INTRODUCTION TO THE iRMX 86™ OPERATING SYSTEM

Manual Order Number: 9803124-02

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

PUBLICATION NOTICE

This document is the second edition of the INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM. The manual reflects the software associated with Release 2.0 of the iRMX 86 Operating System. This edition applies to all future software releases until further notice.

PRINTING HISTORY:

Manual Edition:	Software Release:	Print Date:
First	1.0	April, 1980
Second	2.0	November, 1980

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and may be used only to identify Intel products:

BXP	Intel	Megachassis
CREDIT	Intelevision	Micromap
i	Intellec	Multibus
iCE	iRMX	MULTIMODULE
iCS	iSBC	PROMPT
im	iSBX	Promware
Insite	Library Manager	RMX/80
Intel	MCS	UPI
		μScope

and the combination of ICE, iCS, iRMX, iSBC, iSBX, or MCS and a numerical suffix.

A270/1080/3K DD

PREFACE

If you are looking for a high-level introduction to the iRMX 86 Operating System, this manual will satisfy you. By reading this manual, you will acquire sufficient knowledge of the iRMX 86 Operating System to:

- See how the iRMX 86 Operating System can help you develop your application system in less time and at less expense.
- Begin reading the more detailed iRMX 86 manuals.

This manual, which is written for engineers and managers, is designed to be read completely in one or two sittings. It presents information starting with the most general and familiar terms which it then uses to define specific and new terms.

Throughout this manual, the expression "iAPX 86-based microcomputer" is used to refer to any microcomputer that uses the Intel iAPX 86 microprocessor as its central processing unit.

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

PAGE

ILLUSTRATIONS

TABLES

CHAPTER 1. OVERVIEW OF THE iRMX 86™ OPERATING SYSTEM

The iRMX 86 Operating System is a software package designed for use with Intel's iSBC 86 Single Board Computers and with other iAPX 86-based computers.

The iRMX 86 Operating System is different from many other systems in that it is not turnkey. Rather, it is specifically designed to be incorporated in the products that you build.

The iRMX 86 Operating System consists of a collection of subsystems, each of which provides one or more features that can be used in your product. Based on the features that you need to build your product, you decide which subsystems you want. You then combine these subsystems to form a tailored operating system that precisely meets your needs.

MAJOR CHARACTERISTICS OF THE iRMX 86 SYSTEM

The iRMX 86 Operating System exhibits the following characteristics:

- It can simultaneously monitor and control unrelated events occuring outside the single board computer.
- It can communicate with a wide variety of input, output, and mass storage devices.
- It provides a powerful and flexible means for an operator to observe and modify the behavior of the system.

These characteristics (especially when combined with features discussed in Chapter 4) make the iRMX 86 Operating System an excellent foundation for your software-based products (Figure 1-1).

CUSTOMERS OF THE iRMX 86 OPERATING SYSTEM

The iRMX 86 Operating System is designed for two types of customers. Original Equipment Manufacturers, OEMs, are companies that build products for resale. Volume End Users, VEUs, are companies that build products for use within their organization. Both types of customers can produce products more quickly and at less expense by using the iRMX 86 Operating System.

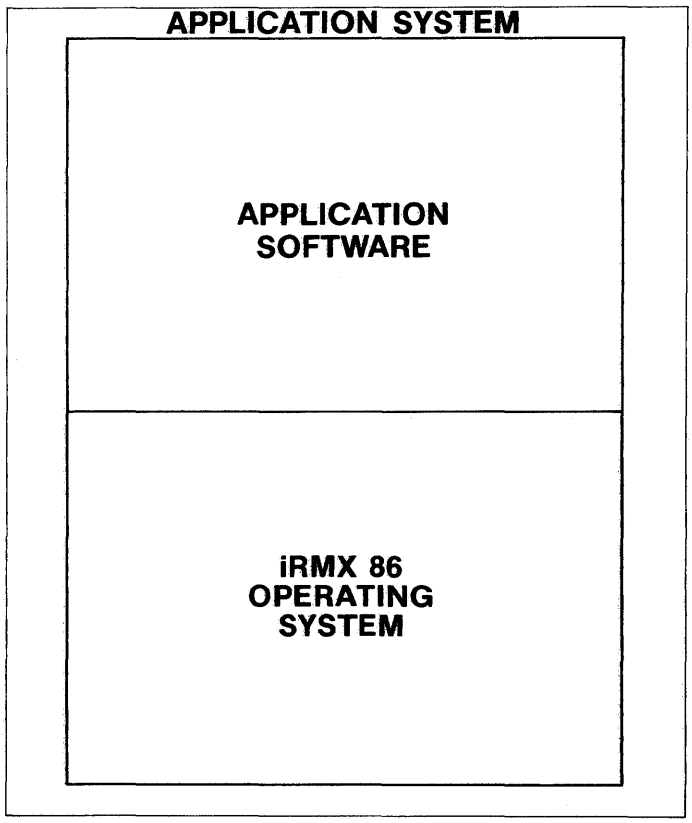

Figure 1-1. The iRMX 86™ foundation for application systems

COMMONLY USED iRMX 86 TERMINOLOGY

The following terms are used frequently in this book:

- Application

  An application is the problem that you solve with your product.

- Application System

  An application system is the product that satisfies the requirements of the application (Figure 1-1).

- Application Software

  The application software is all the software you must add to the iRMX 86 Operating System in order to complete your application system (Figure 1-1).

- User

  The user is the individual or organization who uses your application system.

PURPOSE OF THE iRMX 86 OPERATING SYSTEM

The iRMX 86 Operating System is your shortcut to the marketplace. By supplying you with features that can be used in a large number of application systems, the iRMX 86 Operating System allows you to focus your attention on the specialized application software. Since you spend less time and effort developing sophisticated system software, you can bring your application system to market faster and at a lower price.

ORGANIZATION OF THIS MANUAL

This manual is divided into six chapters. Some of the chapters are designed for managers, some for engineers, and others for both. The following paragraphs identify the audience and purpose of each chapter.

- Chapter 1 - Overview of the iRMX 86 Operating System

  Chapter 1 provides managers and engineers with a very brief introduction to the iRMX 86 Operating System. It provides vocabulary needed in later chapters and contains a statement of purpose for each chapter in this manual.

- Chapter 2 - Considerations Relating to Realtime Software

  Chapter 2 introduces engineers to some of the obstacles that the iRMX 86 Operating System can eliminate. Managers who have had programming experience may want to read this short chapter.

- Chapter 3 - Benefits of the iRMX 86 Operating System

  Chapter 3 provides managers with a discussion of the economic benefits of using the iRMX 86 Operating System. Interested engineers may also want to read this short chapter.

- Chapter 4 - Features of the iRMX 86 Operating System

  Chapter 4 is a tutorial for engineers. It discusses the features of the iRMX 86 Operating System and, at the same time, it defines the vocabulary used in the other iRMX 86 manuals. Engineers who are already proficient at realtime, multitasking programming need only skim this chapter to ascertain the features of the iRMX 86 Operating System.

- Chapter 5 - A Hypothetical System

  Chapter 5 is designed primarily for engineers. It describes a relatively simple application system. The purpose of this chapter is to illustrate the use of some of the features discussed in Chapter 4.

- Chapter 6 - iRMX 86 Literature

  Chapter 6 contains a description of the other manuals associated with the iRMX 86 Operating System. This chapter is designed for engineers who need information more detailed than that provided by this introductory manual.

CHAPTER 2. CONSIDERATIONS RELATING TO REALTIME SOFTWARE

The kinds of difficulties encountered in realtime programming differ significantly from those found in other aspects of programming. This chapter briefly introduces some of the problems that face designers of realtime systems.

The purpose of this chapter is not to discourage you from building a realtime application system. Rather, its purpose is to show you the kinds of hurdles that the iRMX 86 Operating System can help you jump. Consequently, this chapter only poses questions -- it provides no answers. You can find the answers in the discussion of iRMX 86 features in Chapter 4 of this manual.

EVENT DETECTION

Realtime application systems monitor events in the real world. These events occur asynchronously, that is, at seemingly random intervals. When an event occurs, the system could be in the midst of processing information associated with a previous event. Even so, the system must be able to detect and record the occurrence of the second event.

SCHEDULING OF PROCESSING

Assuming that the system can detect and record the occurrence of an event, it still must decide in what order to process recorded events. For that matter, when the system is processing a relatively unimportant event and a critical event occurs, the system must be able to respond correctly. It must be able to postpone the processing of the less significant event until the more important one has been processed. Then, after the higher-priority processing, the system must resume where it left off.

ERROR PROCESSING

Suppose that during the processing of realtime events, an error is detected. How can the error be corrected, or how can its impact be limited, without adversely affecting the running of the system? The whole system, for instance, should not be shut down merely because an error is detected.

DEVICE SENSITIVITY

Many realtime applications use one or more input or output devices. And sometimes, the devices associated with an application system must be changed. By allowing devices to be changed without requiring recompilation, the operating system can save much time and effort.

DEVICE SELECTION

What kinds of devices should an operating system support? Can it handle line printers? Disks? Bubble memories? Tapes?

MASS STORAGE FILE ALLOCATION TRADEOFFS

In any realtime system, performance is an important consideration. One decision that relates directly to performance must be made before formating mass storage files. In some applications, large granularity (large amounts of information located contiguously) results in much faster retrieval. In other applications, large granularity does not improve performance, but does waste space on the device. The operating system must contend with the tradeoff between performance and optimal use of space on the device.

UNNEEDED FEATURES

Some OEM and VEU applications require features that other applications do not. An operating system should provide a means of selecting required features and rejecting unneeded features.

MULTIPLE APPLICATIONS

Sometimes there is a need to run more than one application on the same computer. Several applications might need to share some resources, such as hardware and perhaps some files, while reserving other resources for themselves.

MEMORY REQUIREMENTS

The memory requirements of some applications change according to the events that occur in the real world. If a system can share memory between applications, then the total amount of memory required for the system might be less then the sum of the maximum amounts required by each application.

FILES AND MULTIPLE USERS

Some applications, such as key-to-disk and database-management systems, support more than one user. In such systems, two problems relate to mass storage files.

The first problem pertains to file naming. The users must be able to name files without concern for duplicate names. If they cannot, each user may be forced to guess at names that have not yet been assigned by other users.

The second problem deals with selective sharing of files. Multiuser systems often must be able to share and protect files. For instance, in a key-to-disk system, one operator may be entering data while another simultaneously verifies. This illustrates the need for sharing a file. Now suppose that the file contains confidential information. Once verified, the file must be protected against unauthorized reading and writing. This illustrates the need for restricting access. The system must provide for both sharing and restricted access.

DEBUGGING

Virtually all software, no matter how carefully checked out by manual inspection, contains some bugs. Usually, these bugs are detected by using the system until an error occurs. Once the error is found, an engineer begins tracing backwards from the error to the bug that caused it. When the bug is identified, the engineer modifies the software and eliminates the bug. This process, called debugging, is repeated until no more errors are found.

This debugging process is not always straightforward in realtime systems. Often, bugs in realtime systems are dependent upon events in the real world (outside of the computer). In order to detect some realtime bugs, the system must continue to run even while it is being debugged.

CHAPTER PERSPECTIVE

If the foregoing considerations pertain to your application, then the iRMX 86 Operating System can save you an enormous amount of effort. To see how the iRMX 86 System resolves these and other similar problems, read Chapter 4.

CHAPTER 3. BENEFITS OF THE iRMX 86™ OPERATING SYSTEM

You are reading this manual because you are planning to develop a realtime application system. As an OEM or a VEU, you are interested in developing your application system quickly while still holding down the cost of development. Furthermore, you want to minimize your costs after development. By serving as a foundation for your application software (Figure 3-1), the iRMX 86 Operating System can help you meet your objectives.

