of-day clock.[10] The C source code for *timesrv* is given in Figure 11.6. The algorithm implemented is the following:

1. Create a logfile, `:sd:\timesrv.log`, and write a sign-on message to it.
2. Look up the iRMX object cataloged in the root job's object directory with the name `INARDY`. The token obtained is of no interest, but the fact that it is available is the signal that iNA is available. If the lookup operation fails, it means either that the networking job failed to initialize for some reason or that it was not included in the system configuration.
3. Call *cqcreatecommuser()* to obtain the `user_id` value that will be included in each RB posted to iNA.

[10]The sample server uses *rqsettime()* to set its time of day. This system call modifies only the time of day maintained by the operating system. If this call were changed to *rqsetglobaltime()*, it would set the time on the computer's battery-backed clock.

Figure 11.6 Source code for the *timesrv* server program demonstrating the use of datagrams.

```
/***> timesrv.c <*************************************************
 *
 *  This program provides the current date and time when it receives
 *  datagrams at TSAP 0x0064.
 *
 *    The data portion of a received datagram must contain a four-byte
 *    value.  If the value is non-zero, this server will use it to set
 *    the local system's time of day clock.  Whether the received value
 *    is zero or not, this server will return a datagram containing the
 *    current time of day in iRMX format (number of seconds since
 *    January 1, 1978).
 *
 ******************************************************************/


#include <stdio.h>
#include <time.h>          /* POSIX time functions */
#include <string.h>

#include <rmxc.h>
#include <cqcomm.h>        /* iNA Interface Function Prototypes       */
#include <cqcommon.h>      /* RB common header structure              */
#include <cqtransprt.h>    /* TL structs, opcodes, and response codes */

#include "tlrespcodes.h"

#define E_OK        0
#define NO_WAIT     0
#define ROOT_JOB    3

char            logmessage[80];
time_t          timeNow;        /* POSIX date and time  */

WORD            Status;
TOKEN           user_id, root_job, a_token, rb_segment,
                responseMbx, coConn;


#pragma noalign (Null2)
struct Null2 {
  BYTE          AFI;
  WORD          subnet;
  BYTE          host_id[6];
  BYTE          lsap_selector;
  BYTE          remote_nsap_selector;
  };

#pragma noalign (ta_buffer)
typedef struct ta_buffer {
  BYTE              local_nsap_selector_length;
  BYTE              local_tsap_selector_length;
  WORD              local_tsap_selector;
  BYTE              remote_address_length;
  struct Null2  remote_address;
  BYTE              remote_tsap_selector_length;
  WORD              remote_tsap_selector;
  } TA_BUFFER;
```

Figure 11.6 (*Continued*)

```
#pragma noalign (timesrv_dg_segment)
struct timesrv_dg_segment {
  DATAGRAM_RB    dg_rb;
  TA_BUFFER      ta_buf;
  DWORD          date_time;     /* iRMX date and time buffer  */
  } *dgPtr;

RB_COMMON        *rbPtr;
TA_BUFFER        *taPtr;

/*  main() Starts Here
 *  ------------------------------------------------------------------------
 */
int
main (int argc, char *argv[]) {

/*  Initialization
 *  --------------
 *
 *      Assume we are not able to write to a user's console (perhaps
 *      because we are run by a sysload command), but that we are an
 *      I/O job and can write to a file.  Create a logfile in the root
 *      directory of the system device... it's the only path we are sure
 *      of.
 */
    coConn = rqscreatefile ((STRING *)"\x010:sd:/timesrv.log", &Status);
    rqsopen (coConn, 2, 0, &Status);
    timeNow = time (NULL);  /* POSIX time */
    sprintf (logmessage, "timesrv started -- %s", ctime (&timeNow));
    rqswritemove (  coConn, (BYTE *)logmessage,
                    strlen (logmessage), &Status);
    if (Status != E_OK)
      rqdeletejob ((selector) NULL, &Status); /* tragedy */

/*  Be sure iNA is available before continuing
 */
    root_job = rqgettasktokens (ROOT_JOB, &Status);
    a_token = rqlookupobject (root_job, "\6INARDY", NO_WAIT, &Status);
    if (Status != E_OK) {
      rqswritemove (coConn,
      (BYTE *)"iNA not available, timesrv aborted.\r\n", 37, &Status);
      rqdeletejob ((selector) NULL, &Status);
      }

/*  Allocate all the resources this job will need
 *
 *      These consist of:
 *          comm user ID
 *          datagram RB segment
 *          response mailbox for RBs
 */
    user_id = cqcreatecommuser (&Status);
    if (Status != E_OK) {
      rqswritemove (coConn,
        (BYTE *)"Unable to create comm user, timesrv aborted.\r\n", 46,
        &Status);
      rqdeletejob ((selector) NULL, &Status);
      }
```

Figure 11.6 (*Continued*)

```
    rb_segment =
      rqcreatesegment (sizeof (struct timesrv_dg_segment), &Status);
    if (Status != E_OK) {
      rqswritemove (coConn,
        (BYTE *)"Unable to create RB segment, timesrv aborted.\r\n", 47,
        &Status);
      rqdeletejob ((selector) NULL, &Status);
      }

    responseMbx = rqcreatemailbox (0, &Status);
    if (Status != E_OK) {
      rqswritemove (coConn,
        (BYTE *)"Unable to create resp mbx, timesrv aborted.\r\n", 47,
        &Status);
      rqdeletejob ((selector) NULL, &Status);
      }

/*  Initialize the contents of the single segment we will be using for
 *  all datagram request blocks in this program.
 *
 *      First, the header portion declared as RB_COMMON in cqcommon.h
 */
        rbPtr = (RB_COMMON *) buildptr (rb_segment, 0);
        rbPtr->reserved[0] = 0;
        rbPtr->reserved[1] = 0;
        rbPtr->length = sizeof (DATAGRAM_RB);
        rbPtr->user_id = user_id;
        rbPtr->resp_port = 0xFF;
        rbPtr->resp_mbox = responseMbx;
        rbPtr->rb_seg_tok = rb_segment;
        rbPtr->subsystem = TL_DATAGRAM;
        rbPtr->opcode = RECEIVE_DATAGRAM;
        rbPtr->response = 0;

/*      Now, the datagram arguments --
 *          declared as DATAGRAM_RB in cqtransprt.h
 */
        dgPtr = (struct timesrv_dg_segment *) rbPtr;
        dgPtr->dg_rb.reserved[0] = 0;
        dgPtr->dg_rb.reserved[1] = 0;
        dgPtr->dg_rb.reserved[2] = 0;
        dgPtr->dg_rb.reserved[3] = 0;
        dgPtr->dg_rb.ta_buffer_addr =
            cqcommptrtodword ((void *) &(dgPtr->ta_buf), &Status);
        if (Status != E_OK) {
           rqswritemove (coConn,
             (BYTE *)"cqcommptrtodword failed, timesrv aborted.\r\n", 45,
             &Status);
           rqdeletejob ((selector) NULL, &Status);
           }

         dgPtr->dg_rb.qos = 0;    /* Quality of Service */

         dgPtr->dg_rb.num_blks = 1;
         dgPtr->dg_rb.data_blk_list[0].length = 4;
         dgPtr->dg_rb.data_blk_list[0].address =
           cqcommptrtodword ((void *) &(dgPtr->date_time), &Status);
```

Figure 11.6 (*Continued*)

```
        if (Status != E_OK) {
          rqswritemove (coConn,
            (BYTE *)"cqcommptrtodword failed, timesrv aborted.\r\n", 45,
            &Status);
          rqdeletejob ((selector) NULL, &Status);
          }

/*      Finally, the transport address buffer, declared as TA_BUFFER
 *      in this program
 */
        taPtr = &(dgPtr->ta_buf);
        taPtr->local_nsap_selector_length = 0;
        taPtr->local_tsap_selector_length = 2;
        taPtr->local_tsap_selector = 0x0064;
        taPtr->remote_address_length = sizeof (struct Null2);
        taPtr->remote_address.AFI = 0x49;
        taPtr->remote_address.subnet = 0x0000;
        taPtr->remote_address.host_id[0] = 0;    /* any */
        taPtr->remote_address.host_id[1] = 0;
        taPtr->remote_address.host_id[2] = 0;
        taPtr->remote_address.host_id[3] = 0;
        taPtr->remote_address.host_id[4] = 0;
        taPtr->remote_address.host_id[5] = 0;
        taPtr->remote_address.lsap_selector = 0xFE;
        taPtr->remote_address.remote_nsap_selector = 0;
        taPtr->remote_tsap_selector_length = 2;
        taPtr->remote_tsap_selector = 0x0000;    /* any */

        rqsclose (coConn, &Status);

/*  Main Loop
 *  ---------
 *
 *      Here we post a RB to iNA that indicates we are interested in
 *      all datagrams sent to our TSAP ID, 0x0064.  As datagrams
 *      arrive, we process them, send a datagram in return, and post
 *      another RB to accept another datagram.  This loop never exits.
 *
 */

    for (;;) {

/*      Post a receive_datagram RB, and wait for it to return
 */
        cqcommrb (rb_segment, &Status);
        if (Status != E_OK) {
          rqsopen (coConn, 2, 0, &Status);
          rqswritemove (coConn,
            (BYTE *) "cqcommrb rcv dgm failed, timesrv aborted.\r\n",
            45, &Status);
          rqdeletejob ((selector) NULL, &Status);
          }

        a_token = rqreceivemessage (responseMbx, 0xFFFF, NULL, &Status);
        if (rbPtr->response != OK_EOM_RESP) {
          if (rbPtr->response > TL_RESP_CODES_MAX) rbPtr->response = 0;
          strcpy (logmessage, tlResponseCodes[rbPtr->response]);
          rqsopen (coConn, 2, 0, &Status);
```

Figure 11.6 (*Continued*)

```
            rqswritemove (coConn,
              (BYTE *) logmessage, strlen (logmessage), &Status);
            rqsclose (coConn, &Status);
            break;
            }

          if (dgPtr->dg_rb.buf_len != 4) { /* Sanity check */
            rqsopen (coConn, 2, 0, &Status);
            rqswritemove (coConn, (BYTE *) "Invalid datagram\r\n",
              18, &Status);
            rqsclose (coConn, &Status);
            break;
            }

/*        We have a valid datagram.  Set the local system's time if the
 *        data value is not zero.  Then, put the local system's time in
 *        the same data buffer, and send the datagram back to the sender.
 */

          if (dgPtr->date_time != 0)
            rqsettime (dgPtr->date_time, &Status);
          dgPtr->date_time = rqgettime (&Status);

          rbPtr->opcode = SEND_DATAGRAM;
          rbPtr->response = 0;
          cqcommrb (rb_segment, &Status);
          if (Status != E_OK) {
            rqsopen (coConn, 2, 0, &Status);
            rqswritemove (coConn,
              (BYTE *) "cqcommrb snd dgm failed, timesrv aborted.\r\n",
              45, &Status);
            rqdeletejob ((selector) NULL, &Status);
            }

          a_token = rqreceivemessage (responseMbx, 0xFFFF, NULL, &Status);
          if (rbPtr->response != OK_RESPONSE) {
            if (rbPtr->response > 0x24) rbPtr->response = 0;
            strcpy (logmessage, tlResponseCodes[rbPtr->response]);
            rqsopen (coConn, 2, 0, &Status);
            rqswritemove (coConn,
              (BYTE *) logmessage, strlen (logmessage), &Status);
            rqsclose (coConn, &Status);
            break;
            }

/*        Now reset the necessary parameters to again make our RB segment
 *        into a receive datagram on TSAP 0064 from anyone.
 */
          rbPtr->opcode = RECEIVE_DATAGRAM;
          rbPtr->response = 0;

          dgPtr->dg_rb.num_blks = 1;
          dgPtr->dg_rb.data_blk_list[0].length = 4;
          dgPtr->dg_rb.data_blk_list[0].address =
            cqcommptrtodword ((void *) &(dgPtr->date_time), &Status);

          taPtr->local_nsap_selector_length = 0;
          taPtr->local_tsap_selector_length = 2;
```

Figure 11.6 (*Continued*)

```
        taPtr->local_tsap_selector = 0x0064;
        taPtr->remote_address_length = sizeof (struct Null2);
        taPtr->remote_address.AFI = 0x49;
        taPtr->remote_address.subnet = 0x0001;
        taPtr->remote_address.host_id[0] = 0;
        taPtr->remote_address.host_id[1] = 0;
        taPtr->remote_address.host_id[2] = 0;
        taPtr->remote_address.host_id[3] = 0;
        taPtr->remote_address.host_id[4] = 0;
        taPtr->remote_address.host_id[5] = 0;
        taPtr->remote_address.lsap_selector = 0xFE;
        taPtr->remote_address.remote_nsap_selector = 0;
        taPtr->remote_tsap_selector_length = 2;
        taPtr->remote_tsap_selector = 0x0000;
    }
    return 0; /* never reached */
}
```

4. Create an iRMX segment object to hold each RB and its associated buffers. This program uses just one RB for all its operations, so there is just one segment.
5. Create a mailbox for iNA to return each RB as iNA finishes processing the RB. Because this program makes only one request to iNA at a time, the same token is always returned to the mailbox, so the mailbox serves only as a synchronization mechanism.
6. Fill in the values for a `receive_datagram` RB. These values include the header information that is the same for all iNA RBs, plus those values specific to datagram RBs.
7. Fill in the transport address buffer and initialize the pointers for the datagram data buffers.
8. The program now enters an endless loop in which it:
   a. Posts a `receive_datagram` RB.
   b. Modifies the same RB is to serve as a `send_datagram` RB when a datagram is received, and uses the RB to send a reply datagram containing the server's time back to the sender.
   c Again modify the RB to act as a `receive_datagram` RB when the reply has been sent.
   d. Repeat Loop.

Let's look at the process in a bit more detail. The first step is to understand the data structures defined in the various header files included in the program. The file `cqcomm.h` contains function prototypes for the four preceding procedures (*cqcreatecommuser( )*, *cqcommrb( )*, and the two pointer-conversion routines). The `cqcommon.h` file contains the data structure for

the header part of all iNA RBs described earlier. The fields in this header are referenced using the `rbPtr` pointer variable in `timesrv.c`. The header file `cqtransp.h` contains data structures and definitions used with iNA's Transport layer functions' datagrams and virtual circuits. This header file includes definitions of all the operation codes, such as `SEND_DATAGRAM` and `RECEIVE_DATAGRAM`, as well as names for all the response codes that might appear in the `response` field of the header when a datagram or virtual circuit RB is returned by iNA. The typedef for a `DATAGRAM_RB` is given in this file, as well as other structures used for virtual circuit operations.

The other header file used in this program, `tlrespcodes.h`, (the name is truncated to `tlrespco.h` on DOS file systems) simply establishes an array of string names for the various Transport layer response codes that might be returned by iNA. It is used for formatting any error messages the program writes to its log file.

Sending or receiving a datagram involves two buffers, a `ta_buffer` and a data buffer. The sample program includes its own typedef for a `ta_buffer` structure configured to use Null2 addressing, as well as a single 4-byte data buffer. To illustrate placing the RB and its associated buffers in a single memory segment, the program declares a structure, `timesrv_dg_segment`, that contains a `DATAGRAM_RB` structure, a `TA_BUFFER` structure, and a 4-byte data buffer for sending or receiving a date/time value. This structure is accessed in the program using the `dgPtr` pointer. Although all these structures make the code fairly easy to write, the logic can be a bit difficult to follow because there are several equivalent ways to access the same part of the datagram segment. As Figure 11.7 shows, `rb_segment` is a token (selector) for the segment that contains the RB and the two buffers, and both `rbPtr` and `dgPtr` are pointers that point to the beginning of this segment.

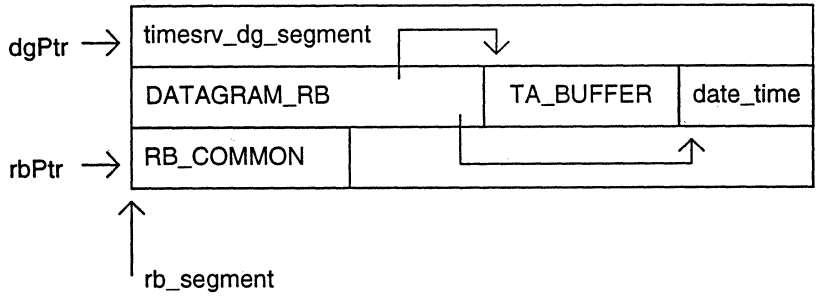


Figure 11.7 Structure of an RB segment used in `timesrv.c`.

Two important concepts regarding network addressing are illustrated by `timesrv.c`. The first is that, as a server, the program does not know the remote addresses of its potential clients. Thus, it fills in the `ta_buffer` with its own TSAP ID, but uses a value of 0 for the remote TSAP ID and remote `host_id` (Ethernet address) fields. iNA interprets the value of 0 as unspecified, and accepts any datagram from any client provided only that the client's remote TSAP ID matches the server's local TSAP ID.

The second concept is that when iNA returns the RB to the server because a datagram has arrived from a client, the server's data buffer is filled with the data supplied by the client and, more importantly, the `ta_buffer` is filled with the fully specified network address for the client. Thus, all the server must do to send a datagram back to a client as a reply is to supply the same `ta_buffer` it used when it posted the `receive_ datagram` request. Before issuing another `receive_datagram` RB to iNA, the server must restore the contents of the `ta_buffer` to the unspecified (0) values for `host_id` and remote TSAP ID.

For completeness, a client for the timeserver *gettime* is given in Figure 11.8. This program first constructs a `send_datagram` RB with a remote TSAP ID, matching the server's local TSAP ID, a local TSAP ID that does not conflict with the server's, and an unspecified `host_id`. It then issues a `receive_datagram` request to receive the server's reply. Because datagram transmission is not guaranteed to be successful, the client repeats its request up to five times at five-second intervals if no server responds.[11]

Figure 11.8 Source code for the *gettime* client program demonstrating the use of datagrams.

```
/***>  gettime.plm  <************************************************
 *
 *       This program is used to get the system time from a remote
 *       system at TSAP 64h.  Local TSAP is 63h.
 *
 ***********************************************************************/

$compact
gettime: DO;

$include (gettime.ext)
$include (cqopcode.lit)
$subtitle ('Declarations')

DECLARE
    E$OK                    LITERALLY   '0',
    E$TIME                  LITERALLY   '1',
    super                   LITERALLY   '0',
    cr                      LITERALLY   '0Dh',
```

[11]The host ID value consisting of 6 bytes of `OFFH` on the line marked (***) must be changed to `OA2h, OA4h, OA6h, OA8h, OAAh, OACh` if *timesrv* and *gettime* are run on the same computer using *ntp4at.job* and no Ethernet controller.

Figure 11.8 (*Continued*)

```
    lf                      LITERALLY   '0Ah',
    five$sec                LITERALLY   '500',

    date_time               WORD_32     INITIAL (0),
    outbuf(81)              BYTE        INITIAL
        (0,'HH:MM:SS  MM DDD YY - Time Set',cr, lf),
    keyword                 STRUCTURE (
        len                     BYTE,
        chars(80)               BYTE),
    (retries,i,is_super)    BYTE,
    atoken                  TOKEN,
    comm_user_token         WORD_16,
    mailbox                 TOKEN,
    rb_segment              TOKEN,
    user_token              TOKEN,
    except_info             STRUCTURE (
        Handler_Ptr             POINTER,
        Mode                    BYTE),
    user_ids                STRUCTURE (
        length                  WORD_16,
        count                   WORD_16,
        id(5)                   WORD_16),
    Status                  WORD_16;

DECLARE
    RB BASED rb_segment     STRUCTURE (
        reserved(2)             WORD_16,
        length                  BYTE,
        user_id                 WORD_16,
        response_port           BYTE,
        return_mailbox          TOKEN,
        segment_token           TOKEN,
        subsystem               BYTE,
        opcode                  BYTE,
        response_code           WORD_16,

        dg_args                 STRUCTURE(
            reserved(4)             BYTE,
            ta_buf_addr             WORD_32,
            qos                     BYTE,
            buf_len                 WORD_16,
            num_blks                BYTE,
            blocks(1)               STRUCTURE(
                addr                    WORD_32,
                len                     WORD_16 ))),

    ta_buf                  STRUCTURE(
        loc_nsap_sel_len        BYTE,
        loc_tsap_sel_len        BYTE,
        loc_tsap_sel            WORD_16,
        rem_nsap_addr_len       BYTE,
            AFI             BYTE,
            subnet          WORD_16,
            host_id(6)      BYTE,
            lsap_selector   BYTE,
            rem_nsap_sel    BYTE,
        rem_tsap_sel_len        BYTE,
        rem_tsap_sel            WORD_16);
```

Figure 11.8 (*Continued*)

```
$subtitle ('main task''s code')
/*  Execution Starts Here
 *  ---------------------------------------------------------------
 */
    rb_segment = rqcreatesegment (128, @Status);

    CALL rq$get$exception$handler (@Except_Info, @Status);
    except_info.mode = 0;
    CALL rq$set$exception$handler (@Except_Info, @Status);

/*  No point in doing anything until iNA is ready
*/
    atoken = rq$get$task$tokens (3, @Status); /* root job's token */
    atoken = rq$lookup$object (atoken, @(6,'INARDY'), 200, @Status);
    IF (Status <> E$OK) THEN
    DO;
        CALL rq$c$send$co$response (NIL, 0,
            @(24,'Network not responding', cr,lf), @Status);
        CALL rq$exit$io$job (Status, NIL, @Status);
    END;

    mailbox = rq$create$mailbox (0, @Status);
    comm_user_token = cq$create$comm$user (@Status);
    IF (Status <> E$OK) THEN
      DO;
        CALL rq$c$send$co$response (NIL, 0,
            @(29,'Unable to Create Comm. User',cr,lf), @Status);
        CALL rq$exit$io$job (Status, NIL, @Status);
      END;
/*
 *    The keyword "set" on the command line causes us to send our
 *    system's time to the server, which will use it to set its own
 *    time.
 */
    CALL rq$c$get$input$pathname (  @keyword,
                                    size (keyword.chars),
                                    @Status);
    IF keyword.len >= 3 THEN
      DO i = 0 TO 2;
        keyword.chars(i) = keyword.chars(i) OR 20h; /* to lower case */
      END;
    IF cmpb (@keyword, @(3,'set'), 4) = 0FFFFFFFFh THEN
      DO;
        user_token = rq$get$default$user (SELECTOR$OF(NIL), @Status);
        user_ids.length = length (user_ids.id);
        CALL rq$inspect$user (user_token, @user_ids, @Status);
        is_super = FALSE;
        DO i = 0 TO user_ids.count - 1;
          IF user_ids.id(i) = super THEN is_super = TRUE;
        END;
        IF NOT is_super THEN
          DO;
            CALL rq$c$send$co$response (NIL, 0,
                @(50,
                'You must be Super to set the time server''s clock',
                cr,lf), @Status);
            CALL rq$exit$io$job (0, NIL, @Status);
          END;
```

Figure 11.8 (*Continued*)

```
      END;
/************************************************************************
 *
 *  Send a datagram to the time server at TSAP 64h in order to
 *  trigger a returned dg containing the date and time at our
 *  TSAP 63h.  Repeat up to 5 times if necessary.
 *
 ************************************************************************/
   retries = 0;
   DO WHILE (retries < 5);

      CALL movb (@(0,              /* loc_nsap_sel_len
*/
                  2,              /* loc_tsap_sel_len                  */
                  63h,0,          /* loc_tsap_sel                      */
                  11,             /* rem_nsap_addr_len (Null2 format) */
                     049h,           /*  AFI                          */
                     0,0,            /* subnet                        */
                     0FFh,0FFh,0FFh,
                     0FFh,0FFh,0FFh, /* host_id (***)                 */
                     0FEh,           /* lsap_selector                 */
                     0,              /* rem_nsap_sel                  */
                  2,              /* rem_tsap_sel_len                  */
                  64h,0),         /* rem_tsap_sel                      */
              @ta_buf, size(ta_buf) );

      RB.reserved(0), RB.reserved(1) = 0;
      RB.length = size(RB);
      RB.user_id = comm_user_token;
      RB.response_port = 0ffh;
      RB.return_mailbox = mailbox;
      RB.segment_token = rb_segment;
      RB.opcode = send_datagram;
      RB.subsystem = 41h;       /* Datagram */
      RB.response_code = 0;

      CALL setb (0, @RB.dg_args.reserved, 0);
      RB.dg_args.ta_buf_addr = cq$comm$ptr$to$dword (@ta_buf, @Status);
      RB.dg_args.qos = 0;
      RB.dg_args.buf_len = 4;
      RB.dg_args.num_blks = 1;
      RB.dg_args.blocks(0).addr = cq$comm$ptr$to$dword (@date_time,
                                                        @Status);
      RB.dg_args.blocks(0).len = 4;

      IF cmpb (@keyword, @(3,'set'), 4) = 0FFFFFFFFh THEN
        DO;
          date_time = rq$get$time (@Status);
          CALL rq$c$send$co$response (NIL, 0,
            @(23,'Setting Server''s Time',cr,lf), @Status);
        END;
      ELSE
        date_time = 0;

      CALL cq$comm$rb (rb_segment, @Status);
      IF (Status <> E$OK) THEN
```

Figure 11.8 (*Continued*)

```
        DO;
          CALL movb (   @(35,'cqcommrb for send_datagram failed: '),
                        @outbuf,
                        36);
          CALL convert$hex (@outbuf, 80, Status, @Status);
          CALL movb (@(cr,lf), @outbuf(outbuf(0) + 1), 2);
          outbuf(0) = outbuf(0) + 2;
          CALL rq$c$send$co$response (NIL, 0, @outbuf, @Status);
          CALL rq$exit$io$job (0, NIL, @Status);
        END;

      atoken = rq$receive$message (mailbox, five$sec, nil, @Status);
      IF (Status <> E$TIME) THEN rb_segment = atoken;
      IF (RB.response_code <> ok_response) THEN
        DO;
          CALL movb (@(25,'dg to Time Server Failed '), @outbuf, 26);
          CALL convert$hex (@outbuf, 80, RB.response_code,
                            @Status);
          CALL movb (@(cr,lf), @outbuf(outbuf(0) + 1), 2);
          outbuf(0) = outbuf(0) + 2;
          CALL rq$c$send$co$response (NIL, 0, @outbuf, @Status);
          CALL rq$exit$io$job (0, NIL, @Status);
        END;

/*
 *  Now wait for a returned datagram from the time server
 */

      RB.reserved(0), RB.reserved(1) = 0;
      RB.opcode = receive_datagram;
      RB.response_code = 0;
      CALL setb (0, @ta_buf.host_id, 6);
      CALL cq$comm$rb (rb_segment, @Status);
      IF (Status <> E$OK) THEN
        DO;
          CALL movb (   @(35,'cqcommrb for recv_datagram failed: '),
                        @outbuf,
                        36);
          CALL convert$hex (@outbuf, 80, Status, @Status);
          CALL movb (@(cr,lf), @outbuf(outbuf(0) + 1), 2);
          outbuf(0) = outbuf(0) + 2;
          CALL rq$c$send$co$response (NIL, 0, @outbuf, @Status);
          CALL rq$exit$io$job (0, NIL, @Status);
        END;


      atoken = rq$receive$message (mailbox, five$sec, nil, @Status);
      IF (Status <> E$TIME) THEN rb_segment = atoken;
      IF (RB.response_code = ok_response) OR
         (RB.response_code = ok_eom_resp) THEN
        DO;

          IF (date_time = 0) THEN
            DO;
              CALL rq$c$send$co$response (NIL, 0,
                  @(47,'Date and Time not Received from Remote System',
                  cr, lf),
                  @Status);
```

Figure 11.8 (*Continued*)

```
            CALL rq$exit$io$job (0, NIL, @Status);
          END;

        CALL rq$set$time (date_time, @Status);
        CALL convert$secs$to$time (@outbuf, 32, date_time, @Status);
        outbuf(0) = outbuf(0) + 2;
        CALL convert$secs$to$date (@outbuf, 32, date_time, @Status);
        outbuf(0) = outbuf(0) + 13;
        CALL rq$c$send$co$response (nil, 0, @outbuf, @Status);
        CALL rq$exit$io$job (0, NIL, @Status);
      END;

    retries = retries + 1;
    CALL rq$c$send$co$response (NIL, 0,
         @(21,'Gettime retrying...',cr,lf), @Status);

  END; /* DO WHILE */

  CALL rq$c$send$co$response (NIL, 0,
       @(30,'No Response from Time Server',cr,lf), @Status);
  CALL rq$exit$io$job (0, NIL, @Status);
END gettime;
```

The five-second interval is timed by placing a time limit on the *rqreceivemessage()* system call that receives the RB back from iNA. The program uses the same RB segment for all five retries, which raises the potential for confusion because it violates the rule that an application should not access an RB from the time it is sent to iNA and the time iNA returns it. No problem exists for this procedure though, because the client makes all five RBs look identical, the client responds identically regardless of which RB is returned, and, with Null2 addressing, datagram transmission is almost always successful, unless there is a physical problem with the physical link.[12]

11.8 Virtual Circuit Operations

Connection-oriented Transport layer operations are performed using the virtual circuit mechanism. A virtual circuit server would typically use an algorithm similar to the following:

1. Ensure iNA is running by looking up the `INARDY` object in the root job's object directory.
2. Call *cqcreatecommuser()* to establish a link to iNA.

[12]Another potential problem with the application is the race condition that exists between the client and the server. If the server responds to the client's request before the client issues a `receive_datagram` RB, the reply will be missed and the client will go through a retry cycle before receiving a response. Normally (meaning when separate RBs are used for sending and receiving), a client would post its receive RBs before its send RB.

3. Use the iNA name server (see the next section) to enter the server's `host_id` and local TSAP ID in the network database using a well-known name. Clients use this information to access the server.
4. Use an RB with an operation code of `open` to obtain a reference number from iNA. This number is used internally by iNA to access the information it keeps for the new connection, called a *connection database* (CDB). Once a virtual circuit has been established by connection with a client, this reference number is used to identify the CDB to iNA without using the addressing process (described next) again.
5. Issue an `await_connect_request_xx` RB. Two different opcodes can be used here. If `xx` is `TRAN` (opcode 2), the Transport layer establishes a virtual circuit whenever a matching `send_connect_request` from a remote system arrives. If `xx` is `CLIENT` (opcode 3), iNA returns the RB when a remote system sends a matching `send_connect_request`, but does not complete the virtual circuit unless the application accepts the connection by issuing an `accept_connect_request` RB. If the application issues a `close` RB instead, the virtual circuit is not established. Note that the term *client* in this opcode name refers to the local application program (a client of the Transport layer) and not to the remote application.
6. Once a virtual circuit has been established, the server supplies iNA with data buffers to receive data from the client using `receive_data` RBs, and uses `send_data` or `send_eom_data` to send data to the client. Since the RB interface is inherently asynchronous, great flexibility exists for how a particular server can manage its flow of data with a client.
7. The `close` RB is used to terminate a virtual circuit (or to reject one as mentioned previously). The close operation does not trigger the transmission of any previously queued output data the way that closing a disk file does. Rather, the server and client must use a separate mechanism, such as special data messages, if they are to provide an orderly termination of their connection. The close operation causes iNA to delete its CDB entry for the virtual circuit.

Several other features of virtual circuits deserve mention. One is that clients and servers can exchange relatively small amounts of data with each other when they establish and terminate a virtual circuit. For example, a server could use `await_connect_request_client`, and use a password supplied by the remote client in its matching `send_connect_request` to decide whether to accept the connection or not. Likewise, a server could provide up to 64 bytes of information when it closes a virtual circuit, such as a text string explaining why it closed a circuit.

A second feature of virtual circuits important for some applications is the provision for expedited data (called *out-of-band* data in the comparable

Internet protocols). Normally, all data sent from one application to another over a virtual circuit is guaranteed to be received in the same order in which it is was sent. Expedited data is sent ahead of any normal data that might be in transit over the network. An example of using expedited data would be a login server that receives single characters as they are typed by the user and echoes them in response. A user might want to cancel text that was typed before receiving the server's response. The user can do so by typing a <^C> character. That character could be sent as expedited data, which the server might receive using a task waiting at a special mailbox set up for `receive_expedited_data` RBs.

The other issue to mention is the management of data buffers. Because iNA often runs on a 16-bit processor, the virtual circuit RB interface requires each data buffer to be used for sending or receiving data to be no greater than 64 KB in length. Within this 64-KB limit, iNA supports gather-write and scatter-read operations. That is, a single RB can reference several disjoint locations in memory as a single data buffer. When writing to the network, iNA gathers data from the locations into a single internal buffer and transmits it as a unit. Likewise, iNA will distribute (scatter) the parts of an incoming message into several locations if that is how the user has defined the receive buffer.

11.9 Name Server Operations

Each entry in iNA's distributed database of network information takes the form of a 4-tuple, {name, property type, uniqueness, property value}. For example, a typical entry might be:

```
{MYHOSTID, 0x0004, FALSE, 0xA0A2A4A6A8AAAC}
```

The name is any arbitrary string of up to 16 bytes, normally ASCII characters. The property type is an unsigned 16-bit numerical value that can be thought of as a modifier for the name. For example, iRMX-Net uses two database entries for each system it services. Both have the same name (the name used in an *attachdevice* command), but two different property types for two pieces of information that iRMX-Net needs to maintain about the system. The virtual terminal facility, which allows remote login to a system, uses the same name along with a third property value for information it needs to know about the system.

The uniqueness database entry is a Boolean value that tells whether a network can have more than one entry with the same {name, property type} combination. For example, every machine on the network has an entry for {`MYHOSTID, 0x0004`}, so that combination is not unique. On the other hand, only one machine can have a particular name and property type for a virtual terminal or file server. If you give a command such as `at-`

`tachdevice systeml as l remote`, no more than one computer can be named `systeml` on the network, so that entry must be unique.

