intel®

iRMX™ 86 INTRODUCTION AND OPERATOR'S REFERENCE MANUAL

For Release 6

iRMX™ 86 OPERATING SYSTEM

Order Number: 146194-001

ASSEMBLY INSTRUCTIONS

Volume: iRMX™ 86 INTRODUCTION AND OPERATOR'S REFERENCE MANUAL
Order No: 146194

INTRODUCTION

This sheet describes how to assemble this iRMX 86 literature packet. The assembly is simple and takes less than 5 minutes.

This literature packet contains:

- The literature in the volume, including this instruction sheet and these manuals:

    - INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM
    - iRMX 86 OPERATOR'S MANUAL
    - iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL

- The first of two cardboard separators.

- Three divider tabs, one for each manual.

- The bottom cardboard separator.

If your literature packet is missing one or more of these items, contact Intel immediately.

ASSEMBLY

Assembling the volume involves inserting the literature packet into a three-ring binder and placing an appropriately labeled divider tab at the front of each manual in the volume.

At this point you have torn open the shrink wrapping, removed the entire literature packet, and extracted this sheet from the packet. Set this sheet aside. You will be referring to it as you go.

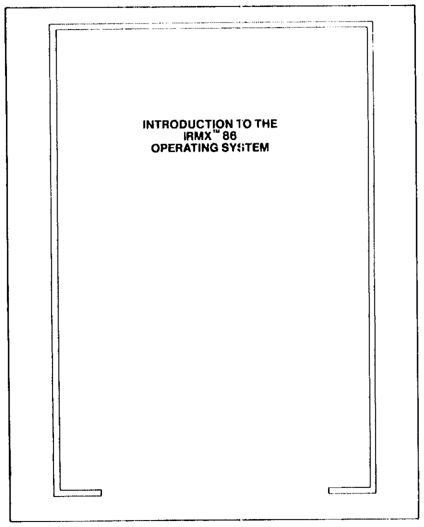
To put the volume together, follow these steps:

1. Separate the divider tabs from the rest of the literature packet. Tear off the shrink wrapping. Discard the cardboard. The divider tabs have these labels and match these manuals:

Label	Manual
Introduction	INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM
Operator	iRMX 86 OPERATOR'S MANUAL
Disk Verify	iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL

(over)

2. Find Page ix, which is at the end of the Volume Contents. Open the binder rings and insert the Front Cover up to and including Page ix into the left side of the open rings. The top page of the literature packet is now the "Introduction" title page, which looks like this:


INTRODUCTION TO THE
iRMX™ 86
OPERATING SYSTEM

3. Insert the divider tab labeled "Introduction" into the left side of the open rings.

4. Insert the text of the Introduction manual into the left side of the binder rings. The last page of the Introduction manual is "Introduction Index-4." The top page of the literature packet should now be the title page of the Operator's manual.

5. Repeat the process for the remaining manuals, matching divider tabs with manuals.

6. Close the binder rings. Discard the shrink wrapping and this instruction sheet.

***

iRMX™ 86 INTRODUCTION AND OPERATOR'S REFERENCE MANUAL

For Release 6

Order Number: 146194-001

REV.	REVISION HISTORY	DATE
-001	Original Issue. Supplies and updates information formerly contained in the *Introduction to the iRMX 86 Operating System,* the *iRMX 86 Operator's Manual,* and the *iRMX 86 Disk Verification Reference Manual.*	3/84

VOLUME PREFACE

This volume, the iRMX 86 INTRODUCTION AND OPERATOR'S REFERENCE MANUAL, contains introductory and operating information about the iRMX 86 Operating System.

MANUALS IN THIS VOLUME

This section briefly describes each iRMX 86 manual in the order they appear in this volume.

INTRODUCTION TO THE iRMX™ 86 OPERATING SYSTEM

Tab Label: Introduction

This manual is designed to introduce engineers and managers to the iRMX 86 Operating System. This manual describes in general terms the most important characteristics of the iRMX 86 Operating System.

iRMX™ 86 OPERATOR'S MANUAL

Tab Label: Operator

This manual describes the iRMX 86 Operating System commands. Introductory material discusses command line editing, iRMX 86 pathnames, wild cards, and other material necessary to use commands from a keyboard terminal. Also, the manual describes how to use the Files Utility.

iRMX™ DISK VERIFICATION UTILITY REFERENCE MANUAL

Tab Label: Disk Verify

This manual documents the iRMX 86 Disk Verification Utility, which can be used to check the file structure of an iRMX 86 volume. The manual also contains a detailed description of the iRMX 86 file structure.

iRMX™ 86 PUBLICATIONS

Because the iRMX 86 documentation set is packaged in bound volumes, you can no longer order manuals individually. Instead, you must order a complete volume of text to get a manual contained in that volume. (Individual manuals no longer have order numbers.)

When ordering individual volumes, you can order the binder, spine card, and literature packet together as a unit or separately. If you wish to order a volume as a unit, use the "order" number that appears on the spine of the binder. This number is also provided in the following list. If you wish to order separate pieces of the volume (e.g., the literature packet only), use the "part" number as labeled on the piece. If you don't know the part number, consult the Intel Literature Guide.

The following list shows volume titles, order numbers, and individual manuals in each of the volumes. Manuals are listed in the order they appear in the volumes. This volume is indicated by boldface type.

1. **iRMX™ 86 INTRODUCTION AND OPERATOR'S REFERENCE MANUAL**
   Order Number: 146545

   - **Introduction to the iRMX™ 86 Operating System**
   - **iRMX™ 86 Operator's Manual**
   - **iRMX™ 86 Disk Verification Utility Reference Manual**

2. iRMX™ 86 PROGRAMMER'S REFERENCE MANUAL, PART I
   Order Number: 146546

   - iRMX™ 86 Nucleus Reference Manual
   - iRMX™ 86 Basic I/O System Reference Manual
   - iRMX™ 86 Extended I/O System Reference Manual

3. iRMX™ 86 PROGRAMMER'S REFERENCE MANUAL, PART II
   Order Number: 146547

   - iRMX™ 86 Application Loader Reference Manual
   - iRMX™ 86 Human Interface Reference Manual
   - iRMX™ 86 Universal Development Interface Reference Manual
   - Guide to Writing Device Drivers for the iRMX™ 86 and iRMX™ 88 I/O Systems
   - iRMX™ 86 Programming Techniques
   - iRMX™ 86 Terminal Handler Reference Manual
   - iRMX™ 86 Debugger Reference Manual
   - iRMX™ 86 Crash Analyzer Reference Manual
   - iRMX™ 86 System Debugger Reference Manual
   - iRMX™ 86 Bootstrap Loader Reference Manual

4. iRMX™ 86 INSTALLATION AND CONFIGURATION GUIDE
   Order Number: 146548

   - iRMX™ 86 Installation Guide
   - iRMX™ 86 Configuration Guide
   - Master Index for Release 6 of the iRMX™ 86 Operating System

RELATED PUBLICATIONS

- iAPX 86,88 Family Utilities User's Guide, Order Number: 121616
- iMMX™ 800 MULTIBUS® Message Exchange Reference Manual, Order Number: 144912

VOLUME CONTENTS

***

INTRODUCTION TO THE iRMX™ 86 OPERATING SYSTEM

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, then uses these terms to define specific and new terms.

Throughout this manual, the expression "iAPX 86,88,186,188,286-based microcomputer" is used to refer to any microcomputer that uses the Intel iAPX 86, 88, 186, 188, or 286 microprocessor as its central processing unit.

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, and defines terms used in later chapters.

- Chapter 2 - Considerations Relating to Real-Time 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 real-time, 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 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.

CONTENTS

PAGE

PAGE

FIGURES

***

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 86,88,186,188,286 Single Board Computers and with other iAPX 86,88,186,188,286-based microcomputers. The Operating System is different from many other operating systems in that 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 occurring outside the single board computer.
- It can communicate with a wide variety of input, output, and mass storage devices.
- It can execute on all members of the iAPX 86,88,186,188,286 microprocessor family.
- It provides a powerful and flexible means for an operator to observe and modify the behavior of the system.
- It provides a base upon which to run a number of languages and other software tools.

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) and Volume End Users (VEUs). OEMs are companies that build products for resale. 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.

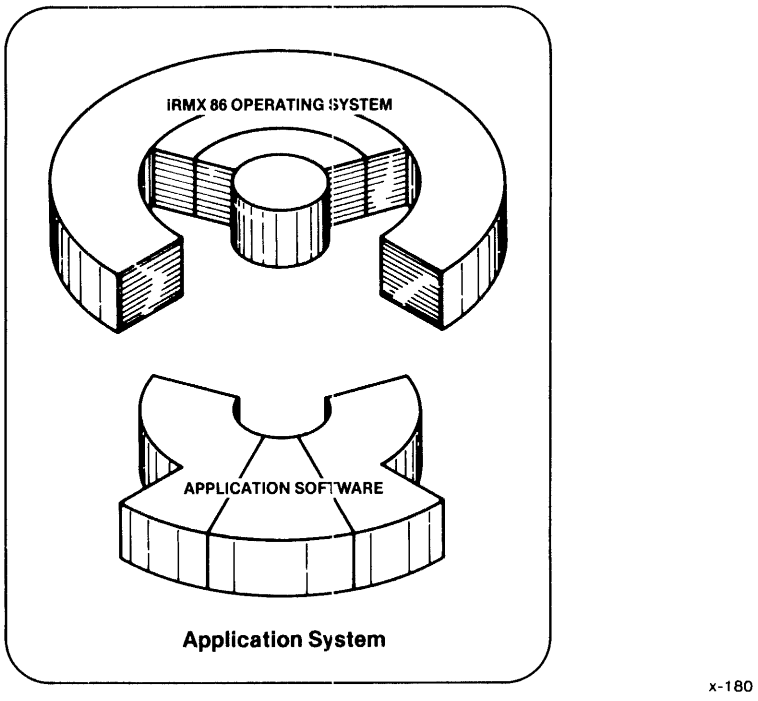


Figure 1-1. The iRMX™ 86 Foundation for Application Systems

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.

***

CHAPTER 2 CONSIDERATIONS RELATING TO REAL-TIME SOFTWARE

The difficulties encountered in real-time programming differ from those found in other types of programming. This chapter briefly introduces some of the problems that face designers of real-time systems.

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

Real-time 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 without affecting the previous 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 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 real-time events, an error is detected. How can the error be corrected, or how can its impact be limited, without adversely affecting the system? The whole system, for instance, should not be shut down merely because an error is detected; it should be able to recover from these errors and continue processing.

DEVICE SENSITIVITY

Many real-time 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.

MASS STORAGE FILE ALLOCATION TRADEOFFS

In any real-time system, file allocation performance is an important consideration. One factor that relates directly to mass storage file allocation performance is the size of each contiguous chunk of data written to and read from a file (the file's "granularity"). In some applications, large granularity 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 trade-off 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 eliminating unneeded features. Because operating systems are complex, the method used to select features should be "human engineered," so that the process is efficient and relatively easy to understand.

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 than the sum of the maximum amounts required by each application.

FILES AND MULTIPLE USERS

Some applications, such as data-entry and database-management systems, support more than one user at a time. In such systems, three major problems must be dealt with.

The first problem pertains to file naming. 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 coined by other users.

The second problem deals with selective sharing of files. Multi-user systems often must be able to share and protect files. For instance, in a data-entry system, one operator may be entering data while another simultaneously verifies the entered data. This illustrates the need for file sharing. 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.

The third problem is that the Operating System must be able to respond simultaneously to more than one terminal. The system must respond quickly to each terminal, and must be able to keep track of tasks and other resources associated with a particular terminal.

HUMAN ENGINEERING

Applications must be controlled by people. Systems often contain critical processes that operators must control with a minimum chance of error. An application system should provide an easily-understood set of interactive commands and messages by which operators may use the system.

APPLICATION DEVELOPMENT

Frequently the hardware on which an application system will be installed includes mass storage devices and file structures. If possible, the operating system should allow application system development using existing hardware. This means that you should be able to use language processors (such as assemblers, compilers, and run-time support systems), linking utilities, editors, and file maintenance utilities. Programmers should be able to install such development tools on the operating system quickly and easily.

DEBUGGING

Real-time application systems require real-time debugging support. Often, logic errors or "bugs" in real-time systems are dependent upon events in the real world (outside of the computer). In order to detect some real-time logic errors, the system should continue to run while you debug it. This type of debugging is called "dynamic" debugging.

If the system crashes, programmers must be able to gather enough information to analyze the cause of the crash. In addition, sometimes it is useful for programmers to "freeze" the system and examine its state. This type of debugging is called "static" debugging.