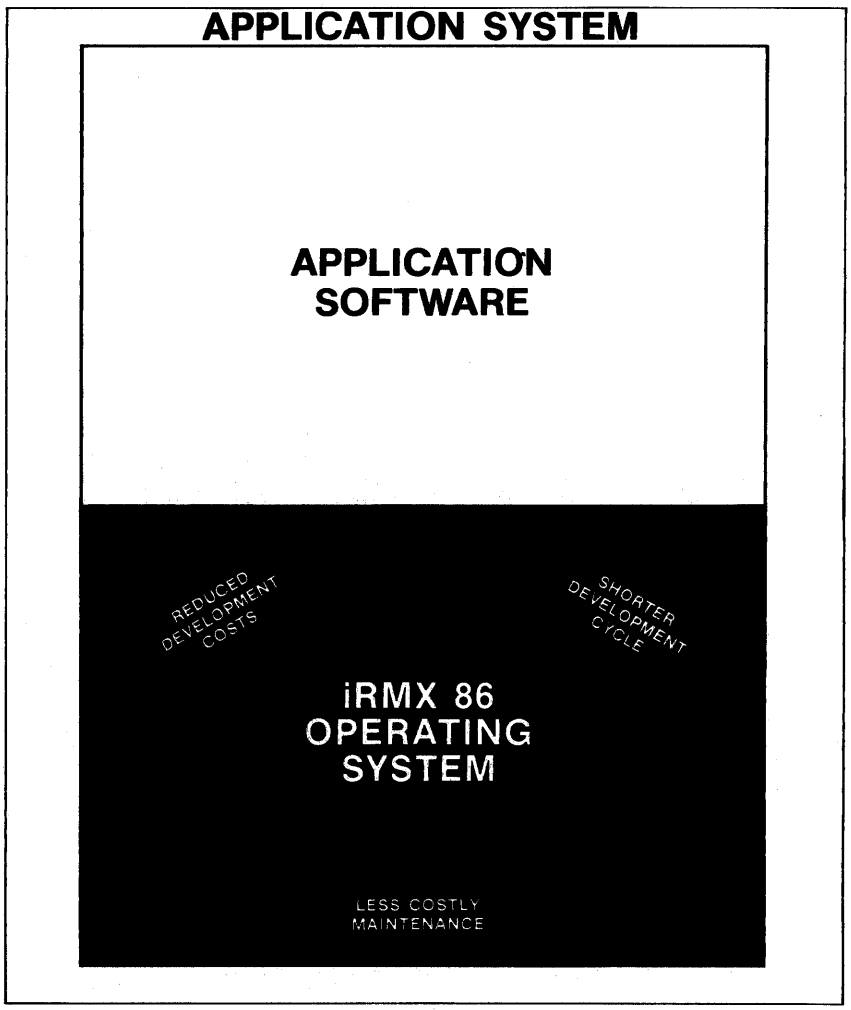

Figure 3-1. The iRMX 86 System provides economic benefits

DEVELOPMENT TIME

The iRMX 86 Operating System helps you develop realtime application systems quickly. Acting as the foundation for your specialized application software, the iRMX 86 Operating System provides services that are required by many realtime applications. Since these services are supplied by the iRMX 86 Operating System, your application engineers spend no time writing software to manage multitasking, dynamic memory allocation, and other functions vital to many realtime applications. Rather, your engineers concentrate their efforts on the software that relates specifically to the application being solved. This greatly reduces the time needed to develop your application system.

COST OF IMPLEMENTATION

The iRMX 86 Operating System helps reduce the cost of implementation in two ways. First, by supplying the general services required by many realtime applications, the iRMX 86 System reduces your manpower requirements. Second, the features of the Operating System simplify the process of development. These features, such as object-oriented architecture, device independence, and others, are discussed in Chapter 4.

COSTS AFTER DEVELOPMENT

After your application system is developed, your major expense is maintenance -- the process of removing bugs, making changes, and adding features. The iRMX 86 Operating System helps minimize these costs.

First, a number of features of the iRMX 86 Operating System smooth the process of system design, reducing the probability of major design errors. These features, which include multitasking and multiprogramming, are described in Chapter 4.

Second, when errors do reveal the presence of bugs in your application software, the iRMX 86 System provides features to help find the bugs. These features include error handlers and an object-oriented, realtime debugger.

Third, the modularity provided by multiple jobs and tasks lets you make changes and additions without severely affecting the system's overall design.

BENEFITS OF THE iRMX 86™ OPERATING SYSTEM

CHAPTER PERSPECTIVE

The iRMX 86 Operating System is your economic ally. It helps you put your realtime application system in the hands of your users in less time and at less expense. It also reduces your maintenance costs after your system is developed.

CHAPTER 4. FEATURES OF THE iRMX 86™ OPERATING SYSTEM

This chapter provides you with moderately detailed descriptions of the following features (Figure 4-1) of the iRMX 86 Operating System:

- Object-Oriented Architecture
- Multitasking
- Interrupt Processing
- Preemptive Priority-Based Scheduling
- Multiprogramming
- Error Handling
- Dynamic Memory Allocation
- Intertask Coordination
- Runtime Binding
- Extendibility
- Terminal Handler
- Application Loading
- Device-Independent Input and Output
- Hierarchical Naming of Mass Storage Files
- File Access Control
- Control over File Fragmentation
- Selection of Device Drivers
- Object-Oriented Debugger
- Bootstrap Loading
- Configurability

Each section of this chapter deals with one of these features and, in case you are already familiar with some features, each section is organized for easy skimming. The first few sentences of each section provide a summary of the feature's value. The feature is then described in moderate detail and, near the end of each section, the advantages of the feature are discussed.

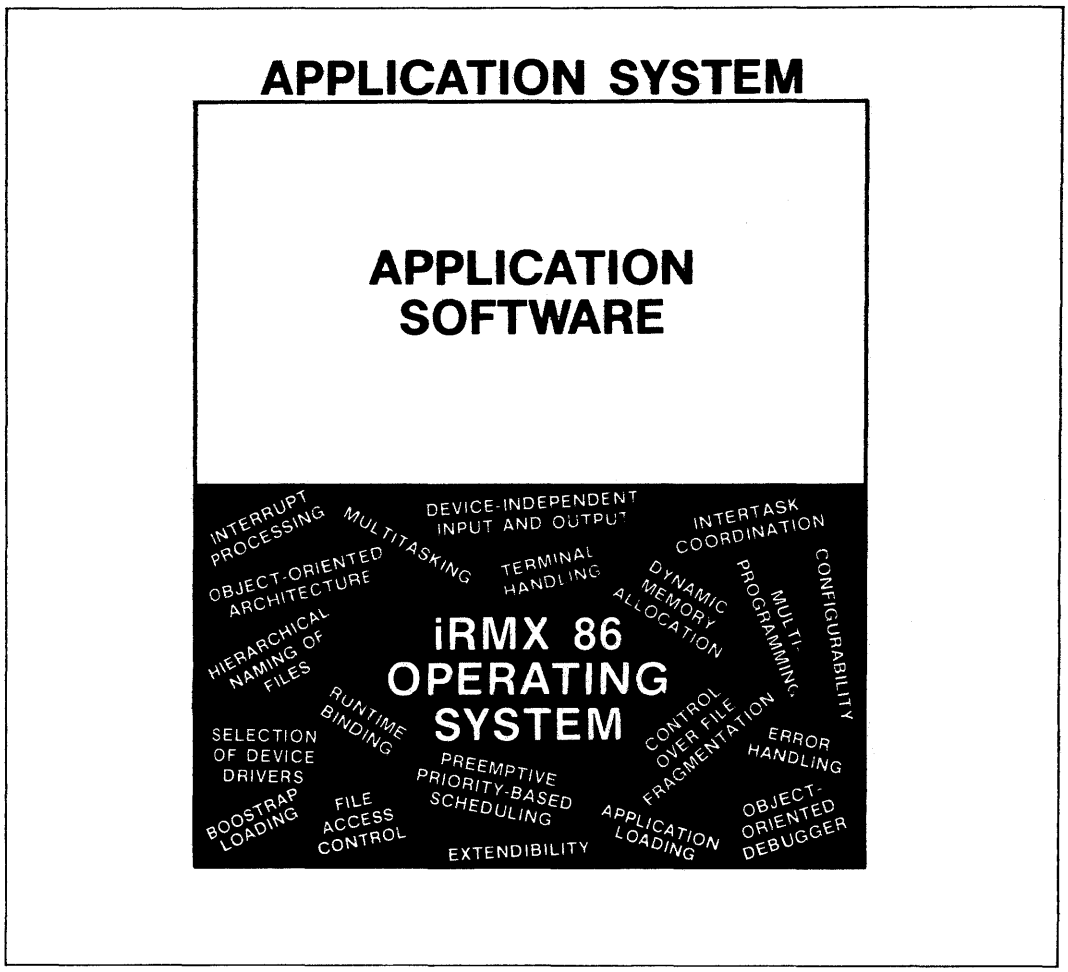

Figure 4-1. Features of the iRMX 86™ Operating System

OBJECT-ORIENTED ARCHITECTURE

The iRMX 86 Operating System uses an object-oriented architecture because it makes the Operating System easy to learn and use.

EXPLANATION OF OBJECT-ORIENTED ARCHITECTURE

An operating system is a collection of software that is meant to be used by software engineers. Many operating systems are so complex that the majority of the engineers using them are unable to fully grasp their organization. In contrast, systems exhibiting object-oriented architectures are easier to understand. Their mechanisms are well defined, and they demonstrate a consistency that makes the operating system seem less awesome.

In other words, an object-oriented architecture is a means of humanizing an operating system. It uses a collection of building blocks that are manipulated by operators. Let's look at a typed architecture that you might be familiar with -- FORTRAN.

FORTRAN exhibits a typed architecture. Its building blocks are variables of several types. For instance, it has integers, real numbers, double-precision real numbers, etc. It also has operators (+, -, *, /, **, and others) that act on variables to produce understandable results. Let's turn back to operating systems and see how object-oriented architecture works in the iRMX 86 System.

The building blocks of the iRMX 86 Operating System are called objects and, as with FORTRAN variables, objects are of several types. There are tasks, jobs, mailboxes, semaphores, segments, and connections. There are also other types of objects, but we already have enough for an introduction.

Just as the variables in a FORTRAN program are acted upon by operators, the objects in an iRMX 86-based application system are acted upon by system calls. In other words, your application software uses system calls to manipulate the objects in your application system. For instance, the CREATE MAILBOX and DELETE MAILBOX system calls do precisely what their names suggest.

How does an object-oriented architecture make a system easier to learn and use? By taking advantage of useful classification. To illustrate this, let's return to FORTRAN. The variables of FORTRAN are classified into types because each type exhibits certain characteristics. For instance, all integer variables are somewhat similar, even though they can take on different values. Once you learn the characteristics of an integer variable, you feel comfortable with every integer variable. This similarity makes FORTRAN easy to master.

For the same reasons, the objects of the iRMX 86 Operating System are classified into types. Each object type (such as a semaphore) has a specific set of attributes. Once you become familiar with the attributes of a semaphore, you are familiar with all semaphores. There are no special cases.

This useful classification also applies to iRMX 86 system calls. Each type of iRMX 86 object has an associated set of system calls. These calls cannot be used to manipulate objects of another type without causing an error. (Our analogy breaks down at this point. FORTRAN operators almost always work on several types of variables.)

The beauty of the object-oriented architecture of the iRMX 86 Operating System can be summed up in one statement. Once you learn the attributes and the system calls associated with a type of object, you have complete knowledge of the behavior of the object type.