The value part of a database entry can be any sequence of bytes desired, not necessarily ASCII characters. The property value for {`MYHOSTID, 0x0004`}, for example, is the 6-byte Ethernet address for the computer. iRMX-Net and the virtual terminal use the Null2 network address as the property value for the entries with the system's name and property types 0x0003 and 0x0008, respectively. The *listname* command normally installed in `:utils:` or `:util286:` displays the portion of the distributed database that resides on the local system.

11.9.1 The RB interface to the name server

Eight opcodes are used in the RB interface to iNA's name server. These functions are documented in the *iRMX-Net User's Reference* manual (Intel, 1991b) rather than the *iNA 960 programmer's Reference* cited earlier in this chapter. RBs for name server operations include the RB header structure used for all iNA RBs, followed by pointers to three buffers: a name buffer, a property value buffer, and an extra buffer used for some of the operations. The RB also has fields for property type and the uniqueness Boolean.

Figure 11.9 gives the source code for a utility program called *namesrv* that allows a user to exercise the eight name-server commands interactively. Each command takes some subset of a database 4tuple and might or might not return information from the database, as described on page 458.

Figure 11.9 Source code for *namesrv*, a program that allows a user to exercise the eight name server functions interactively.

```
/***> namesrv.c <*************************************************************
 *
 *  This program allows users to query and update iNA's distributed
 *  database.
 *
 *      The program accepts command lines consisting of a nameserver
 *      operation code name followed by the appropriate arguments.
 *
 *      Names and values are entered as strings, and types are entered
 *      as integers.  The get_* functions display the result returned
 *      by the nameserver.  The others simply perform their operations.
 *
 *          add_name            Name Type Value
 *          delete_name         Name
 *          get_value           Name Type
 *          change_value        Name Type New Value
 *          delete_property     Name Type
 *          get_name            Type Value
 *          get_spokesman       Name Type
 *          list_table
 *
 ******************************************************************************/
```

Figure 11.9 (*Continued*)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <rmxc.h>
#include <cqcomm.h>       /* iNA Interface Function Prototypes       */
#include <cqcommon.h>     /* RB common header structure              */
#include <cqname.h>       /* NS structs, opcodes, and response codes */

#include "nsrespcodes.h"

#define E_OK             0
#define NO_WAIT          0
#define E_TIME           1
#define ROOT_JOB         3
#define EXIT_PROGRAM    99

/*  Global variables shared with the various functions that
 *  implement the operation codes.
 */
TOKEN           user_id, root_job, a_token, rb_segment, responseMbx;

RB_COMMON       *rbPtr;
NAME_SERVER_RB  *nsPtr;

NAME_BUFFER     *nameBuf;
VALUE_BUFFER    *propValBuf;
EXTRA_BUFFER    *extraBuf;

#pragma noalign (response_struct)
struct response_struct {

    WORD        opcode;
    char        nameString[32];
    WORD        propertyType;
    WORD        valueLength;
    char        valueString[256];
    } response;

static struct {
  char  *name;
  int   len;
  char  *args;
  } opcodes[] = { {"exit", 1, ""},
                  {"add_name", 1, "name type value"},
                  {"delete_name", 8, "name"},
                  {"get_value", 5, "name type"},
                  {"change_value", 1, "name type value"},
                  {"delete_property", 8, "name type"},
                  {"get_name", 5, "type value"},
                  {"get_spokesman", 5, "name type"},
                  {"list_table", 1, ""}
                  };

static  int     i, j;
static  char    *nextPtr;
static  char    commandLine[80];
static  char    hextab[] = "0123456789ABCDEF";
static  WORD    Status;
```

Figure 11.9 (*Continued*)

```
static  union { /* Pointers for interpreting buffers returned by iNA */
  void        *anyPtr;
  char        *charPtr;
  WORD        *wordPtr;
  } p;


/*  getString()
 *  ------------------------------------------------------------------------
 *
 *    This is a utility routine to extract a string from the command
 *    line.  The string may be surrounded by single or double quotes.
 */
static void
getString (char *where) {
int     length;

  nextPtr += strspn (nextPtr, " \t\n\r,");
  if (*nextPtr == '\'') {
    length = strcspn (++nextPtr, "\'");
    }
  else if(*nextPtr == '\"') {
    length = strcspn (++nextPtr, "\"");
    }
  else {
    length = strcspn (nextPtr, " \r\n\t");


/*  addName()
 *  ------------------------------------------------------------------------
    }
  *where = '\0';
  strncat (where, nextPtr, length);
  nextPtr += length + 1;
  return;
  }

/*  parseName()
 *  ------------------------------------------------------------
 *
 *      Get a string for a property name from a command line
 */
static void
parseName (void) {
  getString (response.nameString);
  udistr (response.nameString, response.nameString);
  return;
  }

/*  parseType()
 *  ------------------------------------------------------------
 *
 *      Get a 16-bit property type from a command line.
 *      The user may enter the value in either decimal or hex
 */
static void
parseType (void) {
  response.propertyType = (WORD) strtoul (nextPtr, &nextPtr, 0);
```

Figure 11.9 (*Continued*)

```
  return;
  }

/*  parseValue()
 *  ------------------------------------------------------------
 *
 *      Get a string for a property value from a command line
 */
static void
parseValue (void) {
  getString (response.valueString);
  response.valueLength = strlen (response.valueString);
  return;
  }


/*  doCommand()
 *  ------------------------------------------------------------
 *
 *      Get a command line from the user and parse it
 */
void
doCommand (void) {

  commandLine[0] = '\0';
  while (strlen (commandLine) == 0) {
  printf ("\nEnter nameserver command: ");
  gets (commandLine);
  }

for (i = 0; i < 80; i++) {
  if (commandLine[i] != ' ') break;
  }
nextPtr = &commandLine[i];
response.opcode = -2;
for (i = 0; i < 9; i++) {
  if (strnicmp (nextPtr, opcodes[i].name, opcodes[i].len) == 0) {
    response.opcode = i - 1;
    break;
    }
  }
if (response.opcode == 0xffff) exit (0);
nextPtr += strcspn (nextPtr, " ,\r\n\t\"\'");
switch (response.opcode) {


  case ADD_NAME:          parseName(); parseType(); parseValue();
                          addName(); break;

  case DELETE_NAME:       parseName();
                          deleteName(); break;

  case GET_VALUE:         parseName(); parseType();
                          getValue(); break;

  case CHANGE_VALUE:      parseName(); parseType(); parseValue();
                          changeValue(); break;
```

Figure 11.9 (*Continued*)

```
  case DELETE_PROPERTY:    parseName(); parseType();
                           deleteProperty(); break;

  case GET_NAME:           parseType(); parseValue();
                           getName(); break;

  case GET_SPOKESMAN:      parseName(); parseType();
                           getSpokesman(); break;

  case LIST_TABLE:         listTable(); break;

  default:
          printf ("Valid op code names and arguments are:\n");
          for (i = 0; i < 9; i++)
            printf ("  %-16s   %s\n", opcodes[i].name,
                                      opcodes[i].args);
  }
return;
}
 */
int
addName () {

/*  Fix the Op Code field in the Request Block header for this function;
 *  reset the value of rbPtr->response; post the RB to iNA
 */
    rbPtr->opcode = ADD_NAME;
    rbPtr->response = 0;
    nsPtr->property_type = response.propertyType;

  cqcommrb (rb_segment, &Status);
  if (Status != 0) {
    printf ("cqcommrb failed: %4X\n", Status);
    exit (1);
    }

/*  Wait for iNA to return the RB, and make sure the operation completed
 *  without error.
 */
  rb_segment = rqreceivemessage (responseMbx, 1000, NULL, &Status);
  if (Status != 0) {
    if (Status == E_TIME) {
      printf ("add_name failed: no response from iNA\n");
      exit (1);
      }
    printf ("rqreceivemessage failed: %4X\n", Status);
    exit (1);
    }

  if (rbPtr->response != OK_RESPONSE)
    printf ("add_name request failed: %s",
      nsResponseCodes[rbPtr->response]);

  return;
  }

/*  deleteName()
 *  ------------------------------------------------------------------
```

Figure 11.9 (*Continued*)

```
 */
int
deleteName () {

/*  Fix the Op Code field in the Request Block header for this function;
 *  reset the value of rbPtr->response; post the RB to iNA
 */
    rbPtr->opcode = DELETE_NAME;
    rbPtr->response = 0;
    nsPtr->property_type = response.propertyType;

  cqcommrb (rb_segment, &Status);
  if (Status != 0) {
    printf ("cqcommrb failed: %4X\n", Status);
    exit (1);
    }

/*  Wait for iNA to return the RB, and make sure the operation completed
 *  without error.
 */
  rb_segment = rqreceivemessage (responseMbx, 1000, NULL, &Status);
  if (Status != 0) {
    if (Status == E_TIME) {
      printf ("delete_name failed: no response from iNA\n");
      exit (1);
      }
    printf ("rqreceivemessage failed: %4X\n", Status);
    exit (1);
    }

  if (rbPtr->response != OK_RESPONSE)
    printf ("delete_name request failed: %s",
    nsResponseCodes[rbPtr->response]);

  return;
  }

/*  getValue()
 *  ------------------------------------------------------------------
 */
int
getValue () {

/*  Fix the Op Code field in the Request Block header for this function;
 *  reset the value of rbPtr->response; post the RB to iNA
 */
    rbPtr->opcode = GET_VALUE;
    rbPtr->response = 0;
    nsPtr->property_type = response.propertyType;
    response.valueLength = 256;

    cqcommrb (rb_segment, &Status);
    if (Status != 0) {
      printf ("cqcommrb failed: %4X\n", Status);
      exit (1);
      }
```

Figure 11.9 (*Continued*)

```
/*  Wait for iNA to return the RB, and make sure the operation completed
 *  without error.
 */
    rb_segment = rqreceivemessage (responseMbx, 1000, NULL, &Status);
   if (Status != 0) {
      if (Status == E_TIME) {
        printf ("get_value failed: no response from iNA\n");
        exit (1);
        }
      printf ("rqreceivemessage failed: %4X\n", Status);
      exit (1);
      }
    if (rbPtr->response != OK_RESPONSE) {
      printf ("get_value request failed: %s",
        nsResponseCodes[rbPtr->response]);
      return;
      }

    printf ("The value for %s (assumed to be a unique name) is:\n",
      cstr (response.nameString,response.nameString));
    for (i = 0; i < response.valueLength; i++) {
      printf ("%c%c ", hextab[(response.valueString[i] >> 4) & 0x0f],
                       hextab[(response.valueString[i]     ) & 0x0f]);
      if ((i % 20) == 19) printf ("\n");
      }
  }


/*  changeValue()
 *  ---------------------------------------------------------------------
 */
int
changeValue () {

/*  Fix the Op Code field in the Request Block header for this function;
 *  reset the value of rbPtr->response; post the RB to iNA
 */
    rbPtr->opcode = CHANGE_VALUE;
    rbPtr->response = 0;
    nsPtr->property_type = response.propertyType;

  cqcommrb (rb_segment, &Status);
  if (Status != 0) {
    printf ("cqcommrb failed: %4X\n", Status);
    exit (1);
    }

/*  Wait for iNA to return the RB, and make sure the operation completed
 *  without error.
 */
  rb_segment = rqreceivemessage (responseMbx, 1000, NULL, &Status);
  if (Status != 0) {
    if (Status == E_TIME) {
      printf ("change_value failed: no response from iNA\n");
      exit (1);
      }
    printf ("rqreceivemessage failed: %4X\n", Status);
```

Figure 11.9 (*Continued*)

```
    exit (1);
    }

  if (rbPtr->response != OK_RESPONSE)
    printf ("change_value request failed: %s",
    nsResponseCodes[rbPtr->response]);

  return;
  }


/*  deleteProperty()
 *  ------------------------------------------------------------------------
 */
int
deleteProperty () {

/*  Fix the Op Code field in the Request Block header for this function;
 *  reset the value of rbPtr->response; post the RB to iNA
 */
    rbPtr->opcode = DELETE_PROPERTY;
    rbPtr->response = 0;
    nsPtr->property_type = response.propertyType;

  cqcommrb (rb_segment, &Status);
  if (Status != 0) {
    printf ("cqcommrb failed: %4X\n", Status);
    exit (1);
    }

/*  Wait for iNA to return the RB, and make sure the operation completed
 *  without error.
 */
  rb_segment = rqreceivemessage (responseMbx, 1000, NULL, &Status);
  if (Status != 0) {
    if (Status == E_TIME) {
      printf ("delete_property failed: no response from iNA\n");
      exit (1);
      }
    printf ("rqreceivemessage failed: %4X\n", Status);
    exit (1);
    }

  if (rbPtr->response != OK_RESPONSE)
    printf ("delete_property request failed: %s",
    nsResponseCodes[rbPtr->response]);

  return;
  }

/*  getName()
 *  ------------------------------------------------------------------------
 */
int
getName () {

/*  Fix the Op Code field in the Request Block header for this function;
 *  reset the value of rbPtr->response; post the RB to iNA
```

Figure 11.9 (*Continued*)

```
*/
    rbPtr->opcode = GET_NAME;
    rbPtr->response = 0;
    nsPtr->property_type = response.propertyType;
  cqcommrb (rb_segment, &Status);
  if (Status != 0) {
    printf ("cqcommrb failed: %4X\n", Status);
    exit (1);
    }

/*  Wait for iNA to return the RB, and make sure the operation completed
 *  without error.
 */
  rb_segment = rqreceivemessage (responseMbx, 1000, NULL, &Status);
  if (Status != 0) {
    if (Status == E_TIME) {
      printf ("get_name failed: no response from iNA\n");
      exit (1);
      }
    printf ("rqreceivemessage failed: %4X\n", Status);
    exit (1);
    }

  if (rbPtr->response != OK_RESPONSE) {
    printf ("get_name request failed: %s",
      nsResponseCodes[rbPtr->response]);
    return;
    }
  printf ("The following names have property value type %4X:\n",
    response.propertyType);
  p.charPtr = (char *) &extraBuf->name_list[0];
  for (i = 0; i < extraBuf->count; i++) {
    printf ("%s\n", cstr (p.charPtr, p.charPtr));
    p.charPtr += strlen (p.charPtr);
    }

  return;
  }

/*  getSpokesman()
 *  ------------------------------------------------------------------------
 */
int
getSpokesman () {

/*  Fix the Op Code field in the Request Block header for this function;
 *  reset the value of rbPtr->response; post the RB to iNA
 */
    rbPtr->opcode = GET_SPOKESMAN;
    rbPtr->response = 0;
    nsPtr->property_type = response.propertyType;

  cqcommrb (rb_segment, &Status);
  if (Status != 0) {
    printf ("cqcommrb failed: %4X\n", Status);
    exit (1);
    }
```

Figure 11.9 (*Continued*)

```
/*  Wait for iNA to return the RB, and make sure the operation completed
 *  without error.
 */
  rb_segment = rqreceivemessage (responseMbx, 1000, NULL, &Status);
  if (Status != 0) {
    if (Status == E_TIME) {
      printf ("get_spokesman failed: no response from iNA\n");
      exit (1);
      }
    printf ("rqreceivemessage failed: %4X\n", Status);
    exit (1);
    }

  if (rbPtr->response != OK_RESPONSE) {
    printf ("get_spokesman request failed: %s",
      nsResponseCodes[rbPtr->response]);
    return;
    }

  printf ("The Ethernet address of the spokesman for %s is ",
    cstr (response.nameString,response.nameString));
  p.charPtr = (char *) extraBuf;
  for (i = 0; i < 6; i++) {
    printf ("%c%c ", hextab[(p.charPtr[i] >> 4) & 0x0f],
                     hextab[(p.charPtr[i]     ) & 0x0f]);
    }
  printf ("\n");
  nsPtr->extra_buffer_length = 4096;    /* clobbered by this function */
  return;
  }

/*  listTable()
 *  ---------------------------------------------------------------------
 */
int
listTable () {

/*  Fix the Op Code field in the Request Block header for this function;
 *  reset the value of rbPtr->response; post the RB
 */
    rbPtr->opcode = LIST_TABLE;
    rbPtr->response = 0;

  cqcommrb (rb_segment, &Status);
  if (Status != 0) {
    printf ("cqcommrb failed: %4X\n", Status);
    exit (1);
    }

/*  Wait for iNA to return the RB, and make sure the operation completed
 *  without error.
 */
  rb_segment = rqreceivemessage (responseMbx, 100, NULL, &Status);
  if (Status != 0) {
    if (Status == E_TIME) {
      printf ("list_table failed: no response from iNA\n");
      exit (1);
      }
```

Figure 11.9 (*Continued*)

```
    printf ("rqreceivemessage failed: %4X\n", Status);
    exit (1);
    }

  if (rbPtr->response == E_BUFF_SPACE)
    printf ("Local portion of database > 4 KB.  Output truncated\n");
  else if (rbPtr->response != OK_RESPONSE) {
    printf ("list_table request failed: %s",
      nsResponseCodes[rbPtr->response]);
    return;
    }

  if (extraBuf->count == 0) {
    printf ("Local portion of database is empty\n");
    return;
    }

/*  Interpret the contents of the returned table for display on the
 *  screen.  Because the strings make the size of each object in the
 *  database unknown at compile time, each object has to be interpreted
 *  individually; Use anyPtr to keep track of where we are in the table.
 */
  p.anyPtr = &(extraBuf->name_list[0]);
  printf ("NAME              TYPE  HEXADECIMAL VALUE\n");
  for (i = 0 ; i < extraBuf->count; i++) {
    if ((i % 24) == 23) {
      printf ("more? "); gets (commandLine);
      if (*commandLine == 'q') {
        return;
        }
      }
    printf ("%-16s  ", cstr (p.charPtr, p.charPtr)); /* property name */
    p.charPtr += strlen (p.charPtr) + 2;      /* skip unique flag too */
    printf ("%4X  ", *(p.wordPtr));                  /* property type */
    p.charPtr += sizeof (WORD) + 1;   /* skip property value type too */
    for (j = 0; j < *(p.wordPtr); j++) {
      printf ("%c%c ", hextab[(*(p.charPtr + j + 2) >> 4) & 0x0f],
                       hextab[*(p.charPtr + j + 2) & 0x0f]);
      if ((j % 16) == 15) printf ("\n                          ");
      }
    printf ("\n");
    p.charPtr += (*p.charPtr) + 2;              /* skip to next entry */
    }
  return;
  }

/*  main()
 *  ------------------------------------------------------------------
 */
int
main (int argc, char *argv[]) {

int             i;
WORD            Status;

/*  Initialization
 *  --------------
 *
 *  Be sure iNA is available before continuing, then allocate
 *  all the resources this job will need.
```

Figure 11.9 (*Continued*)

```
*
*       These consist of:
*           comm user ID
*           nameserver RB segment
*           response mailbox for RBs
*/

    root_job = rqgettasktokens (ROOT_JOB, &Status);
    a_token = rqlookupobject (root_job, "\6INARDY", NO_WAIT, &Status);
    if (Status != E_OK) {
      printf ("iNA is not available\n");
      return 1;
      }

    user_id = cqcreatecommuser (&Status);
    if (Status != E_OK) {
      printf ("cqcreatecommuser failed: %4X\n", Status);
      return 1;
      }

    rb_segment =
      rqcreatesegment (sizeof (struct name_server_rb), &Status);
    if (Status != E_OK) {
      printf ("rqcreatesegment failed: %4X\n", Status);
      return 1;
      }

    responseMbx = rqcreatemailbox (0, &Status);
    if (Status != E_OK) {
      printf ("rqcreatemailbox failed: %4X\n", Status);
      return 1;
      }

/*  Initialize the buffer pointers to be used for nameserver names
 *  and values.
 */
    nameBuf = (NAME_BUFFER *) response.nameString;
    propValBuf = (VALUE_BUFFER *) &response.valueLength;
    if ((extraBuf = (EXTRA_BUFFER *) malloc (4096)) == NULL) {
      printf ("malloc failed\n");
      return 1;
      }

 /*  Initialize the contents of the single segment we will be using for
  *  all request blocks in this program.
  *
  *       First, the header portion declared as RB_COMMON in cqcommon.h
  */
         rbPtr = (RB_COMMON *) buildptr (rb_segment, 0);
         rbPtr->reserved[0] = 0;
         rbPtr->reserved[1] = 0;
         rbPtr->length = sizeof (NAME_SERVER_RB);
         rbPtr->user_id = user_id;
         rbPtr->resp_port = 0xFF;
         rbPtr->rb_seg_tok = rb_segment;
         rbPtr->resp_mbox = responseMbx;
         rbPtr->subsystem = NAME_SERVER;
         rbPtr->opcode = LIST_TABLE; /* first operation */
         rbPtr->response = 0;
```

Figure 11.9 (*Continued*)

```
/*      Now, the nameserver arguments, declared as NAME_SERVER_RB
 *      in cqname.h
 */
        nsPtr = (struct name_server_rb *) rbPtr;
        for (i = 0; i < 6; i++) nsPtr->reserved[i] = 0;
        nsPtr->name_buffer_ptr =
          cqcommptrtodword ((void *) nameBuf, &Status);
        if (Status != 0) {
          printf ("cqcommptrtodword failed: %4X\n", Status);
          return 1;
          }
        nsPtr->unique_name_flag = 0xff;  /* Unique for this program */
        nsPtr->property_value_type = 0; /* Only unstructured values */
        nsPtr->pv_buffer_ptr =
          cqcommptrtodword ((void *) propValBuf, &Status);
        if (Status != 0) {
          printf ("cqcommptrtodword failed: %4X\n", Status);
          return 1;
          }
        nsPtr->extra_buffer_ptr =
          cqcommptrtodword ((void *)extraBuf, &Status);
        if (Status != 0) {
          printf ("cqcommptrtodword failed: %4X\n", Status);
          return 1;
          }
        nsPtr->extra_buffer_length = 4096;

/*  Greet the user with a list of the current contents of the database
 */
        listTable();

/*  Main Loop
 *  ---------
 *
 *      Get a string from the user, and dispatch it to the proper
 *      function, based on the opcode
 */
        for (;;) {
          doCommand();
          }
          return 0; /* never reached */
      }
```

`add_name`. The user supplies a name, property type, and property value, which are entered into the database. The *namesrv* program assumes the entry is unique and accepts only a character string for the value, but these restrictions are not imposed by iNA. No information is returned by iNA except a response code telling if the operation was successful or not. If the operation fails, *namesrv* displays a string indicating the reason for failure using the mnemonics defined in `cqname.h`.

`delete_name`. The user supplies a name, and iNA deletes all entries with a matching name, regardless of property type, from the database. No data is returned by iNA, only a response code.

`get_value`. The user supplies a name and property type, and iNA returns the value for the property in the value buffer supplied in the RB. The program sets the uniqueness Boolean to true, so iNA returns only the value found on the local system. The option to specify a value of false is not supported by the program, but if a value of false was specified, iNA would query the entire network to find all matching entries in the distributed database and return a list of values to the user.

`change_value`. The user supplies the same information as for `add_name`, and the value for the corresponding entry in the database is updated accordingly. This operation fails if the name and property type are not already entered in the database. No data is returned by iNA, only a response code.

`delete_property`. The user supplies both a name and a property type, and the entry is deleted from the database. Contrast this with the `delete_name` operation described previously. No data is returned by iNA, only a response code.

`get_name`. The user supplies a property type and a value, and iNA returns a list of the name portion of all matching entries in the database. The list of names is returned in the extra buffer pointed to by the RB. If the extra buffer is not large enough to hold the entire list, an error code is returned, but as much of the list as will fit is placed in the user's buffer.

`get_spokesman`. The user supplies a name and property type, and iNA returns (in the extra buffer) the Ethernet address of the computer that holds the corresponding entry. The spokesperson (spokesmachine?) mechanism is used by iRMX-Net for accessing systems that run iNA but for which there is no operating system software available to initialize the database locally. The HI commands *setname* and *loadname* can be used to make an iRMX system act as a spokesperson for other systems that do not support the iNA nameserver.

`list_table`. The user supplies no arguments; iNA returns the entire local portion of the database in the extra buffer. Using this option with the *namesrv* program produces the same result as the iRMX-Net *listname* command.

11.10 The Network Management Facility

Users have access to many of the parameters, statistics, and other information maintained by iNA through its Network Management Facility (NMF). Services provided by the NMF include:

- Asynchronous event notification. For example, an RB can be returned every time iNA encounters an invalid Transport layer addresses.
- Configuration information about iNA. Such as the maximum TPDU size or the number of virtual circuits that can be open at a time.

- Statistical information, such as the total number of transmitted datagrams or the total number of expedited data bytes transmitted over virtual circuits.
- Connection information, such as a list of all current CDBs (see the previous discussion of virtual circuits) or the local and remote TSAP IDs for a virtual circuit.
- Routing information. This information includes routing tables, configuration parameters, and statistics.
- A dump of iNA's memory. This option is useful to users who implement their own configurations of iNA.
- Remote boot operation. A diskless workstation can obtain an operating system image from a remote computer's disk file using this feature.

An HI command called *inamon* provides interactive access to many of iNA's NMF facilities. A simpler command, called *mynamon*, (pardon the pun) is given in Figure 11.10. *mynamon* prompts the user for an NMF object ID and displays the information that iNA returns about that object. NMF object IDs are 4-digit hexadecimal values used to identify the information the user is interested in seeing. The first two digits identify which part of iNA maintains the information, and the second two digits identify the particular object. iNA recognizes about 200 different object IDs. Their values are given in Appendix A of the *iNA 960 Programmer's Reference* manual. Values for the first two digits are the following:

20, 21, 25	Various Data Link implementations
31	Network
38	Static Routing
39	ES-IS Routing
40	Transport layer virtual circuits
41	Transport layer datagrams
80	iNA NMF itself (system time and version number)
81	Boot Server

Figure 11.10 Source code for *mynamon*, a program that allows a user to examine Network Management Function (NMF) objects interactively.

```
/***>  mynamon.c  <***************************************************
 *
 *  This program allows users to examine Network Management objects
 *  on the local system.
 *
 *      The user enters a network object ID in hex and gets back a
 *      hex dump of the returned object.  Returned items that are
 *      two or four bytes in length are assumed to be unsigned values
 *      and are displayed in decimal as well as hex.
 *
 ********************************************************************/
```

Figure 11.10 (*Continued*)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <rmxc.h>
#include <cqcomm.h>      /*  iNA Interface Function Prototypes        */
#include <cqcommon.h>    /*  RB common header structure               */
#include <cqnmf.h>       /*  NMF structs, opcodes, and response codes */

#include "nmfrespc.h"    /*  NMF response code strings                */

#define E_OK              0
#define NO_WAIT           0
#define E_TIME            1
#define ROOT_JOB          3

/*  Static and global variables
 */
TOKEN           user_id, root_job, a_token, rb_segment, responseMbx;

RB_COMMON       *rbPtr;
NMF_OBJECT_RB   *nmfPtr;

COMMAND_BUFFER  *commandPtr;
RESPONSE_BUFFER *responsePtr;

union {          /* pointer aliases */
  WORD           *wordPtr;
  DWORD          *dwordPtr;
  char           *charPtr;
  BYTE           *bytePtr;
  } p;

static  int      i, n;
static  WORD     Status;
static  char     reply[81];
static  char     hextab[] = "0123456789ABCDEF";

/*  showHexString()
 *  ---------------------------------------------------------------
 *
 *      Display count-many bytes in hex
 */
void
showHexString (unsigned char count, unsigned char *bytes) {
int     i;

  for (i = 0; i < count; i++) {
    printf ("%c%c ",    hextab[(bytes[i] >> 4) & 0x0F],
                        hextab[bytes[i] &0x0F]);
    if ((i % 16) == 15) printf ("\n");
    }
  printf ("\n");
  return;
  }

/*  nmfString()
 *  ---------------------------------------------------------------
```

Figure 11.10 (*Continued*)

```
 *
 *    Return a string from nmfResponseCodes for a response code value.
 */
char *
nmfString (WORD codeVal) {
struct      nmf_codes *rcPtr = &nmfResponseCodes[0];

  while (rcPtr->codeValue != 0xffff) {
    if (codeVal == rcPtr->codeValue) return rcPtr->codeString;
    rcPtr++;
    }
  return "Unknown NMF response code";
  }

/*  main()
 *  ---------------------------------------------------------------------
 */
int
main (int argc, char *argv[]) {

/*  Initialization
 *  --------------
 *
 *  Be sure iNA is available before continuing, then allocate
 *  all the resources this job will need.
 *
 *      These consist of:
 *          comm user ID
 *          nameserver RB segment
 *          response mailbox for RBs
 */

    root_job = rqgettasktokens (ROOT_JOB, &Status);
    a_token = rqlookupobject (root_job, "\6INARDY", NO_WAIT, &Status);
    if (Status != E_OK) {
      printf ("iNA is not available\n");
      return 1;
      }
    user_id = cqcreatecommuser (&Status);
    if (Status != E_OK) {
      printf ("cqcreatecommuser failed: %4X\n", Status);
      return 1;
      }

    rb_segment =
      rqcreatesegment (sizeof (struct nmf_object_rb), &Status);
    if (Status != E_OK) {
      printf ("rqcreatesegment failed: %4X\n", Status);
      return 1;
      }

    responseMbx = rqcreatemailbox (0, &Status);
    if (Status != E_OK) {
      printf ("rqcreatemailbox failed: %4X\n", Status);
      return 1;
      }

/*  Initialize the buffer pointers to be used for commands and replies
 */
```

Figure 11.10 (*Continued*)

```
    if ((commandPtr = (COMMAND_BUFFER *)
      malloc (sizeof (COMMAND_BUFFER))) == NULL) {
      printf ("malloc failed\n");
      return 1;
      }

    if ((responsePtr = (RESPONSE_BUFFER *)
      malloc (sizeof (RESPONSE_BUFFER) + 512)) == NULL) {
      printf ("malloc failed\n");
      return 1;
      }

/*  Initialize the contents of the segment we will be using for
 *  all request blocks in this program.
 *
 *      First, the header portion declared as RB_COMMON in cqcommon.h
 */
        rbPtr = (RB_COMMON *) buildptr (rb_segment, 0);
        rbPtr->reserved[0] = 0;
        rbPtr->reserved[1] = 0;
        rbPtr->length = sizeof (NMF_OBJECT_RB);
        rbPtr->user_id = user_id;
        rbPtr->resp_port = 0xFF;
        rbPtr->rb_seg_tok = rb_segment;
        rbPtr->resp_mbox = responseMbx;
        rbPtr->subsystem = NMF;
        rbPtr->opcode = READ_OBJECT;     /* always */
        rbPtr->response = 0;

/*      Now, the net management arguments,
 *      declared as NMF_OBJECT_RB in cqnmf.h
 */
        nmfPtr = (NMF_OBJECT_RB *) rbPtr;
        nmfPtr->reference = 0;  /* Local Agent Only */
        nmfPtr->resp_buf_ptr =
          cqcommptrtodword ((void *) responsePtr, &Status);
        if (Status != 0) {
          printf ("cqcommptrtodword failed: %4X\n", Status);
          return 1;
          }
        nmfPtr->resp_buf_length = sizeof (RESPONSE_BUFFER) + 512;
        nmfPtr->cmd_buf_ptr =
          cqcommptrtodword ((void *) commandPtr, &Status);
        if (Status != 0) {
          printf ("cqcommptrtodword failed: %4X\n", Status);
          return 1;
          }
        nmfPtr->cmd_buf_length = sizeof (COMMAND_BUFFER);

/*      Finally, the fixed fields of the command buffer
 */
        commandPtr->num_obj = 1;
        commandPtr->obj_info[0].object = 0x4001; /* conn. ID list */
        commandPtr->obj_info[0].modifier = 0;
        commandPtr->obj_info[0].length = 0;

/*  Main Loop
 *  ---------
```

Figure 11.10 (*Continued*)

```
 *
 *      Get a command string from the user.  Exit, if requested.
 *      Otherwise, accept an object id and, optionally, a VC id.
 */

    for (;;) {

      if (argc > 1) {
        printf ("\nEnter an object id in hexadecimal (%X): %s\n",
          commandPtr->obj_info[0].object, argv[1]);
        strcpy (reply, argv[1]);
        argc--; argv++;
        }
      else {
        printf ("\nEnter an object id in hexadecimal (%X): ",
          commandPtr->obj_info[0].object);
        gets (reply);
        }
      if ((*reply == 'e') || (*reply == 'q')) return 0;
      if (strspn (reply, " 0123456789ABCDEFabcdef") != strlen (reply)) {
        printf ("\nEnter q or e to exit,\nenter ");
        printf ("an ID from Appendix A of iNA Programmer\'s ");
        printf ("Reference,\nor press <enter> to accept ");
        printf ("the default value displayed\n");
        continue;
        }
      if (strlen (reply))
        commandPtr->obj_info[0].object = strtoul (reply, NULL, 16);
      if ((commandPtr->obj_info[0].object > 0x4080) &&
          (commandPtr->obj_info[0].object < 0x4093)) {
        printf ("Enter virtual circuit ID (%X): ",
                  commandPtr->obj_info[0].modifier);
        gets (reply);
        if (strlen (reply))
          commandPtr->obj_info[0].modifier = strtoul (reply, NULL, 16);
        }

/*  Post the RB to iNA, wait for reply, check results.
 */
      cqcommrb (rb_segment, &Status);
      if (Status != 0) {
        printf ("cqcommrb failed\n;");
        return 1;
        }

      a_token = rqreceivemessage (responseMbx, 500, NULL, &Status);
      if (Status == E_TIME) {
        printf ("No response from iNA\n");
        return 1;
        }
      if (Status != E_OK) {
        printf ("rqreceivemessage failed\n");
        return 1;
        }

      if (rbPtr->response != OK_RESPONSE) {
        printf ("Request failed: %s\n", nmfString (rbPtr->response));
        continue;
        }
```

Figure 11.10 (*Continued*)

```
      if (responsePtr->obj_info[0].status != E_OK_OBJ_CMND) {
        printf ("Request failed: %s\n",
          nmfString ((WORD) responsePtr->obj_info[0].status));
        continue;
        }

/*    We have a valid response from iNA; display it
 */
      p.bytePtr = &responsePtr->obj_info[0].value[0];
      switch (commandPtr->obj_info[0].object) {

        case 0x4001:        /* Connection ID vector */
          if (responsePtr->obj_info[0].length == 0) {
            printf ("There can be no virtual circuit connection IDs\n");
            break;
            }
          printf ("Virtual circuit connection IDs:\n");
          n = 0;
          for (i = 0; i < responsePtr->obj_info[0].length / 2; i++) {
            if (*p.wordPtr) {
              printf ("    %4X\n", *p.wordPtr);
              if ((++n % 24) == 23) {
              printf ("more? ");
              gets (reply);
              if ((*reply == 'q') || (*reply == 'n')) break;
              }
            }
          p.wordPtr++;
          }
        if (n == 0) printf ("    none\n");
        break;

      case 0x4081: printf ("Local TSAP selector:\n    ");
                   showHexString (responsePtr->obj_info[0].value[0],
                                  &responsePtr->obj_info[0].value[1]);
                   break;

      case 0x4082: printf ("Remote NSAP address:\n    ");
                   showHexString (responsePtr->obj_info[0].value[0],
                                  &responsePtr->obj_info[0].value[1]);
                   break;

      case 0x4083: printf ("Remote TSAP selector:\n    ");
                   showHexString (responsePtr->obj_info[0].value[0],
                                  &responsePtr->obj_info[0].value[1]);
                   break;

      case 0x8049: responsePtr->obj_info[0].value
                     [responsePtr->obj_info[0].length] = '\0';
                   printf ("System time is %s\n",
                     responsePtr->obj_info[0].value);
                   break;

      default:
        if (responsePtr->obj_info[0].length == 2)
          printf ("The value is 0x%X (%d)\n",
                  *p.wordPtr, *p.wordPtr);
```

Figure 11.10 (*Continued*)

```
        else if (responsePtr->obj_info[0].length == 4)
          printf ("The value is 0x%lX (%ld)\n",
                  *p.dwordPtr, *p.dwordPtr);
        else {
          printf ("The value is: ");
          if (responsePtr->obj_info[0].length > 16) printf ("\n");
          showHexString (responsePtr->obj_info[0].length,
                         &responsePtr->obj_info[0].value[0]);
          }
      }

    }
    return 0; /* never reached */
}
```

11.11 Data Link Operations

Like the other iNA modules, the Data Link layer can be accessed using the RB interface. Normally, the Data Link layer is accessed only from within iNA itself, but users are given direct access to this layer using a set of RB commands collectively called the External Data Link (EDL) interface. The EDL commands bypass the Transport and Network layers of iNA to provide users with the ability to program functions that would otherwise be impossible. You should note, however, that this increased functionality comes at a price. You can send and receive datagrams using the EDL, but you must code your application to handle the network protocols that might be used by different packets and the problems of unreliable delivery associated with datagram service. Applications that access the network through the Transport layer do not have to deal with these issues.