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 real-time application system. As an OEM manager, you are interested in developing your application system quickly using the latest Very Large Scale Integration (VLSI) technology, 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.

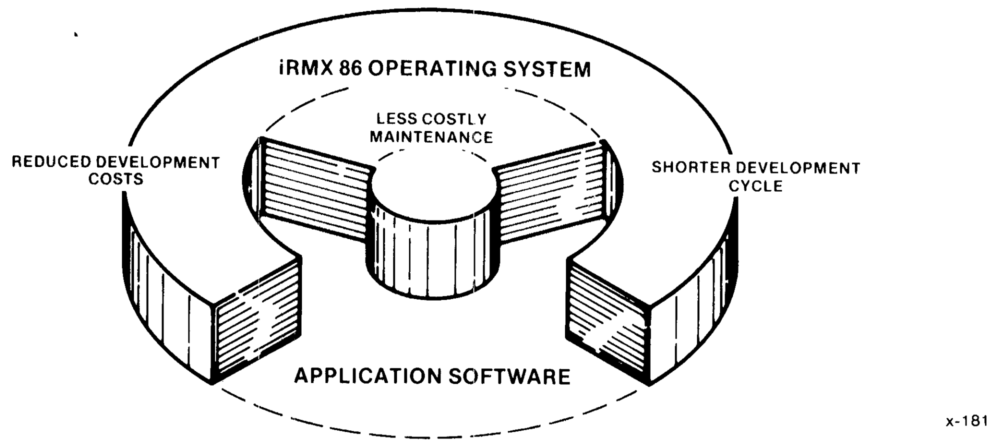


Figure 3-1. The iRMX™ 86 System Provides Economic Benefits

DEVELOPMENT TIME

The iRMX 86 Operating System helps you develop real-time application systems quickly. Acting as the foundation for your specialized application software, the iRMX 86 Operating System provides services that are required by many real-time 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 real-time 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 the following ways:

- By supplying the general services required by many real-time applications, the iRMX 86 System reduces your manpower requirements.
- Industry-standard languages are available for use with the iRMX 86 Operating System. These languages are the same ones used on your Intellec Microcomputer Development System.
- The features of the Operating System simplify the process of development. These features, such as object-oriented architecture and device independence, are discussed in Chapter 4.
- Support for VLSI devices is available now, which results in immediate improvements in speed and performance.

COSTS AFTER DEVELOPMENT

After your application system is developed, your major expense is maintenance -- the process of correcting logic errors, making changes, and adding features. The iRMX 86 Operating System helps minimize these costs in the following ways:

- 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.
- When errors do reveal the presence of bugs in your application software, the iRMX 86 System provides tools to help find the errors. These tools include error handlers, an on-line dynamic debugger, a static system debugger, and a crash analyzer. These tools are described in Chapter 4.
- The modularity provided by multiple jobs and tasks lets you make changes and additions without severely affecting the system's overall design.

CHAPTER PERSPECTIVE

The iRMX 86 Operating System is your economic ally. It helps you put your real-time application system in the hands of your users in less time and at less expense. It also allows you to use the latest improvements in VLSI technology while reducing 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 features of the iRMX 86 Operating System (see Figure 4-1).

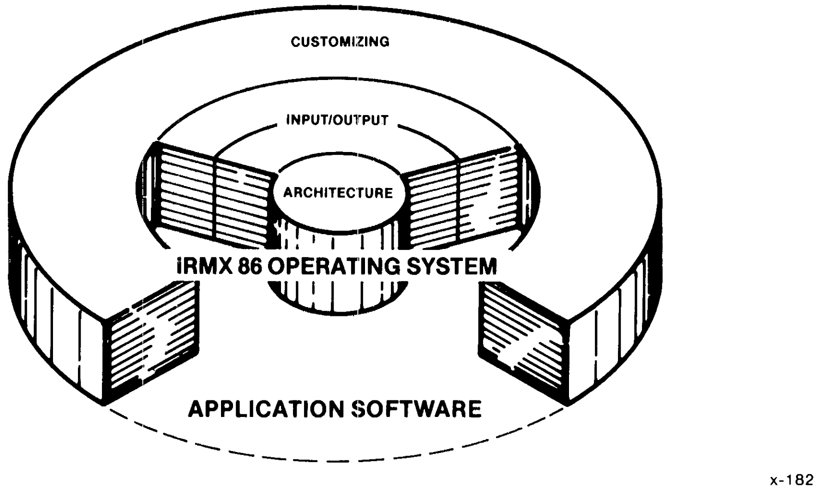


Figure 4-1. Features of the iRMX™ 86 Operating System

The features described in this chapter are:

ARCHITECTURAL FEATURES

- Object-Oriented Architecture
- Multitasking
- Interrupt Processing
- Preemptive Priority-Based Scheduling
- Multiprogramming
- Intertask Coordination
- Extendibility
- Debugging Support
- Processor Selectivity

FEATURES OF THE iRMX™ 86 OPERATING SYSTEM

INPUT/OUTPUT FEATURES

- Choice of I/O Systems
- Device-Independent Input and Output
- Hierarchical Naming of Mass Storage Files
- File Access Control
- Control over File Fragmentation
- Selection of Device Drivers
- Terminal Support Code

CUSTOMIZING FEATURES

- Custom Interactive Commands
- Application Loading
- Run-Time Binding
- Simultaneous Multiple Terminal Support
- Error Handling
- Dynamic Memory Allocation
- Software Interface
- Bootstrap Loading

TOOLS

- Object-Oriented Dynamic Debugger
- System Debugger
- Crash Analyzer
- Installation Systems
- On-Target Development
- Interactive Configuration Utility (ICU)
- File Maintenance Programs

Because you may be familiar with some features, each section is organized for easy skimming as follows:

1. **A brief introduction to the feature (in this typeface).**

2. A detailed and more technical explanation of the feature.

3. **The advantages of the feature (in this typeface).**

ARCHITECTURAL FEATURES

When Intel software engineers designed the iRMX 86 Operating System, they specified the basic processes and data structures of the system, including such characteristics as the partitioning of programs into "tasks," task scheduling, and task communication. These characteristics are referred to as the "architecture" of the system. The important architectural features of the Operating System are described here.

OBJECT-ORIENTED ARCHITECTURE

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

An operating system is a collection of software that is meant to be used by software engineers. Many non-object-oriented operating systems are overly complex and difficult to understand. 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 less intimidating.

In other words, an <u>object-oriented architecture</u> 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.

The building blocks of the iRMX 86 Operating System are called <u>objects</u> 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. Also, 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.

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 real-time events.

The essence of real-time 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.

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.

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. 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.

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 than monolithic code. Modular code is less costly to develop and maintain, and it can be developed more quickly than monolithic 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.

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. Non-preemptive scheduling allows a task to run until it relinquishes the processor. Even if a higher-priority task becomes ready for execution, 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.

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 without abandoning the less-important events.

MULTIPROGRAMMING

Multiprogramming provides your system with the ability to run more than one application on a single iAPX 86,88,186,188,286-based microcomputer. This helps reduce hardware costs.

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.

Multiprogramming provides your application system with two benefits:

- **Multiprogramming increases the amount of work your system can do. By utilizing your hardware more fully, your system can 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.**

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.

As we have already seen, multitasking is a technique used to simplify the designing of real-time 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 <u>that</u> 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 REFERENCE MANUAL.

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

1. Task A is in the process of reading information from a segment.
2. An interrupt occurs and Task B, which has higher priority than Task A, preempts Task A.
3. Task B modifies the contents of the segment that Task A was in the midst of reading.
4. Task B finishes processing its event and surrenders the processor.
5. 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, Task A can be assured of valid information. Somehow, Task A must obtain exclusive use of the table.

The iRMX 86 Operating System provides two types of objects that can be used to provide mutual exclusion -- the semaphore and the region. 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.

Regions allow tasks to share data. Mutual exclusion is achieved because only one task may access a region at a time. The use of regions should be restricted to programmers who have a firm understanding of the iRMX 86 Operating System. For more information on regions, see the iRMX 86 NUCLEUS REFERENCE MANUAL.

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.

Every real-time 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.

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.

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.

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.

DEBUGGING SUPPORT

The iRMX 86 Operating System provides object-oriented debugging facilities.

Intel provides three object-oriented debugging aids for use with the iRMX 86 Operating System: the on-line Dynamic Debugger (or simply the "Debugger"), the System Debugger (SDB), and the Crash Analyzer. You can include these tools during development of your application system, then remove them from your application when it has stabilized, thus reducing the size of the application system.

All three tools are attuned to iRMX 86 objects (tasks, mailboxes, etc.). This eases the debugging of iRMX 86 applications. These tools are discussed in greater detail later in this chapter.

Because the Dynamic Debugger, System Debugger, and Crash Analyzer are "sensitive" to iRMX 86 objects, you can fully debug your application system, including the interaction between tasks. The Dynamic Debugger allows you to debug individual tasks while the remainder of the job continues to execute. The System Debugger and Crash Analyzer allow you to "freeze" the entire system and examine the contents of memory and CPU registers, and the state of each object in use at the time.

Using the iRMX 86 debugging tools, you can reduce development time, time to market, and the cost of implementing and maintaining your application systems.

PROCESSOR SELECTIVITY

The iRMX 86 Operating System supports a number of Intel microprocessor boards. During configuration you select the processor to match your system.

In addition to supporting the iAPX 86,88 microprocessors, the iRMX 86 Operating System can be configured to execute on other iAPX family members, including the iAPX 186, 188, and 286 processors. These processors have a higher level of integration and faster execution times than the iAPX 86, 88 processors.

On the iAPX 186,188 processors, the iRMX 86 Operating System executes in iRMX compatibility mode. On the iAPX 286 processor, the Operating System executes in real address mode. Applications written for iAPX 86,88-based microcomputers will run in real address mode or in iRMX compatibility mode without modification or relinking.

The ability to tailor the Operating System to match your processor means you can upgrade your system to a higher level of integration without the need to modify or relink your application systems to run on the new processor. This provides flexibility in the development of your application systems as well as faster execution times for applications running on the more highly integrated processors.

INPUT/OUTPUT FEATURES

The iRMX 86 Operating System offers the power and flexibility of a general-purpose operating system. Input and output operations will be a large part of most applications, so the Operating System offers a collection of I/O features to speed development of application systems, and to make the I/O of those systems efficient.

CHOICE OF I/O SYSTEMS

To meet the I/O needs of a wide variety of applications, the iRMX 86 Operating System provides two I/O systems: the Basic I/O System and the Extended I/O System. You can use the Basic I/O System only, or you can combine the two I/O systems.

Many features of the iRMX 86 Operating System are useful in most applications, but not all applications. This is especially true of features relating to input and output. The iRMX 86 Operating System provides two I/O systems: the Basic I/O System and the Extended I/O System.

Basic I/O System

For some applications the performance or flexibility of the system is more critical than the time necessary to produce the system. For these applications, the iRMX 86 Operating System provides the Basic I/O System.

The Basic I/O System is the more flexible of the two I/O systems. It provides very powerful capabilities, and it makes few assumptions about the requirements of your application. The following features illustrate the flexibility of the Basic I/O System:

ALLOWS YOU TO DESIGN YOUR OWN BUFFERING ALGORITHM. Rather than automatically providing a buffering algorithm, the Basic I/O System allows you to design and implement your own buffering technique. Using the Basic I/O System, you control the synchronization between I/O and processing.

APPROPRIATE FOR RANDOM I/O OPERATIONS. Perhaps the I/O in your application is random access. This means that rather than reading or writing data in sequential blocks, the application accesses data in blocks that are not adjacent to each other. The Basic I/O System is more appropriate for these operations because of the explicit control the programmer has over I/O operations.

GIVES YOUR TASK CONTROL OF DETAILS. The system calls of the Basic I/O System often have many parameters. Using these parameters, your tasks can closely tailor the behavior of each system call to match the performance requirements of your application system.

The Basic I/O System emphasizes flexibility rather than ease of use. The Basic I/O System provides I/O features that are useful in time-critical or memory-critical applications, and allows the performance of a system to be optimized.

Extended I/O System

The Extended I/O System is designed to be easy to use, and to be efficient for sequential I/O. The important features of the Extended I/O System are described below.

<u>AUTOMATIC BUFFERING OF I/O OPERATIONS</u>. If you want to use multiple-buffered I/O, but do not want to be burdened with writing complex code to check and switch buffers, you can use iRMX 86 Extended I/O System calls. When the application program issues a system call to perform an I/O operation, the iRMX 86 Operating System performs the input or output and returns control to the user program after the data transfer is completed. But before returning control to the user program, the iRMX 86 Operating System starts reading or writing the next block.