ADVANTAGES OF OBJECT-ORIENTED ARCHITECTURE

The advantages of an object-oriented architecture depend upon your point of view. If you are an engineer, the advantage is that you can master the Operating System in a very short time. You can also focus your learning on the objects you plan to use. If you only need a few object types, you can ignore the others.

If you are a manager, you reap economic benefits. Because engineers can quickly become familiar with the iRMX 86 Operating System, you can trim large amounts of time out of your system's development cycle. Your system reaches your users far sooner and at far less cost than it could without object-oriented architecture.

MULTITASKING

The iRMX 86 Operating System uses multitasking to simplify the development of applications that process realtime events.

EXPLANATION OF MULTITASKING

The essence of realtime application systems is the ability to process numerous events occurring at seemingly random times. These events are asynchronous because they can occur at any time, and they are potentially concurrent because one event might occur while another is being processed.

Any single program that attempts to process multiple, concurrent, asynchronous events is bound to be complex. The program must perform several functions. It must process the events. It must remember which events have occurred and the order in which they occurred. It must remember which events have occurred but have not been processed. The complexity obviously grows greater as the system monitors more events.

Multitasking is a technique that unwinds this confusion. Rather than writing a single program to process N events, you can write N programs, each of which processes a single event. This technique eliminates the need to monitor the order in which events occur.

Each of these N programs forms an iRMX 86 task, one of the types of objects mentioned in "Object-Oriented Architecture." Tasks are the only active objects in the iRMX 86 Operating System, as only tasks can issue system calls.

ADVANTAGES OF MULTITASKING

Multitasking simplifies the process of building an application system. This allows you to build your system faster and at less expense. Furthermore, because of the one-to-one relationship between events and tasks, your system's code is less complex and easier to maintain.

INTERRUPT PROCESSING

The iRMX 86 Operating System is an interrupt processor. When an interrupt occurs, the iRMX 86 Operating System schedules a task to process the interrupt. This method of event detection improves the performance of your application system.

EXPLANATION OF INTERRUPT PROCESSING

There are two ways that computer systems can schedule processing associated with detecting and controlling events in the real world -- polling and interrupt processing. Polling, is implemented by having the software periodically check to see if certain events have occurred. An example of polling from a human perspective can be created using a class of students and a teacher. If, rather than spotting raised hands, the instructor specifically asks each student in the class if the student has any questions, then the instructor is polling the students.

Polling has a major shortcoming. A significant amount of the processor's time is spent testing to see if events have occurred. If events have not occurred, the processor's time has been wasted.

The second method of controlling processing is interrupt processing. In this method, when an event occurs the processor is literally interrupted. Rather than executing the next sequential instruction, the processor begins to execute a task associated specifically with the detected event.

The classroom example used earlier to portray a polling situation can also be used to illustrate interrupt processing. If a student has a question, he raises his hand and speaks the instructor's name. The instructor, interpreting this as an interrupt, finishes his sentence and deals immediately with the student's question. Once the instructor has answered the student's question, he returns to what he was doing before he was interrupted.

ADVANTAGES OF INTERRUPT PROCESSING

Interrupt processing of external events provides your application system with three benefits.

- Better Performance

  Interrupt processing allows your system to spend all of its time running the tasks that process events, rather than executing a polling loop to see if events have occurred.

- More Flexibility

  Because of the direct correlation between interrupts and tasks, your system can easily be modified to process different events. All you need to do is write the tasks to process the new interrupts.

- Economic Benefits

  Because interrupt processing allows your system to respond to events by means of modularly coded tasks, your system's code is more structured and easier to understand. Modular code is less costly to develop and maintain, and it can be developed more quickly than unstructured code.

PREEMPTIVE PRIORITY-BASED SCHEDULING

The iRMX 86 Operating System uses preemptive, priority-based scheduling to decide which task runs at any instant. This technique ensures that if a more important task becomes ready while a less important task is running, the more important task begins execution immediately.

EXPLANATION OF PREEMPTIVE PRIORITY-BASED SCHEDULING

In multitasking systems, there are two common techniques for deciding which task is to be run at any given moment. Time slicing, where tasks are run in rotation, is the technique used in time-sharing systems. The second technique, priority-based scheduling, uses assigned priorities to decide which task is to be run.

Within priority-based scheduling, there are two approaches. Nonpreemptive scheduling allows a task to run until it relinquishes the processor. Even if while running it causes a higher-priority task to become ready for execution (for instance, by deallocating a peripheral device), the original task continues to run until it explicitly surrenders the processor.

The second approach to priority-based scheduling is preemptive. In systems using preemptive scheduling, the system always executes the highest priority task that is ready to run. In other words, if the running task or an interrupt causes a higher-priority task to become ready, the operating system switches the processor to the higher-priority task.

ADVANTAGE OF PREEMPTIVE PRIORITY-BASED SCHEDULING

Preemptive, priority-based scheduling goes hand-in-hand with the interrupt processing discussed earlier. The priorities of tasks can be tied to the relative importance of the events that they process. This enables the processing of more important events to preempt the processing of less important events.

MULTIPROGRAMMING

Multiprogramming provides your system with the ability to run more than one application on one iAPX 86-based microcomputer. This helps reduce hardware costs.

EXPLANATION OF MULTIPROGRAMMING

Multiprogramming is a technique used to run several applications on a single application system. By using this technique, the hardware is used more fully. More processing is squeezed out of each hardware dollar.

In order to take full advantage of multiprogramming, you must provide each application with a separate environment; that is, separate memory, files and objects. The reason for the isolation is to prevent independently developed applications from causing problems for each other.

For instance, suppose that two unrelated applications share a temporary file on a disk. If Application 1 writes information to the file and Application 2 writes over the file, Application 1 has problems. The only way to avoid this kind of problem with shared files is to create some form of mutual exclusion. But if the two applications must interact even to the point of excluding each other, they cannot be developed independently. The two engineers creating the applications must coordinate with each other and spend valuable time that could be used within, rather than between, applications. The only alternative is to avoid sharing the file.

The iRMX 86 Operating System provides a type of object that can be used to obtain this kind of isolation. The object is called a job, and it has the following characteristics:

- Unlike tasks, jobs are passive. They cannot invoke system calls.
- Each job includes a collection of tasks and resources needed by those tasks.
- Jobs serve as useful boundaries for dynamically allocating memory. When two tasks of one job request memory, they share the memory associated with their job. Two tasks in different jobs do not directly compete for memory.
- An application consists of one or more jobs.
- Each job serves as an error boundary. When the application detects an error, or when the operator decides to abort an application, a job is a convenient object to delete.

ADVANTAGES OF MULTIPROGRAMMING

Multiprogramming provides your application system with two benefits:

- Multiprogramming increases the amount of work your system can do. By using your hardware a larger percentage of the time, it lets your system run several applications rather than one. This reduces the hardware cost of implementation.
- Because of the correspondence between jobs and applications, new jobs can be added to your system (or old jobs removed) without affecting other jobs. This makes your system much easier and faster to modify.

ERROR HANDLING

The iRMX 86 Operating System allows your application system to specify an error handling procedure for each task.

EXPLANATION OF ERROR HANDLING

Error handling is the process of detecting and reacting to unexpected conditions. The iRMX 86 Operating System supports error handling by doing a substantial amount of validity testing and condition checking within system calls, but it cannot detect every error.

The iRMX 86 Operating System runs on an iAPX 86-based microcomputer. Such computers do not provide memory protection. Consequently, if one of the procedures in your application software contains bugs, it could execute code at random locations and write over parts of the Operating System. The iRMX 86 Operating System cannot detect this kind of error.

Nonetheless, the iRMX 86 Operating System does protect your system from some types of errors. The concepts involved in the iRMX 86 error handling scheme are condition codes, exception handlers, and exception modes. We'll look at these one at a time.

- Condition Codes

  Whenever a task invokes a system call, the iRMX 86 Operating System attempts to perform the requested function. Whether or not the attempt is successful the Operating System generates a condition code. This code indicates two things. First, it shows whether the system call succeeded or failed. Second, in the case of failure, the code shows which unexpected condition prevented successful completion. Successful completion is indicated by a normal condition code, while unsuccessful completion is indicated by an exceptional condition code.

  For the sake of flexibility in processing unexpected conditions, exceptional condition codes are divided into two categories. The first category, environmental condition codes, consists of errors that a task cannot anticipate. An example of such an error is insufficient memory. The second category, programming error codes, consists of two subcategories:

  - Errors Detected by the Processor

    The iAPX 86 microprocessor detects several kinds of error conditions. One of these, for instance, is an attempted division by zero. Such errors can be avoided by using good programming techniques.

  - Incorrect System Calls

    If the Operating System detects parameters or combinations of parameters that are incorrect, the problem is considered a programming error. This kind of error can usually be avoided by good programming techniques.

- Exception Handlers

  An exception handler is a procedure that the Operating System can invoke when a task receives an exceptional condition code. As each task is created, it is assigned an exception handler; therefore an exception handler is an attribute of a task. The alternative to using exception handlers is to process exceptional condition codes in the procedure that issued the system call.

- Exception Modes

  An exception mode is an attribute of a task. When you create a task (using the CREATE TASK system call), you must set the task's exception mode to one of four values. This value governs the processing of condition codes received by the task. The exception mode can be assigned any of these values:

  - Any processing of exceptional conditions must be done within the procedure that issued the system call leading to the exceptional condition.

  - The task's exception handler processes only environmental condition codes. Any processing of programming error codes must be done within the procedure that issued the system call leading to the programming error code.

  - The task's exception handler processes only programming error codes. Any processing of environmental codes must be done within the procedure that issued the system call leading to the environmental condition code.

  - The task's exception handler processes both environmental condition codes and programming error codes.

In summary, exception handling works as follows. The Operating System generates a condition code for each system call. If the code indicates successful completion, the Operating System detected no problems. If the code indicates an exceptional condition, the code can be processed either of two ways: within the procedure that invoked the system call, or by the task's error handler which is invoked by the Operating System. The decision as to which technique is used is a function of the task's exception mode and the category of the condition code (programming error or environmental condition).

You can control the behavior of a task's error handler because you can write the handler. Consequently, the handler can recover from the error, delete the job or task containing the error, warn the operator of the error, or ignore the error altogether. The choice is yours.

ADVANTAGE OF ERROR HANDLING

Error handling provides your application system with several methods for reacting to unusual conditions. One of these methods, having the Operating System automatically invoke your task's error handling procedure, greatly simplifies error processing. The other method, dealing with some or all unusual conditions within your application task, allows you to provide special processing for unusual circumstances. The iRMX 86 Operating System allows your application system to use both methods.

DYNAMIC MEMORY ALLOCATION

The iRMX 86 Operating System supports dynamic allocation of memory. This allows you to reduce your implementation costs by building systems in which applications share memory. It also allows your applications to change the amount of memory they use as their needs change.

EXPLANATION OF DYNAMIC MEMORY ALLOCATION

Although there are numerous techniques for assigning memory to jobs, each technique falls into one of two classes: static allocation or dynamic allocation. Let's look briefly at static allocation first.

Static memory allocation entails assigning memory to jobs when the system is starting up. Once the memory is allocated, it cannot be freed to be used by other jobs. Consequently, the total memory requirements of the system is always the sum of the memory requirements of each job.

Dynamic memory allocation, on the other hand, allows jobs to share memory. Memory is allocated to jobs only when tasks request it. And when a job no longer needs the memory, one of its tasks can free the memory for use by other jobs.

Dynamic allocation also is useful within a job. Some tasks can use additional memory to improve efficiency. An example of this is a task that allocates large buffers to speed up input and output operations.

ADVANTAGE OF DYNAMIC MEMORY ALLOCATION

The dynamic allocation of memory provides your application system with reduced implementation costs. If your application system runs more than one application, chances are fair that memory demands for various jobs will be out of phase. That is, one job will be freeing memory while another needs more. Dynamic memory allocation allows jobs to take advantage of this. Consequently, your application system requires less memory than it would using static allocation.