In addition to supporting the direct transmission and receipt of datagrams, the EDL provides RB functions for multicast filtering and raw packet processing. To understand these two types of functions, you must delve a bit into the Physical layer of the ISO diagram shown in Figure 11.1. First, note that this book always assumes that a LAN is implemented using Ethernet to connect the computers. Actually, iNA provides support for both Ethernet connections (IEEE Standard 802.3) and Token bus connections (IEEE Standard 802.4), using different device controllers, of course.[13]

Everything said so far applies equally to both types of network. Even the information that follows is true conceptually for both Ethernet and Token bus networks, but the details are given in Ethernet terms, just to be concrete.

When a packet is transmitted over the Ethernet, it is sent simultaneously to all device controllers connected to the cable, and all controllers

[13]iNA can also be configured to support other device controllers.

examine the packet header to see if the 6-byte address contained in the packet matches the value embedded in that particular controller. Normally, a controller simply ignores all packets for which the address is not an exact match, but two mechanisms can be used that allow the device controller to accept additional packets and make them available to the Data Link layer. One is called the *broadcast address mechanism*, which uses a special address, 0xFFFFFFFFFFFF, that all Ethernet device controllers accept if they are programmed to do so. The other mechanism, called *multicast addressing*, causes the device controller to accept packets that contain any of several addresses in their header field. A special case of multicast processing, called *promiscuous mode*, causes the device controller to accept *all* packets that pass along the Ethernet cable. The crucial concept here is that a single device controller is not restricted to accepting only those packets specifically addressed to it.

The term *multicast filtering* refers to the ability of an application to determine which network packets are accepted by the local system and which are ignored. For example, our transport layer datagram example *timesrv* could invoke multicast filtering so that its unspecified NSAP address would actually result in the receipt of datagrams from a selected set of remote client machines, rather than all remote machines that sent packets using the proper TSAP ID. The EDL provides two ways to control multicast filtering. One, using the `configure` RB opcode, allows the user to add addresses to (or remove address from) the device controller's list of recognized multicast addresses. The second method is to use the `mc_add` opcode to set up a list of addresses used to perform multicast filtering by the Data Link software.

Normally, all Data Link communication is controlled by matching Link Service Access Point (LSAP) IDs between the sender and the receiver. You have already seen that iNA's Network layer uses LSAP ID 0xFE when you looked at the remote NSAP structure within a `ta_buffer`. This same LSAP ID is used by the nameserver, which operates as a Transport layer application from within iNA. Another LSAP ID (0x08) is used by iNA's NMF module when it accesses information about remote systems to examine or modify data maintained there. Users who wish to perform EDL datagram operations would choose their own LSAP IDs, ensuring that the selected IDs do not conflict with others that might already be in use.

A special LSAP ID, 0x99, is reserved for a special Data Link service, called the raw EDL (RAWEDL) interface. Use of this LSAP ID, along with promiscuous-mode multicast filtering, allows an iRMX application to receive all Ethernet packets not specifically addressed to another ISO LSAP on the local machine. That is, an application can receive network packets that are in the ISO format, but are not specifically addressed to the local machine, and it can receive packets that do not follow the ISO format for packet structure. Receiving packets addressed to other computers can be used to develop a network analyzer program to look at network traffic

loads, isolate connectivity problems, and the like. Receiving non-ISO packets means that an application can be written that will support multiple protocol stacks concurrently.

No problems occur with having multiple types of network share a single physical link, such as a single Ethernet cable. For one thing, each packet sent over the Ethernet includes the address of the destination computer. Two Novell network nodes can send Ethernet packets to each other, which will simply be ignored by all the other computers connected to the same cable simply because they do not have the same addresses. In addition, every network protocol, whether it is ISO, TCP/IP, Novell, DECNET, or any other, includes header information in each packet that it transmits over the network.

When a packet arrives that survives multicast filtering, the Data Link layer performs certain checks, such as a CRC computation, to verify that the packet has not been corrupted and does indeed conform to the format expected for that protocol. Packets that fail these protocol checks are simply discarded. The idea is that a lost packet will result in some higher-level software detecting a communication problem so that the packet will be transmitted again.[14]

iNA's RAWEDL interface allows an application to receive Ethernet packets that fail the Data Link's check for valid ISO protocol format. This means that a single device controller, operating with a single Ethernet address, could be programmed to accept ISO, TCP/IP, and IPX network traffic. Thus, if the Data Link receives a packet not in proper ISO format, it sends it to an application that has posted a `raw_post_receive` RB with an LSAP ID of 0x99. That application can then examine the packet to determine what protocol it does adhere to, and pass it on to the Networking module for the proper protocol stack, or discard it if it cannot be recognized. A TCP/IP protocol stack for iRMX that coexists with the present ISO functions provided by iNA is under development at Intel, as mentioned earlier.

[14]This action is the reason that datagram processing is considered unreliable: if a packet somehow becomes corrupted, the sender is not notified. The different networking protocols in use are designed so that a packet that is valid for one protocol will always appear to be invalid for all other protocols. A Novell node using Novell's IPX network protocols would never accept a TCP/IP packet, for example, even if it contained the proper 6-byte Ethernet address for the node.

Chapter

12

iRMX for Windows

12.1 Overview

DOS, Windows, and iRMX all bring their own characteristics to iRMX for Windows. DOS brings a simple operating system whose popularity has engendered a tremendous amount of application software, as well as a large body of knowledgeable users and developers. DOS, however, is constrained in the areas of memory management, multitasking, and user interface design. Windows addresses all three of these DOS problems, and improves on them as well. Windows also provides good facilities for sharing information among different applications. iRMX's list of contributions to the equation is long, including true multitasking within applications, efficient real-time task management, and good support for customizing the operating system itself.

Two or three operating systems (depending on whether you want to call Windows an operating system or not) that run concurrently on a single computer extract penalties. Each of the three adds its own memory requirements to the system, and each draws on the processing power available from the CPU. The payoff, however, is that the whole is greater than the sum of its parts. By drawing on the strengths of each of the components, iRMX for Windows brings more than just real-time computing to the DOS environment.

iRMX for Windows with DOS alone provides extended memory management and access to iRMX's peer-to-peer networking capabilities, for example. Adding Windows to the configuration provides real-time applications with access to the Dynamic Data Exchange (DDE) mechanism for sharing information among Windows and iRMX applications, and extends the DDE mechanism to include network links among applications, which Windows alone does not support.

Since the design of the iRMX operating system has already been discussed in earlier chapters, this chapter presents iRMX for Windows in

terms of the features it adds to iRMX III. iRMX for Windows actually adds features to DOS, Windows, and iRMX in an integrated fashion, so the discussion does not always maintain the simple view of iRMX for Windows as iRMX with some added features. Nevertheless, that view provides a convenient framework for this information, so this chapter is organized as a presentation of features that fall into the following categories:

- Console management.
- File system compatibility.
- Interrupt management.
- System call compatibility.
- Memory management.
- VM86 protected mode extensions.
- Windows compatibility.
- Network compatibility.
- Run-time configuration.

Many of these topics have been considered tangentially in earlier chapters, and some of these features, such as run-time configuration, will no longer be unique to iRMX for Windows when they are incorporated into other versions of iRMX.

Note that Windows compatibility and network compatibility are included in the preceding list. Despite the name of the operating system, Windows is not required to run iRMX for Windows. Likewise, you do not need a network to run iRMX for Windows, but support for Windows and support for networking is always available.

Instructions for using these facilities are well documented in the iRMX for Windows documentation set. This chapter provides a guide to the facilities and some background on the issues involved in allowing DOS, Windows, networking, and iRMX to coexist on a single computer system.

12.2 Console Ownership

The first feature an iRMX for Windows user normally encounters is the management of the console when DOS, iRMX, and possibly Windows are all running concurrently. *Console* refers to the keyboard and monitor connected directly to the host computer, not terminals that might be connected to the computer over serial links, and not monitors that might be connected to the system through additional display adapters beyond the first. *Ownership* refers to what program is to receive characters typed on the keyboard and what program is to display its output on the monitor at any moment.

Console ownership is switched between DOS and iRMX by entering <alt-SysRq> from the keyboard. Switching the keyboard from the DOS side to iRMX or back is relatively straightforward, but control of the monitor is a bit more complex. For one thing, the information displayed on the screen should be preserved as control is switched back and forth between the two operating systems, but this is subject to two limitations.

First, DOS programs that reset the video adapter destroy the image of the iRMX screen. Before switching to iRMX, it is necessary to restore the display adapter to text mode, and you must press <enter> on the iRMX side to display a new prompt on the screen.

Second, graphics programs (such as Windows) are incompatible with the iRMX use of the console, and <alt-SysRq> is not recognized when these graphics programs are running. Note that this limitation is not in effect when *command.com* is being run from Windows because *command.com* runs with the display adapter set to text mode.

But what happens if an iRMX application tries to generate output while DOS owns the console, or vice versa? The iRMX side will store the output in a buffer and display it on the screen when control returns, but output by DOS programs is discarded when iRMX owns the console.

The console switch between DOS and iRMX can be invoked from programs as well as from the keyboard. For example, Figure 12.1 is an iRMX program that makes a DOS system call to display a message on the screen using the *rqe_dos_request()* system call described in section 12.5.1. Before calling *rqedosrequest()*, the code programmatically switches to the DOS console by invoking the iRMX for Windows system call, *rqealtsysreq()*, and then switches back to the iRMX prompt before exiting the program. When the program is run from the iRMX prompt, the screen switches to the DOS prompt for a fraction of a second, generates its output on the screen, and returns to the iRMX prompt. Manually switching back to the DOS prompt reveals the message on the DOS screen.

Figure 12.1 iRMX program to display a string by making a DOS system call. (This program does not produce legible output when run under Windows.)

```
/***>  hellodos.c  <************************************************
 *
 *  iRMX program to display "Hello, World!" by invoking a DOS system
 *  call to print a string.
 *
 ************************************************************************/
#include <rmxc.h>

#define         NULL                    (void far *) 0
#define         TSRCONTEXT              0
#define         NONE                    0
#define         TRUE                    1
#define         DS_DX                   2
#define         toRMX                   3
```

Figure 12.1 (*Continued*)

```
#define         toDOS                   4
#define         threeSeconds            300

extern BYTE
rqealtsysreq (BYTE, WORD far *);

int
main (int argc, char *argv[]) {

BYTE            helloWorld[] = "Hello, World!\r\n$";
DOS_DATA_STRUCT dosRegisters;
BYTE            consoleCode;
WORD            Status;

    dosRegisters.x.int_num = 0x21;
    dosRegisters.x.tsr_flags = TSRCONTEXT;
    dosRegisters.x.reg_ax = 0x0900;
    dosRegisters.x.xfer_data = TRUE;
    dosRegisters.x.src1_xfer_pair = DS_DX;
    dosRegisters.x.src2_xfer_pair = NONE;
    dosRegisters.x.dest1_xfer_pair = NONE;
    dosRegisters.x.dest2_xfer_pair = NONE;
    dosRegisters.x.src_ptr_1 = helloWorld;
    dosRegisters.x.src_count_1 = sizeof (helloWorld);

    consoleCode = rqealtsysreq (toDOS, &Status);
    rqedosrequest (&dosRegisters, threeSeconds, &Status);
    consoleCode = rqealtsysreq (toRMX, &Status);
    rqexitiojob (0, NULL, &Status);

  }
```

The program can also be invoked directly from Windows by running the *wterm* demonstration application provided with iRMX for Windows, or the *WinTerm* terminal emulator available from Markefield Software. These programs allow users to interact with the iRMX console through a Windows window. Running the sample program from *wterm* or *WinTerm* results in output to the screen, but the message and ASCII string cannot be read because the screen is in graphics mode. Part of the Windows display will be changed (the upper left corner), but no text appears.

The *rqealtsysreq()* function is currently supplied in its own library, `:sd:rmx386/demo/altsys/altsys.lib`. The function prototype is:

```
BYTE
rqealtsysreq (          BYTE                functionCode,
                        WORD far *          exceptionPtr);
```

The possible values for `functionCode` are:

1 Acquire lock
2 Release lock

3 Switch to DOS
4 Switch to iRMX
5 Inquire ownership

The lock referenced for values 1 and 2 is a mutual exclusion mechanism that can guarantee that only one program will attempt to change ownership of the console at one time. The lock was not used in the sample code because the issue is not a crucial. Value 5 is used to determine which operating system currently owns the console. The function returns a Boolean value indicating the owner of the console (0 = DOS, 0xFF = iRMX). For functions 3 and 4, the value indicates the owner before the call was made. For functions 1 and 5, it indicates the current owner. No return value is defined for function 2.

12.3 File System Compatibility

The DOS file system is similar to the iRMX file system in three ways. They both:

- Use a tree structure consisting of directories and files.
- Do not differentiate between uppercase and lowercase letters in file and directory names.
- Allow files to be hidden and/or read-only.

There are, however, significant differences between the two file systems. First, an iRMX file system incorporates the notion of file system users, with different users having different access privileges for individual files and directories. iRMX file attributes are also maintained separately for different users, and the attributes (or permissions) include deletion, reading, appending to the end, and updating. Also, "hidden" is not an attribute for iRMX files, but an effect of a file-naming convention. Hidden iRMX files have names that begin with `r?` or `R?`.

DOS file and directory names follow the 8.3 naming rule (eight characters, an implied dot, and a three character extension), whereas an iRMX file system allows up to 14 characters in virtually any combination for file and directory names.

The two operating systems also use totally different data structures for representing a file system on a disk volume. The iRMX file system structure was introduced in section 8.4 and is documented in the *iRMX Command Reference*, volume 10 of the iRMX for Windows documentation set. The structure of the DOS file system is documented in a number of different places, including the *Disk Explorer* manual provided with the Norton Utilities product for DOS, or the technical reference manuals available from Microsoft for the various versions of DOS.

A DOS file system is required for running iRMX for Windows. The soft-

ware is installed in a DOS partition, and DOS commands are used to initialize the iRMX operating system. Beyond that, you can choose to have any iRMX volumes or not. An iRMX volume can take the form of an entire hard disk drive, a partition on a hard drive, or a diskette. The issues involved in deciding whether to use iRMX volumes or not include convenience, performance, functionality, and security.

The convenience of an iRMX volume is the freedom to use 14-character file and directory names rather than the 8.3 rule for DOS. On the other hand, the DOS file system on a floppy is a convenient way to transfer files from one platform to another. Because DOS is so pervasive, almost all operating systems (including iRMX) can read and write DOS-formatted diskettes.

The performance issue concerns the speed and predictability of the two file system implementations. Data transfers using an iRMX file system are generally faster than DOS, although there are a number of side issues here, notably the hardware or software disk caching that is often done on DOS systems. On the other hand, any caching scheme improves average performance at the expense of response time predictability because a cache miss involves much more overhead than a cache hit. Many real-time systems cannot tolerate such indeterminacy.

Presently, iRMX volumes on an AT platform do not provide any functions not available for DOS volumes other than access protection. One possible difference in function deserves mention, however, in part to clarify some confusing terminology.

Both iRMX III and DOS provide a command named *mirror*. Aside from the fact that the command is not presently available for iRMX for Windows, the two operating systems use the terms in totally different ways. The iRMX command is used to initiate a mode of operation in which everything written to one disk volume is also written to a second volume to ensure high reliability in the face of possible disk failure. The two images of the disk volume are continuously verified to ensure no discrepancies exist. iRMX mirroring can also yield performance advantages if the user elects to have alternate read operations directed to alternate volumes in the mirror set because the system overlap pairs of operations in time. The DOS command, however, is somewhat of a misnomer, since no copying of user data is invoked by that command. Rather, the DOS *mirror* command saves a snapshot of critical parts of a DOS file system (the file allocation table and the partition table) that can be used to recover from a user accidentally formatting a disk. iRMX provides similar functionality to DOS mirroring by allowing the user to reserve space for a copy of the fnode file when formatting a volume. The `backup` option of the *shutdown* command creates a copy of the fnode file in this save area, which can be used to recover accidentally erased or physically damaged files using the *diskverify* utility.

The final issue that might make an iRMX disk volume desirable is system security. Because a DOS file system has no provision for user identifi-

cation or access rights, iRMX must treat all files and directories on all DOS volumes as having full access rights for the World user. iRMX does not purport to be a secure time-sharing system (anyone with access to the development tools for the system could write a program to run with Super user privileges), but the iRMX file system's protection features do prevent accidental destruction of system files and casual unauthorized access to other users' files. When an iRMX system is to be shared among several users, the protected file system can, at the least, prevent unintended conflicts in using the system. Even a system used only by a single individual can benefit from the protected file system. Provided the individual makes a habit of doing development work as a user other than Super and becomes Super only for system administration chores, much inadvertent damage to the file system can be avoided.

12.3.1 Accessing a DOS volume from iRMX

All iRMX for Windows installations must be able to access DOS files from iRMX. This access is accomplished by the iRMX Encapsulated DOS file driver, EDOS. The word *encapsulated* in the name of this driver refers to the fact that it is used with the version of iRMX that encapsulates DOS as a VM86 task. There is nothing encapsulated about the file driver itself.

When file drivers were introduced in chapter 8, it was noted that the file driver acts as an intermediary between application tasks that make iRMX BIOS system calls and a device driver that performs the actual I/O operations involving interaction with the hardware device controller. No iRMX device drivers for disks or other devices, however, directly access DOS disk volumes. Rather, the EDOS file driver performs its I/O operations by making calls to DOS itself, as described later. This technique ensures that the two operating systems do not interfere with each other, but you generally pay a performance penalty for accessing devices through EDOS compared to using a file driver that communicates directly with an iRMX device driver.

Of course, EDOS cannot make a silk purse out of a sow's ear. For example, if you call *rqsgetdirectoryentry( )* with a connection to a directory on a DOS volume, you will get back a data structure that looks just like what you get back from the same call for an iRMX volume. The contents of the entry, however, will be limited to the DOS 8.3 format, with the period in the file name explicitly present to match the iRMX file naming rules. Going the other way, if you try to create a file or directory on a DOS volume using a name too long for the DOS system, any extra characters are silently dropped from the name, the same way DOS 5.0 'supports' long file names. If you try to create a file or directory with an illegal DOS file name (two or more periods, for example) the EDOS file driver will reject the request.

EDOS converts files that have names starting with `R?` or `r?` into DOS hidden files, without the `r?` part of the name. Likewise, DOS hidden files

appear to the iRMX user as if they had `r?` at the beginning of their names. The iRMX *dir* command displays invisible files if you include the `invisible` (or just `i`) parameter. For example,

```
iRMX> dir $ invisible                                    [1]
```

displays the names of all files in the current directory (`$`), including hidden ones.

Finally, EDOS cannot add the iRMX file access protection mechanism to a DOS file system. Commands like *permi* do not return error messages, but all users are treated as the iRMX World user. The DOS `system` and `archive` attributes are not supported at all by EDOS, but the `read-only` attribute can be set for any file by using the *permit* command:

```
iRMX> permit a_file nr u=world                           [2]
```

This command is read, "Permit `a_file` to have no access but reading for user `world`." Any numerical value between 0 and 0xFFFF could have been substituted for `world` with the same effect because all iRMX user IDs are treated as the World user (ID 65535). Giving permission for deletion, reading, or update gives full access to the file for all users.

12.3.2 Accessing an iRMX volume from DOS

When iRMX for Windows is running, you can use *sysload* to install the iRMX-Hosted DOS File Server (RHDFS) job.[1] This iRMX job provides support for the DOS command *rmxuse*. (How iRMX jobs and DOS programs communicate with each other is discussed shortly.) The *rmxuse* command is used to map iRMX logical names to DOS drive letters. The command includes the option to perform an iRMX attachdevice to create the iRMX logical name if it does not already exist. The following are examples that show how the command can be used. (Note the use of `C:>` as our generic DOS prompt; *rmxuse* must be given as a DOS command, not iRMX.)

```
C:> rmxuse F: :SD:                                       [3]
```

The iRMX logical name `:SD:` (the root directory of the system device) can now be referenced from DOS as drive `F:`.

```
C:> rmxuse G: :1: \p=system1 \r                          [4]
```

[1]You can load the RHDFS or Standard Mode Windows job, but not both.

The device name `system1` is attached as the iRMX logical name `:1:`, and can now be referenced from DOS as drive `G:`. The `/r` switch indicates that `system1` is the name of a remote computer system rather than the name of a DUIB on the local system. You cannot do silly things with RHDFS and *rmxuse*, like map a DOS device for which there is an iRMX logical name onto another DOS drive letter:

```
C:> rmxuse H: :b: \p=b_dos                                    [5]
```

This command, if it worked, would let you refer to the DOS diskette in drive `B:` as `H:`. DOS provides its own commands, *subst* and *join*, for doing this sort of command directly without going through an iRMX job. For more examples of *rmxuse*, see the *iRMX Command Reference*.

12.4 Interrupt Management

A common theme throughout the remainder of this chapter is how the processor responds to interrupts in the various contexts in which it may be operating. This section provides an overview of how interrupts are handled. You may want to refer back to chapter 5 for background on the hardware features mentioned here.

An i386 or later processor can operate in real, protected, or VM86 mode. When iRMX code is running, the processor is always in protected mode. When Windows is running the processor is in protected mode, but it puts the processor into real mode to run non-Windows (traditional DOS) applications.[2] iRMX for Windows puts the processor in VM86 mode whenever DOS is running without Windows and whenever Windows tries to put the processor in real mode to run a DOS application. Thus, any given hardware- or software-generated interrupt request could be destined for an iRMX protected-mode interrupt handler, a Windows or Windows application protected-mode interrupt handler, or a DOS real-mode interrupt handler.

In real mode, the processor gets the address of the handler for an interrupt level from a vector of 256 pointers stored in the lowest 256 doublewords of memory. In protected mode, the address of a handler is determined from one of the 256 entries in the Interrupt Descriptor Table (IDT). In VM86 mode, interrupts always vector into the IDT, and the operating system can then either call the corresponding real-mode interrupt handler or process the interrupt itself.

---

[2]Windows has its own modes of operation, but it runs only in its standard mode with iRMX for Windows, so its 386 enhanced mode is ignored in this discussion.

managed by DOS and DOS applications, the protected-mode IDT managed by Windows and Windows applications, and the protected-mode IDT managed by iRMX. iRMX for Windows effectively merges the Windows IDT into its own because the processor can support only one IDT. Managing interrupts destined for DOS's real-mode interrupt vector or the Windows IDT is the job of a routine called the VM86 Dispatcher. When an interrupt destined for Windows occurs, the VM86 dispatcher (using a program called *smw.job*) simply makes a far call to the interrupt handler provided by Windows.

Handling interrupts destined for DOS's real-mode interrupt vector is more complicated. In VM86 mode (DOS always runs in VM86 mode), the processor causes a GP fault (interrupt level 13) if a program tries to execute an "IOPL-sensitive" machine instruction, and the privilege level of the current code segment is numerically greater than the I/O Privilege Level (IOPL) of the processor.

IOPL-sensitive instructions include *int* instructions (used by DOS programs to make system calls), as well as those instructions that enable or disable interrupts or perform I/O transfers. GP faults that occur while the processor is in VM86 mode can be either emulated or ignored by the VM86 dispatcher. For example, by controlling attempts by DOS programs to disable interrupts, the VM86 dispatcher can preserve real-time responsiveness for iRMX tasks at the expense of DOS or Windows performance.

There are two choices for the processor's IOPL when DOS is running. If the IOPL is 0, DOS interrupts cause GP faults, but if the IOPL is 3, the microprocessor does not trap IOPL-sensitive instructions, and DOS programs run with minimal interference from iRMX (at the expense of iRMX's real-time responsiveness). A configuration option called *interrupt virtualization* is selected when an iRMX for Windows system initializes to determine the IOPL to be used when DOS is running. If the `:config:rmx.ini` file contains an entry, `VIE=000h`, in the [`DISPJ`] section, then interrupt virtualization is disabled, the IOPL is set to 3 when DOS runs, and iRMX interferes only minimally with DOS.

If the entry is `VIE=0FFh`, then interrupt virtualization is enabled, the IOPL is set to 0 when DOS runs, and DOS interferes only minimally with iRMX. If I/O-intensive DOS applications need to run efficiently while iRMX is maintaining real-time performance, the solution is to have iRMX manage the I/O operations. For example, an iRMX SCSI device driver can be loaded with iRMX for Windows to provide high-performance disk access to DOS applications even though interrupt virtualization is enabled.

12.5 System Call Compatibility

In chapters 9 and 10 you saw that iRMX application developers have access to the same mechanisms for adding device drivers, system calls, and type managers as the engineers who developed iRMX itself. Thus, it

should come as no surprise that the same mechanism used by the EDOS file driver to make DOS system calls is also available to iRMX for Windows developers. In addition, DOS programs can invoke many (but not all) iRMX system calls, and can use iRMX as a DOS extender (which is a program that allows a real-mode application to use protected-mode features of an 80386 microprocessor). After all, the iRMX for Windows exists to provide facilities for iRMX and DOS applications to interact with each other. The facilities for iRMX-to-DOS and DOS-to-iRMX interactions are discussed in the next two sections.

12.5.1 iRMX access to DOS system calls

DOS programs make DOS system calls by loading parameter values into registers and executing an *int21* instruction. For example, loading register `ax` with 0x0900 and registers `ds:dx` with a pointer to a `$`-terminated string causes the string to be displayed on the DOS console. The following is a sample DOS program that uses in-line assembly code to display a message:

```
#include <stdio.h>
int
main (int argc, char * argv[]) {
char far *a'string=''hello, world\n\r$'';

  asm {
    mov ax,0x0900
    lds dx,a'string
    int 0x21
    }
  return 0;
}
```

Most C compilers for DOS allow you to set up the registers and execute the *int* instruction with a library call, as the following example shows:

```
#include <stdio.h>
#include <dos.h>

int
main (int argc, char * argv[]) {
char      *a'string = n\r$'';
union     REGS  regs;
struct    SREGS sregs;

  regs.x.ax = 0x0900;
  regs.x.dx = FP_OFF (a'string);
  sregs.ds = FP'sEG (a'string);
  intdosx (&regs, &regs, &sregs);

  return 0;
}
```

Why not just run the equivalent program from iRMX? After all, the code runs on the same processor as DOS, so the registers and interrupts must be

the same, right? Not exactly. Remember that DOS runs as a VM86 task (see section 5.5) when iRMX for Windows is running. If this program were compiled and bound using iC-386 and bnd386 to run under iRMX, execution of the *int 21* instruction would be vectored into the iRMX interrupt descriptor table, not the DOS interrupt vector in low memory. Because iRMX does not process DOS system calls, the program would fail.

Instead, iRMX for Windows programs can call *rqe_dos_request()* to make DOS system calls while running under iRMX. The mechanism to support *rqe_dos_request()* consists of two parts, an iRMX first-level job that responds to the iRMX call and a DOS program that performs the DOS system call on behalf of the iRMX application. This DOS program must always be available when iRMX for Windows is running to provide this support, so it is run as a DOS Terminate and Stay Resident (TSR) program, with the appropriate name of *rmxtsr*.

A TSR is a program run from the DOS command line but which executes the DOS *terminate and stay resident* system call to return control to the command processor while remaining in DOS memory. Since DOS is a single-threaded operating system, a TSR can execute only by responding to interrupts. An analogy would be to run an iRMX command using the *background* command or the *sysload* utility, but iRMX tasks do not have to be interrupt-driven to run.

On the iRMX side, the first-level job that responds to *rqe_dos_request()* is called the *VM86 Dispatcher*, which acts as the manager for all DOS VM86-mode operations. As such, it can send messages to *rmxtsr* by means of real-mode software interrupts. Similar to the *intdosx()* function of DOS, *rqe_dos_request()* takes a data structure that contains the values of the processor's registers as one of its arguments. This data structure must be more elaborate than REGS and SREGS in the previous examples, however. The concepts involved can be explained by examining how the "hello world" program would be coded to invoke the DOS print string function from iRMX. The code is given in Figure 12.1.

Looking at the sequence of assignments to fields in `dosRegisters` (which includes the x specifier to select word-wide registers), the first field of interest is `int_num`. This field tells *rmxtsr* which real-mode software interrupt number to use to invoke the system call. Most DOS system calls use interrupt-level 0x21, but DOS uses other levels as well. Furthermore, this interrupt level can be set to values for invoking ROM-BIOS functions directly, if desired.

The `tsr_flags` field is a binary variable. If it is 0, as in the example, the DOS function is executed using the context of *rmxtsr*. A value of 1 invokes the function using the context of the currently running DOS application.[3]

[3]The context of a DOS program is stored in a 256-byte data structure called the Program Segment Prefix (PSP). The PSP contains command line arguments, settings of environment variables, a disk buffer, and other information.

The register initialization for register ax is straightforward, but loading the pointer to the string that is to be printed into the DS:DX register pair illustrates a major issue in making DOS calls from iRMX. The iRMX application occupies memory above 1 MB using the processor's protected-mode addressing mechanism, but *rmxtsr* occupies memory below 1 MB and uses the processor's real-mode addressing mechanism. Therefore, *rmxtsr* (or DOS, for that matter) cannot access the string declared in the iRMX program. It must be copied from iRMX memory to DOS memory, and *rmxtsr* provides four buffers for just this purpose.

Two buffers are available for copying information from iRMX applications to DOS, and two are available for copying information in the opposite direction. The xfer_data field is a Boolean variable that identifies whether any of these buffers are to be used. In the sample code, src_ptr_1 was set to point to the dollar-terminated string to be used for the system call, and src_count_1 was set to the length of the string so *rmxtsr* knows how many bytes to copy into its internal real memory buffer. The srcl_xfer_pair is then set to the constant DS_DX to tell *rmxtsr* that a pointer to this buffer is to be loaded into the processor's ds:dx registers before executing the *int 21* instruction. By setting the other three xfer_pair fields to NONE, *rmxtsr* knows that the other three transfer buffers are not used for this call.

Note that *rmxtsr* knows nothing of DOS or ROM-BIOS calls to provide its services to iRMX clients. By keeping the interface between it and iRMX tasks fully general, it can accommodate the invocation of any function that might be invoked by a software interrupt in the DOS environment. The only restriction is that several DOS functions should not be invoked from any TSR, including *rmxtsr*. These functions are listed as unsupported in the *iRMX System Call Reference* manual documentation for *rqe_dos_request()*.

12.5.2 The DOS Real-Time Extension: making iRMX system calls from DOS

A mechanism called the DOS Real-Time Extension (RTE) is used by DOS programs to make iRMX system calls. Although the RTE interface to iRMX system calls is always the same, push parameters onto the stack and execute an *int B8* instruction), there are several ways in which the *int* instruction is actually handled, which are described in section 12.4.

The RTE code that handles *int B8* instructions examines the arguments on the caller's stack and a function code the caller placed in the processor's ax register, then invokes the appropriate iRMX system call based on the function code. The RTE recognizes codes for 28 different Nucleus system calls, plus two special RTE functions that allow DOS programs to copy information between iRMX protected-mode segments and DOS real-mode segments. The 30 RTE calls are listed in Table 12.1.