For example, if the application is reading a file from disk, the following sequence will occur:

1. When the application program opens a file using an Extended I/O System call, the Operating System starts reading the first block of the file ("initiates" the input).

2. The Operating System returns control to the application program.

3. Later the program requests an Extended I/O System Read. The Operating System has already started reading this data. When the input is complete, the Operating System initiates a read of the next block of the file (called "reading ahead"), and returns control to the calling program.

In this way, whenever the user requests an Extended I/O System Read, the data is either immediately available, or is in the process of being read.

The equivalent output process is performed by "writing behind." When an application program requests an Extended I/O System Write, the iRMX 86 Operating System copies the data to a buffer maintained by the Extended I/O System, and returns to the calling program. Whenever this buffer is filled, the system initiates an output operation.

EFFICIENT SEQUENTIAL I/O OPERATIONS. Another characteristic of the Extended I/O System is that when it does a "read ahead" operation, the Operating System assumes that a series of sequential reads are to be performed. For example, the Operating System will read data from disk address 23, then from disk address 24, and so on. So when your I/O is mostly sequential, (for example, when examining consecutive records of a file) Extended I/O System calls are particularly efficient. Though less efficient, it is still possible to perform random access of a file with the Extended I/O System by preceding operations with a Seek call specifying the offset into the file.

FREE OF TEDIOUS DETAILS. The system calls of the Extended I/O System have relatively few parameters and are easy to code. In many cases a single Extended I/O call will serve the purpose of several Basic I/O System calls. This simplifies your application system, which reduces development time and reduces costs.

The iRMX 86 Operating System allows you to select the features you want. The Basic I/O System gives maximum control of I/O operations for applications requiring finely tuned performance, especially while doing random-access I/O. The Extended I/O System is easy to use. It saves development costs and development time, especially in applications that use sequential I/O.

Finally, remember that you can use both I/O systems when your application system uses I/O for several purposes, some of which are best accomplished by the Basic I/O System, and some of which are best accomplished by the Extended I/O System.

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.

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.

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.

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.

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.

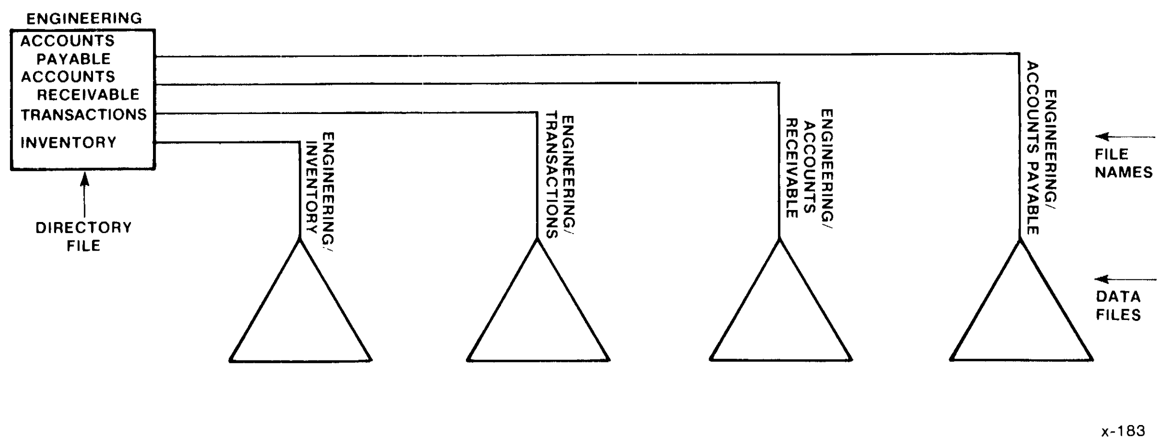


Figure 4-2. An Engineering Directory

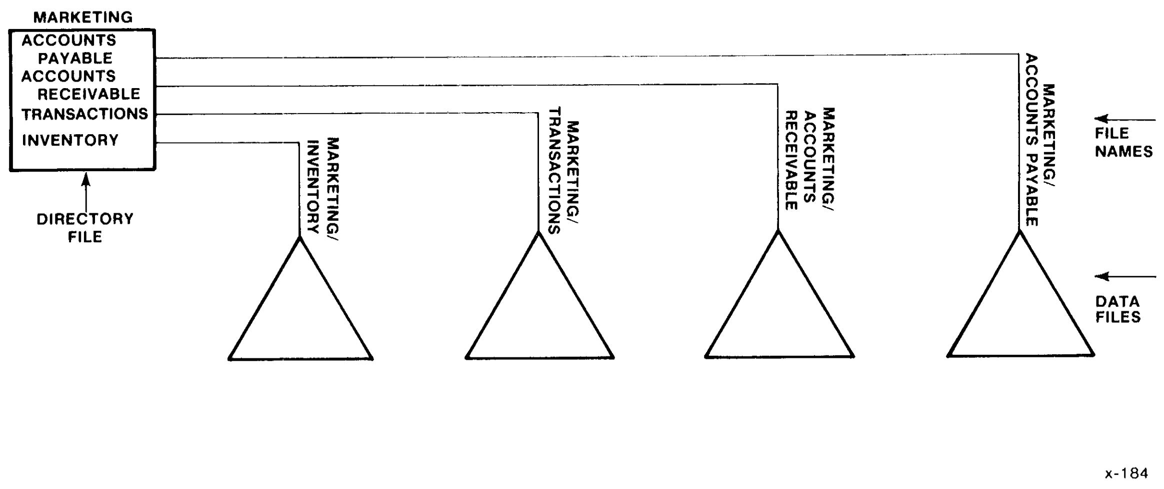


Figure 4-3. A Marketing Directory

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.

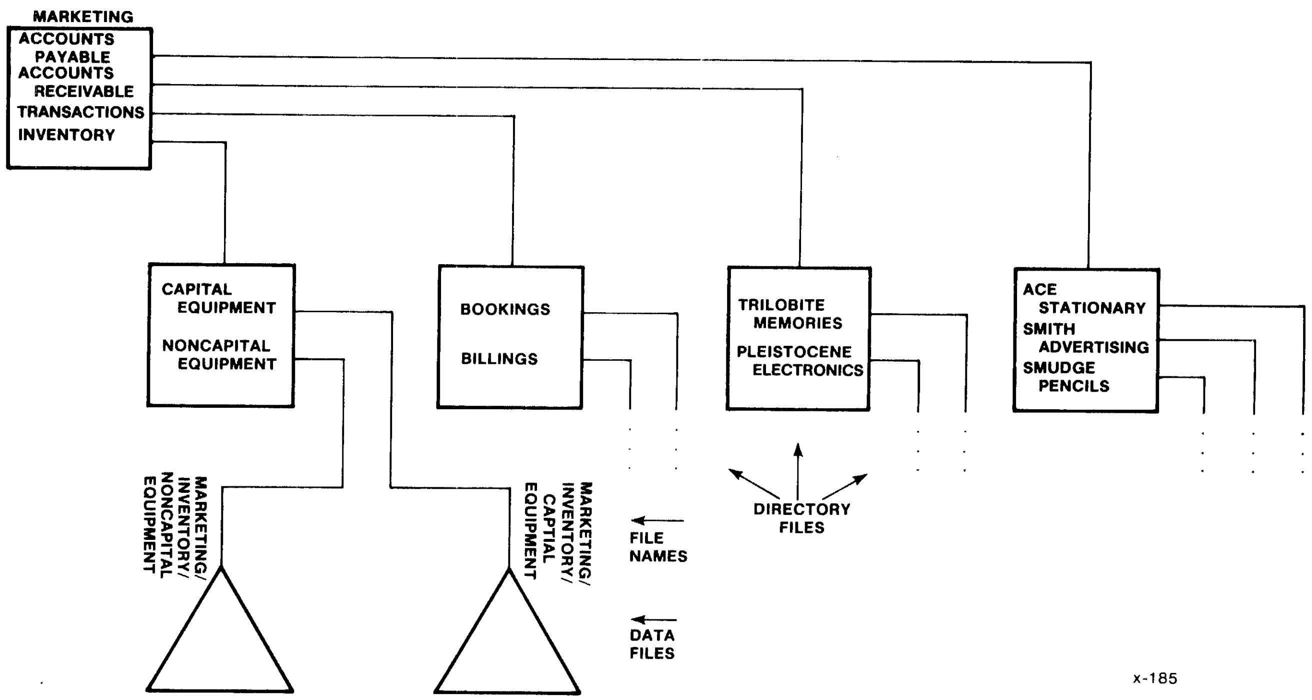


Figure 4-4. Hierarchical Naming of Files

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.

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 ensuring 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 the 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.

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.

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.

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 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. More device space is wasted with larger granularity.

  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.

By allowing you to control 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.

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

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.

TERMINAL SUPPORT CODE

Many brands and types of keyboard terminals are available in the marketplace. The iRMX 86 Terminal Support Code allows you to use nearly any terminal regardless of its individual characteristics. Terminal Support Code also allows programmers or terminal operators to specify a variety of special terminal modes and operations.

Every terminal connected to an iRMX 86 application system communicates with the system via one of two software packages: the iRMX 86 Terminal Handler or the iRMX 86 Terminal Support Code. (The Terminal Handler is described in a later section, "Interactive Configurability.") Terminal Support Code is software that acts as a programmable interface between a terminal driver and the Basic I/O System.

This section describes these major capabilities of the Terminal Support Code:

- Editing and controlling terminal input.
- Type-ahead.
- Controlling terminal output.
- Terminal characterization.

Editing and Controlling Input to a Terminal

A terminal operator has available a set of characters that control and edit terminal input. For example, an operator can:

- Use the RUBOUT key to delete the previous character in an input line. The Terminal Support Code can be set to handle the RUBOUT character differently for a video terminal than it does for a hard-copy terminal.
- Reprint the line to show editing already performed.
- Discard the current input line and start typing a new line.

The Terminal Support Code allows you to replace default control characters with different characters. You can also switch a terminal to "transparent mode," so that editing and control characters have no effect and are not removed from the line of text.

Type-Ahead

If an operator types faster than the Operating System can read, interpret, and respond to input, the Terminal Support Code sends the first line to the I/O System for processing, and saves additional data in a type-ahead buffer.

After the Operating System finishes with the first line, the Terminal Support Code sends additional input data.

Controlling Output to a Terminal

When sending output to a terminal, the Terminal Support Code always operates in one of four modes. An operator can dynamically switch from one output mode to another by entering output control characters. The output modes and their characteristics are as follows:

Normal	The Terminal Support Code accepts output from the application system and immediately passes output to the terminal for display.
Stopped	The Terminal Support Code accepts output from the application system, but it queues the output rather than passing it immediately to the terminal.
Scrolling	The Terminal Support Code accepts output from the application system, and it queues the output as in the stopped mode. However, rather than completely preventing output from reaching the terminal, it sends a predetermined number of lines to the terminal whenever the operator enters a certain character at the terminal.
Discarding	Data sent to the terminal is effectively lost; it is neither displayed nor queued.

Translation

The Terminal Support Code accepts escape sequences (characters preceded by an ESC character) to define characteristics of a terminal. This powerful feature allows you to "characterize" terminals so that the I/O system can use standard control codes and sequences of codes for all terminals. This process is called <u>translation</u>.

Here is how translation works. The Basic I/O System sends standard codes to the Terminal Support Code to, for example, move the video cursor. The Terminal Support Code converts the standard codes into codes recognized by a particular terminal, and sends out these terminal-specific codes.

Besides translation, escape sequences are used to set terminal variables, such as how many lines are displayed when in Scrolling mode. Escape sequences can be sent either from the terminal, or from a program. That is, you can change terminal behavior by keying in escape sequences or by running a program.

Some of the advantages to including the Terminal Support Code in your application are:

- **You can use virtually any ASCII keyboard terminal that can be connected to your hardware.**
- **Your application can include convenient line-editing and output-control functions.**
- **You can define unique characteristics for each terminal in a multiple-terminal system.**
- **You can define new characteristics either from programs or from terminals.**

CUSTOMIZING FEATURES

The iRMX 86 Operating System is designed specifically for OEM and VEU applications. For this reason the application system as a whole can appear unique to the user. Certain features of the Operating System allow an application to be customized in its capabilities and in how it appears to the end user. Let's look at these features.

CUSTOM INTERACTIVE COMMANDS

People interact with your applications by entering commands and receiving information at terminals. The iRMX 86 Operating System allows you to define commands that are appropriate to the application and are meaningful to the operator. This command facility is called the Human Interface.