INTERTASK COORDINATION

The iRMX 86 Operating System provides simple techniques for tasks to coordinate with one another. These techniques allow tasks in a multitasking system to mutually exclude, synchronize, and communicate with each other.

EXPLANATION OF INTERTASK COORDINATION

As we have already seen, multitasking is a technique used to simplify the designing of realtime application systems that monitor multiple, concurrent, asynchronous events. Multitasking allows engineers to focus their attention on the processing of a single event rather than having to contend with numerous other events occurring in an unpredictable order.

However, the processing of several events may be related. For instance, the task processing Event A may need to know how many times Event B has occurred since Event A last occurred. This kind of processing requires that tasks be able to coordinate with each other. The iRMX 86 Operating System provides for this coordination.

Tasks can interact with each other in three ways. They can exchange information, mutually exclude each other, and synchronize each other. We'll now examine each of these.

- Exchanging Information

  Tasks exchange information for two purposes. One purpose is to pass data from one task to another. For instance, suppose that one task accumulates keystrokes from a terminal until a carriage return is encountered. It then passes the entire line of text to another task, which is responsible for decoding commands.

  The second reason for passing data is to draw attention to a specific object in the application system. In effect, one task says to another, "I am talking about that object."

The iRMX 86 System facilitates intertask communication by supplying objects called mailboxes along with system calls to manipulate mailboxes. The system calls associated with mailboxes are CREATE MAILBOX, DELETE MAILBOX, SEND MESSAGE, and RECEIVE MESSAGE. Tasks use the first two system calls to build and eradicate a particular mailbox. They use the second two calls to communicate with each other.

Let's see how tasks can use a mailbox for drawing attention and for sending information. If Task A wants Task B to become aware of a particular object, Task A uses the SEND MESSAGE system call to mail the object to the mailbox. Task B uses the RECEIVE MESSAGE system call to get the object from the mailbox.

NOTE

> The foregoing example, along with all of the examples in this section, is somewhat simplified in order to serve as an introduction. If you want detailed information, refer to the iRMX 86 NUCLEUS, TERMINAL HANDLER AND DEBUGGER REFERENCE MANUAL.

As mentioned previously, tasks can use mailboxes to send information to each other. This is accomplished by putting the information into a segment (an iRMX 86 object consisting of a contiguous block of memory) and using the SEND MESSAGE system call to mail the segment. The other task invokes the RECEIVE MESSAGE system call to get the segment containing the message.

Why don't tasks just send messages directly between each other, rather than through mailboxes? Tasks are asynchronous -- they run in upredictable order. If two tasks want to communicate with each other, they need a place to store messages and to wait for messages. If the receiver uses the RECEIVE MESSAGE system call before the message has been sent, the receiver waits at the mailbox until a message arrives. Similarly, if the sender uses the SEND MESSAGE system call before the receiver is ready to receive, the message is held at the mailbox until a task requests a message from the mailbox. In other words, mailboxes allow tasks to communicate with each other even though tasks are asynchronous.

- Mutual Exclusion

Occasionally, when tasks are running concurrently, the following kind of situation arises:

- Task A is in the process of reading information from a segment.

- An interrupt occurs and Task B, which has higher priority than Task A, preempts Task A.

- Task B modifies the contents of the segment that Task A was in the midst of reading.

- Task B finishes processing its event and surrenders the processor.

- Task A resumes reading the segment.

The problem is that Task A might have information that is completely invalid. For instance, suppose the application is air traffic control. Task A is responsible for detecting potential collisions, and Task B is responsible for updating the Plane Location Table with the new X- and Y-coordinates of each plane's location. Unless Task A can obtain exclusive use of the Plane Location Table, Task B can make Task A fail to spot a collision.

Here's how it could happen. Task A reads the X-coordinate of the plane's location and is preempted by Task B. Task B updates the entry that Task A was reading, changing both the X- and Y-coordinates of the plane's location. Task B finishes its function and surrenders the processor. Task A resumes execution and reads the new Y-coordinate of the plane's location. As a direct result of Task B changing the Plane Location Table while Task A was reading it, Task A thinks the plane is at old X and new Y. This misinformation could easily lead to disaster.

This problem can be avoided by mutual exclusion. If Task A can prevent Task B from modifying the table until after A has finished using it, A can be assured of valid information. Somehow, Task A must obtain exclusive use of the table.

The iRMX 86 Operating System provides a type of object that can be used to provide mutual exclusion -- the semaphore. A semaphore is an integer counter that tasks can manipulate using four system calls: CREATE SEMAPHORE, DELETE SEMAPHORE, SEND UNITS and RECEIVE UNITS. The creation and deletion system calls are used to build and eradicate semaphores. The send and receive system calls can be used to achieve mutual exclusion.

Before discussing how semaphores can provide exclusion, we must examine their properties. As mentioned above, a semaphore is a counter. It can take on only nonnegative integer values. Tasks can modify a semaphore's value by using the SEND UNITS or RECEIVE UNITS system calls. When a task sends N units (must be zero or greater) to a semaphore, the value of the counter is increased by N. When a task uses the RECEIVE UNITS system call to request M units (must be zero or greater) from a semaphore, one of two things happens.

- If the semaphore's counter is greater than or equal to M, the Operating System reduces the counter by M and continues to execute the task.

- Otherwise, the Operating System begins running the task having the next highest priority, and the requesting task waits at the semaphore until the counter reaches M or greater.

How can tasks use a semaphore to achieve mutual exclusion? Easy! Create a semaphore with an initial value of 1. Before any task uses the shared resource, it must receive one unit from the semaphore. Also, as soon as a task finishes using the resource, it must send one unit to the semaphore. This technique ensures the following behavior. At any given moment, no more than one task can use the resource, and any other tasks that want to use it await their turn at the semaphore.

Semaphores allow mutual exclusion; they don't enforce it. All tasks (there can be more than two) sharing the resource must receive one unit from the semaphore before using the resource. If one task fails to do this, mutual exclusion is not achieved. Also, each task must send a unit to the semaphore when the resource is no longer needed. Failure to do this can permanently lock all tasks out of the resource.

- Synchronization

As mentioned earlier, tasks are asynchronous. Nonetheless, occasionally a task must know that a certain event has occurred before the task starts running. For instance, suppose that a particular application system requires that Task A cannot run until after Task B has run. This kind of requirement calls for synchronizing Task A with Task B.

Your application system can achieve synchronization by using semaphores. Before executing either Task A or Task B, create a semaphore with an initial value of zero. Then have Task A issue RECEIVE UNITS requesting one unit from the semaphore. Task A is forced to wait at the semaphore until Task B sends a unit. This achieves the desired synchronization.

ADVANTAGE OF INTERTASK COORDINATION

Every realtime multitasking system must provide for intertask coordination, so this coordination cannot be billed as an advantage. The true advantage arises from the flexible means that the iRMX 86 System provides for accomplishing coordination.

The intertask coordination supplied by the iRMX 86 Operating System is flexible and simple to use. Semaphores and mailboxes can accommodate a wide variety of situations. And your application system is not limited to some arbitrary number of mailboxes or semaphores. It can create as many as it needs.

RUNTIME BINDING

The iRMX 86 Operating System uses runtime binding. This provides your system with three kinds of flexibility. It allows tasks in different jobs to share objects; it lets your procedures use logical names for files and devices; and it simplifies the process of attaching your application software to the iRMX 86 Operating System.

EXPLANATION OF RUNTIME BINDING

Before we look into runtime binding, let's consider binding as it relates to a program. Binding is the process of letting each program know the locations of the variables and procedures that it uses.

Binding can be performed several times during the development and execution of a program. Some binding takes place during the process of compilation. As a program is being compiled, its references to variables and procedures are resolved (that is, converted into machine language) whenever the compiler has sufficient information. Sometimes, however, a program refers to variables or procedures that are part of a separate program. When this happens, the compiler cannot resolve the reference, and binding must be delayed.

Some binding also takes place during linking. Linking is the process of combining several programs that are compiled separately. The purpose of linking is to allow a program to refer to variables and procedures defined in a different program. (Such references are called external references because they refer to information outside of the program under consideration.) When the linking process resolves an external reference, it performs binding that cannot be completed during compilation.

Runtime binding means binding while the system is actually running. The iRMX 86 Operating System provides three kinds of runtime binding:

- Binding objects to tasks.
- Binding files and devices to tasks.
- Binding your application software to the Operating System.

The first two kinds of runtime binding are based on the use of object directories. An object directory is an attribute of a job that allows tasks to provide ASCII names for objects. Tasks use the CATALOG OBJECT, LOOKUP OBJECT, and UNCATALOG OBJECT system calls to define, lookup, or delete the name of an object. In each case, the task using the system call must specify the job whose object directory is to be accessed.

Now we'll look more closely at each type of runtime binding.

Binding Objects to Tasks

When two tasks use a mailbox to pass information, they obviously must both access the same mailbox. But if the programs for the two tasks are compiled and linked independently of one another (as they probably would be if they are in separate jobs), the tasks must use runtime binding to access the same mailbox.

The runtime binding of objects to tasks is accomplished as follows. When a task creates an object that it wishes to share with other tasks, the creator task catalogs the object in an object directory. Other tasks can then access the cataloged object if they know its ASCII name.

Engineers can control the sharing of objects by selectively broadcasting object names. If two engineers wish to share an object, they must agree on both the name and the object directory that is to contain the name. One task then creates the object and the other accesses it through the object directory.

Binding of Files and Devices to Tasks

Suppose you wish to code an application utility program that takes input from any supported input device or from a disk file. Runtime binding can help accomplish this. The utility program can be coded to lookup an input connection under a particular name. Then any program that needs the utility program can create the input connection, catalog it under the agreed-upon name, and invoke the utility program. In effect, the ASCII name in the object directory is the logical name of the input file.

Binding of Application Software to Operating System

The iRMX 86 Operating System uses a third type of runtime binding to allow your application software to communicate with the Operating System. Whenever your application software invokes a system call, an INTEL-supplied interface routine converts the call into a software-generated interrupt. This interrupt causes control to be transferred to a procedure within the iRMX 86 Operating System that performs the desired function. In other words, the software interrupts bind the system calls of your application software to the iRMX 86 procedures.

ADVANTAGES OF RUNTIME BINDING

Runtime binding provides your application system with flexibility. By allowing your system to name objects, the iRMX 86 Operating System provides a means of sharing dynamically created objects between jobs. By supporting logical names for files and devices, the iRMX 86 System allows tasks to work with any combination of files and devices rather than with a single, fixed combination. By using software interrupts to bind your application software to the Operating System, you can reconfigure the Operating System without having to recompile or relink your application software.

EXTENDIBILITY

The iRMX 86 Operating System is extendible. It allows you to create your own object types and to add system calls to the Operating System.

EXPLANATION OF EXTENDIBILITY

Something is extendible if you can add to it, and the iRMX 86 Operating System is extendible. Your system programming engineers can build their own types of objects and the system calls to manipulate those objects. These custom features become a part of the Operating System. From the point of view of the application programming engineer, there is no way to distinguish your custom objects from those supplied by Intel.

ADVANTAGE OF EXTENDIBILITY

The advantage of extendibility is that you can add your features to the iRMX 86 Operating System and obtain the same benefits as supplied by its object-oriented architecture. These benefits include the ability to send your custom-made objects to mailboxes and the ability to put them in object directories. Additionally, your application engineers can more quickly become familiar with your custom features. This shrinks your development time and costs, and it allows you to bring your application system to your users sooner.

TERMINAL HANDLING

The iRMX 86 Operating System includes the software to control one terminal which can be either a teletypwriter or a keyboard and screen.