As you should expect by now, the mechanism that iRMX for Windows

TABLE 12.1 iRMX System Calls That Can Be Made By DOS Programs Using the RTE Mechanism

RTE code	Corresponding iRMX system call
0	*rqcreatemailbox( )*
1	*rqdeletemailbox( )*
2	*rqsendmessage( )*
3	*rqsenddata( )*
4	*rqreceivemessage( )*
5	*rqreceivedata( )*
6	*rqcreatesemaphore( )*
7	*rqdeletesemaphore( )*
8	*rqsendunits( )*
9	*rqreceiveunits( )*
10	*rqcreateregion( )*
11	*rqdeleteregion( )*
12	*rqsendcontrol( )*
13	*rqreceivecontrol( )*
14	*rqacceptcontrol( )*
15	*rqcreatesegment( )*
16	*rqdeletesegment( )*
17	*rqgetsize( )*
18	*rqegetaddress( )*
19	*rqecreatedescriptor( )*
20	*rqedeletedescriptor( )*
21	*rqechangedescriptor( )*
22	*rqcatalogobject( )*
23	*rquncatalogobject( )*
24	*rqlookupobject( )*
26	*rqgettasktokens( )*
27	*rqgettype( )*
28	*rqsleep( )*
30	*rqereadsegment( )*
31	*rqewritesegment( )*

uses to implement the RTE is also available for systems programmers to use to develop their own protected-mode extensions (PMEs) to DOS, as you will see in section 12.8. For example, a DOS program could invoke other system calls besides those listed in Table 12.1 by invoking a PME that recognized function codes for other iRMX calls. Because the VM86 job that owns the DOS task is a first-level job created after the BIOS, DOS programs could, in principle, make any Nucleus or BIOS layer system calls using a PME.

iRMX interrupt handlers are not allowed to make iRMX system calls (see chapter 9), and neither can DOS interrupt handlers. This means that DOS TSRs that connect themselves to hardware interrupts to check for users typing special characters (hot keys, for example) cannot use the RTE mechanism.

There is a nonissue involved in using the RTE mechanism. The objects owned by an iRMX job are automatically deleted when the job terminates,

but the VM86 job does not terminate until the iRMX system shuts down. This would seem to imply that a DOS program could create iRMX objects that would continue to exist after the DOS program terminates. Actually, the VM86 job is able to tell when DOS programs terminate (there is always an interrupt involved), and the RTE mechanism provides a procedure that automatically deletes the appropriate iRMX objects any time any DOS program terminates.

12.5.3 Invoking RTE functions from DOS programs

Making an iRMX system call from DOS involves pushing parameters onto the stack, setting up the processor's `si` register to point to the last parameter, and issuing a software *int* instruction to call the RTE task. If the system call returns a value, it will be returned in the processor's `ax` register, and must be obtained from there. The code to do all this work can be written in assembly language or C if your compiler supports either in-line assembly language or the *int86()* function, as Microsoft, Borland, and others do. Certainly, the easiest way to invoke RTE functions from DOS programs, however, is to use the interface library for RTE calls provided with iRMX for Windows in the file `\rmx386\demo\rte\lib\rmxintfc.c`.[4]

Supplying the code in source form documents the calling convention being used and provides developers with code that can be adapted to different vendors' development tools if necessary. A batch file, `rmxintfc.cmd` is supplied that compiles the code three times to produce object libraries in the small, compact, and large models using the Microsoft C compiler. The object modules created by these compilations are also provided in the files called `dosrtes.lib`, `dosrtec.lib`, and `dosrtel.lib`. Its not really an issue here, but it is worth remembering that Microsoft and Borland DOS development tools define the various memory segmentation models differently from Intel's iRMX development tools, as mentioned in chapter 3.

The source code file is named `rmxintfc.c` in recognition of the analogy between the functions in this file and the interface procedures used to make iRMX system calls from iRMX programs, as discussed in chapters 6 and 10. The difference is that these interface procedures are two steps removed from the iRMX interface procedures. These procedures set up and invoke real-mode software interrupt instructions that are intercepted by

[4]Watch the pathname syntax in this chapter. A / is used as the path component separator in the iRMX context, and a \ is used as the separator in the DOS context. The same files can be accessed from either context for the files discussed in this chapter because they are normally installed on a DOS partition. DOS, however, uses / to introduce command line switches, while the iRMX CLI uses \ as an escape character. The syntax difference helps keep track of which operating system is accessing the files in this chapter.

the iRMX PVAM interrupt manager, which activates an iRMX task, which then calls the iRMX interface procedure to make the system call.

A header file in `\rmx386\demo\rte\lib\rmxintfc.h` is used to compile the interface libraries, but another header file in `\rmx386\demo\rte\inc\rmxintfc.h` should be included in C programs that call the library routines. The `...\inc\rmxintfc.h` version includes monocase, underscore format, and mixed case definitions for all the RTE functions, so the create mailbox system call, for example, could be invoked using *rqcreatemailbox()*, *rq_create_mailbox()*, or *RQCreateMailbox()*, whichever you prefer. You might recall from chapter 3 that system calls invoked from iRMX can be invoked using monocase names if you include `:include:rmxc.h`, or using underscores if you include `:include:rmx_c.h`. (There is currently no mixed-case header file for the iRMX side.)

Before a DOS program generates an interrupt to call an RTE function, it should first determine if iRMX for Windows is actually running. (Generating an *int B8* instruction with no interrupt handler installed is not a good idea.) Two functions can be called from DOS to determine whether the iRMX for Windows TSR has been loaded or not and, if the TSR is present, whether iRMX has been loaded or not. The two functions, *RMX_Interface_TSR_Present()* and *get_RMX_IF_dseg()*, are provided in DOS object module format in the file `\rmx386\demo\rte\lib\rmxutils.obj`. Installing the TSR and loading iRMX are normally done together by submitting the batch file `\dosrmx\rmx.bat`, so a single function that makes calls to both these utilities is normally used to determine if RTE calls can be made or not. Such a function, named *RQEGetRMXStatus()*, is defined in `rmxintfc.c`.

Figure 12.2 illustrates the use of *RQEGetRMXStatus()* to determine whether iRMX for Windows is available followed by two RTE calls to obtain tokens for the caller's task and job. The program uses the code in `\rmx386\demo\rte\lib\rmxintfc.c` as an interface library for the system calls. This code is compiled using a DOS C compiler, and linked with `rmxutils.obj` and `dosrte?.lib` (`? = [s|c|l]`) to produce an executable DOS program, with a name such as `rtesamp.exe`. When it runs, the program displays the tokens for the DOS task and the VM86 job because that is the context in which DOS programs run. The C program in Figure 12.2 can be run as a Windows application rather than a DOS application by using the Borland development tools, which provide a version of *printf()* that writes to a window instead of the DOS screen; the output is the same:

```
This is iRMX task 10D0.
I belong to iRMX job 1040.
```

Figure 12.2 DOS program to invoke iRMX system calls through the Real-Time Executive (RTE) mechanism.

```
/***>  rtesamp.c  <*********************************************
 *
 *      This is a DOS program that makes iRMX system calls using
 *      the iRMX for Windows RTE mechanism.
 *
 ***************************************************************/
#include <stdio.h>
#include "\rmx386\demo\rte\inc\rmxintfc.h"

#define     E_OK  0

WORD    Status;

int
main (int argc, char *argv[]) {
WORD    task, job;


    if (RQEGetRmxStatus() != E_OK) {
      printf ("This program will not run without iRMX for Windows\n");
      exit (1);
      }


    task = rqgettasktokens (0, &Status);
    if (Status != E_OK) {
      printf ("rqgettasktokens() returned %X\n", Status);
      return 1;
      }

    job = rqgettasktokens (1, &Status);
    if (Status != E_OK) {
      printf ("rqgettasktokens() returned %X\n", Status);
      return 1;
      }

    printf
    ("This is iRMX task %X.\n", task);
     printf
    ("I belong to iRMX job %X.\n", job);

    return 0;


    }
```

Figures 12.3 and 12.4 illustrate two other ways to implement RTE calls from DOS programs with the same output as Figure 12.2. Figure 12.3 is another C program but uses in-line assembly language to set up and invoke the RTE functions. Like Figure 12.2, this version of the program can also be run as a Windows application, as long as the run-time library version of *printf()* generates output to a Window instead of the DOS screen. Figure

Figure 12.3 DOS program equivalent to Fig. 12.3, but which uses in-line assembly language to make its RTE calls.

```
/***>  dorte.c  <************************************************
 *
 *      This is a DOS program that makes iRMX system calls using
 *      the iRMX for Windows RTE mechanism.
 *
 *      It uses in-line assembly code to set up the stack and to
 *      invoke the RTE interrupt level, 0xB8.
 *
 ************************************************************/
#include <stdio.h>
#include "\rmx386\demo\rte\inc\rmxintfc.h"

#define     E_OK  0

WORD    Status;

#pragma argsused
int
main (int argc, char *argv[]) {
WORD    task, job;

    if (RQEGetRmxStatus() != E_OK) {
      printf ("This program will not run without iRMX for Windows\n");
      exit (1);
      }

    asm {
        mov     ax,0                /* get token for task */
        push    ax
        mov     ax, SEG Status
        mov     es, ax
        mov     ax, OFFSET Status
        push    es                  /* Far pointer to Status     */
        push    ax
        mov     si,sp
        mov     ax,26     /* function code for rqgettasktokens() */
        int     0xB8                /* invoke the DOS RTE        */
        add     sp, 6
        mov     task, ax
        mov     ax, Status
        cmp     ax, 0
        jne     error

        mov     ax,1                /* get token for job */
        push    ax
        mov     ax, SEG Status
        mov     es, ax
        mov     ax, OFFSET Status
        push    es                  /* Far pointer to Status     */
        push    ax
        mov     si,sp
        mov     ax,26     /* function code for rqgettasktokens() */
        int     0xB8                /* invoke the DOS RTE        */
        add     sp, 6
        mov     job, ax
        mov     ax, Status
```

Figure 12.3 (*Continued*)

```
        cmp     ax, 0
        jne     error
        }

    printf
    ("This is iRMX task %X.\nI belong to iRMX job %X.\n", task, job);
    return 0;

error:
    return 1;
}
```

Figure 12.4 Assembly language version of Fig. 12.3 (Assembled with Borland TASM 3.0).

```
;--->  rteasm.asm  <--------------------------------------------
;
;       This is a DOS program that makes iRMX system calls using
;       the iRMX for Windows RTE mechanism.
;
;       It uses assembly code to set up the stack and to
;       invoke the RTE interrupt level, 0xB8.
;
;-----------------------------------------------------------------

NAME    rteasm
PAGE    58, 132
.MODEL  SMALL
.STACK  100h
.DATA

TRUE    EQU     0FFh
FALSE   EQU     000h

Status  DW      ?
Task    DW      ?
Job     DW      ?

emess   DB      'This program will not run without iRMX for Windows'
        DB      0Dh, 0Ah, '$'

tmess   DB      'This is iRMX task '
tmessx  DB      'XXXX', 0Dh, 0Ah
jmess   DB      'I belong to iRMX job '
jmessx  DB      'XXXX', 0Dh, 0Ah, '$'
smess   DB      'RQ_Get_Task_Tokens failed.  Status is '
smessx  DB      'XXXX', 0Dh, 0Ah, '$'
hextab  DB      '0123456789ABCDEF'

.CODE

        EXTRN   _RMX_Interface_TSR_Present: PROC
        EXTRN   _get_RMX_IF_dseg: PROC
        PUBLIC  _MAIN

;----    TOHEX   ---------------------------------------------------
```

Figure 12.4 (*Continued*)

```
TOHEX   PROC    ; Convert value in ax to 4 hex chars at ds:di
        push    ax

        shr     ax, 12              ; First char
        mov     si, OFFSET hextab
        add     si, ax
        mov     ax, [si]
        mov     [di], al
        inc     di

        pop     ax                  ; Second char
        push    ax
        shr     ax, 8
        and     ax, 0Fh
        mov     si, OFFSET hextab
        add     si, ax
        mov     ax, [si]
        mov     [di], al
        inc     di

        pop     ax                  ; Third char
        push    ax
        shr     ax, 4
        and     ax, 0Fh
        mov     si, OFFSET hextab
        add     si, ax
        mov     ax, [si]
        mov     [di], al
        inc     di

        pop     ax                  ; Last char
        and     ax, 0Fh
        mov     si, OFFSET hextab
        add     si, ax
        mov     ax, [si]
        mov     [di], al

        ret

TOHEX   ENDP

;----   MAIN    ------------------------------------------------

_MAIN   PROC

        mov     ax, @data           ; Set up ds:
        mov     ds, ax

        call    _RMX_Interface_TSR_Present  ; Check for iRMX
        cmp     ax, TRUE
        jne     not_ok
        call    _get_RMX_IF_dseg
        cmp     ax, 0
        jne     ok

not_ok:
        mov     ax, 0900h
        mov     dx, OFFSET emess
```

Figure 12.4 (*Continued*)

```
        int     21h
        mov     ax, 4C00h
        int     21h

;    Get tokens for this task and this job

ok:
        mov     ax,0                ; get token for task
        push    ax
        mov     ax, SEG Status
        mov     es, ax
        mov     ax, OFFSET Status
        push    es                  ; Far pointer to Status
        push    ax
        mov     si,sp
        mov     ax,26    ; function code for rqgettasktokens()
        int     0B8h                ; invoke the DOS RTE
        add     sp, 6
        mov     task, ax
        mov     ax, Status
        cmp     ax, 0
        jne     error

        mov     ax,1                ; get token for job
        push    ax
        mov     ax, SEG Status
        mov     es, ax
        mov     ax, OFFSET Status
        push    es                  ; Far pointer to Status
        push    ax
        mov     si,sp
        mov     ax,26    ; function code for rqgettasktokens()
        int     0B8h                ; invoke the DOS RTE
        add     sp, 6
        mov     job, ax
        mov     ax, Status
        cmp     ax, 0
        jne     error

;   Got tokens ok.  Display them.

        mov     ax, task
        mov     di, OFFSET tmessx
        call    tohex

        mov     ax, job
        mov     di, OFFSET jmessx
        call    tohex

        mov     ax, 0900h
        mov     dx, OFFSET tmess
        int     21h

        mov     ax, 4C00h
        int     21h

;   Error: display status and exit.
```

Figure 12.4 (*Continued*)

```
error:
        mov     di, OFFSET smessx
        call    tohex
        mov     ax, 0900h
        mov     dx, OFFSET smess
        int     21h

        mov     ax, 4C00h
        int     21h

_MAIN   ENDP

        END     _MAIN
```

12.4 is a straight assembly language version of the program that generates exactly the same output as Figures 12.2 and 12.3. This program, however, uses a DOS system call to write its output, and can be run only as a DOS application. Running it under Windows generates similar behavior to the code in Figure 12.1; the program writes to the screen, but the output cannot be read because ASCII text is written when the screen is in graphics mode.

Although it would make sense to look now at the Protected-Mode Extension mechanism that applications can use to develop functions that go beyond the RTE mechanism, that topic is deferred until section 12.8. First, memory management issues are examined, which provide a good background for understanding the PME.

12.6 Memory Management

Two memory management issues are involved in iRMX for Windows. One is to provide DOS, or Windows, programs with access to memory managed by the iRMX operating system, and the second is to have iRMX for Windows coexist with various existing memory management systems that exist for DOS and Windows.

12.6.1 Accessing iRMX memory from DOS

A DOS program can create, delete, and determine the size of iRMX memory segment objects of any size (up to 4 GB) by using the RTE to make the iRMX system calls *rqcreatesegment( )*, *rqdeletesegment( )*, and *rqgetsize( )*. But the token returned by *rqcreatesegment( )* cannot be used by a DOS (real-mode) program to access iRMX (protected-mode) memory directly. Instead, the RTE provides two functions, *rqereadsegment( )* and *rqewritesegment( )*, that copy information between DOS real-mode segments and iRMX protected-mode segments. Of course, only a maximum of 64 KB can be transferred at a time because of the architectural limitation that real-mode addressing imposes on DOS programs.

Because *rqereadsegment()* and *rqewritesegment()* are called from DOS programs, the proper function prototypes for them can depend on the C compiler being used for the DOS application. Like the RTE-supported iRMX system calls, the source code for interface procedures that call these routines is given in `\rmx386\demo\rte\lib\rmxintfc.c`, and the header file with function prototypes suitable for use with Microsoft C and compatible DOS compilers is given in `\rmx386\demo\rte\inc\rmxintfc.h`.

12.6.2 Coexisting with other memory managers

DOS memory management started out fairly simple. The 8086 architecture allowed up to 1 MB of memory. It seemed that 640 KB, ten times the amount of memory available with the previous generation of microprocessors (the 8080 and its competitors), would be adequate to reserve for DOS and the programs it runs. This memory area is often referred to as *program memory* or *conventional memory*. The remaining 360 KB of the 1 MB address space, called the *upper memory area*, is reserved for ROM-resident code and RAM used by various device controllers. Some of the ROM-resident code, the ROM-BIOS, is supplied with the computer, and some of it is supplied by the device controllers. Initially, both the program area below 640K (conventional memory) and the area above 640K (upper memory) were lightly used.

Over time, programs and device controllers increased their demands on the respective regions of the real-mode address space accessible to them. It might seem that programs won the race to exhaust the amount of memory available to them, but this is somewhat illusory. Device controllers for video displays, for example, can require extremely large amounts of memory. A 640 × 480 pixel display with 16 colors per pixel (the resolution of a standard VGA adapter) requires more than 150,000 bytes of storage for a screenful of graphical information. Expanding this to 256 colors per pixel doubles the number of bytes needed; increasing the resolution to 768 × 512 pixels exceeds the 360K of memory space available to all device controllers. Seeing the inevitability of too little memory available to increase graphical capabilities on a regular basis, graphics device controllers make subsets of the entire display memory accessible to the microprocessor at different times. As Figure 12.5 illustrates, only a portion of the actual display memory managed by a display adapter is mapped to the memory accessible to the microprocessor at a time.[5] Other device controllers, such as disk and network adapters, also make demands on the upper memory area above 640K for Direct Memory Access (DMA) buffers.

---

[5]The term *frame buffer* is normally used in computer graphics to refer to what is called display memory here.

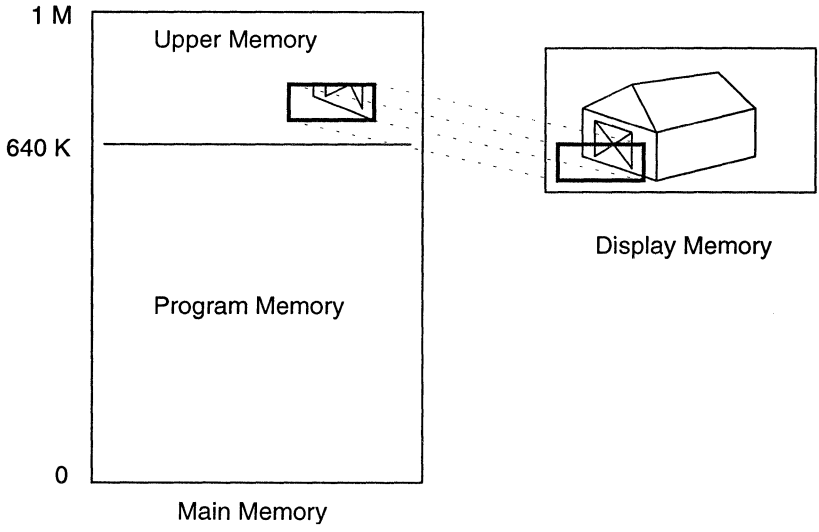


Figure 12.5 The microprocessor can access only part of a display adapter's video memory at a time.

Two basic techniques exist for making additional memory available to DOS programs. One technique, called *expanded memory*, can be used with processors that can access no more than 1 MB of RAM (the 8086 architecture). The other technique, extended memory, can be used with any processor that can access more than 1 MB of RAM. (The 80286 or i386 architectures provide access to 16 MB and 32 GB, respectively.)

Expanded memory. Expanded memory works very much like the video memory technique just discussed. A special memory board is added to the system that can map portions of its memory into part of the upper memory area above 640K. To use an expanded memory board, the user must load a software module called an *expanded memory manager* and then run application programs that make requests for access to various parts of expanded memory by software interrupts. The expanded memory manager responds to the software interrupts by setting registers in the expanded memory board, mapping the desired part of expanded memory into an area of memory above 640K where the application can access it. The RAM cannot be accessed on an expanded memory board except by setting the registers the way the expanded memory manager does.[6]

True expanded memory using an expanded memory board can be used to make more memory available to any processor, including those that oper-

[6]Most expanded memory boards can be configured by changing hardware switches or jumpers to provide extended memory instead of expanded memory. The statement is correct for expanded memory boards configured for expanded memory use.

ate only in real mode, such as the 8086. It is also possible to emulate expanded memory on a processor that can access more than 1 MB of RAM. To perform the emulation, a software module intercepts the same software interrupts as a true expanded memory manager, but, instead of setting hardware registers to map memory from an expanded memory board into the upper memory area, the emulator puts the processor into protected mode. In this mode, the processor can address RAM above 1M, and copies the emulated expanded memory pages between RAM in the upper memory area and RAM above 1M, and then restores the processor to real mode before returning to the application. The application can then access the emulated expanded memory by addressing locations in the upper memory area in the usual way. Naturally, switching the processor in and out of protected mode and copying memory blocks back and forth is slower than using an actual expanded memory board.

Another way to emulate expanded memory is to use the paging mechanism available on i386 processors and later. This technique requires the processor to be in protected mode at all times and for DOS to run as a VM86 task. In this situation, the expanded memory emulator responds to expanded memory software interrupts by setting the processor's page tables to map parts of memory above 1M to the upper memory area.

The software interface between application programs that want to use expanded memory and an expanded memory manager is given in the LIM 4.0 expanded memory specification (Lotus *et al.*, 1987).

Extended memory. Extended memory (XMS) is a generic term for memory above 1M that can be accessed by processors operating in protected mode. Thus, the expanded memory emulators just mentioned would be said to copy or translate between extended memory pages and the upper memory area. DOS programs that want to access extended memory do so by executing an *int 2F* instruction to determine if an XMS manager is available and the address of a routine to call to perform XMS functions. The basic XMS functions are to allocate, free, and copy blocks of extended memory. Extended memory blocks are allocated in 1 KB increments of up to 16 MB on 80286 processors, or up to 64 MB on i386 processors and later. Copying can be done between extended memory blocks or between an extended memory block and a program memory (below 640K) block.

Extended memory managers provide three other types of service in addition to management of extended memory blocks. These are High Memory Area (HMA) support, A20 management, and UMB support. Recall that real-mode addressing on an 80286 or greater processor can generate addresses as large as 0x10FFEF (0xFFFF0 + 0xFFFF). The memory above 1 MB (addresses 0x100000 through 0x10FFEF) is available for use by real-mode programs for both code and data. (Extended memory cannot be used for code for a DOS program) DOS 5.0, for example, can be set to load much of the operating system itself into the HMA. An extended mem-

ory manager can manage access to the HMA through functions that allow programs to request and release use of the HMA area. Only one program can occupy any part of the HMA at a time.

A20 management is closely related to HMA support. The address lines connecting the processor to memory are numbered from right to left starting at A00. Looking at the binary representation of memory addresses below 1 MB compared to those in the HMA, you can see that low addresses have address line A20 equal to 0 and addresses in the HMA have address line A20 equal to 1. Since 8086 processors do not have an A20 line (only lines A00 through A19), any attempt to access the HMA on that processor results in *address wraparound.* That is, accesses to the HMA result in accesses to real memory locations 0x00000 through 0x0FFEF.

Because there are DOS programs that rely on this wraparound, most computers that use an 80286 microprocessor or later provide a mechanism for optionally disabling the A20 address line under software control to simulate the 8086 processor's behavior. Extended memory managers provide functions that can be used to enable or disable the A20 address line using whatever technique is appropriate for a particular type of computer. For example, the extended memory manager provided with DOS 5.0 can be passed a parameter to tell it which of 14 different types of computer it must deal with when manipulating A20. The general rule is to have only the program using the HMA, normally DOS itself, make calls to the extended memory manager to change the state of the A20 line if necessary.

Finally, extended memory managers provide functions to manage access to Upper Memory Blocks (UMBs). UMBs are parts of memory in the 640K to 1M address range for which the computer physically provides RAM and which is not used by the ROM-BIOS or any device controllers. Programs can reserve and release portions of this RAM through two calls to the extended memory manager. UMBs are always a multiple of 16 bytes in size, and can be as large as one real-mode segment, 64 KB.

The Application Program Interface (API) for invoking extended memory manager functions is called the XMS Specification, (Microsoft, 1988). The API is summarized in Table 12.2. All functions are invoked from a DOS program by loading the indicated function code into register `ax` and executing an *int 2f* instruction.

12.6.3 DOS expanded memory and extended memory managers

DOS 5.0 provides an expanded memory emulator for processors numbered 80386 and later, called *emm386.exe*, and an extended memory manager for 80286 processors and later, called *himem.sys*. Both programs are loaded, like device drivers, when the user's `config.sys` file is processed. *emm386.exe* uses *himem.sys* to provide access to the extended memory it needs to emulate expanded memory, so *himem.sys* must be loaded before

TABLE 12.2 Summary of the Version 2.0 Extended Memory Specification (XMS) for DOS.

Code	Function
0x43	Determine if an XMS driver is installed
0x00	Get XMS version number and internal revision number
0x01	Request High Memory Area (HMA)
0x02	Release HMA
0x03	Global Enable A20 (Used only by the program in the HMA.)
0x04	Global Disable A20
0x05	Local Enable A20 (Might be used by conventional DOS programs.)
0x06	Local Disable A20
0x07	Query A20
0x08	Query Free Extended Memory
0x09	Allocate Extended Memory Block (EMB)
0x0A	Free EMB
0x0B	Move EMB
0x0C	Lock EMB (Prevent move operations.)
0x0D	Unlock EMB
0x0E	Get EMB Information (Locking count, block size)
0x0F	Reallocate EMB (Change the size of a block.)
0x10	Request Upper Memory Block (UMB)
0x11	Release UMB

emm386.exe (that is, it must be listed before *emm386.exe* in `config.sys`). *emm386.exe* also manages the upper memory area for systems that want to load device drivers and interrupt handlers into that part of memory.

12.6.4 The DOS Protected Mode Interface (DPMI)

What does all this information have to do with iRMX for Windows? Before answering that question, you still have to look at one more set of functions that affect DOS memory management, the *DOS Protected Mode Interface Specification (DPMI, 1991).*

The basic rationale of the DPMI Specification is to provide an orderly manner in which DOS programs can take advantage of the protected-mode features of the 80286 and later processors. A program, called the *DPMI host* or *DPMI server*, responds to requests for protected-mode operations from real-mode programs called DPMI clients. DPMI clients are DOS programs that can enter protected mode, switch between real and protected mode, and either terminate normally or terminate and stay resident in protected mode to provide services to other protected-mode DPMI clients. While in protected mode, DPMI clients can make a large number of requests of the DPMI host through *int 31* function calls. These functions fall into the following categories:

- Local descriptor table (LDT) management.
- Extended memory management.

- DOS memory management.
- Interrupt management.
- Page management (i386 and above only).
- Translation.
- Debug support.
- Miscellaneous.

Although DPMI hosts provide extended memory management services, the DPMI functions are different from the XMS functions listed in Table 12.2. For example, rather than receive a handle for a new extended memory block, a DPMI client receives the actual linear address of a block that it allocates. The DPMI client must then allocate and initialize Local Descriptor Table (LDT) descriptors to provide addressability to the block. Because DPMI clients normally operate in protected mode, the client can access the extended memory area directly, rather than be forced to use XMS functions to copy information between extended and conventional memory.

Noting the support for interrupt and memory management listed previously, it would seem natural that iRMX for Windows should act as a DPMI server, and that DOS applications that want to access protected-mode services using the DPMI API instead of the iRMX RTE mechanisms should be able to do so. Such is not the case, however, and there seem to be two reasons for this.

One reason is that the DPMI interface is large (81 functions) and would require considerable overhead to implement and ensure that DPMI clients cannot interfere with the real-time requirements of iRMX. The second reason is the crucial one, though. A key DOS application for iRMX for Windows is Microsoft Windows itself, and Windows versions 3.0 and 3.1 do not operate as DPMI clients. Indeed, both versions of Windows provide their own DPMI servers, albeit servers that adhere to an earlier DPMI specification (version 0.9) rather than the current one (version 1.0).

The Windows DPMI server is available only when Windows is running in enhanced mode (available only for i386 and later processors), and not when it is running in standard mode (available for 80286 processors and later). Simply stated, iRMX for Windows cannot operate with Windows running in enhanced mode because to do so, it would have to operate as a DPMI client, which would preclude its operation as a real-time operating system. There is nothing inherent in the design of Windows, by the way, that would prevent it from running as a DPMI client in enhanced mode. If such a version of Windows were to be released, it would undoubtedly be supported by iRMX for Windows. As it stands, iRMX for Windows can run with Windows operating only in its standard mode.

12.6.5 Memory management summary

We can now summarize the characteristics of the iRMX for Windows memory management system. Each item in this summary is based on the fact that iRMX for Windows takes over and uses all of the processor's extended memory.

- Programs that emulate expanded memory, such as *emm386.exe*, cannot be used with iRMX for Windows because they manage extended memory to do their emulation.
- DOS cannot use UMBs for device drivers and TSRs when iRMX for Windows is running because UMBs are managed by *emm386.exe*, and *emm386.exe* is not compatible with iRMX for Windows.
- There is no problem using a true expanded memory board and its expanded memory manager with iRMX for Windows because they do not involve extended memory.
- iRMX for Windows completely honors the state of the HMA as it existed before iRMX was loaded. Thus, *himem.sys* can load DOS into the HMA, for example, and iRMX will preserve the state of the HMA.
- If the HMA is not in use when iRMX is loaded, the iRMX extended memory manager for DOS programs, *himem.job*, can be used to allocate the HMA to any program that wants it. Unlike *himem.sys*, *himem.job* has no mechanism for placing a limit on the minimum amount of memory in the HMA that a program can request.
- Any extended memory blocks allocated by an extended memory manager before iRMX is loaded will become inaccessible; *himem.sys* must put the DOS task into protected mode from VM86 mode, which iRMX does not allow. If *himem.job* is run, it cannot honor requests for access to already-allocated extended memory blocks because of the following:
  1. *himem.job* has no way to associate the original extended memory manager's handle values with actual extended memory regions.
  2. The parts of extended memory allocated by the original extended memory manager might have been overwritten by iRMX itself when it was loaded.
- DPMI hosts, such as Windows running in enhanced mode, cannot be run with iRMX for Windows because the DPMI extended memory management functions conflict with iRMX's management of extended memory.
- DPMI clients cannot be run with iRMX because iRMX does not provide a DPMI host. The iRMX RTE does, however, provide real-mode programs with access to extended memory, and iRMX's VM86 protected mode extension mechanism allows users to build other DPMI-like functions as well.

12.7 PME: VM86 Protected Mode Extensions

A good way to understand the iRMX VM86 Protected Mode Extensions (PME) mechanism is to compare the use of the PME with the operation of a DPMI client. The idea in both cases is for a real-mode DOS program to make use of the processor's protected-mode features. For DPMI, this access is provided by a DPMI host. For iRMX, it is provided by iRMX for Windows itself. In both cases, the real-mode DOS program is actually running with the processor in VM86 mode. In this mode, every software *int* instruction (among others) passes control to its proper interrupt handler using one of the mechanisms described in the earlier discussion of the RTE. If the processor is running in VM86 mode with interrupt virtualization enabled, the transfer is managed by the VM86 dispatcher, which is part of the VM86 job. For both the iRMX VM86 dispatcher and a DPMI host, there are three basic ways to handle the interrupt:

1. Simulate the behavior that the real-mode operating system would perform in response to the interrupt.
2. Reflect the interrupt back to the real-mode operating system's handler for the interrupt.
3. Invoke a protected-mode routine to perform computations not normally be available to real-mode programs.

Many interrupts that occur in the context of DOS programs evoke a response that is a combination of items (1) and (2). For example, every time a DOS program running with iRMX for Windows invokes one of the DOS terminate functions (*int 21* with function code `0x4C00`, for example) the VM86 dispatcher notifies all iRMX programs that need to be informed whenever DOS programs terminate, then causes the corresponding real-mode DOS interrupt handler to be called for normal DOS termination processing.

Item (3) is reminiscent of the iRMX RTE mechanism discussed earlier. In fact, the RTE mechanism is itself implemented using the PME mechanism. For DPMI systems, the mechanism is known as a *real-mode callback*. The idea is that a protected-mode program connects one of its procedures to a real-mode interrupt level. When a real-mode program (actually, a VM86-mode program) invokes the appropriate real-mode software interrupt, the VM86 dispatcher receives control, recognizes the interrupt level, and calls the appropriate protected-mode procedure. When interrupt virtualization is not enabled, or when the DOS task is running in protected mode (under Windows), the DOS task can call the protected-mode interrupt handler directly through the IDT without using the VM86 dispatcher.

A sample PLM program to illustrate the PME mechanism is given in Figure 12.6. The program first displays some information about its iRMX environment using the same code as the I/O job program in Figure 7.3, creates a task to terminate the program in an orderly fashion when the user types <^C>, creates a PME by calling *rqesetvmextension( )*, and then goes into an endless loop in which it receives messages from the PME procedure and displays them on the console.