By designing commands which are appropriate to the type of people who use a system, you can control the human-to-application interface. This can make a system appear "friendly," it can give the application a unique character, and it can reduce operator errors.

Custom Commands = Programs

Because the first word in a command is the name of an executable program file on a mass storage device such as a disk, you are given great flexibility in defining commands. When someone types a command at the terminal, the program having the command name is loaded from the secondary storage and is run by the Operating System. This means:

- You may add or modify commands simply by writing new programs.
- The number of custom commands for a system is not limited by the amount of dynamic memory.
- You do not have to "rebuild" the system to change commands.
- Programs that are used infrequently do not take up memory space when they aren't being run.

Command Line Parsing

System calls are available for retrieving and interpreting parameters of a command. This process is called "parsing a command line."

Consider an application that monitors toxins in the blood of hospital patients. An operator (perhaps a nurse or doctor) can run a task that displays the toxin level of an individual patient, or of all patients being monitored.

One approach would be to have the operator run the task with a command:

```
RUN TOXIN.V3
```

The program might prompt with:

```
Display of which units? --
```

A more "friendly" approach allows a person to use commands that are oriented to the application and to his or her skills, rather than to use computer-oriented commands. In the example, a better command is:

```
TOXIN of John Doe
```

The program TOXIN issues a system call to receive the parameter "John Doe". Because file names frequently are parameters for commands, specialized system calls are also available to interpret file name parameters.

The iRMX 86 Operating System makes it easy to design commands for operators who are not particularly familiar with computers. The ability to define commands enables you to create an application that is easy to use, easy to understand, and resistant to operator errors. New commands may be added by simply writing new programs, rather than making expensive changes to the system.

APPLICATION LOADING

The iRMX 86 Operating System allows your application to read programs from disk into memory and to run them. (This capability is briefly described under Custom Interactive Commands in this section.) Also, the Operating System allows a program to be broken into segments called overlays, so that the entire program does not have to be in memory at one time.

Load-Time Location

The explanation of Custom Interactive Commands mentioned that programs can be loaded from a mass storage device like a disk or bubble memory. The iRMX 86 Application Loader is designed so that programs may be loaded anywhere in available memory. The loader will modify the appropriate addresses in the program at the time the program is loaded. This capability, Load-Time Location, offers great flexibility in the design of application systems. As new programs are added, existing programs do not have to be rebuilt ("linked") in order to run together. Or if more memory is added to the system, the memory can be readily used.

Overlay Loading

Occasionally a program is large enough that it is necessary to break it into pieces called overlays. Each overlay runs at a different time, and occupies the same area of memory. A program containing overlays consists of a "root" that is always present while the program is running, and of two or more overlays. The overlays are loaded by system calls issued from the root.

An overlay facility allows programs to be run even if the programs are too large to fit in memory. Naturally, some care must be exercised to ensure that functions performed by separate overlays do not have to run simultaneously. Also, a program with overlays will execute somewhat slower than one that does not contain overlays.

The iRMX 86 Application Loader gives a programmer great flexibility in the way programs use memory. The system can load programs anywhere in available memory, and programs can execute even though they are actually larger than the memory available.

SIMULTANEOUS MULTIPLE TERMINAL SUPPORT

Operating systems are characterized as either single-terminal or multi-terminal systems. The iRMX 86 Operating System, being a multi-terminal system, can be accessed by more than one terminal at the same time.

The iRMX 86 Operating System offers two ways that you can implement multi-terminal support:

- Multi-access Human Interface (both standard multi-access and modified multi-access).
- Simultaneous Multiple Terminal Support with I/O Programs.

This section explains both approaches.

Multi-Access Human Interface

The iRMX 86 Operating System can communicate with multiple terminals simultaneously. The Human Interface part of the Operating System provides multi-access, which is high-level support for this communication. From a terminal in a multi-access system an operator can execute commands, run development programs (like editors, compilers, and so on), and run other application programs. Here is how multi-access works.

The Operating System detects when a terminal is turned on and assigns an operating environment for the terminal. This environment consists of an identifier (ID), an area of memory in which programs can run, and a priority at which the programs run. The Operating System then starts a program called the initial program. (This is true even if you have only one terminal.)

If the initial program is the one that comes with the Human Interface, we call this standard multi-access. But you can replace the Intel-supplied initial program with one of your own; we call this modified multi-access.

STANDARD MULTI-ACCESS. With standard multi-access, the initial program is an Intel-supplied Command Line Interpreter, (CLI, for short). A CLI is a program that parses commands an operator enters. As each command is entered the CLI divides it into a program name and parameters, runs the program indicated by the command, and passes the parameters to the program.

MODIFIED MULTI-ACCESS. You have the option of providing your own initial program. This initial program might be a CLI of your own design, or it might be a completely different kind of program. For example, you can write a Command Line Interpreter that checks a password before allowing the user to access the system. Or if you want a particular terminal to be used only for BASIC-language programs, a BASIC interpreter might be the initial program.

Multi-access is particularly versatile because you can select, on a terminal-by-terminal basis, what initial program runs. For example, one terminal might run the Intel CLI, another run a special CLI, and a third terminal might always run a word-processing program.

Multiple Terminal Support with I/O Programs

You can implement multiple terminal support with your own programs. That is, you can replace the iRMX 86 multi-access mechanism just described with programs that you write.

In this case, your programs communicate with terminals through I/O system calls. You might do this if you need to implement functions not available with multi-access, or if you want to leave out the Human Interface layer. (A later section, "Interactive Configurability," describes how you can include and exclude parts of the iRMX 86 Operating System.)

The obvious advantage of this feature is that the your system can be accessed by more than one person at a time, which enables you to build systems that are more cost-effective and more powerful than single-terminal systems.

Because of the variety of ways that the Operating System supports multiple terminals, you can build specialized systems, you can make your application system easy to use, and you can protect data from accidental changes.

RUN-TIME BINDING

The iRMX 86 Operating System uses "run-time binding," the process of linking objects, files and devices with the tasks that use them. 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.

Before we look into run-time 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.

Run-time binding means binding while the system is actually running. The iRMX 86 Operating System provides three kinds of run-time binding:

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

The first two kinds of run-time 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.

Let's look more closely at each type of run-time 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 run-time binding to access the same mailbox.

The run-time 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 and its object directory.

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. Run-time 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 run-time 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.

Run-time 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.

ERROR HANDLING

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

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.

Nonetheless, the iRMX 86 Operating System does protect your system from most types of errors. The concepts involved in the iRMX 86 error handling scheme are condition or exception codes, and exception handlers. 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, it is called an exception code and shows which unexpected condition prevented successful completion.

- Exception Handlers. An exception handler is a procedure that the Operating System can invoke when a task receives a condition code indicating failure of the function requested. As each task is created, it is assigned an exception handler; either one written by the programmer, or a default exception handler provided by the Operating System.

The alternative to using exception handlers is to process exception codes in the procedure that issued the system call.

Because you can write the exception handler, you can control the behavior of an application when it receives an exception code. The handler can recover from the error, delete the task containing the error, warn the operator of the error, or ignore the error altogether. The choice is yours.

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 exceptional code can be processed either of two ways: within the procedure that used the system call, or by an exception handler invoked by the Operating System. The technique used is a characteristic of a task, and is established when the task is incorporated into the system.

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.

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.

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.

SOFTWARE INTERFACE

The iRMX 86 Operating System may be used to run various language translators (PASCAL, FORTRAN, PL/M-86, ASM86 Macro Assembler, etc.). A standard, flexible protocol, the Universal Development Interface (UDI), allows language translators, language run-time packages, and other software development tools to run on the iRMX 86 Operating System.

The UDI protocol consists of a set of system calls by which language software uses the Operating System. (Language processors might be compilers, interpreters, assemblers, or run-time systems.) Any language may be run on the iRMX 86 Operating System if the language processor uses the UDI standard system calls. In addition, the same language processor can, without modification, be run on any other operating system which includes the UDI system calls. (Intel markets a variety of operating systems which use UDI for language support.)

There are at least two major advantages to the UDI software interface:

- **A language processor can use well-defined, appropriate, standard calls to communicate with the iRMX 86 Operating System. Existing languages can be adapted easily to run on the Operating System.**

- **Any language processor or software tool using UDI system calls can run on several Intel operating systems. This feature is commonly termed "portability," and is becoming a major consideration in software design because of obvious economic benefits.**

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).

A bootstrap loader is a program that resides in ROM on your application hardware. When your system's 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, into RAM.

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

- **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.

- **The iRMX 86 Bootstrap 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.**

TOOLS

Along with the iRMX 86 Operating System, Intel provides software tools to help you develop an application system. Sometimes you use the features listed in this section as part of your system, and sometimes you use them only while developing the system. But each feature simplifies the process of developing a complex system.

OBJECT-ORIENTED DYNAMIC DEBUGGER

The iRMX 86 Operating System provides a special dynamic 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 real-time environment.

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 Dynamic Debugger (or simply the "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.

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

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.

By simplifying the process of debugging interaction 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.

SYSTEM DEBUGGER

The iRMX 86 Operating System includes a System Debugger (SDB), which extends the capabilities of your system monitor. The System Debugger provides "static" debugging facilities for those times when the system hangs or crashes, when the Nucleus is inadvertently overwritten or destroyed, when you wish to "freeze" the system and examine it, or when synchronization requirements preclude the debugging of selected tasks.

As we saw earlier, the Dynamic Debugger lets you debug one or more tasks while the rest of the application system continues to run.

In contrast, the System Debugger stops the system (if it's not already stopped) and allows you to examine the state of the system at that very instant in time. If possible, after examining the state of the system you can continue execution from where it stopped.

The SDB extends the capabilities of the iSDM 86,286 System Debug Monitor (which you must purchase separately) or the iSBC 957B Monitor (which you may already have) by allowing you to:

- Identify and interpret iRMX 86 system calls.
- Display information about iRMX 86 objects.
- Examine a task's stack to determine system call history.

The System Debugger provides the facilities necessary for diagnosing system crashes. By stopping the system, the SDB provides a global view of the system, which can help you find errors not easily found with the Dynamic Debugger. Development time and costs are reduced because you can track down and fix errors in a more timely manner.

CRASH ANALYZER

The iRMX 86 Operating System includes a Crash Analyzer that dumps the contents of memory to a file, and then formats the information for display or printing.

When your system crashes -- as any system might during development -- the Crash Analyzer provides a "snapshot" of the contents of memory. After a system crash (or whenever you want to "freeze" the system) the Dumper portion of the iRMX 86 Crash Analyzer writes the contents of selected memory to a file on an Intellec Microcomputer Development System. The Analyzer portion then reads the file and produces a formatted print file that includes:

- Every iRMX 86 object.
- The state of each object; for example, how many units are in a semaphore, whether a task is ready or suspended or asleep, the objects queued at a mailbox, the size of memory segments.
- The state of the hardware, including the contents of registers.

In short, the Crash Analyzer is oriented to the characteristics of the iRMX 86 Operating System. Therefore an iRMX 86 memory dump doesn't show just the usual raw data; it formats the data according to iRMX 86 data types.

Having a memory dump facility helps to solve difficult and sometimes obscure problems in your application system. There are four major advantages to using the Crash Analyzer:

- **It is a "smart" analysis utility that knows about the iRMX 86 system and environment, so it can save programmers laborious searches through raw memory data. The result is an application system that can be debugged and refined quickly, and a system that you know is working according to design.**
- **A system failure does not have to be debugged at the time it occurs. The problem can be handled later, perhaps by another person, and perhaps at a different site.**
- **The Analyzer finds and identifies probable errors, such as linked-lists that are broken, and stack overflow.**
- **Crash Analyzer software is divided between a relatively small Dumper program that is part of the application system, and an Analyzer program that runs on the host development system. Therefore you can use the Crash Analyzer with little loss of application system memory.**

INSTALLATION SYSTEMS

The iRMX 86 release package contains two systems that are ready to be used. These ready-to-run systems are referred to as installation systems.

The iRMX 86 Operating System is a "building block" operating system, with pieces you can put together to create your system. The release diskettes contain two installation systems, built and ready to run. Intel provides one installation system for iAPX 86-based computers and one for iAPX 286-based computers. Also on the release diskettes are the definition files that were used to create these installation systems as well as definition files that you can use to create systems for the iAPX 186/03, 186/51, and 188/48 processor boards, with little or no modification.