EXPLANATION OF THE TERMINAL HANDLER

The iRMX 86 Terminal Handler is a job that runs under the control of the iRMX 86 Operating System. This job, which serves as the interface between the terminal and the tasks of your application system, provides the following capabilities:

- Your tasks can communicate with the terminal asynchronously.
- The operator using the terminal can edit lines before they are seen by the application software.
- The operator using the terminal can suppress or slow down the display of output generated by the application system.

Using these features, your application system can interact with a human operator.

ADVANTAGES OF THE TERMINAL HANDLER

The advantages of the Terminal Handler are economic. By relieving you of the chore of creating an interface with a terminal, the Terminal Handler reduces your system's cost of implementation and time to market. Furthermore, because it has been debugged by Intel, the Terminal Handler also reduces your maintenance costs.

APPLICATION LOADING

The iRMX 86 Operating System allows your application to read programs from disk into memory.

EXPLANATION OF APPLICATION LOADING

Application systems occasionally contain some programs that are infrequently used. If the programs are stored on disk, the application system can load them into main memory whenever they are required. This loading process is called Application Loading.

ADVANTAGE OF APPLICATION LOADING

The iRMX 86 Operating System allows you to store infrequently used programs on disk rather than in main memory. This reduces the amount of memory that you must incorporate in your system's hardware.

DEVICE-INDEPENDENT INPUT AND OUTPUT

The input and output capabilities of the iRMX 86 Operating System are device independent. This adds flexibility to your system by allowing you to easily reroute input or output to different devices.

EXPLANATION OF DEVICE-INDEPENDENT INPUT AND OUTPUT

Device independence is a relatively simple yet powerful concept. A system provides device-independent I/O if it has one set of system calls for communicating with all I/O devices. The alternative to device independence is to provide different calls for each type of device. Let's first examine the alternative and then move on to device independence.

Consider an operating system that does not provide device independence. The system calls controlling input and output operations are explicitly related to the I/O devices being used. For instance, the system call for writing to the line printer might be PRINT, while the system call for writing to the terminal might be TYPE. Once you have written a procedure in such a system, the procedure is locked into a particular combination of devices. The only way you can reroute input or output is to edit the source code and recompile.

Now consider an operating system that is device independent: the iRMX 86 Operating System. Because the iRMX 86 System supports device-independent I/O, the system calls are not device dependent. The READ system call is always used for input, and the WRITE system call is always used for output. The device is specified by a parameter of the system call. Consequently, by using a variable as the parameter that selects the device, you can create I/O procedures that are completely independent of the devices they use.

ADVANTAGES OF DEVICE-INDEPENDENT INPUT AND OUTPUT

Device independence makes your application system very flexible. If you write a procedure to log events on a line printer, you can use the same procedure to log events on a terminal or, for that matter, on a disk. You need not recompile or otherwise modify your system.

HIERARCHICAL NAMING OF MASS STORAGE FILES

The iRMX 86 Operating System supports hierarchical naming of files on mass storage devices. This naming technique provides your application systems with additional flexibility by simplifying the process of organizing and naming files.

EXPLANATION OF HIERARCHICAL NAMING

Hierarchical naming is one of three common techniques used to name files on mass storage devices such as disks, bubble memories, or drums. The other two techniques are called simple naming and directory naming. The advantages of hierarchical naming become clear when that technique is compared to the other two. First we'll look at simple naming.

Simple naming allows you to provide files with a descriptive name. For instance, you might decide to name files ACCOUNTS PAYABLE, ACCOUNTS RECEIVABLE, TRANSACTIONS, and INVENTORY. These names are certainly descriptive, but what happens when a different application running in the same system also decides to use one of these names? This question is avoided by using a more powerful naming technique: directory naming.

Directory naming allows different applications (or different application engineers, for that matter) to use the same file name. Each application (or engineer) is given one special-purpose file, called a directory. This directory contains only file names; it does not contain data. Figures 4-2 and 4-3 provide examples of directories.

When application software refers to a specific file, it first names the directory and then names the file. For instance, in Figure 4-2, the TRANSACTIONS file associated with Engineering would be designated ENGINEERING/TRANSACTIONS. The comparable file for Marketing, in Figure 4-3, would be designated MARKETING/TRANSACTIONS.

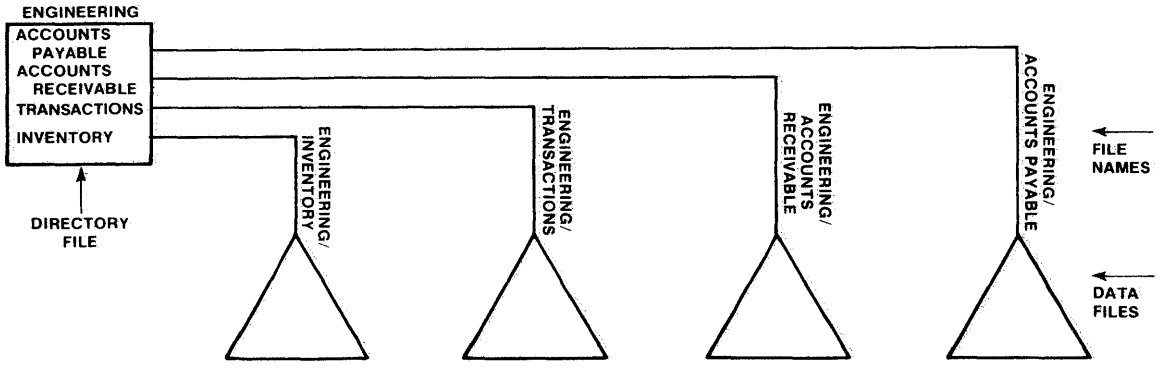

Figure 4-2. An Engineering directory

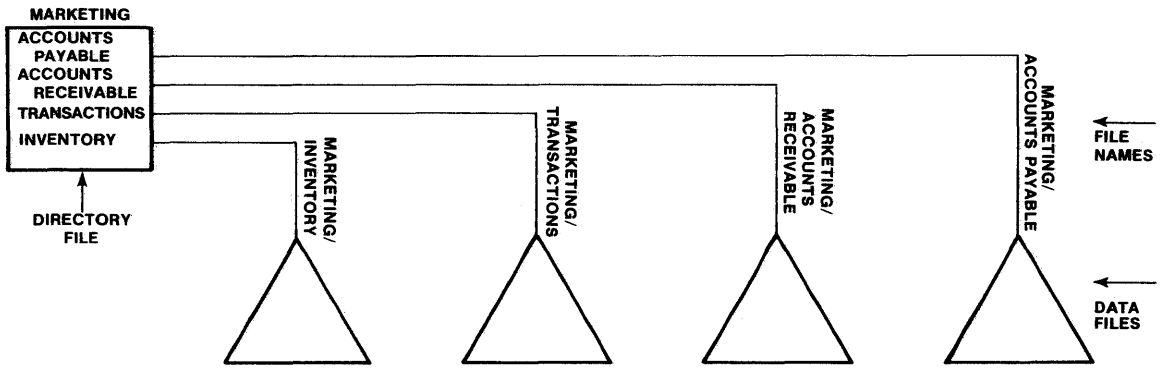

Figure 4-3. A Marketing directory

The advantage of directory naming over simple naming is that directory naming allows the file names to reflect the relationships between files. In Figure 4-2, all the files pertaining to Engineering are in the directory called ENGINEERING. This grouping of related files is not supported by simple naming.

What about situations in which more than one level of directory is required? This situation is illustrated in Figure 4-4. This figure differs from 4-3 only in that a second level of grouping has been included.

Just as Figure 4-4 shows that single-level directory naming is not sufficient for all collections of files, another figure could be constructed to show that two-level directory naming is not always sufficient. Consequently, the iRMX 86 Operating System supports any number of levels of directories. This n-level directory naming is called hierarchical naming of files.

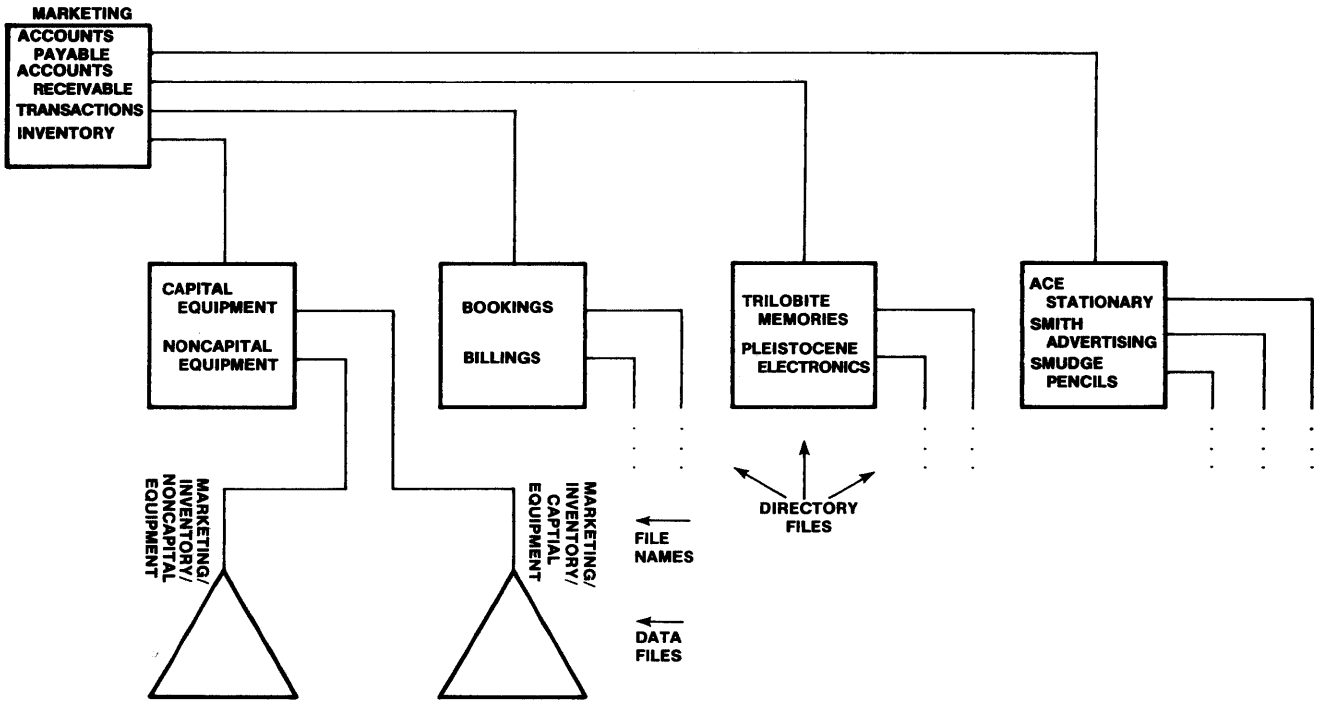

Figure 4-4. Hierarchical naming of files

ADVANTAGES OF HIERARCHICAL NAMING

Hierarchical naming of files simplifies the process of adding new applications to your system. One concern about expanding your system is the naming of mass storage files associated with a new application. Names of new files must differ from names of existing files. If your system uses only a few mass storage files, you can expect little difficulty in assigning unique file names. But if your system uses a large number of files, the problem of assuring uniqueness becomes more significant. This uniqueness problem becomes particularly difficult if file names are assigned by an operator in a system having more than one operator.

Hierarchical file naming eliminates this problem. Whenever you add a new application to your system, you can assign it a directory. The new application can then use this directory to provide unique names to any number of files. Also, each operator can be assigned a unique directory which can then be used to provide unique names.

FILE ACCESS CONTROL

The iRMX 86 Operating System allows your application system to control access to hierarchically named files. This facilitates file sharing while still preventing valuable data from being copied, modified, or destroyed by unauthorized users.