Figure 12.6 PLM program that installs a Protected Mode Extension (PME).

```
/***>  PME.PLM  <*****************************************************
 *
 *     This is an HI command that demonstrates the Protected Mode
 *     Extension mechanism.  The main task sets up the PME, then
 *     displays messages that the PME sends to it.  The PME is invoked
 *     by a DOS int C0 instruction.
 *
 ***********************************************************************/
$compact (exports pmeproc, cntrlCtask)
$title ('Sample Program to Create a Protected Mode Extension.')

pme: DO;
$include (pme.ext)

/*  Global Variables
 *  ----------------
 */
DECLARE
        CR          LITERALLY       '0Dh',
        LF          LITERALLY       '0Ah',

mess (*)     BYTE INITIAL (0,'This is the initial task: xxxx.', CR, LF,
                            '  I belong to job xxxx.', CR, LF,
                            '  My priority is xxxx.', CR, LF,
                            '  My maximum priority is xxxx.', CR, LF,
                            '  Now I will create a PME', CR, LF),
pmemess (*) BYTE INITIAL (0, CR, LF, 'PME job: xxxx.', CR, LF,
                            '  Task: xxxx.', CR, LF,
                            '  Priority: xxxx.', CR, LF,
                            '  Max priority: xxxx.', CR, LF, LF),

        buffer (128)                BYTE,
        hextab (*) BYTE INITIAL ('0123456789ABCDEF'),

        (myjob, mytoken,
        cntrlCtaskTkn, PMEmbx)      TOKEN,
        myprio                      BYTE,
        maxprio                     BYTE,
        actual                      WORD_16,
        Status                      WORD_16;

/*  Procedure to Convert a Hexadecimal Value to ASCII Characters
```

Figure 12.6 (*Continued*)

```
    ------------------------------------------------------------
*/
word2hex: PROCEDURE (value, where);
DECLARE
        value       WORD_16,
        i           INTEGER,
        where       POINTER,
        xxxx BASED where (1) BYTE;

  DO i = 3 TO 0 BY -1;
    xxxx(i) = hextab(value AND 0Fh);
    value = shr (value, 4);

  END;
END word2hex;

/*      Procedure to be executed by the Control-C Task
 *      ----------------------------------------------
 */
cntrlCtask: PROCEDURE PUBLIC;
DECLARE
        cntrlCsem                       TOKEN,
        Status                          WORD_16;

    cntrlCsem = rqcreatesemaphore (0, 1, 0, @Status);
    CALL rqcsetcontrolc (cntrlCsem, @Status);

/*  Delete the PME when the user terminates the program
 *  ---------------------------------------------------
 */
      actual = rqreceiveunits (cntrlCsem, 1, 0FFFFh, @Status);
      CALL rqesetvm86extension (0C0h, NIL, NIL, @Status);
      CALL rqcsendcoresponse (  NIL, 0,
                                @(13,'PME removed', CR, LF),
                                @Status);
      CALL rqexitiojob (0, NIL, @Status);

END cntrlCtask;

/*      Procedure to be executed as the Protected Mode Extension
 *      --------------------------------------------------------
 */
pmeproc: PROCEDURE (DOSState, Flags) BYTE PUBLIC;
DECLARE
        DOSState                        POINTER,
        Flags                           DWORD,

        (mytasktoken, myjobtok)         TOKEN,
        mypriority                      BYTE,
        maxpriority                     BYTE,
        Status                          WORD_16;

/*      Format a Message to send to the HI Command job
 *      ----------------------------------------------
 */
    pmemess(0) = length (pmemess) -1;
    mytasktoken = rqgettasktokens (0, @Status);
```

Figure 12.6 (*Continued*)

```
    myjobtok = rqgettasktokens (1, @Status);
    mypriority = rqgetpriority (selector$of(NIL), @Status);
    CALL rqsetpriority (selector$of(NIL), 0, @Status);
    maxpriority = rqgetpriority (selector$of(NIL), @Status);
    CALL rqsetpriority (selector$of(NIL), mypriority, @Status);
    CALL word2hex (WORD (myjobtok), @pmemess(12));
    CALL word2hex (WORD (mytasktoken), @pmemess(27));
    CALL word2hex (mypriority, @pmemess(46));
    CALL word2hex (maxpriority, @pmemess(69));
/*      Send it
 *      -------
 */
    CALL rqsenddata (PMEmbx, @pmemess, length (pmemess), @Status);

    RETURN 0FFh;  /*  Notify DOS that processing is complete  */

END pmeproc;

/*
 *  Initial Task Starts Here
 *  ------------------------------------
 */
    mess(0) = length (mess) -1;

/*  Format Initial Message and Display It
 *  -------------------------------------
 */
    mytoken = rqgettasktokens (0, @Status);
    myjob = rqgettasktokens (1, @Status);
    myprio = rqgetpriority (selector$of(NIL), @Status);
    CALL rqsetpriority (selector$of(NIL), 0, @Status);
    maxprio = rqgetpriority (selector$of(NIL), @Status);
    CALL word2hex (WORD (mytoken), @mess(27));
    CALL word2hex (WORD (myjob), @mess(52));
    CALL word2hex (myprio, @mess(76));
    CALL word2hex (maxprio, @mess(108));
    CALL rqcsendcoresponse (NIL, 0, @mess, @Status);

/*  Create a Task That Will Delete the PME When This Job Terminates
 *  ---------------------------------------------------------------
 */
    cntrlCtaskTkn = rqcreatetask (0, @cntrlCtask, selectorof(@Status),
                    NIL, 8192, 0, @Status);

/*  Create the Protected Mode Extension and Display Messages From It
 *  ----------------------------------------------------------------
 */
  PMEmbx = rqcreatemailbox (20h, @Status);  /*  Data Mailbox  */
  CALL rqesetvm86extension (0C0h, @pmeproc, NIL, @Status);

  DO WHILE 1;
    actual = rqreceivedata (PMEmbx, @buffer, 0FFFFh, @Status);
    CALL rqcsendcoresponse (NIL, 0, @buffer, @Status);
  END;

END pme;
```

When the program starts running, it displays some information about its iRMX job and initial task that might look like this:

```
This is the initial task: BAD8.
I belong to job B9A0.
My priority is 008E.
My maximum priority is 008D.
Now I will create a PME
```

It then sets up a task that runs when the iRMX user types <^C>, sets up a procedure to be called as a PME, and goes into an endless loop waiting for messages to arrive from the PME procedure. Each time a DOS program issues an *int 0xC0* instruction, the iRMX VM86 Dispatcher calls the PME procedure, which formats a message with information about itself, sends it to the mailbox that is being monitored by the job's initial task. A typical message would look like:

```
PME job: 1040.
Task: 10D0.
Priority: 00FE.
Max priority: 0000.
```

When the user types <^C> at the iRMX console, the control-C task wakes up, deletes the PME so that further DOS *int 0xC0* instructions will not be recognized by the VM86 Dispatcher, and exits the iRMX job.

As the sample output indicates, the PME procedure is executed in a totally different context from the initial task of the HI job. The PME procedure's job is the VM86 job that owns the DOS task, and the PME procedure's code is executed by (in the context of) that same DOS task. Its task priority is 254, the level assigned to the DOS task in order to give higher priority to real-time tasks running under iRMX.

Note that the PME procedure calls *rqsetpriority( )* with a value of 0 as its first parameter, which sets the task's priority to the maximum allowed for its job, which is 0 in the case of the VM86 job. If the sample program were coded to leave the DOS task's priority at 0, DOS (which constantly polls for I/O) would prevent any iRMX code from running.

Because the VM86 job is a first-level job, there is no console associated with it, so PMEs do not have access to console I/O functions. Most PMEs are installed to provide DOS programs with services that do not involve interaction with an iRMX user. In a more typical application, the job that sets up the PME is installed using *sysload*, and remains in effect for as long as the iRMX operating system is running. Such jobs would not do any iRMX console I/O and would not need a <^C> handler as the sample program does. Still, the sample code does illustrate an important point: the PME procedure is not being executed by a task that belongs to the iRMX job initiated by *sysload*, it is executed by a task that belongs to the job that owns the VM86 dispatcher, the DOS task. Figure 12.7 gives the code for a

DOS program that could be run to invoke the PME of Figure 12.6. It simply ensures that iRMX is running and issues an *int C0* to invoke the PME.

Figure 12.7 DOS program that invokes the Protected Mode Extension set up by the code in Fig. 12.6.

```
/***>  dopme.c  <************************************************
 *
 *      This is a DOS program that generates an interrupt 0xC0
 *      It will trigger an iRMX Protected Mode Extension that
 *      has been installed at that interrupt level.
 *
 ********************************************************************/
#include <stdio.h>
#include <dos.h>
#include "\rmx386\demo\rte\inc\rmxintfc.h"

#define     E_OK  0

int
main (int argc, char *argv[]) {
union   REGS    regs;
struct  SREGS   sregs;

  if (RQEGetRmxStatus() != E_OK) {
    printf ("This program will not run without iRMX for Windows\n");
    return 1;
    }

  for (;;) {
    printf ("Invoking the iRMX PME with int 0xC0\n");
    int86x (0xC0, &regs, &regs, &sregs);
    printf ("Done.  Again? ");
    if (getch() != 'y') break;
    }
  return 0;
  }
```

The RTE is a PME that invokes a specific set of iRMX system calls on behalf of DOS RTE clients. But DOS programs are not limited in their access to iRMX services by the design of the RTE. The PME mechanism can be used to build an iRMX server that provides access to any iRMX functionality an application might require.

12.8 DDE: Communication with Windows Applications

Windows provides three mechanisms programs can use to interact with each other: Dynamic Link Libraries (DLLs), Dynamic Data Exchange (DDE), and Object Linking and Embedding (OLE). DLLs are really a way for Windows applications to share object code; they do not actually provide a mechanism for programs to communicate or synchronize with one an-

other. DDE and OLE both allow Windows programs to share data and communicate with each other. Because iRMX for Windows does not presently provide support for OLE, the focus in this section is the DDE mechanism.

Any Windows application can be programmed to act as a DDE server or DDE client, and the Windows kernel acts as the switchboard for passing messages between DDE servers and clients. iRMX for Windows provides a program called the *DDE Router* that acts as a surrogate Windows application for iRMX programs that want to use the DDE mechanism, as either servers or as clients. The DDE Router, normally installed as `\windows\rmx\router.exe`, can be thought of as an extension to *rmxtsr*, described earlier, which acts a surrogate for iRMX programs that want to make DOS system calls.

An interaction between Windows applications that use the DDE begins when a client sends a message addressed to a particular {*<application>*, *<topic>*} tuple, where *<application>* and *<topic>* are strings that specify the pathnames of a particular Windows application and a particular document with which the application is to work. The Windows kernel broadcasts the message to all active Windows applications. When the kernel receives an acknowledgment from an application that recognizes the tuple, it opens a channel by which the client and server can communicate with each other using a set of DDE commands. An example of a DDE exchange between two Windows applications might be word processor macro which, when run, establishes a conversation with a spreadsheet program using a particular spreadsheet as the conversation topic. The macro might obtain the value of a particular cell in the spreadsheet to supply a value to use in the text of a report.

The DDE Router provides a significant extension to the Windows DDE mechanism by recognizing an extended form for the *<application>* string, which includes a machine-name component used to identify the network node on which the DDE server resides. Since the machine name and the application name are embedded in a single string, the mechanism is transparent to the Windows kernel. When the kernel broadcasts an extended-form *<application>* string, the DDE Router recognizes it by the presence of a <%> character in the string, which is used to separate the machine name from the application name.[7] The DDE Router then redirects the request to the appropriate network node, where a copy of DDE Router must be running to receive the request and pass it on to the proper DDE server on the remote system.

[7]This means that the DDE Router also intercepts attempts to initiate conversations with Windows DDE servers that have a <%> character in their names. If this is a problem, the separator character can be changed in the [DDERouter] section of the system's `win.ini` configuration file.

To operate, the DDE Router requires iRMX for Windows to be running with its networking software in place. You saw how iRMX used iNA 960 to supply ISO-compatible network support to iRMX applications in chapter 11, and you will see how this support is integrated with various DOS networking options in the next section. An example scenario for which the iRMX DDE mechanism is useful might be the use of a PC to perform real-time control over a manufacturing process while running a task that acts as DDE server. A remote user could monitor or even control that process using a custom Windows application working as a DDE client.

Figure 12.8 is a sample iRMX DDE client, *ddeinq*, that illustrates the use of the DDE mechanism. The program takes two command-line arguments, a node name for a computer running the DDE router and the name of an application program that can operate as a DDE server. The program first tries to establish a conversation using the node and application name supplied on the command line using the *client_dde_initiate()* library call. If this call succeeds, the program returns a 16-bit value called a *conversation ID*, which is used as the first argument to other DDE calls the way an iRMX token is used to identify a particular object in iRMX system calls.

Figure 12.8 An iRMX DDE client.

```
/***>  ddeinq.c  <**************************************************
 *
 *  Allows an iRMX user to determine topics available for a DDE server.
 *
 *  Command line:
 *
 *       ddeinq <node> <application>
 *
 *       The <node> must be the name of a computer running DDE Router
 *       with the win.ini file containing a [DDERouter] section with a
 *       line in the form pcname=<node>.
 *
 ********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <rmxdde.h>

static  WORD        conversation, Status;
static  char        topicsList[4096], *this;
static  CONFIGBUF   configBuf;


/*  Status check utility
 *  --------------------
 */
void
chkStat (char *message) {
  if (Status) {
    printf ("%s failed. Status is %X\n", message, Status);
    client_dde_terminate (conversation, &Status);
    exit (1);
```

Figure 12.8 (*Continued*)

```
    }
  return;
  }

/*  Main Program Starts Here
 *  ------------------------
 */
int
main (int argc, char *argv[]) {

  if (argc != 3) {
    printf
      ("Give machine node and application name on command line.\n");
    return 1;
    }

  dde_library_init (&configBuf, &Status);
  chkStat ("dde_library_init");

/*  Most DDE apps provide a "system" topic for housekeeping
 */
    conversation = client_dde_initiate (argv[1],
                                        argv[2],
                                        "system",
                                        (LINKFUNCPTR) 0,
                                        &Status);

    chkStat ("client_dde_intiate");


/*  Now ask for a list of topics
 */
  client_dde_request (conversation, "topics", topicsList, (WORD) sizeof
  (topicsList), &Status);
  chkStat ("client_dde_request");

  this = strtok (topicsList, "\t");
  printf ("Application %s on machine %s has the following topics:\n",
        argv[2], argv[1]);

  while (this) {
    printf ("  %s\n", this);
    this = strtok (NULL, "\t");
    }

  client_dde_terminate (conversation, &Status);
  chkStat ("client_dde_terminate");

  return 0;
  }
```

The value returned is not actually an iRMX token, however. It is a small integer used by the DDE library routines to index into an internal list of active conversation data structures. If the call fails, the program receives a nonzero exception code, which takes on one of the values given in Table 12.3.

TABLE 12.3 Exception Codes Returned by iRMX DDE Library Functions.

Value	Name	Meaning
0xE000	dde_busy	Remote program is busy.
0xD000	dde_denied	Remote program denied a request.
0xC800	dde_no_response	No response from the remote program.
0xC0XX		XX is an application-defined exception code returned by the remote application.
0x0000 through 0x3FFF		Standard iRMX exception codes.

The topic name used to establish the conversation is *system*. Many Windows applications able to operate as DDE servers recognize this special topic name, which is used to supply general information without accessing a particular document or worksheet. The sample program then makes a call to *client_dde_request()* with a data item name of *topics*. A more typical type of data item name might be the name of a cell or range of cells in a spreadsheet or a bookmark in a wordprocessing document, but *ddeinq* takes advantage of the fact that many DDE servers return a list of currently available topics (e.g., the currently open worksheets or currently open documents) when a client uses the topics data item name for a conversation based on the system topic. The list is returned as a <tab>-separated list terminated by a <nul>. The sample program uses the ANSI *strtok()* function to extract the individual topic names from the returned list, and displays them. Three types of DDE application programs might be developed to run under iRMX:

Simple client. A simple client application can request information from a server, as illustrated by *ddeinq*, tell the server to change the value of a data item using the *client_dde_poke()* function, or pass a set of commands for the server to execute using the *client_dde_execute()* function.

Client link. Two types of links can be established between a client and server. A *hot link* causes the server to send the value of a data item to the client any time the value changes, whereas a *warm link* causes the server simply to notify the client when a value changes The client can then obtain the value using *client_dde_request()* if it wants. The client is notified of the new value or of the change in value by a callback mechanism. When the client issues the call to *dde_library_initiate()* to establish a conversation with the server, it can include a pointer to a function that will be called by the library whenever a server updates a hot or warm link. (The *ddeinq* sample program coded the pointer as (LINKFUNCPTR) 0 because it does not use links.)

Server. An iRMX server supplies pointers to two functions when it calls *server_dde_register()* to inform the DDE library that the application is ready to act as a DDE server. The first pointer identifies a function known as the server's conversation callback, which is called by the DDE library any time a client attempts to initiate or terminate a new conversation with the server. The second pointer identifies the server's data callback, which is called by the DDE library whenever the client requests a data operation, including to request data, poke data, or establish a hot or warm link.

There is nothing to prevent an application from acting in a mixed fashion with respect to these three modes. Because of the multitasking nature of iRMX, there would be no problem, for example, in having an application performing client operations with one task while the DDE library asynchronously makes calls to conversation and data callback functions declared previously by a call to *server_dde_register()*.

The DDE functions supplied by the iRMX DDE library are not unique to iRMX. Rather, they are functions provided by the library that allow iRMX applications to access standard Windows DDE functions. The Windows DDE protocol is discussed in some detail in Petzold (1992).

12.9 Network Compatibility

Two basic approaches can be taken for integrating iRMX and DOS networking. The simpler but less powerful approach is simply to maintain an existing DOS network and let iRMX applications access network drives using the EDOS file driver from the iRMX side or the *rmxuse* command from the DOS side. For example, if your computer is running a Novell network that gives you access to a networked disk as your DOS drive G:, you can give the iRMX command:

```
iRMX> attachdevice g_dos as g edos                    [6]
```

After this command, iRMX can access the networked drive using the logical name `:G:` without regard to the fact that it is being accessed over the network. This approach is easy to use, but requires quite a bit of overhead to access the networked disk, and no provision exists for operating as anything but a network client or a dedicated network server.

The second approach is to provide the network access from the iRMX side and let DOS and Windows applications make use of the iRMX networking facilities. This approach is more efficient for iRMX applications and provides the added benefit of allowing the computer to operate concurrently as a network server and client. This section explores the situation when the network device controller is managed from the iRMX side.

Chapter 11 introduced the native iRMX networking environment: a software module called iNA 960 provides (in addition to name-server and

network-management functions) an ISO standard implementation of the Transport, Network, and Data Link layers, which can be interfaced to Ethernet (802.3) or Token Bus (802.4) network device controllers. iNA includes support for multiple protocol stacks at the Data Link layer, which means that the same device controller can be used to support concurrent operation with different networking protocols, such as TCP/IP and Novell, in addition to ISO protocols. Here, you will see how iRMX for Windows integrates the iRMX networking capabilities provided by iNA with DOS networking. In the process, you will also see how this integration provides for networked DDE communication, introduced in the previous section.

Figure 12.9 shows the structure of the software components that contribute to iRMX for Windows networking. In this figure, the box labeled iNA 960 would normally include all the iRMX-Net components shown in Figure 11.2. On an iRMX for Windows system, networking software is loaded by a *sysload* command, normally when the system initializes. Different versions of the network job are used, depending on what network device controller and system bus iNA is configured to operate with. For example, the networking file *netat.job* contains iRMX-Net for PC/AT bus systems. iRMX-Net loads iNA into the memory of the network device controller as it initializes.

Another networking file that might be used instead is *ntp4at.job*, which is a version of iNA that does not require any network device controller at

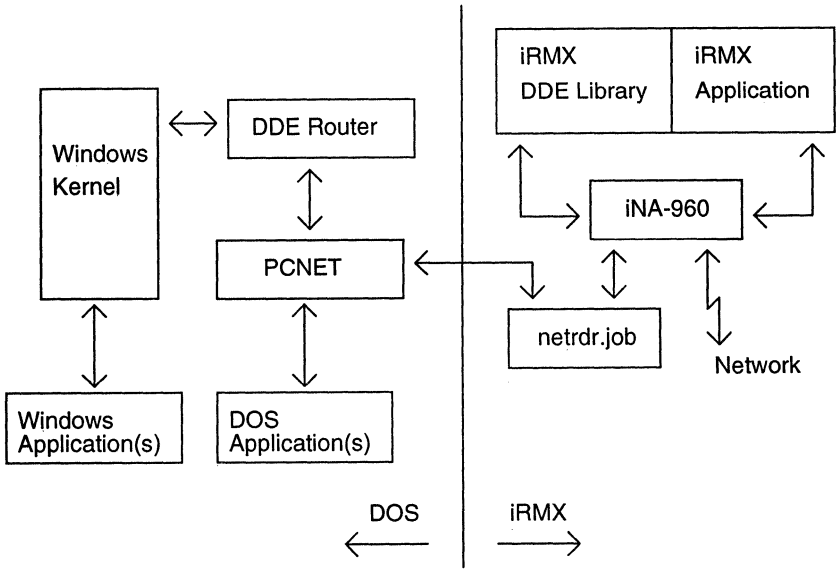


Figure 12.9 Relationships among DDE servers and clients using the iRMX DDE Router.

all. When this job is loaded, iNA provides the ISO Transport layer interface, TP4, coupled with a null Data Link layer. Because no network controller is involved, *ntp4at.job* does not include any iRMX-Net support, however. The significance of *ntp4at.job* is that the iRMX support for Windows' DDE mechanism is based on a networking model for its operation, and *ntp4at.job* allows the DDE mechanism to be used on a standalone computer system.

The use of *ntp4at.job* also allows programs to perform many of the iNA functions described in chapter 11 on a non-networked computer; it just does not support iRMX-Net and the Remote file driver. The *timesrv*, *gettime*, *namesrv*, and *mynamon* programs presented in chapter 11 all run successfully on iRMX for Windows systems running *ntp4at.job*.

As Figure 12.9 shows, an iRMX application linked to the iRMX DDE Library communicates with DOS through iNA, using the Request Block interface described in chapter 11. The RB requests are intercepted by *netrdr.job*, another job normally started by means of a *sysload* command when the system initializes. The *netrdr.job* includes a PME that communicates with a DOS TSR supplied with iRMX for Windows named *pcnet.exe*.[8] When an iRMX application makes a DDE library call, the library encapsulates the request as an iNA network request. iNA running on the remote system receives the message over the network, where it is received by that system's copy of *netrdr.job*. If the DDE request is addressed to the local computer, the local copy of *netrdr.job* receives the request directly from iNA. In either case, local or remote DDE request, *netrdr.job* transmits the request to *pcnet.exe* on the DOS side, which passes the request to the DDE Router, the Windows application that acts as a surrogate DDE client and server for iRMX DDE operations.

The reverse path is followed when a Windows application makes a DDE protocol request using a conversation based on an extended application name that includes a node name and <%> character, as introduced previously. The DDE router intercepts the request and uses *pcnet.exe* to send it to *netrdr.job*, which sends it over the network where it is received by the remote system's *netrdr.job* and passed up to the DDE Library code that interacts with the iRMX application.

Figure 12.9 also shows a direct connection between the iRMX application and iNA. This link represents the Request Block (RB) interface to iNA described in chapter 11. This same RB interface is used by *netrdr.job* for its interface to iNA, but the actual form it takes (datagrams or virtual circuits) is invisible to the DDE user.

---

[8]You must install the *pcnet.exe* TSR after loading *netrdr.job* and the iRMX iNA job. If you load iRMX from `autoexec.bat` and your iRMX `:config:loadinfo` loads the networking jobs, you can put the `pcnet` statement in the `autoexec.bat` file after the statements to load iRMX.

What remains to be examined is the connection between DOS applications and the network, as managed by iRMX and shown by the connection between *pcnet.exe* and DOS applications in Figure 12.9. One key to the success of this part lies in the fact that all DOS networking software, whether MS-Net, TCP/IP, or Novell, uses a common interface to the network, called NetBIOS, based on data structures called *Network Control Blocks* (NCBs) sent to a networking device driver using *int 5C* instructions. The *pcnet.exe* TSR intercepts these interrupts, transforms the NCBs into iNA RBs, and passes them on to *netrdr.job*, which passes them to iNA for the actual network operations.

The second key to the integration of non-ISO networks such as Novell and TCP/IP with iRMX networking is the availability of the RAWEDL interface to iNA described in section 11.11. By using RAWEDL RBs, *pcnet* and *netrdr* are able to send and receive Novell and TCP/IP packets through a single network device controller without interfering with each other or with concurrent ISO operations.

Although this section has presented the conceptual structure of the components involved in iRMX for Windows and DOS networking, many procedural details must be followed to make the entire process work. In particular, integration of Novell and TCP/IP networking on the DOS side requires the installation of MS-Net software, available from Intel for use with its PCL2(A) network device controller; Novell client software from Novell (and a Novell server on the network); the PC/TCP Ethernet device driver from FTP Software, Inc.; and a special Novell shell available from Brigham Young University. Details for integrating all of this software are given in the documentation supplied with MS-Net and in volume 8 of the iRMX for Windows documentation set, *iRMX Network Concepts.*

12.10 Run-Time Configuration

System configuration refers to the process of tailoring the software structure of iRMX to match the particular hardware present on a system and the particular functional needs of a user. Adding a device driver to support a particular device controller, or deleting unneeded software modules to conserve system memory requirements, are both examples of system configuration. Before iRMX for Windows was introduced, configuration of an iRMX system was always performed using the Interactive Configuration Utility (ICU) described in chapter 9.

The ICU is a menu-based editing program used to modify a special file called a *definition file* and then to generate a set of command files based on the definition file that, when submitted, would assemble and compile various tables of information, link everything together, and generate a new operating system image file that could be bootstrap loaded. The process of editing a definition file and generating a new image file might take only 15 to 20 minutes on a fast machine, and several image files can be kept on disk

so the desired features for the system can be selected by bootstrap loading the proper image.

iRMX for Windows introduced the notion of selectively loading device drivers and operating system software layers at system initialization time. A text file, `:config:loadinfo` (`:config:` is normally `:sd:/rmx386/config`), contains *sysload* commands to install the desired set of device drivers and software layers. Once the core part of the operating system has been loaded (from DOS), the `loadinfo` file is automatically submitted for execution, thereby running the set of *sysload* commands selected by the user. Users do not need to deal with the ICU, which is a bit complicated to learn to use, and there is no need to keep multiple images of the operating system on disk. Each layer or driver is stored just once, and multiple versions of the `loadinfo` file can be stored to select different configurations selected by the user. A nice feature of using *sysload* to install software is that it allows command-line arguments, such as device driver interrupt numbers or I/O port addresses, to be passed to the module as it is being loaded. With the ICU, it is necessary to change the definition file and rebuild the system every time such a change is made. The ICU might give developers the ability to change certain features of a system that cannot be done with *sysload*, but, with the introduction of *sysload* and the `loadinfo` file in iRMX III.2, it is clear that Intel's intention is to provide most of the functionality users need without using the ICU.

But what about the layers of the operating system always included in an iRMX for Windows configuration, such as the Nucleus? Users need a way to set configuration parameters for these parts of the operating system as well. To deal with this problem, a second configuration file is used based on the model used in Windows' `win.ini` file. The iRMX file, `:config:rmx.ini`, is divided into sections marked by bracketed names for the various layers of the operating system, each of which contains a series of lines in the form of <parameter>=<value>. For example, the file might begin:

```
[NUC]
UML=FFFFFFFFH;                  Upper Memory Limit
OSX=14H;                        Number of user OS Extensions
```

These lines cause the Nucleus layer to use of all memory available on the system (0xFFFFFFFF is the same as no limit) and to reserve `0x14` descriptor table slots for user-installed system calls, described in chapter 10. Volume 2 of the iRMX for Windows documentation set, *System Configuration and Administration*, contains detailed information on managing the `rmx.ini` file.

12.11 Summary

The iRMX operating system provides a rich set of resources for developing robust real-time systems. iRMX for Windows adds to this by allowing iRMX applications to coexist and interact with either DOS or Windows applications in ways that build on the advantages that each operating environment has to offer. The primary contribution of DOS is its widespread availability and familiarity. From our point of view, the primary contribution of Windows is its DDE mechanism for exchanging information among applications, although its graphical interface and task switching capabilities can also be useful in developing user interfaces to real-time systems. iRMX contributes its real-time resources to the system. The major real-time feature, of course, is rapid, deterministic scheduling of real-time tasks. In the tradition of real-time operating systems, however, iRMX also provides developers with a great number of resources for customizing the operating system to the needs of an application using the same facilities used by the developers of the operating system itself.

iRMX for Windows, to take advantage of the DOS and Windows environments, adds several features to iRMX that cannot be provided in an iRMX-only configuration. Features like console sharing, interrupt management, extended memory management, and access to DOS file systems are basic extensions necessary to enable DOS, Windows, and iRMX to coexist successfully. Features like *sysload*, device drivers, and run-time configuration that were introduced largely as convenience items in iRMX for Windows do not need to be unique to the DOS environment and are being incorporated into other versions of iRMX as well, most notably iRMX III. Finally, incorporating iRMX's networking facilities into iRMX for Windows has generated a facility that goes beyond what either system could provide alone: networked DDE.

Appendix

A

SoftScope III Command Summary

NOTE: This Appendix is extracted from chapter 5 of the *SoftScope III Reference Manual*, reprinted with permission of Concurrent Sciences, Inc. The full manual is included as volume 13 of the iRMX for Windows documentation set.

BPSCOPE

Syntax

BPSCOPE [TASK | JOB | GLOBAL]

TASK	=	Sets the scope so only one task (as specified by the TASK command) can trigger a breakpoint.
JOB	=	Sets the scope so only the tasks in a job (specified by the TASK command) can trigger a breakpoint.
GLOBAL	=	Sets the scope so any task can trigger a breakpoint.

Description

BPSCOPE determines the scope of all breakpoints which are set after the BPSCOPE command is issued. A breakpoint can be triggered within a task, within any task in a job, or within any task in the system.

BPTIMEOUT

Syntax

BPTIMEOUT [*decnumber32*]

decnumber32	=	A 32-bit unsigned integer.

Description

BPTIMEOUT sets or displays the maximum time Soft-Scope III will wait for a breakpoint to be hit. If this time is exceeded, Soft-Scope III will report < Task running > and change the prompt to a running prompt (an exclamation point (!) will precede the prompt).

BREAKPT

Syntax

BREAKPT [-] [*coderef*] [TASK | JOB | GLOBAL]
BREAKPT [-] WRITE | ACCESS *memref* [TASK | JOB | GLOBAL]
Abbreviation: **BR**

coderef	=	*address*
	=	[*:modname*]*#linenum*
	=	[*:modname*].*codesym*
memref	=	*address*
	=	*:modname*
	=	[*:modname*][.*codesym*]*.*datasym*
	=	[*:modname*].*codesym*
	=	[*:modname*]*#linenum*
address	=	A logical, physical, or linear address, (eg. DS:1000, 1000P, or 0FFFFL).
modname	=	A module name.
linenum	=	A line number found in the current module or in *modname*.
codesym	=	The name of a procedure or label.
datasym	=	The name of a symbol.
- *(dash)*	=	Delete breakpoint.
WRITE	=	Break when written to.
ACCESS	=	Break when read from or written to.
TASK	=	Sets the scope so only one task (as specified by the TASK command) can trigger a breakpoint.
JOB	=	Sets the scope so only the tasks in a job (as specified by the TASK command) can trigger a breakpoint.
GLOBAL	=	Sets the scope so any task can trigger a breakpoint.

Description

BREAKPT manages a list of static execution and data breakpoints. A breakpoint tells Soft-Scope III to stop execution when a condition is met.

CONSOLE

Syntax

CONSOLE *devicename* [*termtype*]
CONSOLE

devicename	=	A host-system-dependent name for the device.
termtype	=	The physical type of the second terminal.

Description

You can redirect Soft-Scope III output to a second terminal with the CONSOLE command. CONSOLE with no parameters will direct Soft-Scope III output back to the original terminal.

DISASM

Syntax

```
[count] DISASM [ALL] [NOLINES] [coderef] [TO coderef]
```

Abbreviation: **DIS**

count = An integer in the range 1 to 32,767.
coderef = *address*
= *:modname*
= [*:modname*]*#linenum*
= [*:modname*]*.codesym*
address = A logical, physical, or linear address, (eg. DS:1000, 1000P, or 0FFFFL).
modname = A module name.
linenum = A line number.
codesym = A procedure name or label.
ALL = Display op-codes and comments.
NOLINES = Don't display source lines.

Description

DISASM disassembles the instructions found at the specified address, and if the corresponding high-level lines can be determined, displays them.

DUMP

Syntax

```
[count] DUMP [ BYTE | WORD | DWORD ] [memref]
DUMP [ BYTE | WORD | DWORD ] memref [TO memref]
```

count = An integer in the range 1 to 32,767.
memref = *address*
= [*:modname*][*.codesym*]*.datasym*
= [*:modname*]*.codesym*
= [*:modname*]*#linenum*
address = A logical, physical, or linear address, (eg. DS:1000, 1000P, or 0FFFFL).
modname = A module name.
datasym = The name of a symbol.
codesym = A procedure name or label.
linenum = A line number from the current module, or from modname.
BYTE = Display in BYTE format (byte order: 1 2 3 4).

WORD	=	Display in WORD format (byte order: 2 1 4 3).
DWORD	=	Display in DWORD format (byte order: 4 3 2 1).

Description

DUMP displays blocks of memory with a hexadecimal display on the left and the corresponding ASCII field on the right. Memory dumps can be displayed in BYTE, WORD, or DWORD format (BYTE is the default). WORD and DWORD formats display memory addresses in word- or dword-length groupings.

EVAL

Syntax

EVAL [*memref* | *coderef*]

memref	=	*address*
	=	[*:modname*] [*.codesym*] * *.datasym*
	=	[*:modname*] *.codesym*
	=	[*:modname*] *#linenum*
coderef	=	*address*
	=	*:modname*
	=	[*:modname*] *#linenum*
	=	[*:modname*] *.codesym*
address	=	A logical, physical, or linear address, (eg. DS:1000, 1000P, or 0FFFFL).
modname	=	A module name.
datasym	=	The name of a symbol.
codesym	=	A procedure name or label.
linenum	=	A line number from the current module, or from modname.