The installation-system concept provides these specific advantages:

- **The installation systems may be used as-is on Intel's Integrated Systems such as the iSYS 86/310, iSYS 86/330A, iSYS 86/380, iSYS 286/310 and iSYS 286/380 systems. No hardware or software changes are required to install iRMX 86 on these systems.**
- **You can become familiar with the Operating System immediately, and perhaps even run your application software without rebuilding the Operating System.**
- **The installation systems have commands to perform common file operations. (See "File Maintenance Programs" later in this chapter.) These commands can be used in developing your application, and may themselves be included in your application.**
- **The actual files used to create the installation systems provide an example of how to put together an iRMX 86 system, and may be be used as a starting point for creating your own system.**

ON-TARGET PROGRAM DEVELOPMENT

You may develop an iRMX 86 application system on a "host" computer system, and then transfer the system to a "target" computer. An alternative is to develop programs on the target computer. With certain hardware and software options, the iRMX 86 Operating System can provide an ideal program development environment. Although program development on a target system is not practical for all applications, for some applications it is very worthwhile.

The emphasis of most of this chapter has been on building a specialized software product using the iRMX 86 Operating System as a basis for the application.

Typically the programs you write are developed on an Intellec Microcomputer Development System, and the application system is then "migrated" onto an iSBC 86,88,186,188,286 board (the target computer).

In contrast, you can develop programs directly on the target computer by using certain capabilities of the iRMX 86 Operating System. Each of these features has already been described, and here is how they combine to provide a program development system.

- File support. The iRMX 86 file system supports creation of source, object, and loadable files. Many programmers can use the same disk because of the hierarchical structure and protection mechanisms of the iRMX 86 file system.

- Languages and software tools. The Universal Development Interface makes it easy to support language processors and run-time support systems. You can perform all phases of program development using editors, linkers, and other tools of the programming trade; these software tools are available from Intel and run on the iRMX 86 Operating System.

- Convenience. The Application Loader makes it easy to load and execute software. Also, the Human Interface provides a powerful facility for parsing the names of files that are used by language processors, editors, and linkers.

- Debugging. Programs developed on an iRMX 86 Operating System can be debugged using the iRMX 86 object-oriented Dynamic Debugger and the static System Debugger.

On-target program development using the iRMX 86 Operating System is useful for these reasons:

- If your application system has spare resources (processing time, memory, mass-storage space) you can use the system more efficiently.

- Programmers can make changes on-site, which has economic and scheduling advantages.

INTERACTIVE CONFIGURABILITY

The iRMX 86 Operating System is configurable. By selecting only the parts of the Operating System that you need, you can reduce the amount of memory required for your application system. The configuration process is straightforward and certain because Intel provides the Interactive Configuration Utility (ICU), a utility that guides you through the process.

A system is configurable if you can select the pieces of it that you want and discard the pieces that you don't want. During the process of configuration, you select the desired parts and combine them to form the system.

Configuration is Making Choices

To configure an application system based on the iRMX 86 Operating System, you first select the parts of the Operating System that your application system requires, as shown in Figure 4-5. (You also specify important characteristics of each module, such as memory requirements.) Then you combine Operating System modules with your application software, with Intel-supplied software and with software from vendors. This forms the complete application system. Finally you install the application system on the target hardware.

Configuration is Interactive

The iRMX 86 Operating System includes the Interactive Configuration Utility (ICU), which guides you through the configuration process by displaying a series of "menus." Each menu describes a number of features, and then allows you to accept or change an existing (or default) value for each feature. Also, the ICU allows you to save the results of a previous configuration, so that you can make a small change quickly without re-answering all of the questions.

Parts of the iRMX™ 86 Operating System

The iRMX 86 Operating System consists of a number of major subsystems, also called layers. During the process of configuration you specify which of these subsystems, shown in Figure 4-5, to include in your application system. The functions of these layers are:

- The Nucleus. The Nucleus is the heart of the iRMX 86 Operating System. All 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.

  With Intel's iOSP 86 Support Package, supplied on release diskette, you can replace approximately two-thirds of the Nucleus with the code contained in the 80130 Operating System Firmware component. This device decreases the number of hardware components required for your system.

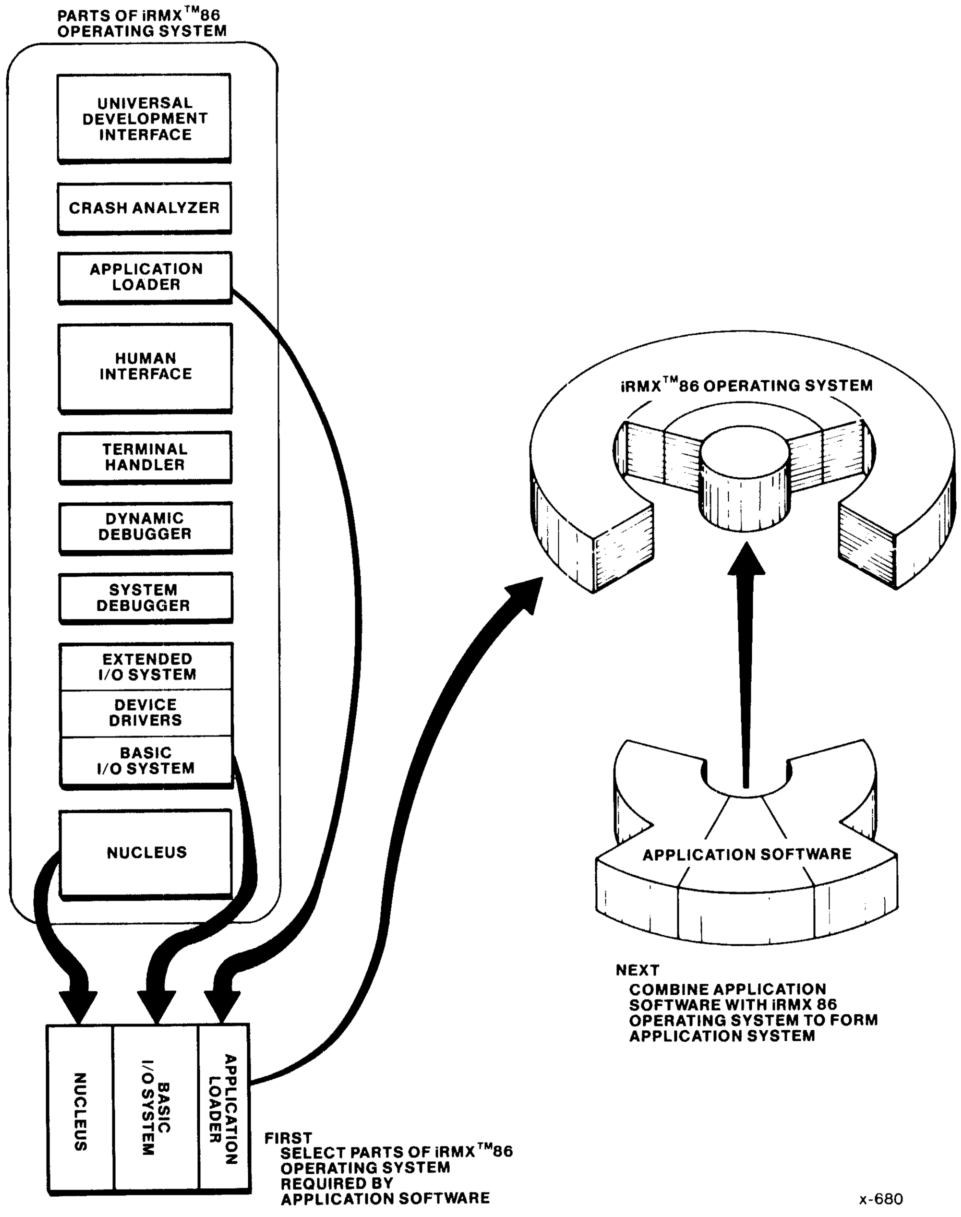


Figure 4-5. Configuration of an iRMX™ 86 System

- The I/O Systems. The I/O Systems (Basic and Extended) provide file management and the device-independent interface to input and output devices. The I/O Systems are optional components of the iRMX 86 Operating System, so they can be excluded from the Operating System if they are not needed. The user may include the Basic I/O System without including the Extended I/O System. The Extended I/O System requires the Basic I/O System.

- Device Drivers. Device drivers are the interface between an application and the I/O devices connected to the application. Any device drivers selected during configuration (including terminal drivers and Terminal Support Code) become part of the Basic I/O System.

- The Human Interface. The Human Interface may control the application system with commands entered at a terminal. The Human Interface includes a set of commands for commonly used operations. You can also create your own commands. Like the I/O Systems, the Human Interface is an optional component and can be left out of the Operating System if it is not required. If the Human Interface is included, it requires all other system layers.

- The Application Loader. The Application Loader allows your application to load programs and overlays from disk into main memory. The Application Loader is an optional part of a system, but if included requires the I/O Systems.

- The Dynamic Debugger. The Dynamic 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 System Debugger. The System Debugger (SDB), also optional, extends the capabilities of the iSDM 86, 286 System Debug Monitors by supplying "static" debugging information about the system after a crash or at any time you need to freeze and examine the system. As with the Dynamic Debugger, you can include the SDB in your system during development, then remove it to reduce the size of your finished application system.

- Terminal Handler. The Terminal Handler, another optional piece of software, allows you to use a terminal without using the I/O System or Human interface. It is possible to configure the terminal so that it is only an output device.

- The Crash Analyzer. The Crash Analyzer, an optional component, produces a post-mortem dump of memory. It allows a user to dump memory to a file, and later format and print the file, showing each iRMX 86 object. The program to dump memory to a file becomes part of the application system; formatting and printing is done with your development system.

- The Universal Development Interface (UDI). The Universal Development Interface is a software interface that allows language translators and other software development tools to access the facilities of the iRMX 86 Operating System. The UDI is the outermost layer of any application system but may be excluded if not needed. However, if it is included it requires the Human Interface and all other system layers.

Figure 4-5 illustrates the advantage of a configurable Operating System. An iRMX 86 Operating System -- consisting of the Nucleus, Basic I/O System, and Application Loader -- is being combined with application software. By excluding unnecessary subsystems of the Operating System, you reduce the amount of memory needed by your system.

Two advantages to using Intel's interactive utility for configuration are:

- **Configuration of application systems, even complex systems, is relatively easy.**

- **Choices you make during the configuration process are saved in a file, and you can make changes to this file and re-use it. This means that having once configured your application system, it is easy to make changes to the configuration.**

FILE MAINTENANCE PROGRAMS

The iRMX 86 Operating System is delivered with programs which allow you to manipulate iRMX 86 files.

As you develop an application, you need to work with files. You can write programs to perform these operations. But the Operating System already has programs to perform operations that are usually necessary in developing an application system. (These programs operate as commands to the system, as explained in the earlier section "Custom Interactive Commands.")

Here is a sample of some of the programs supplied with the Operating System:

- COPY, which copies or creates files.

- FORMAT, which formats an iRMX 86 secondary storage device such as a disk or diskette.

- DIR, which displays a directory of the files on an iRMX 86 device.

- DOWNCOPY and UPCOPY, which are used to move files between Intellec development system devices and iRMX 86 devices.

- RENAME, which allows you to rename files.
- CREATEDIR, which allows you to create a new file directory.
- SUBMIT, which automatically executes commands contained in an iRMX 86 file.
- BACKUP and RESTORE for saving all of the files on a device.

The most important advantage of these programs is that you will save time and money in developing your application system, because you already have the software tools necessary to manipulate files during the development process. In addition, the file maintenance programs may be included as part of your application system if this is appropriate.

CHAPTER PERSPECTIVE

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 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 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 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 (MCU) consists of a terminal with a screen and a keyboard. This terminal, which operates under control of the single board computer, allows one individual to monitor and control the entire system.