EXPLANATION OF FILE ACCESS CONTROL

In the multiprogramming environment provided by the iRMX 86 Operating System, the sharing of files can be useful. But the job that owns a file may wish to share it with only certain other jobs rather than all other jobs. Furthermore, the job owning a file may wish to restrict the nature of the shared access. For example, the owning job may wish to allow a particular file to be read but not written. The ability to specify how and with whom a file is shared is called file access control.

The iRMX 86 Operating System provides powerful file access control by allowing the owner of a file to specify who can use the file and how they can use it. In fact, a file's owner can even grant different combinations of access (reading only, writing only, reading and writing, etc.) to each user of a file.

ADVANTAGES OF FILE ACCESS CONTROL

By controlling who can access a file and how they can access it, your system becomes more reliable and secure. There is less chance for an unauthorized task to accidentally modify a valuable file, and there is less opportunity for an unauthorized task to read a confidential file.

Your application software can, in fact, expand file access protection into a file security system. For instance, suppose that your application involves several operators accessing files on disk. By providing each operator with a password, so an individual's identity can be verified, your application software can strictly control which operators have access to which files.

CONTROL OVER FILE FRAGMENTATION

The iRMX 86 Operating System allows you to specify the granularity of each mass storage file. This lets you trade faster I/O for more efficient use of space on the mass storage device.

EXPLANATION OF FILE FRAGMENTATION

When information is stored on a mass storage device, space is allocated in chunks rather than one byte at a time. These chunks, called granules, can be large or small, but all granules within one file must be the same size. This size is called the file granularity, and it is specified by the engineer who creates the file.

A file's granularity affects the use of a storage device in three ways.

- Data Transfer Rate

  The file granularity directly affects the speed at which the Operating System can transfer information to or from the storage device. The larger the granularity, the faster the Operating System transfers data.

- Access Time

  The smaller the granules, the more time is required to access a series of random locations in the file. Larger granules reduce access time.

- Wasted Device Space

  The file granularity directly affects the amount of wasted space on the device. The larger the granularity, the more device space is wasted.

  Here's an example. (For the sake of simplicity, we will ignore any information stored on the device on behalf of the Operating System.) Consider a file containing 20010 bytes. If the granularity is 10000 bytes, the file occupies three granules, each of which is 10000 bytes long. The first two granules are full and the third contains only 10 useful bytes. This file wastes almost 10000 bytes of storage space.

  If we change the file granularity to 200 bytes, the file occupies 101 granules. Each of the first 100 granules is full and the last granule contains only 10 useful bytes. The file now wastes only 190 bytes of storage space.

ADVANTAGES OF CONTROL OVER FRAGMENTATION

By allowing you to control the file granularity, the iRMX 86 Operating System lets you trade device space for performance. If your application has many mass storage units and space is readily available, you can specify a large file granularity. This provides you with faster average transfer rates and shorter access times, but it wastes some of your device space.

If, on the other hand, you have only one small mass storage unit, you might want to sacrifice some performance for better use of space. This trade would be particularly desirable if you do not use the device often enough to be concerned with the rate of data transfer.

SELECTION OF DEVICE DRIVERS

The iRMX 86 Operating System offers you your choice of Intel-supplied device drivers. It also allows you to write your own drivers.

EXPLANATION OF DEVICE DRIVERS

A device driver is a software module that serves as the interface between a device's controller (which is hardware) and the iRMX 86 I/O System. The purpose of the driver is to make all devices look alike to the I/O System. In effect, the driver hides the idiosyncrasies of a device from the I/O System.

ADVANTAGES OF HAVING A SELECTION

By selecting and creating device drivers, you can attach any device to your application system. This means that you are not limited to a few specific devices. You can select devices on any basis at all -- performance, cost, reliability, availability, whatever. The choice is yours.

NOTE

The iRMX 86 Operating System is being released in several phases. The second release contains drivers for the iSBC 206 rigid disk controller, the iSBC 204 flexible disk controller and the USART on the iSBC 86 Single Board Computer. Later releases will contain additional drivers.

OBJECT-ORIENTED DEBUGGER

The iRMX 86 Operating System provides a special debugger that is attuned to iRMX 86 objects. This debugger simplifies the process of removing the bugs in the interaction between tasks of the application system. It also facilitates debugging in a realtime environment.

EXPLANATION OF AN OBJECT-ORIENTED DEBUGGER

We have already discussed the object-oriented architecture of the iRMX 86 Operating System. Reviewing briefly, each iRMX 86 job is a community of tasks, and each task can manipulate objects. A special set of objects (mailboxes and semaphores) provides a means for tasks to coordinate with one another.

The iRMX 86 Debugger has two capabilities that greatly simplify the process of debugging in a multitasking environment. First, the Debugger allows you to debug several tasks while the balance of the application system continues to run in real time. Second, the debugger lets you see which tasks or objects are queued at mailboxes and semaphores.

These two capabilities help you debug your application system at two levels. You can look into the behavior of an individual task, and you can examine the interaction between tasks. Both levels must be thoroughly debugged before your system is fully implemented.

ADVANTAGE OF AN OBJECT-ORIENTED DEBUGGER

The object-oriented Debugger gives your application system flexibility while simultaneously providing economic benefits.

- Added Flexibility

  By allowing you to debug several tasks while the system continues to run in real time, the Debugger lets you check out new tasks in a running system. This simplifies the process of adding new tasks to an existing application system.

- Economic Benefits

  By simplifying the process of debugging the interplay between tasks, the Debugger lessens the amount of time needed to debug your application system. This directly reduces the time to market, the cost of implementation, and the cost of maintenance.

BOOTSTRAP LOADING

The iRMX 86 Operating System contains a bootstrap loader that allows your application system to reside on disk and be loaded into RAM (random-access memory).

EXPLANATION OF BOOTSTRAP LOADER

A bootstrap loader is a program that resides in ROM on your application hardware. When your iAPX 86 microprocessor is reset the bootstrap loader receives control and loads the rest of the software, including the iRMX 86 Operating System and the application software.

ADVANTAGES OF A BOOTSTRAP LOADER

The iRMX 86 Bootstrap Loader provides your application system with two major advantages:

- Reduced Manufacturing Costs

  By placing the iRMX 86 Bootstrap Loader in ROM, you can shift the rest of your application system to RAM. Since the rest of your system is probably one or two orders of magnitude larger than the Bootstrap Loader, this displacement substantially decreases the amount of ROM required to implement your application.

  This decrease in the amount of ROM required for your application leads directly to reduced manufacturing costs. ROM, unlike RAM, requires that information be "burned" or masked into memory. By decreasing the amount of ROM in your system, the Bootstrap Loader reduces your masking or "burning" expenses.

- Reduced Software Maintenance Costs

  The iRMX 86 Boostrap Loader simplifies the process of providing updated software to your customers. Rather than shipping ROMs containing the modified software, you can ship diskettes. This greatly reduces the cost of updating your software.

CONFIGURABILITY

The iRMX 86 Operating System is configurable. By allowing you to select only the parts of the Operating System that you need, configurability reduces the amount of memory required for your application system.

EXPLANATION OF CONFIGURABILITY

A system is configurable if you can select the pieces of it that you want and discard the pieces that you don't want. For example, Figure 4-5 shows a system that consists of six parts. During the process of configuration, you select the desired parts and combine them to form the system.

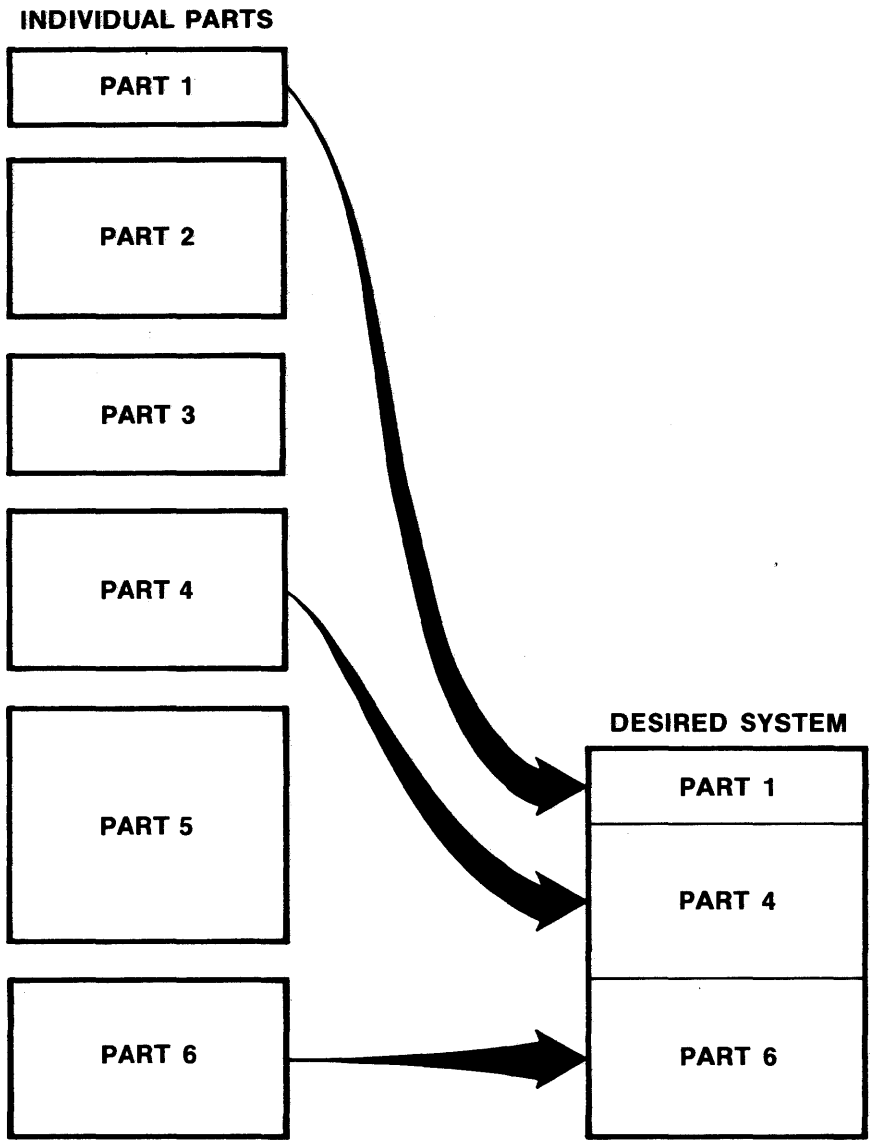

Figure 4-5. Configuration of a hypothetical system

When you configure an application system that is based on the iRMX 86 Operating System, you have two objectives. First, you select the parts of the iRMX 86 Operating System that your application system needs. And second, you combine those parts with your application software to form the complete application system. These two objectives are depicted in Figure 4-6.