Description

Evaluating a procedure displays the procedure's module name, line numbers, starting and ending addresses, and length. Evaluating pointers displays the descriptor entry and physical address associated with that pointer. Using EVAL on other kinds of symbols will produce the same display as you would see if you entered the symbol name without EVAL.

EXIT

Syntax

EXIT

Description

This command exits Soft-Scope III and returns you to system command level. You can also use QUIT.

GO

Syntax

GO [WRITE | ACCESS] *memref*
GO *coderef*
GO RETURN
Abbreviation: **G**

coderef = *address*
= [*:modname*]*#linenum*
= [*:modname*]*.codesym*
memref = *address*
= [*:modname*][*.codesym*]*.datasym*
= [*:modname*]*.codesym*
= [*:modname*]*#linenum*
address = A logical, physical, or linear address, (eg. DS:1000, 1000P, or 0FFFFL).
modname = A module name.
datasym = The name of a symbol.
codesym = A procedure name or label.
linenum = A line number from the current module, or from *modname*.
WRITE = Go till coderef is written to.
ACCESS = Go till coderef is read or written to.
RETURN = Go till return from the current procedure.

Description

GO tells Soft-Scope III to transfer execution to your application. The program will start executing at the current execution point. GO *coderef* sets a temporary breakpoint at the desired code reference -- a line number, label name, procedure name, or absolute address. Execution then proceeds at full speed until that (or any other) breakpoint is hit.

HELP

Syntax

HELP [*topic*]
topic = Soft-Scope III command name or Help topic.

Description

HELP provides on-line assistance with Soft-Scope III syntax and usage. Each Soft-Scope III command has a HELP entry associated with it. HELP with no parameters displays the command syntax summary, as well as a list of other topics for which help text is available.

LINE

Syntax

LINE [*coderef*]
Abbreviation: *<carriage return>*

coderef = *address*
= *:modname*
= [*:modname*]*#linenum*
= [*:modname*]*.codesym*

address = A logical, physical, or linear address, (eg. DS:1000, 1000P, or 0FFFFL).

modname = The name of a module from your program.

linenum = A line number from the current module, or the module given by *modname*, if supplied.

codesym = The procedure name or label.

Description

LINE directs Soft-Scope III to display as much information as it can about the whereabouts of the specified address. The display will include some or all of the following: line number, module name, procedure name, and source line or assembly instruction.

LIST

Syntax

[*count*] **LIST** [*lineref* | TO *lineref*]
LIST *lineref* TO *lineref*

Abbreviation: **L**

count = An integer in the range 1 to 32,767.

lineref = *:modname*
[*:modname*]*#linenum*
[*:modname*]*.codesym*

modname = A module name.

linenum = A line number from the current module or from *modname*.

codesym = A procedure name or label.

Description

Use LIST to display source lines from a module's listing, or find a specified string in a source file. Soft-Scope III uses the lines from the compiler-generated listing file.

LOAD

Syntax

LOAD [SYMBOLS] *filename*

filename = A host system dependent identifier for a disk file. (eg. /SRC/FILE).

SYMBOLS = Load only symbols.

Description

The LOAD command loads symbols from the specified file, and (with the exception of LOAD

SYMBOLS), loads the code and data into iRMX III free space, through the Application Loader. This command is designed to be used with applications written to be run under the Human Interface and generated by BND386 (or BND286). The application loaded under Soft-Scope III will be automatically deleted upon exit from the debugger.

The LOAD SYMBOLS command allows you to load symbolics without disturbing the selected application or changing register values. This command is appropriate for first-level jobs or device drivers embedded in the iRMX III boot file.

LOADSEGS

Syntax

LOADSEGS [*segtoken jobtoken filename*]

segtoken = *hexnumber16*
= *datasym*
jobtoken = *hexnumber16*
= *datasym*
filename = a host system dependent identifier for a disk file.
hexnumber16 = A 16-bit hexidecimal number.
datasym = A name of a symbol.

Description

You may access and debug files loaded by your application through the iRMX system call RQALOAD() with the macro LOADSEGS. This macro will also load the symbolic information for the file specified. You must follow the RQALOAD() call with a RQECREATEIOJOB(). The *segtoken* is the token returned to the caller of RQALOAD() via a mailbox. The *jobtoken* is returned directly by the RQECREATEIOJOB() system call. The *filename* is the name of the file passed to RQALOAD().

LOG

Syntax

LOG [*devicename* | *filename*]
LOG ON | OFF

devicename = A host-system-dependent name for the device.
filename = A host-system-dependent identifier for a disk file. (eg. /src/output.ss).
ON = Log to device.
OFF = Stop logging to device.

Description

Use LOG *filename* to create or open a file and begin copying most Soft-Scope I/O to that file.

MACRO

Syntax

```
MACRO [LIST]
MACRO LOAD filename
MACRO DELETE [macroname]
MACRO STEP [macroname]
```

filename = A host-system-dependent identifier for a disk file. (eg. /src/samp.mac).
macroname = The name of a macro from the currently- loaded macro file.

Description

This command gives you the basic tools you will need to manipulate Soft-Scope III macros. For information about how to create your own macros, see *Macros* (*Chapter 7* of the full SoftScope Manual.

MODULE

Syntax

```
MODULE [:modname = filename]
       modname = The name of a module from your program.
```

modname = A module name.
filename = A host-system-dependent identifier for a disk file. (eg. /SRC/FILE.LST).

Description

MODULE displays the current listing file assignments. MODULE :*modname* = *filename* assigns a listing file to a program module.

QUIT

Syntax

```
QUIT
```

Description

This command exits Soft-Scope III and returns you to system command level. You can also use EXIT.

REG

Syntax

```
REG [ALL | FLOAT]
```

ALL = Show system registers.
FLOAT = Show Floating Point Registers.

Description

REG displays the contents of the CPU registers. When the 386 is running in protected mode, REG ALL gives a fuller display.

RESUME

Syntax

RESUME *tasktoken*
tasktoken = *hexnumber16*
datasym
hexnumber16 = A 16-bit hexidecimal number.
datasym = A name of a symbol.

Description

The RESUME macro allows you to restart a suspended task. This macro, in conjunction with SUSPEND, is useful when you are trying to debug a task and its interaction with another task is preventing you from determining the problem. You could suspend one task while you locate the problem and resume the task once the problem is solved.

SET

Syntax

SET [*optionname* [= *optionvalue*]]

Description

Soft-Scope III maintains a list of options for the Soft-Scope III environment and their associated values. If you set up these values in the Soft-Scope III configuration file SS.SET, they will be configured whenever you bring up Soft-Scope III. Soft-Scope III uses these options for specific operations, but only looks at a value when it is needed, so it's possible to specify an invalid option and not generate an error until that option is used by some other Soft-Scope III command.

The following options are available:

Option	Description
sym.case	Consider case in searches.
sym.pointer	Type of FAR pointer to use.
sym.descriptor	Descriptor Type Override type.
src.path.ext	Pathname for source-file searches.
src.path	Pathname for all file searches.
src.tab	Tab equivalence for any files.

cmd.history	Number of commands available to recall.
cmd.macro	Initial macro file(s) to load.
cmd.prompt	Soft-Scope III prompt to use.
cmd.initial	Initial command or macro to execute when Soft-Scope III is invoked.
cmd.silent	Disable the bell.

STACK

Syntax

```
[count]      STACK [TRACE] [LINES]
       STACK USAGE | RESET
```

count = Number of levels to view.
TRACE = Display calling statements.
LINES = Display source line.
USAGE = Display current stack level.
RESET = Clear unused stack area.

Description

STACK TRACE shows procedure call nesting. It tells you what procedure called what procedure, starting at the your current execution point and proceeding backwards. If the stack display is longer than can fit on one screen, you will have the option to continue tracing backwards along the stack. STACK LINES displays the source line that made each procedure call.

STEP

Syntax

```
[count] STEP [ASM] [INTO]
```
Abbreviation: **S**

count = number of source lines to execute.
ASM = Step one assembly instruction at a time.
INTO = Step into all calls.

Description

The STEP command executes source code one line at a time. Soft-Scope III displays the next line to be executed. If the execution doesn't start at the beginning of a line, you will see the "[Inside]" prompt, telling you that the first step began in the middle of the assembly code generated for that line. STEP ASM displays disassembled instructions and steps in assembly-language increments. STEP steps over all calls. Specify STEP INTO to step into all calls.

SUSPEND

Syntax

```
SUSPEND tasktoken
        tasktoken = hexnumber16
                    datasym
        hexnumber16 =  A 16-bit hexidecimal number.
        datasym     =  A name of a symbol.
```

Description

The SUSPEND macro in conjunction with the RESUME macro allows you to suspend a task and then restart it. This is useful when you are trying to debug a task and its interaction with another task is preventing you from determining the problem. You could suspend one task while you locate the problem and resume the task once the problem is solved. This macro corresponds exactly to an iRMX RQSUSPEND() system call.

SYSTEM

Syntax

SYSTEM *program*

Abbreviation: **SYS**

program iRMX command to execute.

Description

SYSTEM allows you to execute operating system commands from inside Soft-Scope III. When you are finished, you will return to the Soft-Scope III command line.

TASK

Syntax

```
TASK [tasktoken] | [ALL]
        ALL         =  All tasks from all Soft-Scope III sessions.
        tasktoken   =  hexnumber16
                       datasym
        hexnumber16 =  A 16-bit hexidecimal number.
        datasym     =  A name of a symbol.
```

Description

The TASK macro allows you to determine the status of other tasks being debugged and to change the current task context. TASK reports information on which tasks are at a breakpoint, and will print source-level information about the breakpoint, if possible. The task whose

context Soft-Scope III is currently using is denoted by the asterisk in the left-most column. TASK ALL lists all tasks at break from all Soft-Scope III sessions.

TYPE

Syntax

```
TYPE [memref | coderef]
        coderef  = address
                 = :modname
                 = [:modname]#linenum
                 = [:modname].codesym
        memref   = address
                 = :modname
                 = [:modname][.codesym]*.datasym
                 = [:modname].codesym
                 = [:modname]#linenum
        address  = A logical, physical, or linear address, (eg. DS:1000, 1000P, or
                   0FFFFL).
        modname  = A module name.
        datasym  = The name of a symbol.
        linenum  = A line number from the current module, or from modname.
        codesym  = A procedure name or label.
```

Description

TYPE displays all available information about a variable's data type. In addition to the type, the display will show scope (global, module, or local), and storage class (static, stack-based, parameter, or based). TYPE allows you to look at the composition of large, complex data structures without looking at the contents of these variables. Use TYPE *memref* to see if a variable is stack-based and only reachable from within the procedure where it is declared.

"V" MACROS (SDB)

Syntax

```
VT objtoken                 (view token)
VJ [jobtoken]               (view jobs)
VO jobtoken                 (view object)
VD jobtoken                 (view directory)
VU tasktoken                (view unwind)
VK                          (view ready)
VS                          (view stack)
VC segment:offest           (view call)
VF                          (view free)
VB devicename               (view duib)
VR segtoken                 (view iors)
VH                          (view help)
VMI [hexnumber16]           (view message input)*
VMO [hexnumber16]           (view message output)*
VMF                         (view message failsafe)
```

* The interactive mode of these commands (using a comma, ',') is not supported.

objtoken = *hexnumber16*
= *datasym*
jobtoken = *hexnumber16*
= *datasym*
tasktoken = *hexnumber16*
= *datasym*
segtoken = *hexnumber16*
= *datasym*
segment = *hexnumber16*
offset = *hexnumber16*
= *hexnumber32*
devicename = A host-system-dependent name for the device.
hexnumber16 = A 16-bit hexadecimal number.
hexnumber32 = A 32-bit hexadecimal number.
datasym = A name of a symbol.

Description

You can access the iRMX III System Debugger (SDB) through a set of macros in SS.MAC. Soft-Scope III provides information about iRMX system objects, such as mailboxes, tasks, jobs, semaphores, segments, and regions. It also displays stack and system call information. If you are using a Multibus II system, you can display the input or output message buffer of the Message Passing Coprocessor (MPC), or toggle its fail-safe timeout feature.

VERSION

Syntax

VERSION

Description

VERSION displays Soft-Scope III's version number and information about its host operating system.

Appendix

B

Terminal Support Code

This appendix describes the technique for adapting the iRMX for Windows console driver to work with ANSI X3.64 escape sequences. The technique uses the translation feature of the Terminal Support Code (TSC), a layer of software that acts as a programmable filter between terminal I/O drivers and application programs. The material in this appendix is derived from Appendix C of the *Device Driver Programming Concepts* manual of the iRMX for Windows documentation set (Chapter 2 of the *Device Driver User's Guide* in the documentation sets for other versions of iRMX), from the *Aedit* Reference manual, and from experimentation.

When a program writes characters to the screen, they pass through the TSC on their way to the driver. If the TSC has been told to, it will intercept X3.64 sequences and translate them into character codes appropriate for the specific type of non-ANSI terminal that is to receive the output. An example of a non-ANSI terminal is the iRMX console driver for the PC, which has the DUIB name D_CONS. On input, the driver accepts keyboard scan codes and puts the proper character codes in the application's input buffer. On output, the driver accepts character codes from the application's output buffer and determines what to place in the PC's video memory to get the proper characters to appear on the screen.

The following table lists some X3.64 escape sequences and the corresponding D_CONS control codes.

Function	X3.64	D_CONS
Cursor Forward	<esc>	<0x19>
Cursor Backward	<esc>	<0x1F>
Cursor Up	<esc>	<0x1E>
Cursor Down	<esc>	<0x1C>
Cursor Position	<esc>	see text
Clear Screen	<esc>	<0x0C>

In the table, <esc> is the ASCII escape character <0x1B>, and 99 represents an arbitrary number which, if omitted, defaults to 1.[1] D_CONS cursor positioning is performed with a sequence of bytes consisting of a lead-in character <0x04> and two bytes giving the row and column as binary numbers offset by 0x20. For example, the cursor could be positioned to row 12, column 34 by sending three bytes with the values 0x04, 0x2B, and 0x41 in that order. (Row and column numbering start at zero.) Another way to position the cursor using the D_CONS driver would be to clear the screen and send a sequence of 11 cursor-down <0x1C> and 33 cursor-forward <0x19> characters. The file hellocon.cad illustrates the use first technique (cursor addressing):

```
File hellocon.cad: <0x0C><0x04><0x2B><0x41>Hello There!
```

The file hellocon.cmv illustrates accomplishing the same thing using cursor movement codes:

```
File hellocon.cmv:

<0x0C><0x1C><0x1C><0x1C><0x1C><0x1C><0x1C><0x1C><0x1C><0x1C><0x1C>
<0x1C><0x1C><0x19><0x19><0x19><0x19><0x19><0x19><0x19><0x19><0x19>
<0x19><0x19><0x19><0x19><0x19><0x19><0x19><0x19><0x19><0x19><0x19>
<0x19><0x19><0x19><0x19><0x19><0x19><0x19><0x19><0x19><0x19><0x19>
<0x19><0x19><0x19><0x19><0x19>Hello There!
```

To generate these files yourself, you must insert the indicated hexadecimal codes into the files using a text editor. With *Aedit*, you can use the HI (hex insert) command from command mode or the <^R> command while in insert mode. To use these files, just copy them to the screen using the iRMX *copy* command. For example:

```
rmx> copy hellocon.cmv
```

When the D_CONS device driver receives the characters from the file (because the copy command writes them to :co:), it inserts the proper characters into the display adapter's video RAM to cause a message to appear in the middle of the screen. The .cmv file contains a clear screen character, followed by a sequence of cursor-down and cursor-forward characters to do the positioning. The .cad file accomplishes the same thing using direct cursor addressing.

If you try typing the hellocon.* files on a DOS console (with ANSI.SYS installed) or copy them to the screen of a terminal that recog-

[1]Some of the codes in this appendix can be difficult to read. The symbols > and < never appear in any of the codes being described; they are always used to delimit single character codes. Where ASCII names like <esc> and <cr> are not available, the hexadecimal values of the necessary character codes are given using C-language syntax inside the angle brackets.

nizes ANSI sequences, they will not accomplish the same thing because the D_CONS control codes are not the same as the corresponding ANSI escape sequences. The files helloans.cmv and helloans.cad are equivalent to the hellocon files, but illustrate the use of X3.64 escape sequences. These files can be copied to the screen of a DOS or ANSI terminal with the desired effects, but they do not work on the iRMX for Windows console because the D_CONS driver does not recognize them. These files will be looked at after we solve problem of getting the D_CONS driver to accept ANSI escape sequences.

The terminal support code saves the day. I programmed the TSC to translate X3.64 escape sequences into D_CONS control codes. The TSC is programmed by writing TSC control codes to the device. These control codes begin <esc]> as opposed to ANSI codes that begin with <esc>[. When the TSC receives the <esc>] sequence, it interprets everything until it receives the sequence <esc>\ as information for its own use, and does not pass on any of those characters to the device driver. Here is an example of a TSC escape sequence:

```
<esc>]T: E2=25 <esc>\
```

The T: part tells TSC that this sequence applies to the terminal as opposed to an I/O connection, for example. E2=25 tells TSC that X3.64 escape sequence number 2 is equivalent to decimal code 25 for this terminal. Table C-5 in *Device Driver Programming Concepts* (Table 2-3 in *Device Drivers User's Guide*) must be consulted to determine that X3.64 sequence number 2 is the cursor-forward sequence (<esc>[99C). The value 25 is 0x19, which is the D_CONS code to move the cursor forward. Any number of translations can be set up with a single TSC sequence:

```
<esc>]T: E2=25, E3=31, E4=30, E5=28, E6=4, E30=12 <esc>\
```

E2 through E5 correspond to the four cursor movement functions, E6 is for cursor addressing, and E30 is the clear screen function. To perform cursor addressing, TSC must also be told the order in which the row and column are specified (row, then column, in the case of D_CONS), and the offset value for row and column numbers (32 for D_CONS). This information can be specified by the following sequence:

```
<esc>]T: F=1, U=32 <esc>\
```

This string was determined by consulting Table C-4 in the *Device Driver Programming Concepts*. (Table 2-2 in the *Device Driver User's Guide*.) The field F=1 tells the TSC that rows are specified before columns, and U=32 tells the offset value for converting encoded row and column numbers to lines and column numbers on the screen. You can also view and set these

parameters with the HI *term* command. With no command-line arguments, *term* displays current settings. The command term `yx over=32` will have the same effect as this TSC sequence.

The file `ansi.osc` contains a TSC string for all of the preceding plus a few other translation values, such as for erasing a line or part of the screen. I chose the `.osc` extension for the file name because Intel documentation calls these strings for the TSC *Operating System Command* (OSC) strings. There is another type of string that the TSC recognizes called an *Application Program Command (APC) string*, but that is not of concern here. Here is `ansi.osc`:

```
File ansi.osc
<esc>] T:F=1, U=32, E2=25, E3=31, E4=30, E5=28, E6=4,E26=05, E28=01,
E30=12, E31=03, E33=02, E35=06<esc>\
```

Just setting all the translation rules and cursor addressing information is not enough. It is also necessary to turn on TSC translation to get the TSC to intercept X3.64 sequences and pass the correct control codes on to the device driver. This can also be done using the *term* command, but it is usually done with a TSC string that looks like:

```
<esc>] T: T=1 <esc>\
```

There is one more messy detail to handle before proceeding to show the files for ANSI control of the `D_CONS` driver. *Aedit* and the CLI have their own way of controlling the screen display and processing such keyboard characters as the arrow keys. They access the `:config:termcap` file to find out which control codes to send to the device driver, assuming no TSC translation is in effect. The termcap file contains strings such as `AFMR=19`, which *Aedit* and the CLI interpret as "the character code to move the cursor to the right is <0x19>." The point here is that TSC translation must be off for *Aedit* and the CLI to work correctly. Either that, or you have to provide a complete mapping of X3.64 codes to `D_CONS` codes, turn translation on, and use the CLI's *set term*=`vt100` command to tell *Aedit* and the CLI to use ANSI sequences rather than `D_CONS` codes. VT100 is the entry in `:config:termcap` that is the same as X3.64.

Finally, here are the files that use ANSI escape sequences to accomplish the same thing as the files `hellocon.cad` and `hellocon.cmv` presented previously.

```
File helloans.cad
<esc>] T: T=1<esc>\<esc>[2J<esc>[12;34HHello
There!<esc>[9;9HAgain!!<cr><lf><esc>] T: T=0<esc>\
```

```
File helloans.cmv
<esc>] T: T=1<esc>\<esc>[2J<esc>[11B<esc>[33CHello
There!<cr><lf><esc>] T: T=0<esc>\
```

You can see that the files contain the TSC string to turn on translation, followed by the ANSI sequences to clear the screen and position the cursor in the middle. Next comes the message text, including <cr><lf>, followed by the TSC string to turn translation back off so that *Aedit* and the CLI will still work after the file is copied to the screen. If you *type* this same file on the DOS console, the message will still show up in the proper place, but you will also see the TSC string on the screen because ANSI.SYS does not know what to do with it.

Although all the examples in this appendix have shown the contents of files, application programs can generate the same effects simply by writing the same codes to the console output device. For example, you could have a small program that writes the equivalent of `ansi.osc` to the terminal, and include that program in your `r?logon` file (or your `:config:r?init` file). The TSC information only needs to be output once. After that, you could use *term* to turn translation on and off as desired, and have your applications generate ANSI escape sequences for all cursor control operations.

Appendix

C

Stream I/O

C.1 Overview

This appendix shows how to use the iRMX stream file driver to implement intertask communication. Understanding streams helps explain two totally different features of the operating system: IORS processing by device drivers and command processing by a CLI. First, the structure of an application that uses streams for intertask communication will be reviewed. The main issue in designing an application that uses streams is to understand how the streams device driver processes IORSs. (Device drivers and IORS processing are covered in chapter 9.) I then show how streams can be used to implement I/O redirection by a CLI.

C.2 Stream IORS Processing

Two tasks communicate using a stream file by performing read and write operations on the same stream file. One task writes to the file, and the other task reads from it. The stream device driver copies data from the writer's output buffer in RAM directly to the reader's input buffer. No actual I/O device is involved in the process. Any number of tasks read or write a given stream file, possibly even just one task does both the writing and the reading.

As explained in chapter 8, the iRMX I/O model is based on the iRMX object type called an I/O connection. I/O connections to devices encapsulate information about the device driver and the file driver to be used for performing I/O operations and are sometimes called *device connections.* I/O connections to files identify the particular file to be accessed on a device and are sometimes called *file connections.* Both device connections and file connections are really just one (composite) type of object as far as the iRMX Nucleus is concerned, but the BIOS differentiates between the

two in two important ways:

1. No more than one connection can exist to a particular device driver at any time on an iRMX system. Any number of file connections can exist based on a single device connection, however.
2. Any task can use a particular device connection, but only tasks that belong to the same job can use the same file connection. The BIOS does, however, allow a task to use a file connection belonging to another job to create a new file connection, one that belongs to the calling task's owning job.

A stream device driver and a stream file driver are provided on virtually all iRMX systems. (It would be possible to configure an iRMX system that does not support streams by using the ICU.) Furthermore, an I/O connection to the stream device is normally established automatically as the system initializes. The logical name for this device connection is `:STREAM:`.

As with I/O connections for other device drivers and file drivers, there is just one I/O connection to the stream device, but there can be any number of file connections based on the connection to the device. A single file connection does not uniquely identify a particular file, however. For example, a single disk file can be accessed concurrently by two different programs using two different connections to the file (provided only that the two connections are opened with compatible file sharing modes). For disk files, what uniquely identifies a file is the file's fnode data structure.

The fnode data structure is stored on the disk itself, and a copy is stored in memory whenever connections to the file exist. There is an analogous data structure to a disk file's fnode for each different stream file that exists, called the stream file's file node. Thus, multiple sets of tasks can be communicating by means of different stream files at the same time. The different sets of tasks use file connections with different stream file node numbers. Just as two tasks read or write the same disk file if their I/O connections specify the same fnode number, two tasks read or write the same stream file if their I/O connections specify the same stream file node number.

The BIOS creates a new stream file node number each time a file connection is created based on the device connection to the stream device driver. Two ways to do this are the following:

1. Call *rqacreatefile( )* or *rqsattachfile( )* with the `prefix` parameter set to the token for the stream device connection and the `subpathPtr` set to null.
2. Call *rqscreatefile( )* or *rqsattachfile( )* with the logical name for the stream device (`:STREAM:`) as the pathname.

The two ways to create a new stream connection with a stream file number that matches an existing one are:

1. Call *rqacreatefile( )* or *rqaattachfile( )* with the `prefix` parameter set to a token for an existing stream file connection.
2. Catalog a stream file connection in some job's object directory by calling *rqscatalogconnection( )* or *rqcatalogobject( )* and use the resulting logical name as the pathname argument to *rqscreatefile( )* or *rqsattachfile( ).*

When a task reads or writes using a stream connection, the stream file driver creates an IORS for the operation and calls the device driver's *queueIO( )* procedure to enter the IORS on the driver's queue of work to be done. The device driver tries to match the new IORS with any others that it has already received with the same stream fnode number. If a match exists, the device driver copies data from the writer's buffer to the reader's buffer, and returns both IORSs to their response mailboxes. If no match is found, the new IORS is simply added to the driver's queue of work to be done. If the number of bytes being read is different from the number of bytes being written, the driver completes any I/O requests it can and leaves any extra bytes pending until further IORSs arrive. Note that the mailbox used by the sample device driver in chapter 9 does not work as the stream driver's IORS queue because the stream device driver needs to be able to access more than one IORS at a time.

Figure C.1 is a program that uses BIOS system calls to illustrate stream IORS processing. The main task creates a new stream file connection and then does a series of I/O transfers: a read of 12 bytes, two writes of 9 bytes each, and a read of 6 bytes. The first read completes only after both write operations are performed. The second write operation provides 6 more bytes than the first read requested, so the driver uses those 6 bytes to satisfy the second read request. The sequence in which the read and write operations are performed is arbitrary as far as correct operation of the program is concerned. The task that monitors the response mailbox terminates the program when no IORSs arrive in a 2-second period.

Figure C.1 A program that performs a series of stream file transfers using BIOS (asynchronous) system calls.

```
/***>  rwwr.c  <************************************************
 *
 *       One of a series of programs which performs asynchronous
 *       stream I/O.
 *
 *       This program performs two reads and two writes in the
 *       sequence r-w-w-r.  The total number of bytes read and
 *       written are equal (24).
 *
 ****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <rmxc.h>
```

Figure C.1 (*Continued*)

```
#define chk(x) if (Status) printf ("%X %s\n", Status, x)

/*
 *      Global variables
 */
TOKEN           pipe, respMbx, iorsTkn, streamDev;
A_IORS_DATA_STRUCT *iorsPtr;

/*
 *      Task to monitor the response Mailbox and display
 *      information about IORSs that arrive there during
 *      reading and writing
 *      -----------------------------------------------
 */
void far task1 (void) {
WORD            i, Status;
char            *bufPtr, hexTab[] = "0123456789ABCDEF";
char            *funcNames[] = {"read",
                                "write",
                                "seek",
                                "special",
                                "attach device",
                                "detach device",
                                "open",
                                "close"};

  for (;;) {
    iorsTkn = rqreceivemessage (respMbx, (unsigned short) 600, NULL,
&Status);
    if (Status == 0x0001) {
      printf ("time out\n");
      exit (0);
      }
    iorsPtr = (A_IORS_DATA_STRUCT *) iorsTkn;
    printf ("%s transferred %ld bytes with status %4X\n",
        funcNames[iorsPtr->funct],
        iorsPtr->actual,
        iorsPtr->status);
    bufPtr = (char *) iorsPtr->buf_ptr - (int) iorsPtr->actual;
    printf ("Data transferred:");
    for (i = 0; i < iorsPtr->actual; i++) {
      printf (" %c%c", hexTab[(*bufPtr >> 4) & 0x0F], hexTab[*bufPtr &
0x0F]);
      bufPtr++;
      }
    printf ("\n");
    }

}

/*
 *      Main task - initializes everything and does the reading
 *      and writing.
 *      -----------------------------------------------------
 */
int main (int argc, char *argv[]) {
BYTE            buffer[30];
BYTE            myPrio;
```

Figure C.1 (*Continued*)

```
WORD            Status;

/*
 *      Initialize
 */
  myPrio = rqgetpriority ((selector) NULL, &Status);
  streamDev = rqslookupconnection ("\x008:STREAM:", &Status);
  respMbx = rqcreatemailbox ((unsigned short) 0, &Status);

  rqacreatefile ((selector) NULL,              /* user */
                 streamDev,                    /* prefix */
                 (selector) NULL,              /* path */
                 (unsigned char) 0x0F,         /* access */
                 (unsigned short) 0x0000,      /* granularity */
                 (unsigned long) 0x00000000,   /* size */
                 (unsigned char) 0x00,         /* must create */
                 respMbx, &Status);
  pipe = rqreceivemessage (respMbx, (unsigned short) 0xFFFF, NULL,
&Status);
  chk ("main create");
  iorsPtr = (A_IORS_DATA_STRUCT *) pipe;
  rqaopen (pipe, (unsigned char) 0x03, (unsigned char) 0x03, respMbx,
&Status);
  iorsTkn = rqreceivemessage (respMbx, (unsigned short) 0xFFFF, NULL,
&Status);
  iorsPtr = (A_IORS_DATA_STRUCT *) iorsTkn;
  chk ("main open");

  rqcreatetask (myPrio, &task1, (selector) NULL, NULL,
    (unsigned long) 8192, (unsigned short) 0, &Status);

/*
 *      Exercise the stream connection.  Display a progress message
 *      and delay for a second after each operation so the user may
 *      observe when the other task blocks relative to this one.
 */
  rqaread  (pipe,
            buffer,
            (unsigned long) 12,
            respMbx,
            &Status);
  chk ("main read");
  printf ("read\n");
  rqsleep ((unsigned short) 100, &Status);

  rqawrite (pipe,
            (unsigned char *) "Message 1",
            (unsigned long) 9,
            respMbx,
            &Status);
  chk ("main write 1");
  printf ("write 1\n");
  rqsleep ((unsigned short) 100, &Status);

  rqawrite (pipe,
            (unsigned char *) "Message 2",
            (unsigned long) 9,
            respMbx,
```

Figure C.1 (*Continued*)

```
            &Status);
  chk ("main write 2");
  printf ("write 2\n");
  rqsleep ((unsigned short) 100, &Status);

  rqaread  (pipe,
            buffer,
            (unsigned long) 6,
            respMbx,
            &Status);
  chk ("main read");
  printf ("read\n");
  rqsleep ((unsigned short) 100, &Status);

/*      The other task exits the job
 *      when it stops receiving IORSs.
 */
  rqsuspendtask ((selector) NULL, &Status);
}
```

The code in Figure C.1 uses a single task and a single I/O connection for all reading and writing to illustrate basic stream I/O without having to deal with the intricacies of steam I/O synchronization. There are two intricacies to consider: EIOS serialization of I/O operations on a connection, and management of arbitrary data transfer sizes. These two issues are addressed in the next two sections.

C.2.1 EIOS serialization

The EIOS serializes all I/O transfers performed using a single I/O connection. It was mentioned earlier that EIOS transfers are synchronous in the sense that they do not return to the caller until the data transfer completes. In addition, the EIOS does not release a request for an I/O operation using a connection until all previous operations using that same connection have completed. If the main task in Figure C.1 had used EIOS calls for reading and writing, execution would never have proceeded past the first read operation because it could not complete until another task performed a matching write operation. The point being made here is that another task could not perform the matching write operation using the same connection object as the main task (the token `pipe`) because the EIOS would not actually issue that other task's write request to the BIOS until the read request using that same connection had completed. This deadlock can be avoided by having the main task catalog its connection in some job's object directory and have the second task call *rqscreatefile( )* or *rqsattachfile( )* using the logical name specified by the initial task to obtain a second connection to the same stream file node. Alternatively, the second task could use the main task's token as the `prefix` parameter in a call to *rqacreatefile( )* or

rqaattachfile( ). In either case, the second task *could* use a second connection to the same stream file to initiate a second I/O operation.

C.2.2 Managing arbitrary transfer sizes

iRMX streams are often used in situations where the reading or writing task does not know how much data the other side of the stream is transferring. A sample program shown later in this appendix, for example, shows a task that reads an arbitrary number of characters from the user's console and writes them to a stream that acts as the console input for an arbitrary task that could be trying to read any number of bytes at a time.

The *rq[as]special( )* system calls provide function codes 0 and 1 to handle this problem for streams. Function code 0 is called `query`, and function code 1 is called `satisfy`. A task uses the `query` function to find out how many bytes have been requested by a read or write operation on a stream. The call returns an IORS that the caller examines to determine the value of the `count` field, which can then be used as the count operand for a matching write or read operation. The `satisfy` function is used to force another task's read or write request to complete, with the actual number of bytes transferred being equal to the number of bytes already matching the pending request, even if that number is less than the number of bytes requested. Examples of these calls being used are shown after the system calls provided by the Human Interface (HI) for command-line processing are discussed.

C.3 Command-Line Processing

The HI provides support for executing command lines programmatically through the *rqcsendcommand( )* system call. The function prototype for this call is the following:

```
void
rqcsendcommand (            TOKEN           commandConn,
  STRING far *                              commandLine,
  WORD far *                                commandExceptionPtr,
  WORD far *                                exceptionPtr);
```

The `commandLine` argument is simply an iRMX string (byte count followed by characters) that looks like a command line typed in by a user. In fact, the iRMX CLI reads command lines typed by a user and uses just this system call to pass them to the HI for processing. The buffer into which the CLI reads command lines is pointed to by the `CommandLine` argument to the call. Application programs can invoke any HI command programmatically by putting a pointer to any command string in this argument.