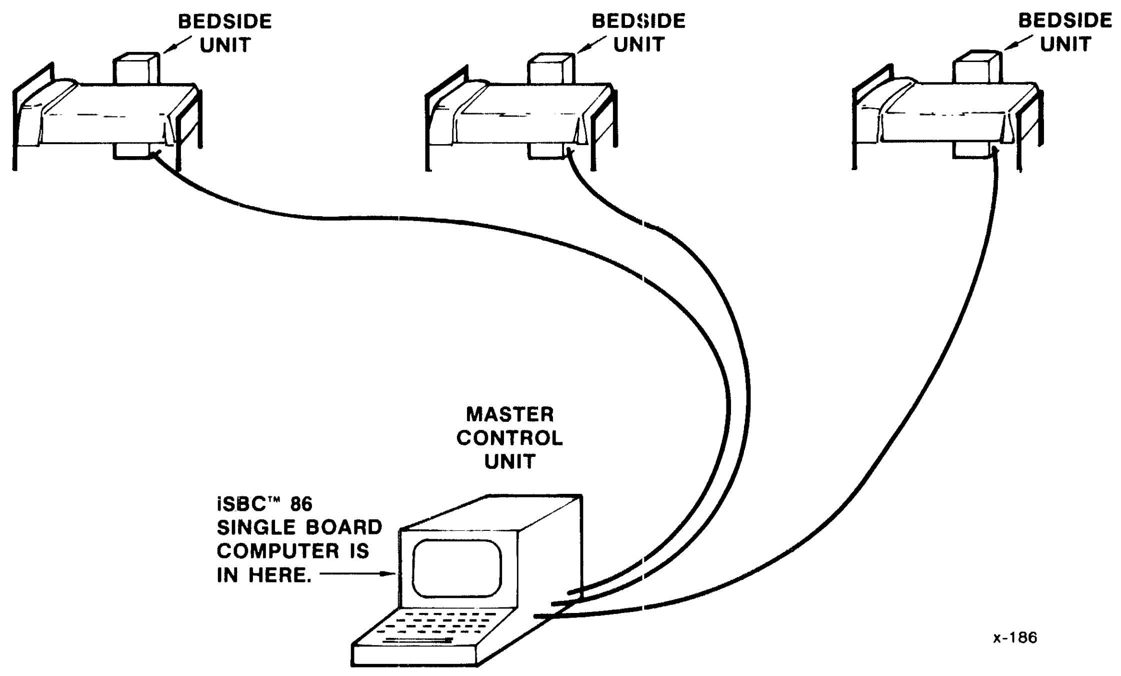


Figure 5-1. The Hardware Of The Dialysis Application System

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 MCU 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 MCU.
- 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 steadily declining toxicity level.

This means that 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.

HUMAN INTERFACE

Interaction between medical personnel and the system can be "human engineered." Information can be requested and displayed in a form that is meaningful to the operators of the system. Also, new capabilities may be added to the system by simply adding new programs.

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 run-time binding.

RUN-TIME BINDING

As mentioned earlier in this manual, run-time 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 cataloged names.

Run-time 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 run-time 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 provides a way to move information between main memory and the disk. Your application software need not include code to perform these functions.

If mass storage devices were added to the system, it would be possible to do program development on the system, so that new programs could be written and tested at the site. This is a powerful addition to a system, although it is not appropriate for every application.

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 send 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 the iRMX 86 technical manuals.

***

CHAPTER 6
iRMX™ 86 LITERATURE

This chapter describes the iRMX 86 Operating System documentation set. This chapter lists the four volumes and the manuals contained therein, describes the technical content and audience level of each manual, and correlates individual manuals with the features described in previous chapters.

The iRMX 86 documentation set consists of four bound volumes organized into these functional categories: introductory and operational information, programming information (two volumes), and installation and configuration instructions. Each volume contains two or more iRMX 86 manuals that conform to the volume's functional theme.

Because the iRMX 86 documentation set is packaged in bound volumes, you can no longer order manuals individually. Instead, you must order a complete volume of text to get a manual contained in that volume. (Individual manuals no longer have order numbers.)

When ordering individual volumes, you can order the binder, spine card, and literature packet together as a unit or separately. If you wish to order a volume as a unit, use the "order" number that appears on the spine of the binder. This number is also provided in the following list. If you wish to order separate pieces of the volume (e.g., the literature packet only), use the "part" number as labeled on the piece. If you don't know the part number, consult the Intel Literature Guide.

The following list shows volume titles, order numbers, and individual manuals in each of the volumes. Manuals are listed in the order they appear in the volumes.

1. iRMX™ 86 INTRODUCTION AND OPERATOR'S REFERENCE MANUAL
   Order Number: 146545

   - Introduction to the iRMX™ 86 Operating System
   - iRMX™ 86 Operator's Manual
   - iRMX™ 86 Disk Verification Utility Reference Manual

2. iRMX™ 86 PROGRAMMER'S REFERENCE MANUAL, PART I
   Order Number: 146546

   - iRMX™ 86 Nucleus Reference Manual
   - iRMX™ 86 Basic I/O System Reference Manual
   - iRMX™ 86 Extended I/O System Reference Manual

3. iRMX™ 86 PROGRAMMER'S REFERENCE MANUAL, PART II
   Order Number: 146547

   - iRMX™ 86 Application Loader Reference Manual
   - iRMX™ 86 Human Interface Reference Manual
   - iRMX™ 86 Universal Development Interface Reference Manual
   - Guide to Writing Device Drivers for the iRMX™ 86 and iRMX™ 88 I/O Systems
   - iRMX™ 86 Programming Techniques
   - iRMX™ 86 Terminal Handler Reference Manual
   - iRMX™ 86 Debugger Reference Manual
   - iRMX™ 86 Crash Analyzer Reference Manual
   - iRMX™ 86 System Debugger Reference Manual
   - iRMX™ 86 Bootstrap Loader Reference Manual

4. iRMX™ 86 INSTALLATION AND CONFIGURATION GUIDE
   Order Number: 146548

   - iRMX™ 86 Installation Guide
   - iRMX™ 86 Configuration Guide
   - Master Index for Release 6 of the iRMX™ 86 Operating System

Each of these volumes (and the manuals contained in each) serves a well-defined set of readers. For each volume, this chapter describes the general content of the volume and the content, purpose and intended readership of each manual in that volume. (Table 6-1 correlates features with manuals.) Also, the chapter provides some time-saving tips to bear in mind as you read the documentation.

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 effect.

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 identified, in whatever manual they are described, by a caution regarding their effect.

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 (this 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.

iRMX™ 86 INTRODUCTION AND OPERATOR'S REFERENCE MANUAL

This volume contains introductory and operations-specific information about the iRMX 86 Operating System. The information in this volume is designed for first-time users of the system.

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 OPERATOR'S MANUAL

This manual describes the Human Interface commands. For example, the Human Interface provides commands:

- To copy, delete, and otherwise manage files.
- To display directories.
- To format and verify mass storage volumes.

The manual also describes:

- File pathnames and other file information necessary to use commands.
- Standard logical names.

- The Multi-Access Human Interface.
- The iRMX 86 Files Utility.

If you intend to include the iRMX 86 Human Interface in your system, you will be interested in this manual.

iRMX™ 86 DISK VERIFICATION UTILITY REFERENCE MANUAL

This manual documents the Disk Verification Utility, a software tool that runs as a Human Interface command, verifying and modifying the data structures of iRMX 86 disk structures (files, directories, and physical volumes). The manual describes how to invoke the utility and contains detailed descriptions of all utility commands. Because users of the Disk Verification Utility must be familiar with the structure of iRMX 86 volumes in order to use the utility intelligently, the manual describes in detail the structure of iRMX 86 file and directory structures.

You will use this manual to maintain disk file volumes, and perhaps to recover files that are "lost" or corrupted.

iRMX™ 86 PROGRAMMER'S REFERENCE MANUAL, PART I

This volume contains detailed information about the iRMX 86 Nucleus and Basic and Extended I/O systems. The information in this volume is intended for system and application programmers.

iRMX™ 86 NUCLEUS REFERENCE MANUAL

This reference manual is written for engineers planning to use the iRMX 86 Operating System. It is the information warehouse for the Nucleus. It contains concise but detailed discussions of these topics:

- The nature of objects in general and of tasks, jobs, semaphores, mailboxes, and segments in particular.
- Error processing.
- Interrupt processing.
- The creation and deletion of extensions to the Operating System.
- Region exchanges.
- Enabling and disabling the deletion of objects.
- Adding new types of objects to the Operating System.

The heart of the iRMX 86 NUCLEUS REFERENCE MANUAL is a chapter describing all the information necessary to use each system call relating to the Nucleus, including:

- the calling sequence,
- a description of parameters,
- an explanation of exception codes that may be returned by each call, and
- a sample program showing its usage.

iRMX™ 86 BASIC I/O SYSTEM REFERENCE MANUAL

This manual describes the iRMX 86 Basic I/O System. The manual contains descriptions of:

- File directories, and the types of files supported (named, stream, and physical).
- User objects, and access rights associated with user objects.
- I/O operations a programmer may use with the iRMX 86 Operating System.
- Attaching and detaching devices.
- System calls a programmer may use in accessing the facilities of the Basic I/O System.
- Terminal Support Code.

iRMX™ 86 EXTENDED I/O SYSTEM REFERENCE MANUAL

This manual describes the iRMX 86 Extended I/O System, the easier-to-use I/O system. It includes descriptions of the types of iRMX 86 files, and a general description of the Extended I/O System. The most useful part of the manual is a chapter describing all the information necessary to use each Extended I/O System call.

The Extended I/O System unburdens programmers from details of I/O operations. In particular, Extended I/O data transfers are synchronous, meaning that the operating system performs multiple-buffering operations, automatically synchronizing I/O operations with processing.

iRMX™ 86 PROGRAMMER'S REFERENCE MANUAL, PART II

This volume contains detailed information about the iRMX 86 programming utilities and advanced programming techniques for writing application and system programs. The information in this volume is intended for system and application programmers.

iRMX™ 86 APPLICATION LOADER REFERENCE MANUAL

This manual describes the Application Loader. The Application Loader is used for two purposes:

- To load and run programs that reside on secondary storage. These programs are invoked by Human Interface commands.
- To load overlays by invoking system calls.

The manual describes the types of code that can be loaded by the Application Loader (absolute, position-independent, and load-time locatable). The manual also provides detailed descriptions of the Application Loader system calls.

If your application uses the Application Loader (chances are that it will), then you will want to use this manual.

iRMX™ 86 HUMAN INTERFACE REFERENCE MANUAL

The iRMX 86 Human Interface is an optional layer of the iRMX 86 Operating System. The Human Interface allows operators to load and run programs from a console terminal, and provides facilities for custom commands. In addition, the Human Interface has a number of commands (COPY, FORMAT, RENAME, etc.) that are documented in the iRMX 86 OPERATOR'S MANUAL.

If you want to use the powerful facilities provided by the Human Interface, the manual provides information you need, such as:

- Descriptions of Human Interface system calls. These system calls are used to parse custom commands, to control programs run by the Human Interface, to send and receive messages, and generally to control command functions.
- An explanation of the Multi-Access Human Interface, which allows the Operating System to communicate with many terminals simultaneously.
- Instructions for building command programs.

iRMX™ 86 UNIVERSAL DEVELOPMENT INTERFACE REFERENCE MANUAL

Designed for system and application programmers, this manual outlines general programming considerations for using the Universal Development Interface (UDI). The UDI is a software interface that allows language translators and other software development tools to access the facilities of the iRMX 86 Operating System. The manual describes in detail the UDI system calls that provide this access.

You will be interested in this manual if you need a general introduction to the application development process and to the use of the UDI.

GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX™ 86 AND iRMX™ 88 I/O SYSTEMS

This manual gives detailed instructions for writing a device driver that is compatible with both the iRMX 86 I/O System and the iRMX 88 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.

iRMX™ 86 PROGRAMMING TECHNIQUES

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

iRMX™ 86 TERMINAL HANDLER REFERENCE MANUAL

If you wish to use the iRMX 86 Operating System without using the iRMX 86 I/O System, you might need software to communicate with a terminal. This is the function of the Terminal Handler. The iRMX 86 Terminal Handler provides basic character echoing and line editing functions. The manual also describes an optional form of the software called the Output-Only Terminal Handler.

If your application is not using the I/O System or Human Interface to communicate with terminals, you will be interested in the information in this manual.

iRMX™ 86 DEBUGGER REFERENCE MANUAL

This manual describes the Dynamic Debugger (or simply the "Debugger"), an interactive debugging tool used with the iRMX 86 Operating System. The Debugger is especially useful because it is "sensitive" to iRMX 86 objects and it lets you debug one or more tasks while the rest of the system continues to run. This manual includes descriptions of iRMX 86 Dynamic Debugger commands.

iRMX™ 86 SYSTEM DEBUGGER REFERENCE MANUAL

This manual describes the System Debugger, a static debugging tool that is useful in diagnosing system crashes and other "freeze" situations. The System Debugger, like the Dynamic Debugger, is attuned to iRMX 86 objects. The System Debugger is an extension of the iSDM 86 or 286 System Debug Monitor. This manual includes descriptions of System Debugger commands.

iRMX™ 86 CRASH ANALYZER REFERENCE MANUAL

This manual describes the iRMX 86 Crash Analyzer, a utility used to produce post-mortem memory dumps, and to print a formatted display that describes iRMX 86 objects (memory segments, tasks, jobs, etc.) along with the state of each object at the time the system failed.