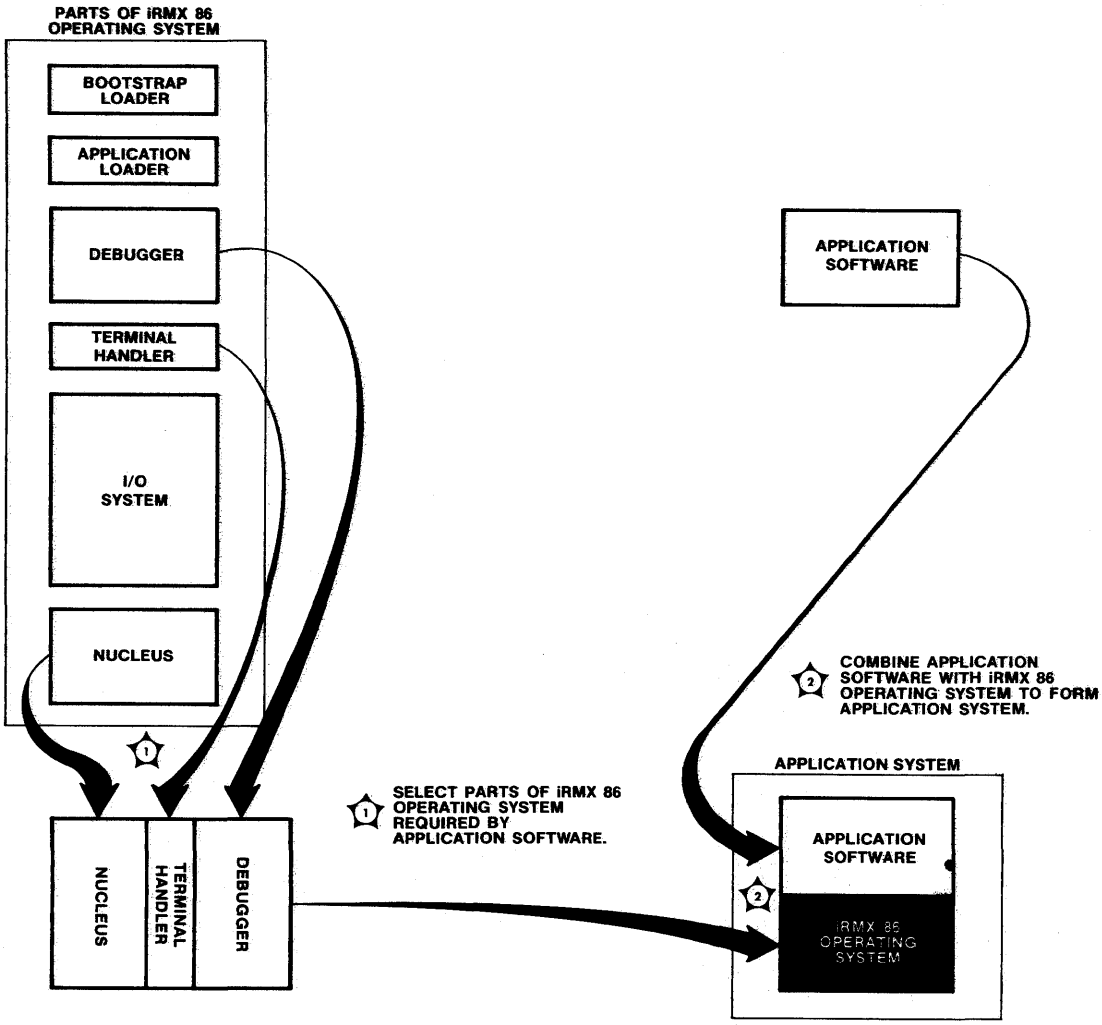

Figure 4-6. The dual objective of iRMX 86™ configuration

The iRMX 86 Operating System consists of six major parts. During the process of configuration you must specify which of these parts are to be included in your application system. The six parts are:

- The Nucleus

  The Nucleus is the heart of the iRMX 86 Operating System. All of the other pieces of the Operating System use the Nucleus, so it must be included in every application system built upon the iRMX 86 Operating System.

- The I/O System

  The I/O System provides file management and the device-independent interface to input and output devices. The I/O System is an optional component of the iRMX 86 Operating System, so it can be excluded from the Operating System if it is not needed.

- The Terminal Handler

  The Terminal Handler is the software interface for the terminal. It can be used alone or through the device-independent interface of the I/O System. Like the I/O System, the Terminal Handler is an optional component and can be left out of the Operating System if it is not required.

- The Debugger

  The Debugger is also an optional component of the iRMX 86 Operating System. While the application system is being developed, the Debugger is a very useful tool. By including it in your system during the development period, you can take advantage of its powerful capabilities. Then, once development is completed, you can remove the Debugger and reduce the size of your finished application system.

- The Application Loader

  The Application Loader allows your application to load programs from disk into main memory. This part of the iRMX 86 Operating System is optional but, if included, requires the I/O System.

- The Bootstrap Loader

  The Bootstrap Loader is an optional component of the iRMX 86 Operating System. The Bootstrap Loader reads your application system from disk into main memory whenever you reset the hardware of the application system.

ADVANTAGES OF CONFIGURABILITY

Figure 4-6 shows the advantage of configurability. In this figure, an iRMX 86 Operating System consisting only of the Nucleus, Terminal Handler, and Debugger is being combined with application software. By excluding from your finished product the subsystems of the iRMX 86 System that you don't need, you reduce the amount of memory needed by your system.

CHAPTER PERSEPCTIVE

In this chapter we discussed some features of the iRMX 86 Operating System. We also saw some of the advantages that each feature lends your application system. Next we'll see how some of these features work together.

CHAPTER 5. A HYPOTHETICAL SYSTEM

In the previous chapter, you were shown some of the features of the iRMX 86 Operating System. The features were discussed individually. This chapter revisits some of these features using a hypothetical system to show you how features combine to form a powerful environment for your application software.

During the following discussion, a hypothetical application system is used to illustrate the relationship between your application software and the iRMX 86 Operating System. The system monitors and controls a variable number of kidney machines in a hospital. These machines remove toxins from the blood of patients whose kidneys are not functioning correctly.

The system, which is portrayed in Figure 5-1, consists of three main hardware components.

- Intel iSBC 86 Single Board Computer

  The single board computer provides the intelligence for the entire system. It contains the software to monitor and control the function of all the machines in the system.

- Bedside Units

  One of these units is located at the side of each patient's bed. Connected by cable to the iSBC 86 Single Board Computer, each of these units performs four distinct functions:

  - Measuring the level of toxins in the blood as the blood enters the unit.

  - Displaying information so medical personnel at the bedside can monitor the dialysis process.

  - Accepting commands from the bedside personnel.

  - Removing toxins from the blood.

  Each bedside unit performs these functions under the control of the single board computer. That is, commands and measurements are sent to the single board computer which then adjusts the rate of dialysis and generates the bedside display.

- Master Control Unit

  The system's master control unit consists of a terminal with a screen and a keyboard. This terminal, which allows one individual to monitor and control the entire system, operates under control of the single board computer.

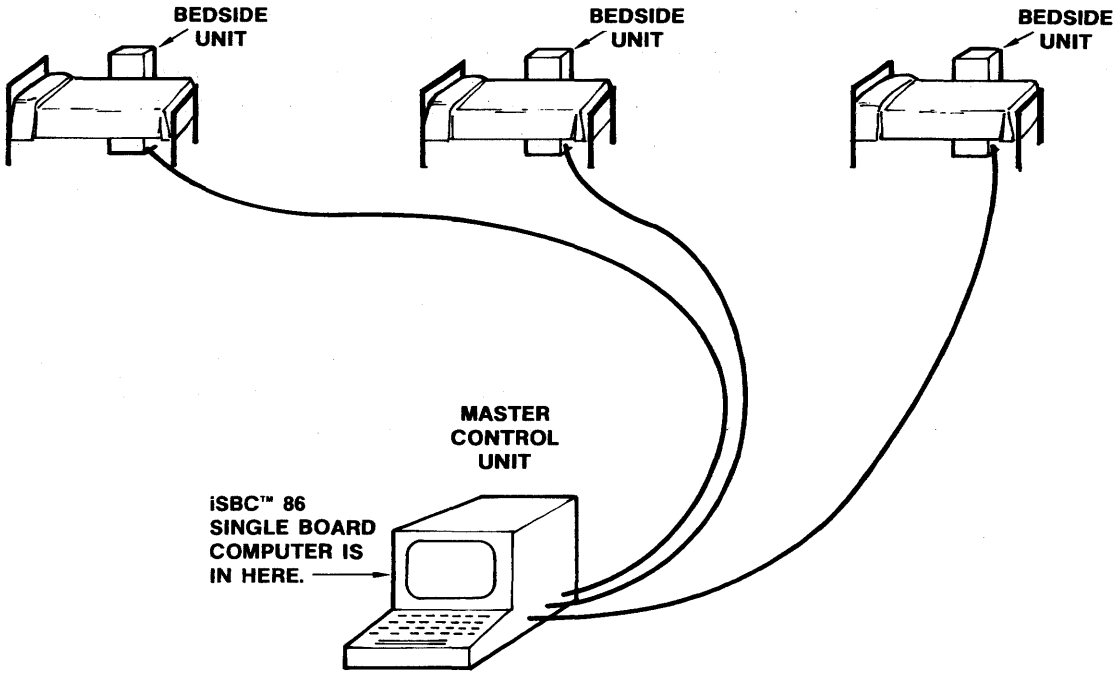

Figure 5-1. The hardware of the dialysis application system

So, in summary, the system consists of one master control unit and a variable number of bedside units, all operating under control of the software within an Intel iSBC 86 Single Board Computer. Now let's look at the software.

The application software controls the dialysis process. In order to do this, the software must:

- Obtain commands from the master control unit.
- Obtain commands (if there are any) from each of the active bedside units.
- Reconcile the commands from the master control unit and the commands of the active bedside units.

- Obtain a toxicity level from each of the active bedside units.
- Create a display at each active bedside unit.
- Create a display at the master control unit.
- Control the rate of dialysis at each of the active bedside units.

Now that we have roughly examined the nature of the system, let's investigate how the iRMX 86 Operating System fits in. Let's start with interrupt processing.

INTERRUPT PROCESSING

Two kinds of information flow from the bedside units to the single board computer -- commands and toxicity levels. Before we delve into the technique used to process this information, we must know more about the form of the information.

The toxicity levels, measured as the blood enters the bedside unit, are not subject to violent change. The machine slowly removes toxins from the blood while the patient's body, even more slowly, puts toxins back in. The result is a rather steadily declining toxicity level.

This means that the toxicity levels must be monitored regularly, but not too frequently. Let's suppose that each bedside unit computes the toxicity levels once every ten seconds and sends a signal when the computation is complete. When the signal line goes high, the levels are available until the signal line goes low and then high again for the next computation.

The command information changes less predictably than the toxicity levels. Persons at the patient's bedside can enter commands through the bedside unit. Let's suppose that after encoding the information they press a button labeled ENTER, and that this button sets a line high. When the line goes high, the command information is available until the ENTER button is pressed for the next command.

Now let's see how the interrupt processing of the iRMX 86 Operating System fields the commands. (The toxicity levels can be fielded in precisely the same manner so, for the sake of brevity, they are not discussed.) By attaching all the signal lines to a Multibus interrupt line, we convert the signal into an interrupt level. Each interrupt level has an interrupt task that is executed when the level goes high. So, when the ENTER line from any bedside unit goes high, the interrupt task for bedside commands begins running.

You must write the interrupt tasks for your system's custom devices, so the bedside-command task may serve as an example for you. In brief, the task performs the following steps.

- It determines which bedside unit received the command.
- It puts the command information, along with the number of the bedside unit that received the command, into a message.
- It sends the message to a predetermined mailbox. The only task that waits at this mailbox is the task that reconciles bedside commands with the commands from the master control unit.
- It surrenders the processor to the iRMX 86 Operating System.

One advantage of interrupt processing now becomes clear. Instead of wasting time polling the bedside units to see if a command has been issued, the application system can do other things until interrupted by one of the units. When an interrupt (an event) does occur, it is quickly converted into a message and is placed into a mailbox for processing by a task. The system then returns to its normal priority-based, preemptive scheduling. This technique enables your system to deliver more throughput.

Interrupt processing also provides the application system with flexibility. For instance, you can add more bedside units without modifying the system's software at all.

TERMINAL HANDLER

The iRMX 86 Terminal Handler simplifies the process of fitting the master control unit into the application system. Rather than dealing with the mechanics of a terminal, your tasks wait at a mailbox for input information and send output information to the output mailbox. Alternatively, your tasks can use the Terminal Handler through the I/O System. This latter technique preserves device independence.