Two different condition codes are associated with this system call. The code pointed to by `exceptionPtr` can be set to a nonzero value for two

different situations. One is when the commandLine is terminated with an ampersand, which is the iRMX continuation mark for command lines. In this case, the HI returns a condition code value of 0x0083 (E_CONTINUED), and the program that called *rqcsendcommand()* should then repeat the call, supplying additional parts of the command line. The HI assembles all the parts of the command line into an internal buffer and runs the command when a line with no continuation mark is received. The second case in which the exceptionPtr condition code is nonzero is if an invalid command line is entered that prevents the HI from launching a new HI command job. Examples of this situation include naming a command that the HI cannot find in the disk directories searched for command files, or naming a command file that is not in valid STL load module format.

Once a command starts running, it might terminate with an error code. The commandExceptionPtr argument points to a word that will be set to the termination code specified by the value of the first parameter of the command's call to *rqexitiojob()*. If the job terminates because the user types <^C> or because the program encountered a processor fault, the condition code for the command is set to 0x0080 (E_CONTROLC) or 0x800X, where X is the fault code (0x0C for a stack fault, 0x0D for a general protection fault, etc.). Note that a task's call to *rqcsendcommand()*, if it does have a zero value for its normal condition code, does not complete until the child job created by the command completes processing. That is, this call is *not* an asynchronous call, even though it returns two different condition codes at two different times.

Before making a call to *rqcsendcommand()*, a program must obtain a token for an object called a *command connection.* This object is simply a memory segment in which the HI stores information about the command connection; it is not an iRMX composite object type. To create a command connection, call *rqccreatecommandconnection()*:

```
TOKEN
rqccreatecommandconnection (      TOKEN          ciConnection,
                                  TOKEN          coConnection,
                                  WORD           connectionFlags,
                                  WORD far *     exceptionPtr);
```

The first two parameters of this call are tokens for the connections that the HI sets up as :CI: and :CO: for the programs run programatically based on the command connection returned by this call. For normal commands issued through the iRMX CLI, these two tokens are simply the tokens for the CLI's own :CI: and :CO: logical names. In the next section, the use of stream files for these two connections is illustrated.

The connectionFlags parameter is a Boolean value. If it is true (any odd value), calls to *rqcsendcommand()* return a condition code value of 0x0085 (E_ERROROUTPUT) if a child command calls *rqcsendeoresponse()*, that is, if it tries to write to its error output device instead of its console

output. At the time of this writing, the value of `connectionFlags` has no effect on C programs that write to *stdout*.

C.4 I/O Redirection Using Streams

A common use of the stream device and file drivers is to implement command-line redirection and submit files by the CLI. A submit file can be thought of as command input redirection applied to an entire set of commands rather than to a single command using the < character. Commands invoked from a submit file read from the submit file itself when they read from their `:CI:` device. The basis for this I/O redirection by a CLI is illustrated in Figures C.2 through C.4. The main task (Figure C.2) creates two stream files and catalogs them in the local job's object directory using the logical names `:PIPE1:` and `:PIPE2:`. It then creates two tasks. Task 1 reads from `:CI:` and writes to `:PIPE1:`, and Task2 reads from `:PIPE2:` and writes to `:CO:`. The main task then operates as a simple CLI: it creates a command connection using the two streams as the connection's input and output devices, and then enters an endless loop in which it reads command lines from the console and sends them to the HI for processing by calling *rqcsendcommand( )*. The program exits when the user enters a *quit* command or types a null line (<\Z at the beginning of a line).

Figure C.2 Main module for a stream-based command line interpreter (CLI).

```
/***>  strmcli.plm  <*********************************************
 *
 *  Stream I/O sample program
 *  Creates three tasks and two pipes.
 *      Task 1 copies :CI: to pipe 1
 *      Task 2 copies pipe 2 to :CO:
 *
 *  The main task then acts as a CLI, using the pipes in place of
 *  the console.
 *
 ********************************************************************/
$compact (exports task1, task2)
strmcli: DO;
$include (strmcli.ext)

DECLARE
        STREAM$QUERY            LITERALLY       '0',
        STREAM$SATISFY          LITERALLY       '1',
        E$CONTINUED             LITERALLY       '0083h',
        LF                      LITERALLY       '0Ah',
        CR                      LITERALLY       '0Dh',

        ehstruct                                STRUCTURE (
            handler     POINTER,
            mode        BYTE)                   INITIAL (NIL, 0),
        (coConn, p1, p2, t1, t2)                TOKEN,
        (siConn, soConn, cmdConn)               TOKEN,
        buffer (81)                             BYTE,
        (Status, cmdStatus)                     WORD_16;
```

Figure C.2 (*Continued*)

```
/*  Test Condition Code Utility
 *  ---------------------------
 */
chkstat: PROCEDURE (StatusVal, MessagePtr, abortFlag) PUBLIC REENTRANT;
DECLARE
        StatusVal                                   WORD_16,
        MessagePtr                                  POINTER,
        Message BASED MessagePtr (1)                BYTE,
        abortFlag                                   BYTE,
        actual                                      WORD_32,
        Status                                      WORD_16;

/*
 *      If StausVal is non-zero, display its value and a
 *      message; exit the job if the abortFlag is set.
 *      Otherwise, simply return.
 */
  IF StatusVal <> 0 THEN DO;
    CALL movb (MessagePtr, @buffer, Message(0) + 1);
    CALL rqcformatexception (@buffer, 80, StatusVal, 0, @Status);
    CALL movb (@(0Dh, 0Ah), @buffer(buffer(0) + 1), 2);
    buffer(0) = buffer(0) + 2;
    actual = rqswritemove ( coConn,
                            @buffer(1),
                            DWORD(buffer(0)),
                            @Status);
    IF abortFlag THEN CALL rqexitiojob (0, NIL, @Status);
    END;
  RETURN;

  END chkstat;

task1: PROCEDURE EXTERNAL;
END task1;

task2: PROCEDURE EXTERNAL;
END task2;

/*  Main Program Starts Here
 *  ------------------------
 *
 *      First, open connection to actual console for use by chkstat.
 */
    coConn = rqsattachfile (@(4,':CO:'), @Status);
    CALL rqsopen (coConn, 2, 0, @Status);

/*
 *      Set up: in-line exception handling,
 *              streams,
 *              I/O tasks.
 */
    CALL rqsetexceptionhandler (@ehstruct, @Status);

    p1 = rqscreatefile (@(8,':STREAM:'), @Status);
    CALL chkstat (Status, @(6,'pipe1 '), TRUE);
    CALL rqscatalogconnection ( selectorof(NIL), p1, @(7,':PIPE1:'),
                                @Status);
```

Figure C.2 (*Continued*)

```
    CALL chkstat (Status, @(5,'cat1 '), TRUE);

    p2 = rqscreatefile (@(8,':STREAM:'), @Status);
    CALL chkstat (Status, @(6,'pipe2 '), TRUE);
    CALL rqscatalogconnection ( selectorof(NIL), p2, @(7,':PIPE2:'),
                                @Status);
    CALL chkstat (Status, @(5,'cat2 '), TRUE);

    t1 = rqcreatetask ( 0,
                        @task1,
                        selectorof (@Status),
                        NIL,
                        8192,
                        0,
                        @Status);
    CALL chkstat (Status, @(3,'t1 '), TRUE);
    t2 = rqcreatetask ( 0,
                        @task2,
                        selectorof (@Status),
                        NIL,
                        8192,
                        0,
                        @Status);
    CALL chkstat (Status, @(3,'t2 '), TRUE);

/*
 *      Set up command connection to use for CLI operations
 */
    siConn = rqsattachfile (@(7,':PIPE1:'), @Status);
    CALL chkstat (Status, @(14,'task3 attach1 '), TRUE);
    CALL rqsopen (siConn, 1, 0, @Status);
    CALL chkstat (Status, @(12,'task3 open1 '), TRUE);

    soConn = rqsattachfile (@(7,':PIPE2:'), @Status);
    CALL chkstat (Status, @(14,'task3 attach2 '), TRUE);
    CALL rqsopen (soConn, 2, 0, @Status);
    CALL chkstat (Status, @(12,'task3 open2 '), TRUE);

/*
 *  Create a command connection based on the pipes
 */
    cmdConn = rqccreatecommandconnection (  siConn, /* stream input  */
                                            soConn, /* stream output */
                                            1,      /* detect eo out */
                                            @Status);
    CALL chkstat (Status, @(23, 'task3 create cmd conn '), TRUE);
/*
 *  Prompt for a command line.
 *  Exit on zero-length input or a quit command.
 */
    DO WHILE 1;
      CALL rqcsendcoresponse (@buffer, 80, @(9 ,'Command: '), @Status);
      IF (buffer(buffer(0)) <> lf) THEN
        CALL rqcsendcoresponse (NIL, 0, @(2, cr, lf), @Status);
      IF (buffer(0) = 0) OR
         (cmpb (@buffer(1), @('quit'), buffer(0) -2) = 0FFFFFFFFh) THEN
         CALL rqexitiojob(0, NIL, @Status);
      CALL rqcsendcommand (cmdConn, @Buffer, @cmdStatus, @Status);
```

Figure C.2 (*Continued*)

```
      DO WHILE Status = E$CONTINUED;
        CALL rqcsendcoresponse (    @buffer, 80, @(10, 'Continue: '),
                                    @Status);
        IF buffer(buffer(0)) <> lf THEN
          CALL rqcsendcoresponse (NIL, 0, @(2, cr, lf), @Status);
        CALL rqcsendcommand (cmdConn, @Buffer, @cmdStatus, @Status);
        END;
      CALL chkstat (Status, @(13, 'send command '), FALSE);
      CALL chkstat (cmdStatus, @(15,'command failed '), FALSE);
      END;

END strmcli;
```

Figure C.3 Input task for the stream-based CLI. This task reads from the console (`:CI:`) and writes to the stream named `:PIPE1:`.

```
/***>  task1.plm  <*********************************************
 *
 *      Read from :CI:, write to :PIPE1:
 *
 ****************************************************************/
$compact (exports task1)
task1: DO;
$include (task1.ext)

DECLARE
        STREAM$QUERY            LITERALLY       '0',
        STREAM$SATISFY          LITERALLY       '1',
        LF                      LITERALLY       '0Ah',
        CR                      LITERALLY       '0Dh';

chkstat: PROCEDURE (StatusVal, MessagePtr, abortFlag) EXTERNAL;
DECLARE
        StatusVal                               WORD_16,
        MessagePtr                              POINTER,
        abortFlag                               BYTE;
END chkstat;

/*  task1: Read from :CI:, write to :PIPE1:
 *  ---------------------------------------
 */
task1: PROCEDURE PUBLIC;
DECLARE
        buffer(80)                              BYTE,
        (actualr, actualw)                      WORD_32,
        ehStruct STRUCTURE (
            handler             POINTER,
            mode                BYTE),
        (ciConn, pipeConn)                      TOKEN,
        iors                                    STRUCTURE (
            actual              WORD_32,
            device              WORD_16,
            unit                BYTE,
            funct               BYTE,
            subfunct            WORD_16,
            device_loc          WORD_32,
            buf_ptr             POINTER,
```

Figure C.3 (*Continued*)

```
            count                   WORD_32,
            aux_ptr                 POINTER),

        Status                                  WORD_16;

/*
 *  Handle Exceptions in-line
 */
    ehStruct.mode = 0;
    CALL rqsetexceptionhandler (@ehStruct, @Status);

/*
 *  Set up connections to :ci: and :pipe1:
 */
    ciConn = rqsattachfile (@(4,':CI:'), @Status);
    CALL chkstat (Status, @(14,'task1 attach1 '), TRUE);
    CALL rqsopen (ciConn, 1, 0, @Status);
    CALL chkstat (Status, @(12,'task1 open1 '), TRUE);

    pipeConn = rqsattachfile (@(7,':PIPE1:'), @Status);
    CALL chkstat (Status, @(14,'task1 attach3 '), TRUE);
    CALL rqsopen (pipeConn, 2, 0, @Status);
    CALL chkstat (Status, @(12,'task1 open3 '), TRUE);

 *
 *  When another task tries to read from the stream,
 *  read input from keyboard and write it to the stream.
 */
    DO WHILE 1;
      CALL rqsspecial (pipeConn, STREAM$QUERY, NIL, @iors, @Status);
      CALL chkstat (Status, @(12, 'task1 query '), TRUE);
      actualr = rqsreadmove (ciConn, @buffer, 80, @Status);
      CALL chkstat (Status, @(11,'task1 read '), FALSE);

      IF Status = 0 THEN DO;
        actualw = rqswritemove (pipeConn, @buffer, actualr, @Status);
        CALL chkstat (Status, @(12,'task1 write '), TRUE);
        IF actualr < iors.count THEN DO;
          CALL rqsspecial (pipeConn, STREAM$SATISFY, NIL, NIL, @Status);
          CALL chkstat (Status, @(14, 'task1 satisfy '), TRUE);
          END;
        END;
      END;

  END task1;          /* procedure */

END task1;            /* module */
```

Figure C.4 Output task for the stream-based CLI. This task reads from the stream named `:PIPE2:` and writes to the user's screen (`:CO:`).

```
/***>  task2.plm  <*********************************************
 *
 *      Read :PIPE2:, write to :CO:
 *
 ****************************************************************/
```

Figure C.4 (*Continued*)

```
$compact (exports task2)
task2: DO;
$include (task2.ext)

DECLARE
        STREAM$QUERY            LITERALLY       '0',
        STREAM$SATISFY          LITERALLY       '1',
        LF                      LITERALLY       '0Ah',
        CR                      LITERALLY       '0Dh';

chkstat: PROCEDURE (StatusVal, MessagePtr, abortFlag) EXTERNAL;
DECLARE
        StatusVal                               WORD_16,
        MessagePtr                              POINTER,
        abortFlag                               BYTE;
END chkstat;

/*  task2: Read from :PIPE2:, write to :CO:
 *  ---------------------------------------
 */
task2: PROCEDURE PUBLIC;
DECLARE
        buffer(80)                              BYTE,
        ehStruct STRUCTURE (
            handler             POINTER,
            mode                BYTE),
        (pipeConn, coConn)                      TOKEN,
        iors                                    STRUCTURE (
            actual                WORD_32,
            device                WORD_16,
            unit                  BYTE,
            funct                 BYTE,
            subfunct              WORD_16,
            device_loc            WORD_32,
            buf_ptr               POINTER,
            count                 WORD_32,
            aux_ptr               POINTER),
        (actualr, actualw)                      WORD_32,
        Status                                  WORD_16;

/*
 *  Handle Exceptions in-line
 */
    ehStruct.mode = 0;
    CALL rqsetexceptionhandler (@ehStruct, @Status);

/*
 *  Set up connections to :pipe2: and :co:
 */
    pipeConn = rqsattachfile (@(7,':PIPE2:'), @Status);
    CALL chkstat (Status, @(14,'task2 attach1 '), TRUE);
    CALL rqsopen (pipeConn, 1, 0, @Status);
    CALL chkstat (Status, @(12,'task2 open1 '), TRUE);

    coConn = rqsattachfile (@(4,':CO:'), @Status);
    CALL chkstat (Status, @(14,'task2 attach2 '), TRUE);
    CALL rqsopen (coConn, 2, 0, @Status);
    CALL chkstat (Status, @(12,'task2 open2 '), TRUE);
```

Figure C.4 (*Continued*)

```
/*
 *  Read input from the stream, and echo it to the screen.
 */
    DO WHILE 1;
      CALL rqsspecial (pipeConn, STREAM$QUERY, NIL, @iors, @Status);
      CALL chkstat (Status, @(12,'task2 query '), TRUE);
      actualr = rqsreadmove (pipeConn, @buffer, iors.count, @Status);
      CALL chkstat (Status, @(11,'task2 read '), TRUE);
      actualw = rqswritemove (coConn, @buffer, actualr, @Status);
      CALL chkstat (Status, @(12,'task2 write '), TRUE);
    END;

  END task2;     /* procedure */

END task2;       /* module */
```

As it stands, the stream I/O performed by the sample CLI does not do anything special. The command connection could just as well have been created using tokens for `:CI:` and `:CO:` instead of `:PIPE1:` and `:PIPE2:`. However, this CLI does work, so it serves as an illustration of tasks that communicate successfully using stream I/O. Furthermore, the sample CLI could easily be extended by having Task 1 read from a disk file or some other device rather than from `:CI:`, or by having Task 2 write to a disk file or other device rather than to `:CO:`, thereby implementing I/O redirection. The logic of the CLI itself remains unchanged, and the commands executed by the calls to *rqcsendcommand()* never know the difference.

There is a significant difference between the environment in which commands are run using the sample CLI and those run using the normal iRMX CLI. Under the iRMX CLI, commands use actual connections to `:CI:` and `:CO:` unless the user invokes command line redirection using the < and/or > characters or uses the *submit* command. I/O through `:CI:` and `:CO:` is processed by a terminal I/O device driver, which means that an application program can make *rq[as]special()* system calls specific to that driver to perform such operations as changing the line edit mode, character echoing, signal character recognition, and the like. The stream device driver supports none of these operations. Because the sample CLI always uses stream I/O, no command invoked from it can successfully make any *rq[as]special()* system calls that change terminal or connection attributes. To give a concrete example, *aedit* will not run properly in its interactive mode using the sample CLI because *aedit* needs to control the terminal's character-echoing and line-editing modes. This same restriction applies to the iRMX CLI only in those situations where it uses stream I/O command-line redirection and the *submit* command.

The sample CLI is intended only as a demonstration of the use of stream I/O through the EIOS. It lacks many features of the iRMX CLI, such as command history, aliases, *background* commands, and a *super* mode.

Appendix

D

iRMX System Calls

The following list of iRMX system call names is adapted from Tables 1-1 through 1-7 of the iRMX System Call Reference manual, volume 9 of the iRMX for Windows documentation set. Refer to that manual for complete documentation for each system call. The system calls to access iNA and the Name Server at the end of this list are documented in the *iRMX Network Concepts and Network Programmer's Reference* manual.

The names in this appendix are appropriate for C programs that include the header file `rmxc.h`, for PLM programs, and for assembly language programs. C programs that include the header file `rmx_c.h` use names with embedded underscore characters for easier reading.

C programs that use the networking system calls listed here should include the header file `cqcomm.h`; PLM programs that use networking system calls should include the file `cqcomm.ext`. C programs that use the UDI system calls listed here should include the header file `udi.h` or `udi_c.h`; PLM programs that use UDI system calls should include the file `udi.ext`.

It might be necessary to link to three different libraries to use these system calls. Programs that use system calls with names beginning with *rq* or *rqe* must be linked to the library `rmxifx.lib`, where `x` depends on the model of compilation and word size C, C32, or L. Programs that use UDI calls must link to `udiifx.lib`, and programs that make use network system calls must link to `cqx.lib`.

Application Loader System Calls

rqaload	Loads an object file from secondary storage into memory.
rqaloadiojob	Creates an I/O job with a memory pool of up to 1 Mbyte, loads a specified object

	file, and creates a task to execute the loaded code.
rqealoadiojob	Creates an I/O job with a memory pool of up to 4 GB, loads a specified object file, and creates a task to execute the loaded code.
rqsloadiojob	Loads an object file and creates an I/O job for it. This call is similar to rqesloadiojob; it is provided for compatibility with older versions of the iRMX OS.
rqesloadiojob	Creates an I/O job with a memory pool of up to 4 GB, loads a specified object file, and creates a task to execute the loaded code.
rqsoverlay	Loads an overlay module into memory.

BIOS System Calls

BIOS job-level system calls

rqencrypt	Encrypts a specified string of characters.
rqgetdefaultprefix	Returns the default prefix of a specified job.
rqgetdefaultuser	Returns the default user object of a specified job.
rqsetdefaultprefix	Sets the default prefix for a specified existing job.
rqsetdefaultuser	Sets the default user object for a specified existing job.

BIOS device-level system calls

rqaphysicalattachdevice	Attaches the specified device to the BIOS.
rqaphysicaldetachdevice	Detaches a device that was attached using *rqaphysicalattachdevice( ).*
rqeinstallduibs	Installs a cluster of Device Unit Information Blocks (DUIBs) into the BIOS.
rqaspecial	Enables tasks to perform a variety of device-level functions.

BIOS file/connection-level system calls

rqaattachfile	Creates a connection to an existing file of any type.
rqacreatedirectory	Creates a directory file.
rqacreatefile	Creates a file and returns a token for the new file connection.
rqadeleteconnection	Deletes a file connection created by *rqacreatefile( )*, *rqacreatedirectory( )*, or *rqaattachfile( )*.
rqadeletefile	Marks a stream, named data or named directory file for deletion.

BIOS file-modification system calls

rqachangeaccess	Changes the access rights to a named data or directory file.
rqarenamefile	Changes the pathname of a named data or directory file.
rqatruncate	Truncates a named data file at the current setting of the file pointer.

BIOS file input/output system calls

rqaclose	Closes an open file connection for any type of file.
rqaopen	Opens an asynchronous file connection for I/O operations for any type of file.
rqaread	Reads the requested number of bytes on an open connection for any type of file.
rqaseek	Moves the file pointer of an open file connection.
rqaupdate	Updates a device by writing all buffered partial sectors.
rqwaitio	Returns the concurrent condition code for the prior call to the calling task.
rqawrite	Writes data from the calling task's buffer to a connected physical, stream, or named data file.

BIOS get status/attribute system calls

rqagetconnectionstatus	Returns information about the connection status of a specified file.
rqagetdirectoryentry	Returns the filename associated with an entry number in a named or EDOS directory.
rqagetfilestatus	Returns status and attribute information about a specified file.
rqagetpathcomponent	Returns the name of a data or directory file, as cataloged in its parent directory.

BIOS user object system calls

rqcreateuser	Creates a user object, accepts a list of IDs, and returns a token for the new object.
rqdeleteuser	Deletes a user object.
rqinspectuser	Accepts a token for a user object and returns a list of the IDs contained in the user object.

BIOS extension data system calls

rqagetextensiondata	Writes the extension data for a named data or directory file; not valid for DOS files.
rqasetextensiondata	Stores a named file's extension data; not valid for DOS files.

BIOS time/date system calls

rqgettime	Returns the date/time value from the BIOS's local clock.
rqsettime	Sets the date and time for the BIOS's local clock.
rqgetglobaltime	Reads the time of day from the battery-backed-up hardware clock.
rqsetglobaltime	Sets the battery-backed-up hardware clock to a specified time.

EIOS System Calls

EIOS I/O job calls

rqcreateiojob	Creates an I/O job containing one task with a memory pool of up to 1 Mbyte.

rqecreateiojob	Creates an I/O job containing one task with a memory pool of up to 4 Gbytes.
rqexitiojob	Sends a message to a previously designated mailbox and deletes the calling task.
rqstartiojob	Starts the initial task in an I/O job.

EIOS logical name calls

rqscatalogconnection	Creates a logical name for a connection by cataloging the connection in the object directory of a job.
rqsgetdirectoryentry	Returns a directory entry filename to the caller.
rqsgetpathcomponent	Returns the name of a named file as the file is known in its parent directory.
rqhybriddetachdevice	Temporarily removes the correspondence between a logical name and a physical device.
rqlogicalattachdevice	Assigns a logical name to a physical device.
rqlogicaldetachdevice	Removes the correspondence between a logical name and a physical device, and removes the logical name from the root object directory.
rqslookupconnection	Returns a token for the connection associated with the specified logical name.
rqsuncatalogconnection	Deletes a logical name from the object directory of a job.

EIOS file and connection calls

rqsattachfile	Creates a connection to an existing file.
rqscreatedirectory	Creates a new directory file and automatically adds a new entry to the parent directory.
rqscreatefile	Creates a new physical, stream, or named data file.
rqschangeaccess	Changes the access list for named file.
rqsrenamefile	Changes the pathname of a directory or data file.

rqsclose	Closes an open connection to a named, physical, or stream file.
rqsopen	Opens a file connection.
rqsreadmove	Reads a number of contiguous bytes from a file associated with a connection to a buffer specified by the calling task.
rqsseek	Moves the file pointer for any open physical or named file connection.
rqstruncatefile	Removes information from the end of a named data file.
rqswritemove	Writes a collection of bytes from a buffer to a file.
rqsdeleteconnection	Deletes a file connection, not a device connection.
rqsdeletefile	Deletes a stream, named data, or named directory file created by the BIOS or the EIOS.

EIOS device call

rqsspecial	Allows tasks to communicate with devices, device drivers, and the stream file driver to perform various operations.

EIOS status calls

rqsgetconnectionstatus	Provides status information about file and device connections that were created by the BIOS or the EIOS.
rqsgetfilestatus	Obtains information about a physical, stream, or named file created by the BIOS or the EIOS.
rqgetlogicaldevicestatus	Provides status information about logical names that represent devices.

EIOS user-related calls

rqgetuserids	Returns the user ID(s) associated with a user defined in the User Definition File (UDF).
rqverifyuser	Verifies a user's name and password.

Human Interface System Calls

HI I/O processing calls

rqcgetinputconnection	Returns an EIOS connection object for the specified input file.
rqcgetoutputconnection	Returns an EIOS connection object for the specified output file.

HI command parsing calls

rqcbackupchar	Moves the parsing buffer pointer back one character for each occurrence of the call.
rqcgetchar	Gets a character from the parsing buffer and moves the parsing buffer pointer to the next character.
rqcgetinputpathname	Gets a pathname from the list of input pathnames in the parsing buffer.
rqcgetparameter	Retrieves one parameter from the parsing buffer and moves the parsing pointer to the next parameter.
rqcgetoutputpathname	Gets a pathname from the list of output pathnames in the parsing buffer.
rqcsetparsebuffer	Permits parsing the contents of a buffer other than the command line buffer whenever the parsing system calls are used.
rqcgetcommandname	Obtains the pathname of the command entered by the operator.

HI message processing calls

rqcformatexception	Creates a default message for a given exception code and writes that message into a user-provided string.
rqcsendcoresponse	Sends a message to `:co:` and reads a response from `:ci:`.
rqcsendeoresponse	Sends a message to and reads a response from the operator's terminal.

HI command processing calls

rqccreatecommandconnection	Returns a token for a command connection object required to invoke commands programmatically instead of interactively.
rqcdeletecommandconnection	Deletes a command connection object previously defined in a ccreatecommandconnection call and frees the memory used by the command connection's data structures.
rqcsendcommand	Stores a command line in the command connection created by the ccreatecommandconnection call, concatenates the command line with any others already stored there, and (if the command invocation is complete) invokes the command.

HI program control call

rqcsetcontrolc	Changes the default response to a <Ctrl-C> entry to a response that meets the needs of your task.

Nucleus System Calls

Nucleus job calls

rqcreatejob	Creates a job containing one task with a memory pool of up to 1 Mbyte and returns a token for the job.
rqecreatejob	Creates a job containing one task with a memory pool of up to 4 Gbytes and returns a token for the job.
rqdeletejob	Deletes a specific job.
rqoffspring	Returns a token for the a segment containing tokens of the child jobs of the specified job.
rqeoffspring	Fills the specified data structure with tokens of the child jobs of the specified job.

Nucleus task calls

rqcreatetask	Creates a task and returns a token for it.
rqdeletetask	Deletes a specific, non-interrupt task.
rqgetpriority	Returns the static priority of a specific task.
rqgettasktokens	Returns a token for either itself, its job, its job's parameter object, or the root job.
rqresumetask	Decreases a task's suspension depth by one.
rqsetpriority	Changes the priority of a non-interrupt task.
rqsleep	Places the calling task in the asleep state for a specified amount of time.
rqsuspendtask	Increases a task's suspension depth by one.

Nucleus mailbox calls

rqcreatemailbox	Creates a mailbox and returns a token for it.
rqdeletemailbox	Deletes a specific mailbox.
rqreceivedata	Receives a data message from a data mailbox.
rqreceivemessage	Receives a signal message from an object mailbox.
rqsenddata	Sends a data message of up to 80H characters to a mailbox.
rqsendmessage	Sends a signal object to a mailbox.

Nucleus semaphore calls

rqcreatesemaphore	Creates a semaphore and returns a token for it.
rqdeletesemaphore	Deletes a specific semaphore.
rqreceiveunits	Requests a specific number of units from a semaphore.
rqsendunits	Sends a specific number of units to a semaphore.

Nucleus segment and memory pool calls

rqcreatesegment	Creates a segment and returns a token for it.
rqdeletesegment	Returns a segment to the memory pool from which it was allocated or deletes a descriptor from the Global Descriptor Table (GDT).
rqgetpoolattributes	Returns the memory pool attributes of the calling task's job.
rqegetpoolattrib	Returns the same information as getpoolattributes for any job, plus the amount of memory borrowed and the token of the parent job.
rqgetsize	Returns the size, in bytes, of a segment.
rqsetpoolmin	Sets the minimum attribute of the memory pool of the caller's job.

Nucleus buffer pool calls

rqcreatebufferpool	Creates a buffer pool object that can be associated with one or more ports.
rqdeletebufferpool	Deletes a buffer pool object.
rqreleasebuffer	Returns previously allocated buffer space to the specified buffer pool.
rqrequestbuffer	Gets a buffer from an existing buffer pool.

Nucleus descriptor calls

rqechangedescriptor	Changes the base physical address and size of a descriptor in the GDT.
rqecreatedescriptor	Builds a descriptor for a memory segment, places the descriptor in the GDT, and returns a token for that descriptor.
rqedeletedescriptor	Removes a descriptor entry from the GDT.

Nucleus object calls

rqcatalogobject	Places an entry for an object in an object directory.
rqechangeobjectaccess	Changes the access rights of iRMX segments or composite objects.

rqegetaddress	Returns the physical address of an object.
rqegetobjectaccess	Returns the access type of an object whose token is specified.
rqgettype	Returns the type code for the specified object.
rqlookupobject	Returns a token for the specified cataloged object name.
rquncatalogobject	Removes an entry for an object from an object directory.

Nucleus exception handler calls

rqgetexceptionhandler	Returns the address of the calling task's exception handler and the current value of the task's exception mode.
rqsetexceptionhandler	Assigns an exception handler and exception mode attributes to the calling task.

Nucleus interrupt management calls

rqdisable	Disables a specific interrupt level.
rqenable	Enables a specific interrupt level.
rqendinittask	Informs the root task that a synchronous initialization process has completed. Not available to iRMX For Windows users.
rqenterinterrupt	Sets up a previously-specified data segment base address for the calling interrupt handler.
rqexitinterrupt	Used by interrupt handlers to send an end-of-interrupt to hardware.
rqgetlevel	Returns the interrupt level of the highest priority interrupt that an interrupt handler is currently processing.
rqresetinterrupt	Cancels the assignment of an interrupt handler to a level.
rqsetinterrupt	Assigns an interrupt handler and, if desired, an interrupt task to an interrupt level.
rqsignalinterrupt	Used by interrupt handlers to invoke interrupt tasks.
rqetimedinterrupt	Puts the calling interrupt task to sleep

	until either it is called into service by an interrupt handler or a specified time period elapses.
rqwaitinterrupt	Puts the calling interrupt task to sleep until it is called into service by an interrupt handler.

Nucleus composite object calls

rqaltercomposite	Replaces components of composite objects.
rqcreatecomposite	Creates a composite object and returns a token for it.
rqdeletecomposite	Deletes a composite object but not its component objects.
rqinspectcomposite	Returns a list of the component tokens contained in a composite object.

Nucleus extension object calls

rqcreateextension	Creates a new object type and returns a token for it.
rqdeleteextension	Deletes an extension object and all composites of that type.

Nucleus deletion control call

rqdisabledeletion	Makes an object immune to ordinary deletion.
rqenabledeletion	Makes an object susceptible to ordinary deletion.
rqforcedelete	Deletes objects whose disabling depths are zero or one.

Nucleus OS extension calls

rqesetosextension	Attaches or deletes the entry-point address of a user-written OS extension to a call gate.
rqsetosextension	Supported by iRMX I only. Attaches or deletes the entry-point address of a user-written OS extension to a call gate.
rqsignalexception	Used by OS extensions to signal the occurrence of an exceptional condition.

Nucleus region calls

rqacceptcontrol	Provides access to data protected by a region only if access is immediately available.
rqcreateregion	Creates a region and returns a token for it.
rqdeleteregion	Deletes a specific region.
rqreceivecontrol	Allows the calling task to gain access to data protected by a region.
rqsendcontrol	Relinquishes control to the next task waiting at the region.

Nucleus communication service calls

rqattachbufferpool	Associates a buffer pool with one or more ports.
rqattachport	Forwards all messages sent to the port that issued the call to a sink port.
rqbroadcast	Sends a control message to every message passing host.
rqcancel	Performs synchronous cancellation of RSVP message transmission.
rqconnect	Creates a connection between the sending task and a remote task.
rqcreateport	Creates a port object that can be used in message passing.
rqdeleteport	Deletes a specific port.
rqdetachbufferpool	Ends the association between a buffer pool and a port.
rqdetachport	Ends message forwarding from the source port to the sink port.
rqgethostid	Returns the host ID of the board that the task is running on.
rqgetportattributes	Returns information about the specified port.
rqreceive	Accepts a message at a port.
rqreceivefragment	Accepts a fragment of an RSVP data message.
rqreceivereply	Accepts a message that is a reply to an earlier request.

rqreceivesignal	Receives a signal from a remote host at a specified port.
rqsend	Sends a data message from a port to a port on another board.
rqsendrsvp	Initiates a request/response message exchange.
rqsendreply	Sent in response to the rqsendrsvp system call.
rqsendsignal	Sends a signal message to a remote host through the specified port.

Nucleus multibus II interconnect calls

rqgetinterconnect	Retrieves the contents of the specified interconnect register.
rqsetinterconnect	Alters the contents of an interconnect register to a specified value.

UDI System Calls

UDI program control calls

dqexit	Exits from the current application job.
dqoverlay	Loads an overlay module.
dqtrapcc	Designates an interrupt procedure that takes control when <Ctrl-C> is entered.