Because the Crash Analyzer is such a useful tool for development, you will probably want to include it in the early stages of your development cycle. The reference manual describes everything that you need to know about hardware requirements, operating the Crash Analyzer, and interpreting the formatted output.

iRMX™ 86 BOOTSTRAP LOADER REFERENCE MANUAL

This manual describes the Bootstrap Loader. The Bootstrap Loader is used to start a system running by loading the Operating System from a secondary storage device, and transferring control to the Operating System. The Bootstrap Loader is usually in the computer ROM.

If your application uses the Bootstrap Loader, you will want to use this manual.

iRMX™ 86 INSTALLATION AND CONFIGURATION GUIDE

This volume describes how to get your iRMX 86 Operating System up and running on your hardware. The information in this volume is designed for system managers and/or engineers.

This volume also contains the Master Index for the iRMX 86 documentation set, which is useful to any user of the system.

iRMX™ 86 INSTALLATION GUIDE

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 iSDM 86,286 System Debug Monitor, depending on your hardware.

The iRMX 86 INSTALLATION GUIDE tells you how to use the iRMX 86 software and the iSDM package in conjunction with your Intellec Microcomputer Development System and any Intel single board computer.

The manual describes everything necessary to get the installation systems running. This includes instructions for installing wire jumpers on an Intel Single Board Computers, loading and running an installation system, and using the Human Interface commands included in the installation systems. If you are familiar with the Intellec Microcomputer Development System, this manual will prove very useful.

iRMX™ 86 CONFIGURATION GUIDE

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. Once you have made these decisions and have written your application software, you can 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 describes the iRMX 86 Interactive Configuration Utility (ICU). The ICU leads a system programmer through the process of configuration by displaying a logical series of "menus" that describe each choice he or she must make. Each menu then allows a default answer, or allows the programmer to change the default. The ICU, using these answers, creates a file that automatically links and locates the application system software.

MASTER INDEX FOR RELEASE 6 OF THE iRMX™ 86 OPERATING SYSTEM

The Master Index is your road map to the four-volume iRMX 86 documentation set. It is intended for all levels of users.

Table 6-1. Correlation of Manuals and Features

Title	Feature
iRMX 86 Nucleus Reference Manual	Object-Oriented Architecture Multiprogramming Multitasking Interrupt Processing Preemptive, Priority-based Scheduling Error Handling Dynamic Memory Allocation Intertask Coordination Run-Time Binding Extendibility Processor Selectivity
iRMX 86 Basic I/O System Reference Manual and iRMX 86 Extended I/O System Reference Manual	Choice of I/O Systems Hierarchical Naming of Mass Storage Files File Access Control Control of File Fragmentation Device-Independent I/O Terminal Support Code
iRMX 86 Human Interface Reference Manual	Custom Interactive Commands Multi-Access Human Interface
iRMX 86 Operator's Manual	File Maintenance Commands Multi-Access Human Interface
iRMX 86 Bootstrap Loader Reference Manual	Bootstrap Loading
iRMX 86 Application Loader Reference Manual	Application Loading
iRMX 86 Installation Guide	Installation Systems
iRMX 86 Configuration Guide	Interactive Configuration
iRMX 86 Debugger Reference Manual	Object-Oriented Dynamic Debugger Debugging Support
iRMX 86 System Debugger Reference Manual	System Debugging (static) Debugging Support
iRMX 86 Crash Analyzer Reference Manual	Crash Analysis Debugging Support
Guide to Writing Device Drivers for the iRMX 86 and iRMX 88 I/O Systems	Selection of Device Drivers Device-Independent I/O
iRMX 86 Universal Development Interface Reference Manual	Software Interface

***

INDEX

Primary references are underscored.

***

iRMX™ 86 OPERATOR'S MANUAL

PREFACE

This manual is the primary reference for operators who access the iRMX 86 Operating System through a terminal using Human Interface commands. In addition to Human Interface commands, this manual also discusses the line-editing and control characters supported by the iRMX 86 Operating System and two utilities available to iRMX 86 operators: the Patching Utility and the Files Utility.

The manual is divided into the following chapters:

Chapter 1 discusses the Line-editing and control characters available to terminals that access the iRMX 86 Operating System. This chapter applies to all application systems, regardless of whether they include the Human Interface.

Chapters 2 through 4 discuss the Human Interface. Chapter 2 introduces the operator to the Human Interface and describes the general process of using it. Chapter 3 provides a detailed description of Human Interface commands in alphabetical order. Chapter 4 contains examples of Human Interface operations.

Chapter 5 discusses the Patching Utility, a utility that runs on both iRMX 86-based systems and Series III Microcomputer Development Systems.

Chapter 6 discusses the Files Utility, an iRMX 86 application system that you can use to format and maintain iRMX 86 secondary storage volumes.

Appendix A contains a list of iRMX 86 condition codes with short descriptions of the codes.

NOTATIONAL CONVENTIONS

This manual uses the following notational conventions to illustrate syntax:

UPPERCASE	In examples of command syntax, uppercase information must be entered exactly as shown. You can, however, enter this information in uppercase or lowercase characters.
lowercase	In examples of command syntax, lowercase fields indicate information to be supplied by the user. You must enter the appropriate value or symbol for lowercase fields.

underscore	In examples of dialog at the terminal, user input is underscored to distinguish it from system output.
< >	Angle brackets surround variable fields in messages displayed by the Human Interface commands and by the utilities. This information can vary from message to message.

All numbers, unless otherwise noted, are assumed to be decimal. Hexadecimal numbers include the "h" radix character (for example OFFh).

CONTENTS

PAGE

***

CHAPTER 1
LINE EDITING AND CONTROL CHARACTERS

Every terminal connected to an iRMX 86 application system communicates with the system via one of two software packages: the iRMX 86 Terminal Handler or the Terminal Support Code feature of the Basic I/O System.

The Terminal Handler is an independent layer of the Operating System that provides terminal I/O facilities for application systems that do not include the Basic I/O System. Because this manual assumes you are using an application system that includes the Basic I/O System, it does not discuss how to communicate with an application system via the Terminal Handler. Refer to the iRMX 86 TERMINAL HANDLER REFERENCE MANUAL for information about the line-editing and control characters available with the Terminal Handler.

The Terminal Support Code is a software package that interfaces to terminal device drivers to provide terminal communication for systems that include the Basic I/O System. This manual assumes that your terminal communicates with the iRMX 86 application system via the Terminal Support Code.

The Terminal Support Code provides a set of line-editing and control characters that give you the basic editing and control functions you need when entering text at a terminal. You can use these characters in addition to the Human Interface commands described later in this manual. This chapter discusses, along with the terminal support code, the line editing features and control characters which are available. However, the Terminal Support Code contains many features other than those discussed in this chapter. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for a complete description of the Terminal Support Code.

TYPE-AHEAD

When you enter characters at the terminal, you can use the type-ahead feature to enter a number of lines at one time. The Terminal Support Code sends the first line to the Operating System for processing and stores additional lines in a type-ahead buffer. It sends the next line in the buffer to the Operating System after the Operating System finishes with the first line. If the type-ahead buffer becomes full, the Terminal Support Code sounds the terminal bell and refuses to accept input.

CONTROLLING INPUT TO A TERMINAL

The Terminal Support Code provides several characters that you can enter to control and edit terminal input. Some of these characters correspond to single keys on your terminal (such as carriage return or rubout). For others, called control characters, you must press the CTRL key, and while holding it down, also press an alphabetical key. This manual designates control characters as follows:

CTRL/character

The editing and control characters are processed by the Terminal Support Code. With the exception of the line terminator, they are not normally included in the input line that is sent to the Operating System.

The control characters listed in this section are the default characters. Each can be replaced with a different character by means of a selection procedure described in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL. The default editing and control characters for terminal input include:

CARRIAGE RETURN or LINE FEED	Terminates the current line and positions the cursor at the beginning of the next line. Entering either of these characters adds a carriage return/line feed pair to the input line.
RUBOUT	Deletes (or rubs out) the previous character in the input line. In response to the RUBOUT, your terminal display changes in one of two ways, depending on the configuration of the Terminal Support Code. In one configuration, each RUBOUT removes a character from the screen and moves the cursor back to that character position. In the other configuration, each RUBOUT echoes the deleted character back to the terminal. In the second configuration, also called hard-copy mode, the Terminal Support Code surrounds the echoed characters with the "#" character to distinguish the echoed characters from the surrounding text.
CTRL/p	A "quoting" character, which removes, from the character that follows it, any meaning that is special to the Terminal Support Code. It literalizes the next character, causing it to be sent on to the Operating System, even if it is a control character that the Terminal Support Code understands. All control characters (except for output control characters) sent to the Operating System in this manner are not processed as control characters. Output control characters (such as CTRL/s and CTRL/q) perform their special functions even if preceded by a CTRL/p. The CTRL/p does not echo at the terminal.

CTRL/r	If the current input line is not empty, this character reprints the line with editing already performed. This control character enables you to see the effects of the editing characters entered since the most recent line terminator. If the current line is empty, this character reprints the previous line, up to the point of the line terminator. Additional CTRL/r characters display previous lines, until there are no more lines in the type-ahead buffer. Subsequent CTRL/r characters display the last line found.
CTRL/u	Discards the entire contents of the type-ahead buffer.
CTRL/x	Discards the current input line. This character echoes the "#" character, followed by a carriage return/line feed, at the terminal.
CTRL/z	If entered as the only character in a line, this character specifies an end-of-file, terminating a read from the terminal. If entered on a non-empty line, it terminates the line without appending a carriage return/line feed pair to the line.

CONTROLLING OUTPUT TO A TERMINAL

When sending output to a terminal, the Terminal Support Code always operates in one of four modes. You can switch the current output mode dynamically to any of the other output modes by entering output control characters. The output modes and their characteristics are as follows:

Normal	The Terminal Support Code accepts output from the application system and immediately passes the output to the terminal for display.
Stopped	The Terminal Support Code accepts output from the application system, but it queues the output rather than immediately passing it to the terminal.
Scrolling	The Terminal Support Code accepts output from the application system, and it queues the output as in the stopped mode. However, rather than completely preventing output from reaching the terminal, it sends a predetermined number of lines (called the <u>scrolling count</u>) to the terminal whenever the operator enters a control character at the terminal.
Discarding	The Terminal Support Code discards output from the application system without displaying or queuing the output.

The following control characters, when entered at the terminal, change the output mode for the terminal. Like the input control characters, these are defaults. They can be changed by a selection process described in the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

CTRL/o	Places the terminal in discarding mode if the terminal is in a mode other than discarding mode. If the terminal is already in discarding mode, the CTRL/o character returns the terminal to its previous output mode.
CTRL/q	Resumes previous output mode. If you enter this character after stopping output with the CTRL/s character, output continues in the same manner as before you entered the CTRL/s (that is, if your terminal was in scrolling mode before you entered CTRL/s, output resumes in scrolling mode). Entering CTRL/q at any other time places your terminal in normal mode (that is, all output is displayed at the terminal without waiting for permission to continue). Therefore, you can use CTRL/q to reverse the effect of a CTRL/w and get your terminal out of scrolling mode.
CTRL/s	Places the terminal in stopped mode (stops output). You can resume output without loss of data by entering the CTRL/q character. If the terminal is in discarding mode (as a result of a CTRL/o character), the CTRL/s character has no effect on output.
CTRL/t	Places the terminal in scrolling mode and sets the scroll count to one. This means that you must enter another CTRL/t character after each displayed line in order to continue the display.
CTRL/w	Places the terminal in scrolling mode. In this mode, the terminal displays output several lines at a time (usually, enough lines to fill the screen) and then waits for user input to continue. When you enter another CTRL/w character, the terminal displays the next screen of information. The scrolling count is selectable; refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for more information. Entering the CTRL/w character while the terminal is scrolling increments the scrolling count by the original scrolling count value. Therefore, you can use CTRL/w to increase the number of lines the terminal displays before stopping. Entering an input line resets the scroll count to its original value.

An additional control character is supported which, although it doesn't affect the output mode of the terminal, can affect output to the terminal. This character is:

CTRL/c	Deletes the type-ahead buffer and causes the Operating System to abort the currently-executing program. If you enter a Human Interface command to initiate a program, you can enter CTRL/c to stop it.

For an overview of the control characters see Table 1-1 and Table 1-2.