MULTITASKING

The entire application system is based on the multitasking capability of the iRMX 86 Operating System. Tasks are run using the preemptive, priority-based, scheduling that was discussed in Chapter 4. This allows the more important tasks (such as those controlling the bedside units) to preempt lower priority tasks (such as those of the Terminal Handler).

INTERTASK COORDINATION

The only form of intertask coordination used in our hypothetical dialysis system is intertask communication. The system uses a number of mailboxes to send information from one task to another. The simplicity of mailboxes allows engineers to divide the system into tasks on the basis of modularity, rather than on the basis of minimizing intertask communication.

MULTIPROGRAMMING

Although multiprogramming has not yet been of use in our hypothetical example, its potential is high. Suppose that we extend the example to include cardiac monitoring in addition to dialysis. The two functions could advantageously be performed in different jobs. Why? Because they need share very few resources.

If the cardiac application has very little to do with the kidney application, there is no need for them to share mailboxes, tasks, or any other objects. By splitting them into two different jobs, there is less chance that one application can affect the environment of the other.

But what happens if the two applications need to share only a little information? How can the shared data be passed from one job to another without losing the benefits of isolation? The iRMX 86 Operating System provides for this contingency in its implementation of runtime binding.

RUNTIME BINDING

As mentioned earlier in this manual, runtime binding provides a means for tasks of different jobs to share objects. As tasks create objects to be shared, the tasks catalog the objects in an object directory. Then the tasks that need the objects can look them up by using their ASCII names.

Runtime binding also allows you to change the configuration of the iRMX 86 Operating System without recompiling or relinking your application software. For instance, suppose you have been including the iRMX 86 Debugger in systems delivered to your customers. The advantage in doing this is that it allows some debugging on systems as they are being used. But now, a year or so after you started delivering systems, your product has stabilized. Virtually no new bugs are being found. If you delete the Debugger from your system, you can reduce the amount of memory required in any new systems you sell. The runtime binding of the system to your application software allows you to remove the Debugger from your system without making any changes to your application software.

MASS STORAGE FILES

As specified, the hypothetical system does not require mass storage files. However, a very reasonable extension of the current specification could include recording information about patients.

The iRMX 86 I/O System allows you to record information in files on flexible and hard disks. The system provides device handlers, disk formatting and allocating, and the means of actually moving information between main memory and the disk. Your application software need not include code to perform these functions.

DEVICE INDEPENDENCE

Even if the application system uses mass storage devices, device independence is not necessarily required. But, if the application is extended to allow the operator at the MCU to direct recorded data to any of several devices (say teletypewriter, line printer, magnetic tape or disk), device independence becomes more important. The device-independent I/O System lets you implement recording without adding code specific to each possible device.

CHAPTER PERSPECTIVE

In this and the previous chapters, you were introduced to some of the features and benefits associated with the iRMX 86 Operating System. If you want more detailed information, you will find the next chapter very useful. It contains descriptions of all the iRMX 86 technical manuals.

CHAPTER 6. iRMX 86™ LITERATURE

Nine manuals relating to the iRMX 86 Operating System are currently available. They are:

- INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM
  Manual no. 9803124
- iRMX 86 NUCLEUS, TERMINAL HANDLER, AND DEBUGGER REFERERNCE MANUAL
  Manual no. 9803122
- iRMX 86 I/O SYSTEM AND LOADER REFERENCE MANUAL
  Manual no. 9803123
- iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL
  Manual no. 142721
- iRMX 86 INSTALLATION GUIDE FOR ISIS-II USERS
  Manual no. 9803125
- iRMX 86 CONFIGURATION GUIDE FOR ISIS-II USERS
  Manual no. 9803126
- iRMX 86 PROGRAMMING TECHNIQUES
  Manual no. 142982
- iRMX 86 POCKET REFERENCE
  Manual no. 142861
- GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 I/O SYSTEM
  Manual no. 142926

Each of these manuals is designed for a well-defined set of readers and each has a different purpose. This chapter describes the readers at whom each manual is aimed and the purpose of each manual. (Table VI-I correlates features with manuals.) Also, the chapter provides some time-saving tips to bear in mind as you read the documentation.

TABLE VI-I

CORRELATION OF MANUALS AND FEATURES

TITLE	FEATURE
iRMX 86 Configuration Guide for ISIS-II Users	Configurability
iRMX 86 Nucleus, Terminal Handler and Debugger Reference Manual	Object-Oriented Architecture Multiprogramming Multitasking Interrupt Processing Preemptive, Priority-based Scheduling Error Handling Dynamic Memory Allocation Intertask Coordiation Runtime Binding Terminal Handling Object-Oriented Debugger
iRMX 86 I/O System And Loader Reference Manual	Device-Independent I/O Hierarchical Naming of Mass Storage Files File Access Control Control of File Fragmentation Applicaton Loading
iRMX 86 System Programmer's Reference Manual	Intertask Coordination Expandibility Selection of Device Drivers Runtime Binding File Access Control Device Independence Bootstrap Loading
Guide to Writing Device Drivers for the iRMX 86 System	Selection of Device Drivers

The following descriptions deal with engineers in two classes -- system programmers, and application programmers. System programmers are responsible for configuring the system, extending the Operating System, writing interrupt handlers, and performing other functions that affect the entire application system. Application programmers, on the other hand, are responsible for writing application software. This distinction is drawn because the actions of the system programmer have a more global affect.

Specifically, some system calls can, if used improperly, cause problems for all the tasks in your system; other system calls can affect only the task invoking the call. As a matter of policy, the more powerful system calls should be used only by system programmers and, even then, only within Operating System extensions. To emphasize this distinction, the more powerful system calls are all explained in one manual, the SYSTEM PROGRAMMER'S REFERENCE MANUAL.

The following sections describe the iRMX 86 manuals.

INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM

This manual, the one you are presently reading, is aimed at a wider variety of readers than any of the other iRMX 86 manuals. Being an introduction, it can be understood by anyone who has some experience programming or managing programming projects. It is designed to introduce managers and engineers to the iRMX 86 Operating System.

iRMX 86 NUCLEUS, TERMINAL HANDLER, AND DEBUGGER REFERENCE MANUAL

This reference manual is written for any engineers planning to use the iRMX 86 Operating System.

It is the information warehouse for the Nucleus, the Terminal Handler, and the Debugger. It contains concise but detailed discussions of the following topics:

- The nature of objects in general and of tasks, jobs, semaphores, mailboxes, and segments in particular.
- Error processing.
- Interrupt processing.
- The requirements and behavior of the Nucleus system calls available to all engineers.
- How to use the Terminal Handler directly (without the I/O System), from both the programming and the operating points of view.
- The requirements and behavior of the Debugger commands available to all engineers.

Each of the foregoing topics is presented in a manner suitable for reference purposes and, to a lesser degree, for instructional purposes.

iRMX 86 I/O SYSTEM AND LOADER REFERENCE MANUAL

The iRMX 86 I/O System is distinct from, but built upon, the iRMX 86 Nucleus. The I/O System provides device independence, hierarchical naming of mass storage files, control over fragmentation of files, and file access control. The iRMX/86 I/O SYSTEM AND LOADER REFERENCE MANUAL tells you how to use these features. It also tells you how to load programs from disk to memory while the system is running.

The manual is aimed at any engineer who is familiar with the information presented in the iRMX 86 NUCLEUS, TERMINAL HANDLER, AND DEBUGGER REFERENCE MANUAL.

iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL

This manual describes aspects of the iRMX 86 Operating System that should be used only within extensions of the iRMX 86 Operating System. If these features are used by application software that has not been incorporated into the Operating System, your application system will not be compatible with future releases of the iRMX 86 Operating System. Also, readers of the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL must be familiar with the information presented in the iRMX 86 NUCLEUS, TERMINAL HANDLER, AND DEBUGGER REFERENCE MANUAL and the iRMX 86 I/O SYSTEM AND LOADER REFERENCE MANUAL.

This reference manual discusses the following topics:

- The creation and deletion of extensions to the Operating System.
- Region exchanges and related system calls.
- Enabling and disabling the deletion of the objects.
- Attaching and detaching devices.
- Creation and deletion of user objects.
- Adding new types of objects to the Operating System.
- Using the Bootstrap Loader

The foregoing topics are presented in a manner suitable for reference and, to a lesser degree, for instruction.

iRMX 86 INSTALLATION GUIDE FOR ISIS-II USERS

The INTELLEC Microcomputer Development System is a general purpose tool for programming microcomputers. When you purchase the iRMX 86 Operating System, you receive the iRMX 86 software on several flexible disks and you receive the iSBC 957A package. The iRMX 86 INSTALLATION GUIDE FOR ISIS-II USERS tells you how to use the software and the iSBC 957A package in conjunction with your Microcomputer Development System and an iSBC Single Board Computer. These four elements form the development environment for your application system.

This manual is designed to lead the reader, step by step, through the process of adding the iRMX 86 software and hardware to the INTELLEC system. If you are not responsible for performing this addition, you can gain very little from reading this manual. If, on the other hand, you are responsible for the installation and you are familiar with the ISIS-II Operating System, this manual will prove very useful.

iRMX 86 CONFIGURATION GUIDE FOR ISIS-II USERS

As you build an application system upon the iRMX 86 Operating System, you must decide which optional iRMX 86 features you want in your system. For instance, if your system uses no features of the I/O System, you can save a substantial amount of memory by excluding the iRMX 86 I/O System.

Once you have made these decisions and have written your application software, you must configure your system. Configuration is the process of building a complete system from the iRMX 86 Nucleus, your application software, and iRMX 86 options that you have selected. The iRMX 86 CONFIGURATION GUIDE FOR ISIS-II USERS leads you through the process of configuration.

The readers of this manual must be thoroughly familiar with the ISIS-II operating system and the language translators being used to compile the application software. Furthermore, they must know which parts of the iRMX 86 System are used by the application software, and they must be fluent in the vocubulary of the iRMX 86 Nucleus, Terminal Handler, Debugger, I/O System, Bootstrap Loader, and Application Loader.

iRMX 86 PROGRAMMING TECHNIQUES

This manual provides system and application programmers with techniques that can save time during system development.

iRMX 86 POCKET REFERENCE

This manual summarizes the information contained in the iRMX 86 NUCLEUS, TERMINAL HANDLER AND DEBUGGER REFERENCE MANUAL and the iRMX 86 I/O SYSTEM AND LOADER REFERENCE MANUAL.

GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 OPERATING SYSTEM

This manual gives detailed instructions for writing a device driver that is compatible with the iRMX 86 I/O System. System programmers can use this manual to add new devices to application systems. Readers of this manual must be very familiar with the I/O System and the Nucleus.

READING TIPS

The following pointers can save you a substantial amount of time:

- No one individual need become intimately familiar with all of the documents associated with the iRMX 86 Operating System. Read only the documents that relate to your responsibilities.
- Before reading one of the documents, read its preface and scan its table of contents to see if the manual contains the kind of information you seek.
- Read the Introductory Manual before reading any of the others.

By following these tips, you can quickly focus your attention on the information that is of most value to you.

INDEX

For your convenience, the primary reference of each multiple-page topic is underscored.

REQUEST FOR READER'S COMMENTS

Intel Corporation attempts to provide documents that meet the needs of all Intel product users. This fo you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestio improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of docume needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

Please check here if you require a written reply. ☐

'D LIKE YOUR COMMENTS . . .

document is one of a series describing Intel products. Your comments on the back of this form help us produce better manuals. Each reply will be carefully reviewed by the responsible on. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
3585 S.W. 198th
Aloha, Oregon 97005

.M.S. Technical Publications

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA 95051 (408) 987-8080

Printed in U.S.A.