UDI file-handling calls

dqattach	Creates a connection to a file.
dqchangeaccess	Changes access rights to a file or directory.
dqchangeextension	Changes the extension of a file name in memory.
dqclose	Closes the specified file connection.
dqcreate	Creates a file.
dqdelete	Deletes a file.
dqdetach	Closes a file and deletes its connection.
dqfileinfo	Returns data about directory and data files.
dqgetconnectionstatus	Returns information about a file connection.

dqopen	Opens a file for a particular type of access.
dqread	Reads bytes from a file.
dqrename	Renames a file.
dqseek	Moves the file pointer of a file.
dqspecial	Sets the mode of a console input device.
dqtruncate	Truncates a file at the position specified by the file pointer.
dqwrite	Writes data to a file.

UDI memory management calls

dqallocate	Requests a memory segment.
dqfree	Returns a memory segment to the system.
dqgetmsize	Returns the size of a segment allocated by dqmallocate.
dqgetsize	Returns the size of a specified segment.
dqmallocate	Requests a logically contiguous memory segment of a specified size.
dqmfree	Returns memory allocated by dqmallocate to the Free Space Pool.
dqreserveiomemory	Sets aside memory for I/O operations.

UDI exception-handling calls

dqdecodeexception	Converts a condition code into its equivalent mnemonic.
dqgetexceptionhandler	Returns the address of the current exception handler.
dqtrapexception	Substitutes an alternate exception handler.

UDI utility and command parsing calls

dqdecodetime	Decodes the specified binary date/time value to ASCII characters.
dqgetargument	Returns an argument from the command line.
dqgetsystemid	Returns the identity of the OS environment.
dqgettime	Obsolete: included for compatibility.

dqswitchbuffer	Selects a new command line buffer.

Windows- and DOS-Specific System Calls

rqereadsegment	Allows a DOS application program to transfer data from a Protected Virtual Address Mode (PVAM) segment to a Real Mode segment.
rqewritesegment	Allows a DOS application program to transfer data from a Real Mode segment to a PVAM segment.
rqesetvm86extension	Installs and removes a Virtual 8086 Mode (VM86) extension at the specified interrupt level.
rqedosrequest	Makes DOS/ROM BIOS requests and other software interrupts handled by DOS applications.

System Calls for Access to iNA and the Name Server

cqcommdwordtoptr	Converts a 32-bit absolute address to the corresponding pointer.
cqcommptrtodword	Converts a pointer to the corresponding 32-bit absolute address.
cqcommrb	Delivers a request block to iNA or to the Name Server for processing.
cqcreatecommuser	Creates a user ID for programmatic access to iNA.

Glossary

8.3 rule The DOS file naming rule that requires an eight-character base part, followed by a period, followed by a three-character extension part.

ACRONYM A Contrived Reduction of Nomenclature Yielding Mnemonics.

activation record A data structure consisting of a procedure's parameter values, return address, and local dynamic variables.

AL Application Loader. The layer of an iRMX system that loads programs into memory for execution, such as when a user enters a command at the console.

ANSI American National Standards Institute.

API Application Programming Interface application task. A task that runs at a priority level that does not disable any interrupt levels.

ASCII American Standard Code for Information Interchange. A 7-bit character code specified in ANSI standard number X3.4.

AT bus The system bus architecture used in most PCs. Also known as the ISA bus.

base address The address in physical or virtual memory at which a memory segment begins. Derived from a selector when a processor is operating in real mode or from a descriptor identified by a selector when a processor is operating in protected mode.

base register A register that is added to an index and a displacement to compute the effective memory address of an operand for a machine language instruction.

binary compatible An executable program that can run on different systems without being recompiled or relinked.

BIOS Basic I/O System. The layer of an iRMX system that provides an asynchronous interface to the input/output system. The same acronym is used in the context of PCs for the code that resides in ROM to supply OS-independent access to the computer's I/O system.

bootstrap loader Unless the operating system resides in ROM, it must be loaded into memory from a disk using a program called a *bootstrap loader*. For iRMX, the disk may be either local or accessed over the network.

buffer pool An iRMX object type managed by the Nucleus layer. Buffer pools provide a way to manage sets of memory segments efficiently.

C start-off routine The initial code for a C program that calls *main()* and which receives control when *main()* returns.

callback A function that is called by another program in response to some event.

CDB Connection Database. iNA maintains information about each virtual circuit that it manages in a data structure called a CDB. It returns an unsigned integer to identify the CDB when a program issues an open Request Block.

cheat To use knowledge about the internal representation of an encapsulated object in an application program.

client On a network, a client is a computer which makes requests for a remote system that acts as a server. For example, a client could request a remote server to supply it with data from one of the server's disk files.

compile time The time at which source code is translated into an object module. If the source code is assembly language rather than a high-level language, the process is called *assembly* rather than *compilation*, and is said to occur at assembly time.

composite object An iRMX object that is made up of other objects. All application-defined object types use composite objects.

condition The name of a synchronization primitive introduced by POSIX.4a.

connection A composite object type managed by the BIOS. Although there is only one connection object type, the internal format of a connection object depends on whether it was created by an attach device or an attach file system call.

consumer Another name for a network client.

conventional memory A DOS term used to refer to the portion of the real mode address space below 640K.

CPU Central Processing Unit. In this book, the same as a microprocessor.

CRC Cyclic Redundancy Check. A parity-based mechanism for checking the integrity of data retrieved from a memory or communication link.

CST Context switch time. The time it takes to get the CPU to stop executing code for one task and start executing code for another one.

CUSP Commonly Used Systems Program.

datagram A message passed over a network on a best-effort basis. The network is not guaranteed to deliver datagrams in the order in which they are sent, if at all. Successful transmission of a datagram does not mean that it was actually received anywhere.

DDE Dynamic Data Exchange. A technique for sharing information between Windows applications, such a fields within word processing documents or cells within spreadsheets. iRMX for Windows extends the DDE to allow applications to communicate over a network.

deadline Time limit on when a task can complete processing an event.

default prefix An I/O connection, normally to a directory, that is used as the starting point for locating a file or directory that does not include a complete path in its name.

descriptor An 8-byte data structure taken from a descriptor table in memory. Descriptors provide about information memory segments, system call procedures, interrupt and trap handling procedures, and hardware-supported tasks.

development system A computer system used to build application software that will be debugged on, and ultimately, integrated with, another computer called the *target system.*

device connection A BIOS connection object that was created by `rqaphysicalattachdevice()` or `rqlogicalattachdevice()`. A device connection identifies the DUIB, and hence the device driver, that will handle I/O requests for a particular device.

device controller The hardware interface between the CPU or system bus and a device unit.

device driver The software interface between the BIOS and a device controller.

device independence A feature of an OS that makes it possible for application programs to do I/O to disk files or different physical devices without having to change the program itself in any way.

device unit A single I/O device, such as a terminal or a disk drive.

DHDT DOS-Hosted Development Tool. A compiler, binder, or other development tool that runs under MS-DOS on a PC.

disk volume A single floppy diskette or hard disk drive that has been formatted with a particular type of file system.

DMA Direct Memory Access. A device controller design in which the processor provides the controller with the address of a memory buffer and receives only a single interrupt from the controller when it has completed the data transfer for the entire buffer.

DOS Disk Operating System. In this book, the MS-DOS or PC-DOS operating system of Microsoft or IBM.

DUIB Device Unit Information Block, also known as a device name or physical device.

EDL External Data Link. The name for the collection of Data Link operations that can be invoked directly by applications using the iNA RB interface.

EDOS Encapsulated DOS. The iRMX file driver for use with MS-DOS disks.

effective address The address of a memory operand referenced by a machine language instruction. Computed as the sum of a displacement value, the contents of an index register, and the contents of a base register.

EIOS Extended I/O System. The layer of an iRMX system that provides a synchronous interface to the BIOS layer.

EISA Extended ISA bus. A system bus architecture for PCs. Competes with MCA.

embedded system Any device or piece of equipment that is controlled by one or more microprocessors. Normally, users interact with an embedded computer only through the instrument's controls, if at all.

encapsulation A characteristic of object-based systems. The representation of an object is hidden from the user of that object. Only the object's type manager can manipulate an object. Users of an object can manipulate the object only by invoking type manager functions for the object.

EOI End of Interrupt. A command that is sent to a Programmable Interrupt Controller to indicate that an interrupt handler has completed processing an interrupt.

event loop A code structure characterized by an endless loop in which a task is idle while it waits for an event, performs some computation when an event occurs, then returns to the waiting state until its next event occurs.

exception handler A program that receives control when an error is detected by the operating system or by the hardware.

exchange Generic name for semaphores, mailboxes, and regions, which are object types managed by the Nucleus that allow tasks to synchronize and communicate with one another.

execution breakpoint An instruction in a program that causes a transfer of control to a debugging program. Implemented by replacing the original machine language instruction at the break point with an interrupt 3 instruction.

exit procedure A procedure called by a system call procedure when it is ready to return to an application that called it. The exit procedure performs the machine language operations needed to set up the condition code and any return value for the system call, and then returns to the interface procedure.

expedited data Virtual circuit data that can be transmitted and received ahead of normal data already en-route over the circuit.

explicit function A function called by name from an application program.

FIFO First in, first out.

file driver Any of five software interfaces to the BIOS that provide a uniform, file-oriented view of I/O devices. The four file drivers are the Named, Physical, Remote, Stream, and EDOS drivers.

flags register A CPU register used to hold status information resulting from arithmetic operations, and some of the bits to control the operating mode of the processor, such as the interrupt enable flag.

FSM Free Space Manager. The type manager provided by the Nucleus for creating and reclaiming memory segment objects.

gather-write An output option that allows several discontinuous parts of memory to supply the data to be written, rather than use a single contiguous buffer.

GDT Global Descriptor Table. A memory segment that contains descriptors for code, data, and descriptor table segments. All processor tasks have access to a common GDT.

GP fault General Protection fault. Protected mode processors generate interrupt number 13 to indicate a GP fault whenever an attempt is made to violate the processor's protection mechanism. For example, invalid program addresses cause GP faults.

hardware task A thread of execution that can be managed directly by the processor with little or no operating system intervention. A Task State Segment (TSS) is used to save and restore the task's processor state for context switches.

help file A file with a name ending in `.hlp` and residing in one of the directories normally searched by the help command.

HI Human Interface. The layer of an iRMX system responsible for interactions with people who use the system.

high-water mark When the rate at which data arrives is too great for a serial device driver to buffer, it sends a signal to the device unit telling it to cease transmission temporarily. The amount of buffered data that triggers this action is called the high-water mark.

HMA A DOS term for the High Memory Area, the part of the real-mode address space between 1M and 1M + 64K.

hot key A special combination of keys that activates a software function that was previously loaded into the computer's memory.

hot link In the context of the Windows DDE mechanism, a hot link causes a DDE server to send the new value of a data item to a DDE client any time the value changes.

I/O connection A composite iRMX object type managed by the BIOS layer. There can be no more than one I/O connection to a device at a time. There can be any number of I/O connections to a file on a device.

I/O job A composite iRMX object type managed by the EIOS layer. Only tasks belonging to I/O jobs are allowed to make EIOS system calls.

I/O port A device controller that can connect to one serial device, such as a terminal, or parallel device, such as a printer.

I/O port address The 16-bit address used to reference registers and buffers in a device controller.

I/O user A composite iRMX object type managed by the BIOS layer. An I/O user object is a list of user ID values that are used for checking access rights to named or remote files.

ICU Interactive Configuration Utility. A tool for generating customized configurations of the iRMX operating system.

IDT Interrupt Descriptor Table. A memory segment containing call, trap, or interrupt gates. Used to vector interrupts to the proper handler when the processor is in protected mode.

IEEE Institute of Electrical and Electronics Engineers, a professional organization that includes the Computer Society (IEEE-CS). The IEEE-CS Technical Committee on Operating Systems (TCOS) is developing the POSIX standard.

implicit function A function that must be bound to an application to satisfy a reference generated by the compiler, but which is not explicitly named in the application's source code.

iNA Intel Network Architecture. The software module that implements transport layer networking using OSI protocols.

incremental bind Binding an application in two or more steps to control the selection of object modules that share common public symbol names.

indirect function A function that is not called by an application program, but from a run-time function that is linked to the application.

interface procedure A procedure that is linked to an application so it can make system calls. The interface procedure sets up the registers for the call, invokes the proper call gate, and checks for errors on return.

interrupt handler Code that is executed in response to a hardware interrupt request without incurring an operating system context switch.

interrupt response time The time between the moment that an I/O controller makes an interrupt request until the CPU starts executing instructions to process that request.

interrupt task A task that is scheduled for execution in response to an interrupt event.

interrupt virtualization An iRMX for Windows configuration parameter set by the `VIE` entry in the `rmx.ini` file. If virtualization is enabled, iRMX manages interrupts and DOS performance may suffer. If interrupt virtualization is disabled, DOS I/O can interfere with iRMX real-time performance.

IOPL I/O Privilege Level. When a processor is operating in protected or VM86 mode, the current IOPL in the flags register and the privilege level of the current code selector determine the response of the processor to I/O instructions.

IORS Input/Output Request Segment or Input/Output Result Segment. The data structure sent to a device driver by the BIOS to initiate an I/O operation. When the operation is complete, the segment is returned to a mailbox that was specified in the BIOS system call that initiated the operation.

IPC Inter-Process Communication. A Unix term for various mechanisms used for information exchange between processes.

iRMK Intel's Real-time Multitasking Kernel. A software module that may be used to implement small real-time systems. Some iRMX implementations use iRMK as the basis for the Nucleus layer and allow applications to make direct calls to some iRMK functions.

iRMX Intel's Real-time Multitasking eXecutive.

IRT Interrupt Response Time. The time that elapses form the moment a device controller signals that an event has occurred until an interrupt handler starts executing in response to the event.

iRUG The Intel Real-Time User's Group. P.O. Box 91130, Portland, Oregon 97291, 800-255-IRUG (800-255-4784). Also manages real-time forum on CompuServe.

ISA Industry Standard Architecture. The name of the system bus used on most PCs. Also known as the AT bus.

ISO International Standards Organization.

ISO-Latin-1 An 8-bit character code that includes ASCII as the first 128 codes and international extensions in the upper 128 codes.

job One of the fundamental iRMX object types. Jobs have memory pools and own objects.

LAN Local Area Network. In this book, a LAN is assumed to be a set of computers connected by an Ethernet cable. A different medium, such as a Token Ring, could be substituted without affecting the concepts involved.

layer A part of the operating system that supplies a set of system calls. A single layer can implement zero, one, or several type managers, and might or might not have a corresponding job.

LDT Local Descriptor Table. A memory segment that contains code and data descriptors. Each processor task has its own LDT.

linear address A virtual memory address. If paging is not being done, the same as a physical memory address. If paging is being done, a linear address consists of three fields: a page directory, page table, and an offset. Linear addresses are generated by the segmentation unit.

link time The time at which an object module is combined with other object modules to produce a load module.

load module The contents of a disk file that is in a suitable format to be loaded into primary memory and executed. iRMX load modules are in a format called Single Task Loadable (STL) format. A load module contains executable code, data constants, information needed to reserve memory for variables and a stack, and initial values for key processor registers. A load module can also contain symbolic information about all or parts of a program for use by an interactive debugging program.

local On a network, the computer on which software is running is called the local computer. The local computer may be either a client or a server, or both.

logical address The combination of a selector and an effective address that are combined by the processor's segmentation unit to produce a linear address.

logical device A composite iRMX object type managed by the EIOS layer. A logical device is an I/O connection to a device and an associated logical name.

logical name A name for a device or file connection. Up to 12 characters long, plus surrounding colons. Logical names (without the colons) and the token for the associated I/O connection object are cataloged in some job's object directory.

low-water mark The point at which a device driver for a serial device controller generates a signal that allows the remote device unit to resume sending data. See also high-water mark.

LSAP Link Service Access Point. The ID number that a Network layer implementation uses to identify itself to the Data Link Layer in an ISO network.

MAC Media Access Control. The 6-byte Ethernet address for a computer node on a network.

mailbox An iRMX object type managed by the Nucleus layer. Mailboxes are used for intertask synchronization and communication.

MCA Microchannel Architecture. A proprietary system bus used in some IBM personal computers.

memory segment (1) The architectural device used by x86 processors to access memory. Each segment is identified by a 16-bit value called a *selector*. In real mode, the value of the selector is multiplied by 16 to give the base address of the segment in memory. In protected mode, the selector contains an index into a table of descriptors which specify the segments base address, size, and access permissions. (2) A fundamental iRMX object type managed by the Free Space Manager part of the Nucleus layer. Implemented as hardware memory segments.

message port A composite iRMX object type managed by the Nucleus layer. Used for interprocessor communication in Multibus II systems.

MIP Message Interprocessing Protocol. Originally a Multibus I standard for communication among processors over the Multibus I system bus, now used in several configurations for message passing between iNA and applications.

mnemonics Mellifluous Notation Enabling Mastery Of Nomenclature in Concatenate Sequence.

monitor A program used for debugging at the machine language level. A monitor is often kept in ROM, but for iRMX for Windows, the iSDM III monitor is loaded with the OS.

Multibus I, II System bus architectures. Multibus I uses a 16-bit data bus; Multibus II uses a 32-bit data bus. iRMX II and III support message passing on Multibus II systems.

multicast filtering The selection of network packets to be accepted by the network device controller or by the Data Link software layer.

mutex A mechanism for guaranteeing mutually exclusive access to a shared resource by a set of cooperating tasks. The iRMX semaphore and region object types can be used as mutex mechanisms. Also, the name of a specific mechanism for this purpose defined in POSIX.4a.

Named The file driver used to access disk volumes that have been formatted with an iRMX file system.

NCB Network Control Block. The data structure, analogous to iNA Request Blocks, used to invoke DOS networking functions using the NetBIOS interface.

NMF Network Management Facility. An iNA module that supplies statistics, parameters, and data maintained internally by iNA. The information can be examined, and in some cases altered as well.

NSAP Network Service Access Point. The ID number a Transport layer module uses to identify itself to the Network layer in an ISO network.

object An instance of a data structure that can be accessed only through a set of functions provided by a type manager.

object directory A data structure that is part of a job object. Object directories are used for sharing objects across jobs or tasks. An object directory is a list of tokens and names for those tokens. The names are normally made of ASCII characters.

object module The contents of a disk file produced by the compilation of a single source module. In addition to binary machine instructions and data constants, an object module contains information about external and public symbols as well as

memory requirements for variables and stack. The object module can also include symbolic information (symbol names, data types, and locations) to be passed on to a symbolic debugger.

octet Eight bits. This term is used by the networking community instead of byte because there are some machines that define bytes in sizes other than eight.

offset A 16-bit quantity (32-bits for the i386 architecture) that is added to a segment base address to obtain a physical or virtual memory address. The offset can be part of a pointer or can be the effective address for a machine language instruction.

OMF Object Module Format. The specification for how information is to be stored in object modules, libraries, load modules, and bootstrap-loadable files.

operating system extension An object type added to the system by a user-supplied or iRMX-supplied layer.

OS Operating System.

OS/2 Registered trademark of International Business Machines Corporation.

OSE Operating System Extension. An iRMX object type managed by the Nucleus layer. OSEs are used to create new object types.

OSI Open Systems Interconnection. The ISO seven-layer reference model for network communication.

out-of-band data The Internet protocol term for out-of-band data.

pathname The unique identification of a particular file on a file system that is given by naming all the directories, starting with the root, that must be traversed to locate the file.

PC Personal Computer. An IBM trademark, but used generically in this book.

peer-to-peer Networking in which computers can act as both servers and consumers (clients), rather than being dedicated to one or the other class of operation.

physical The file driver used to access disks on a block-by-block basis (rather than as a file system), tapes, terminals, and other serial devices.

physical address If paging is not operating, a physical address is the same as the linear address generated by the segmentation unit. If paging is operating, the physical address is the linear address transformed by the paging unit.

physical memory The primary memory accessed by the CPU when it accesses machine instructions and operands.

PIC Programmable Interrupt Controller. An integrated circuit that can resolve simultaneous interrupt requests and negotiate a sequence of CPU interrupt requests based on the priorities of the requesting device controllers.

PME VM86 Protected Mode Extension. DOS programs can trigger the execution of an iRMX task by issuing a software interrupt instruction. The procedure to be executed by the iRMX task is defined using the PME mechanism.

portable The ability of a single program to run on different computer systems. The term is used to describe a variety of programs, ranging from those that require

"minor" changes to the source code when moving from one system to another, to those that require no changes to the executable code, even to run on systems with different processor architectures.

POSIX Portable Operating System Interface for Computer Environments. A set of Unix-based standards being developed by the IEEE Computer Society's Technical Committee on Operating Systems.

POSIX.1 The IEEE Standards committee that defined the C Language API to POSIX systems.

POSIX.16 The IEEE Standards committee concerned with multiprocessing extensions to POSIX.

POSIX.4 The IEEE Standards committee concerned with real-time extensions to POSIX.

POSIX.4a The IEEE Standards committee concerned with the threads extension to POSIX.

primary memory The memory that holds programs as they are being executed by the CPU. Can be either RAM (volatile, read-write memory) or ROM (nonvolatile, read-only memory).

priority inversion A situation in which a high-priority task is effectively prevented from running by a lower-priority task even though the lower-priority task has nothing to do with the higher-priority task. iRMX region objects eliminate this problem.

process An independently scheduled thread of execution. In iRMX, a task. In Unix, a thread of execution plus an address space.

protected mode A mode of operation of 80286 and later microprocessors in which the processor checks the legitimacy of memory accesses.

PSP Program Segment Prefix. The data structure used by DOS to store information about a command that is loaded into memory.

PVAM See VM86.

RAM Random Access Memory. Strictly speaking, any type of memory in which access time does not depend on the position (address) of the data to be read or written. Common usage reserves RAM for memory that can be both read and written and which loses its stored information when electrical power is removed from the circuit. (See ROM.)

RAWEDL Raw External Data Link. The Request Block interface to the Data Link layer of iNA that allows an application to receive and send packets that do not conform to ISO standards.

RB Request Block. The data structure that is used for communicating network requests to iNA and receiving results in return.

real mode An operating mode available on all x86 processors. No memory access checking is done by the processor.

region An iRMX object type managed by the Nucleus layer. Regions are used to prevent priority inversions that might occur if semaphores are used for intertask synchronization.

Remote (1) The file driver that is used to access files or devices located on a remote computer. (2) Any computer that is accessed over a network. The remote computer can act as either a client, a server, or both.

RHDT iRMX-Hosted Development Tools. Development Tools that run on iRMX. See DHDT.

RISC/CISC Reduced Instruction Set Computer / Complex Instruction Set Computer. Two styles of computer architecture. RISC computers use many simple but fast instructions to accomplish the same work as CISC computers do with a few complex, but relatively slow, instructions. iRMX runs on Intel's CISC processors, using the x86 architecture.

ROM Read Only Memory. A type of random access memory circuit that can be read, but not written. An important feature of ROM is that it retains the information stored in it even after electrical power is removed from the circuit.

RPC Remote Procedure Call. A mechanism that allows a task on one computer to invoke a system call (or other procedure) on another computer by passing an appropriate request over a network.

RTE Real-Time Extension. DOS programs can make certain iRMX system calls by using the iRMX for Windows RTE mechanism.

run time The time at which a program is executed by the CPU. The code must first be compiled, linked, and loaded.

run-time library A library of object modules supplied with a high-level programming language. Functions in the library can be called by an application explicitly, such as the `printif( )` function for C, implicitly, such as doubleword multiplication and division for PLM and C, or indirectly, such as `cq_exit` for C.

scatter-read An input option in which arriving data is distributed to several disjointed parts of memory instead of into a single contiguous buffer.

segment A contiguous region of memory, up to 64 Kilobytes (KB) for 80286 processors and lower, but up to 4 Gigabytes (GB) for i386 processors and higher. Also, a type of iRMX object which provides unstructured access to memory segments.

segment descriptor An 8-byte data structure that provides the physical base address, the size, and the access rights for a segment.

selector A 16-bit value used to identify a memory segment.

semaphore An iRMX object type managed by the Nucleus layer. Semaphores are used for intertask synchronization.

server On a network, a server is any computer which responds to requests from remote systems. A file or print server allows remote systems to access local disks or printers, for example.

SIL System Implementation Language. A high-level language suitable for coding a systems program or an operating system itself. PLM is a SIL, C can be used as a SIL, Pascal is not a SIL.

SMB Server Message Block. A format for messages used to communicate between network servers and clients. The message formats were defined by Microsoft and are used by OpenNet software.

source module The unit of compilation. For iC-x86 and PLM-x86, one disk file contains the ASCII text for one source module plus optional compiler directives. One of these directives, "include," allows additional text to be inserted into the source module from other disk files at compile time.

spawn The phrase "all jobs spawned by a job" is used to refer to both the direct children of the job as well as their descendants.

stack frame An activation record.

static login A terminal that automatically comes up with a particular user already logged in when the system starts.

STL Single Task Loadable. The object module format (OMF) that is used for programs loaded by the iRMX Application Loader.

stream The file driver that provides access to a pseudo-I/O device. Used for intertask communication by means of I/O system calls.

system call A subroutine supplied by the OS to provide some service, such as I/O processing or memory allocation.

system definition file A file that can be edited by the ICU editor to build a specification for a customized configuration of iRMX. Once the definition file is complete, the ICU uses it to generate the files needed to build the new copy of the operating system.

target system A computer that runs the iRMX OS and real-time applications, as distinguished from a development system.

task An independently scheduled thread of execution. Same as a *process* in the general OS literature. One of the fundamental object types in iRMX.

token A unique identifier for an object. In iRMX systems, a selector for the memory segment containing the representation of the object.

TPDU Transport Protocol Data Unit. The size in bytes of the largest packet that can be sent over the network by the Transport layer. Single messages larger than the TPDU size are broken up by the Transport layer, sent over the net, and reassembled at the other end.

TSAP Transport Service Access Point. A number, sometimes called a *selector*, used to identify an individual application to the Transport layer of an ISO network.

TSR A DOS program that terminates (allows DOS to return to the command interpreter), but stays resident in memory so that it can provide services for other DOS programs or respond to interrupts. The equivalent iRMX mechanism uses the `sysload` command to install a program.

type manager The set of procedures for manipulating objects belonging to a particular object type.

UDI Universal Development Interface. A set of system calls that can be invoked by programs running on different operating systems, including iRMX, MS-DOS, and VAX/VMS.

UNIX Registered trademark of UNIX Systems Laboratories. Not an acronym. Also known as Unix.

upper memory A DOS term referring to the real-mode address space between 640K and 1M.

user ID A 16-bit number used to identify an individual or group of users on iRMX systems. User ID 0x0000 is the Super user, who can read all files and change their permissions. User ID 0xFFFF is the World user, which is a member of every individual's I/O user object by default.

virtual circuit A network connection managed by Transport layer software that provides reliable end-to-end sequential transmission of data.

virtual memory Primary memory as addressed by machine language instructions but subject to mapping into physical memory by paging hardware in the 80386 architecture. For iRMX, which does not use paging, virtual memory is the same as physical memory.

virtual root The list of public directories and devices offered to the network by a computer system.

VM86 Virtual 8086 Mode. A mode of operation for the i386 and later microprocessors in which a hardware task seems to be running in real mode.

VM86 dispatcher The iRMX for Windows code that receives control when a VM86 task causes a software interrupt, issues an I/O instruction, or performs any operation that the processor reserves for protected mode operations. The VM86 dispatcher can ignore the event, emulate it, or invoke the VM86 task's interrupt handler, as appropriate.

volume block The smallest amount of storage that can be allocated or freed on a disk. Equivalent to the volume's granularity, the size of a volume block is always a multiple of the size of a disk sector.

volume granularity The smallest number of bytes that can be read from or written to a disk volume at one time. This value is often equal to the sector size of the disk but it can be made to be a multiple of the disk sector size when the disk is formatted.

VT Virtual Terminal. A mechanism for supporting remote login to a computer using OpenNET.

warm link In the context of the Windows DDE mechanism, a warm link causes a DDE server to notify a DDE client any time the value of a data item changes.

x3.64 The ANSI standard for extended terminal control codes based on ASCII, which in turn is ANSI standard number X3.4.

x86 Generic name for a family of microprocessors that are upwardly compatible with the Intel 8086 CPU. These include the 8086, 80186, 80286, i386, and i486. Also means "either 86, 286, or 386" in names of programs for which there are versions specific to different architectures, such as "iC-x86."

References

Andleigh, P. K. 1990. *UNIX System Architecture.* Englewood Cliffs, NJ: Prentice-Hall.

Bach, M. J. 1986. *The Design of the UNIX Operating System.* Englewood Cliffs, NJ: Prentice-Hall.

Comer, D. E. 1988. *Internetworking with TCP/IP: Principles, Protocols, and Architecture.* Englewood Cliffs, NJ: Prentice-Hall.

Comer, D. E. and Fossom, T. 1988. *Operating System Design. Vol. I: The Xinu Approach.* Englewood Cliffs, NJ: Prentice-Hall.

Deitel, H. M. 1990. *Operating Systems, 2nd Ed.* Reading, MA: Addison-Wesley.

DPMI Committee. 1991. *DOS Protected Mode Interface (DPMI) Specification Version 1.0.* Order code 240977-001. Santa Clara, CA: Intel Corporation.

Harbison, S. P. and Steele, G. L. 1991. *C: A Reference Manual, Third Edition.* Englewood Cliffs, NJ: Prentice-Hall.

IEEE Computer Scociety TCOS. 1990. *Information Technology: Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language].* New York, NY: IEEE.

Intel Corportation. 1991. *iNA 960 Programmer's Reference Manual.* Order code 467686-001. Santa Clara, CA: Intel Corporation.[1]

Intel Corporation. 1991. *iRMX-Net User's Reference.* Order code 467727-001. Santa Clara, CA: Intel Corporation.[2]

Kernighan, B. and Pike, R. 1984. *The UNIX Programming Environment.* Englewood Cliffs, NJ: Prentice-Hall.

Kernighan, B. and Ritchie, D. 1978. *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall.

Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Reading, MA: Addison-Wesley.

Lotus Development Corporation, Intel Corporation, Microsoft Corporation. 1987. *Lotus/Intel/Microsoft Expanded Memory Specification Version 4.0.* Order code 300275-005. Santa Clara, CA: Intel Corporation.

Microsoft Corporation. 1991. *eXtended Memory Specification (XMS): Version 3.0..* Redmond, WA: Microsoft Corporation.

Milenkovic, M. 1992. *Operating Systems Concepts and Design, 2nd Ed.* New York, NY: McGraw-Hill.

Petzold, C. 1992. *Programming Windows, 3rd Edition.* Redmond, WA: Microsoft Press.

Plauger, P. J. 1992. *The Standard C Library.* Englewood Cliffs, NJ: Prentice-Hall.

Rochkind, M. J. 1985. *Advanced UNIX Programming.* Englewood Cliffs, NJ: Prentice-Hall.

Rose, M. T. 1990. *The Open Book.* Englewood Cliffs, NJ: Prentice-Hall.

Snow, C. R. 1992. *Concurrent Programming.* Cambridge, MA: Cambridge University Press.

Stevens, W. R. 1990. *Unix Network Programming.* Englewood Cliffs, NJ: Prentice-Hall.

---

[1] A replacement for this manual is available in *iRMX Network Concepts and Network Programmer's Reference, Order Number 610489-001.*

[2] A replacement for this manual is available in *iRMX Network Concepts and Network Programmer's Reference,* Order Number 610489-001.

Stone, H. S. 1987. *High Performance Computer Architecture.* Reading, MA: Addison-Wesley.
Tanenbaum, A. *Operating Systems Design and Implementation.* Englewood Cliffs: Prentice-Hall, 1987.

Journal Articles

Feriozi, D. T. 1991. A C programming model for OS/2 device drivers. *IBM Systems Journal* 30 (3): 322–335.
Finlayson, R. S. 1991. Object-oriented operating systems. *IEEE TCOS Newsletter* 5 (1): 17–21.
Gonzalez, M. J. 1977. Deterministic Processor Scheduling. *ACM Computing Surveys* 9 (3): 173–204.
Heller, M. 1990. Programming 32-bit OS/2. *BYTE* 15 (11): 97–104.
Kogan, M. S. and Rawson, F. L. III. 1988. The Design of Operating System/2. *IBM Systems Journal* 27 (2): 90–104.
Liu, J. W. S. and Yang, A. T. 1973. Optimal Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* 20 (1): 46–61.
Mastriani, S. J. 1991. OS/2 device drivers. *BYTE* 16 (7): 241–350.
Vickery, C. 1991. Loadable device drivers for iRMX. *iRUG Newsletter* 2: 1–3.

Conference Proceedings

Vickery, C. 1990. Real-Time X Windows: the BSD Socket Mechanism. Proceedings of the Seventh International iRUG Conference, St. Louis, MO.
Wegner, P. 1987. Dimensions of object-based language design. OOPSLA Conference Proceedings. Cited in Finlayson, 1991.

Index

$29.95 U.S.A.

Programming Languages

Real-time systems programming made easy

REAL-TIME AND SYSTEMS PROGRAMMING FOR PCS

Real-time applications programming demands the same set of skills traditionally required of systems programmers, namely careful management of memory allocation, of concurrent threads of execution, of network links, and of I/O devices. Whether you want to use a PC to build high-performance real-time systems or to learn first-hand how to do real-world systems programming, this book will serve as an invaluable reference.

Now, Intel's official guide to the iRMX® for Windows operating system explains clearly and concisely all the ins and outs of iRMX for Windows, which runs iRMX software concurrently with DOS or Windows software on a single PC. You'll learn how to perform common real-time and systems programming operations, including resource allocation, concurrency management, and I/O transfers. You will also find everything you need to know about device drivers, OS extensions, type managers, network programming, and DDE communication between iRMX and Windows applications.

All the information you need to use the iRMX system effectively and efficiently is contained in this one, easy-to-use guide. The book is equally valuable for use as the text for a university-level laboratory course in real-time or systems programming and as a self-study reference for programmers and engineers designing critical real-time systems.

Cover Art: Dan Mandish

ISBN 0-07-067466-3

9 780070 674660 90000

McGraw-Hill, Inc.
Serving the Need for Knowledge
1221 Avenue of the Americas
New York, NY 10020