Table 1-1. Overview of Default Control Characters

Characters	Results
Default Input Control Characters	
carriage return or line feed	terminates current line and puts cursor at start of next line
rubout	deletes single character
CRTL/p	removes any meaning special to terminal support code
CRTL/r	reprints line
CRTL/u	discards type-ahead buffer
CRTL/x	discards current input line
CRTL/z	specifies an end of file
Default Output Control Characters	
CRTL/o	places terminal in discarding mode
CRTL/q	resumes output mode
CRTL/s	stops output
CRTL/t	scrolls output one line at a time
CRTL/w	scrolls output
CRTL/c	aborts currently executing program

ESCAPE SEQUENCES

The Terminal Support Code also accepts escape characters that you can enter to further define your terminal. (For example, you could set the scroll count or switch your terminal into transparent mode so that control characters have no effect.) You can enter these escape characters from the terminal, or you can write them to the terminal from a program. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for more information about these escape characters.

***

CHAPTER 2
USING THE HUMAN INTERFACE

This chapter discusses how to use the Human Interface. It doesn't provide detailed descriptions of individual commands. These descriptions are in Chapter 3. However, it does address the following topics:

- Requirements for including the Human Interface in your system.
- Configurable features of the Human Interface.
- The process of loading and accessing the Human Interface.
- The iRMX 86 file structure and file-naming conventions (including wild cards).
- Devices supported by the Human Interface.
- Automatic Device Recognition.
- The general syntax of a command.
- The system manager.

REQUIREMENTS

This section explains the basic software and hardware requirements for running the Human Interface.

iRMX™ 86 LAYERS

The Human Interface is a layer of the iRMX 86 Operating System. To include the Human Interface in your application system, you must also include the following additional layers:

- Nucleus
- Basic I/O System
- Extended I/O System
- Application Loader

During command execution, the Human Interface invokes the services of these other iRMX 86 layers in a way that is transparent to the operator.

Therefore, an operator needs little or no knowledge of operating system structures to load and execute programs from the console keyboard. For more information about iRMX 86 configuration, refer to the iRMX 86 CONFIGURATION GUIDE.

HARDWARE REQUIREMENTS

You can implement the iRMX 86 operating system on five different Intel microprocessors. These CPUs are the iAPX 86, iAPX 88, iAPX 186, iAPX 188, and the iAPX 286 microprocessors. The iRMX 86 INSTALLATION GUIDE explains how to install the Operating System on boards containing each of the microprocessors.

Although you can use different Intel microprocessors to run the iRMX 86 operating system, the Human Interface does not change its appearance to the user. Every command in this manual retains the format you see described regardless of the microprocessor you use.

CONFIGURABLE FEATURES OF THE HUMAN INTERFACE

The Human Interface, like the other layers of the iRMX 86 Operating System, is configurable. Thus, any description of how to use the Human Interface depends a great deal on its configuration. This manual describes several features of the Human Interface that may be different (or not present at all) in your system. The configurable items that are the most visible to the operator include:

- Multi-access. If your Human Interface is configured for multi-access, several users can access the Human Interface at once via separate terminals. One of the users, the system manager, has more capabilities than other users and is responsible for managing system resources and controlling who can use the system. Users of a multi-access Human Interface are concerned about user IDs, access rights to files, and attaching and detaching devices -- all in relation to the other users of the system.

  However, if your Human Interface is configured for single access, you are less interested in much of this information. You are the only user accessing the system; therefore you are not as interested in user IDs and the system manager. You have no great concern about file access rights since all the files on the system are yours.

  This manual attempts to satisfy both users. It explains all the information that the user of a multi-access Human Interface needs, but it also points out cases where information does not apply to users of single-access systems. In all cases, the information required by a user of a single-access Human Interface is a subset of the information required by a user of a multi-access system.

- Initial program. During initialization, the Human Interface starts an initial program for each terminal. This initial program is a Command Line Interpreter (CLI), a program that reads commands and starts those programs running.

  This manual assumes that the initial program for all users is the CLI supplied by the Human Interface. If your Human Interface is configured with a different initial program, the information in this manual might not describe your Human Interface accurately. The system prompts might be different, the command syntax might be different, or you might be restricted to using a special program such as an interpreter or a transaction processor. If you suspect that your initial program is not the standard CLI, contact the person who configured your system to determine the differences.

- RAM Disk. The RAM Driver treats a reserved area of RAM as if that area of RAM were a physical device. Reserving RAM for the RAM Driver occurs at system configuration time. Once you have allocated an area of memory for the RAM Driver, you can use such commands as FORMAT and ATTACHDEVICE to manipulate the contents of of the reserved memory.

There are other configuration options that affect how the system appears to a user. When describing these items, this manual points out their configurable nature and urges you to consult the iRMX 86 CONFIGURATION GUIDE. If you are not involved in iRMX 86 configuration, contact the person who configured your system to obtain more information.

LOADING THE OPERATING SYSTEM

Before you can access the Human Interface, someone must first load the Operating System into the memory of your iRMX 86 system and start it running. This process can vary from system to system (depending on such things as the monitor you use), but generally it involves one of the following procedures:

- Connecting the target system (the iRMX 86 system) to an Intel Microcomputer Development System and using the iSDM 86 or iSDM 286 package to load the Operating System from development system files to memory in the iRMX 86 system. This procedure is normally done during the development phase of an application system, when some of the system elements are still undergoing development. Refer to the iSDM 86 SYSTEM DEBUG MONITOR REFERENCE MANUAL and the iSDM 286 SYSTEM DEBUG MONITOR REFERENCE MANUAL for more information.

- Using the iRMX 86 Bootstrap Loader to load the Operating System from iRMX 86 files to memory.

LOADING THE OPERATING SYSTEM ON AN OEM SYSTEM

On systems which are not part of Intel's family of integrated systems, such as the System 86/310, you must verify that the device contains the correct volume and then perform the following steps:

1. Reset the system; usually, reset involves pressing a RESET button on the system chassis. A series of characters (usually asterisks) should appear at the system terminal (the one connected to the processor board). If you have an iSDM 286 monitor, no characters appear on the screen; you simply go to step 2.

2. Type an uppercase U at the system terminal. This procedure accesses the resident monitor. The monitor displays the following information:

   ```
   iSDM xxx MONITOR Vx.y
   COPYRIGHT <year> INTEL CORPORATION

   .
   ```

   The period (.) is the monitor prompt. iSDM xxx indicates which monitor you are using and Vx.y which version of the monitor you are using.

3. Use the monitor's B command to bootstrap load the Operating System. In most cases you do this by entering:

   ```
   .B
   ```

For the default configuration of the Bootstrap Loader, this command loads a file with pathname SYSTEM/RMX86 from the first available device. If your Operating System resides on a file with a different pathname, you must specify that pathname in the B command. Refer to the iRMX 86 LOADER REFERENCE MANUAL, and the iRMX 86 CONFIGURATION GUIDE.

LOADING THE OPERATING SYSTEM ON INTEL INTEGRATED SYSTEMS

On Intel integrated systems (such as the System 86/310), you must verify that the device contains the correct volume and then perform the following steps:

1. Turn on the power to the terminals and to the system. If your system contains multiple chassis (such as the 86/380 Microcomputer System), turn on the power to the PERIPHERAL chassis before turning on the power to the PROCESSOR chassis.

2. Within a few seconds, the terminal connected to the system's processor board (J1 connector) should begin displaying a series of asterisks if the microprocessor is an iAPX 86 CPU. If the microprocessor running the system is an iAPX 286 CPU, the display will show only one astrisk. In either case, the system requires no further input from you to load the Operating System from your hard disk. In approximately 12 seconds, the System Confidence Test (SCT), a diagnostic program residing in PROM which performs an initial check of your hardware, begins to run automatically. The SCT executes in its Terse Mode output (a short version of the original SCT's output). If your want to run other modes of the SCT, consult the SYSTEM x86/300 SERIES DIAGNOSTIC MAINTENANCE MANUAL for more information. The system assumes a baud rate of 9600 for your terminal.

3. When the SCT successfully completes its check of the system, you will see a display which is similar to Figure 2-1. Figure 2-1 shows the display you would see if your microcomputer was a member of the System 286/300 series. But every display which the SCT shows in its Terse Mode contains the same type of information.

4. When the SCT is successful, it invokes the Bootstrap Loader, which attempts to load a file with the pathname /SYSTEM/RMX86. The Bootstrap Loader, along with the SCT, resides in PROM.

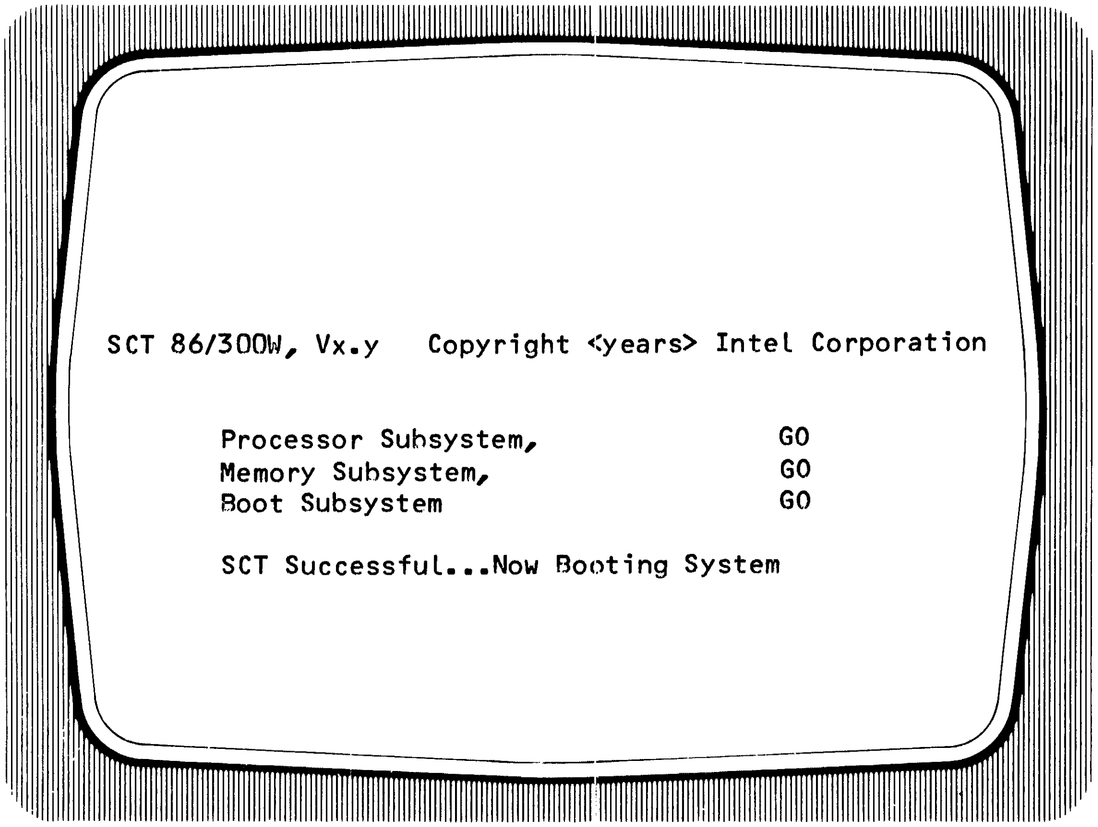
5. If the SCT encounters a problem, it transfers control to the monitor. Refer to the SYSTEM 86/300 SERIES DIAGNOSTIC MAINTENANCE MANUAL if the Operating System fails to load. When the Bootstrap Loader completes loading, you can access the Human Inteface from any terminal, as described in the next section.

LOADING THE OPERATING SYSTEM WITHOUT A RESIDENT MONITOR

For some custom systems which do not include one of the monitors, you simply ensure that an iRMX 86-formatted volume containing the Operating System resides in the proper device and reset the system.

LOADING FROM A PROM-BASED OPERATING SYSTEM

Some systems contain the entire Operating System in PROM and do not require you to load additional information from secondary storage. The usual process for starting these systems is simply to reset the system. If you were not involved in the configuration of your system and are unsure about how to load and start the Operating System, contact the person who configured your system.



```
SCT 86/300W, Vx.y   Copyright <years> Intel Corporation


          Processor Subsystem,             GO
          Memory Subsystem,                GO
          Boot Subsystem                   GO

          SCT Successful...Now Booting System
```

Figure 2-1. Example Output of a System 86/300 SCT (Terse Mode)

ACCESSING THE HUMAN INTERFACE

Assuming that the Operating System software is loaded into the system, you access the Human Interface by powering on your terminal. If your application system is configured for automatic baud rate recognition, you must also enter the following character at the keyboard:

U (uppercase U)

This character allows the Operating System to determine the baud rate of your terminal.

When the Human Interface starts running, it creates an environment for you to enter commands. This environment is an iRMX 86 job, which this manual refers to as an interactive job.

As part of creating this interactive job, the Human Interface assigns you a user ID. This user ID is your "identity" in the system. It determines your access to files and devices. Whenever you create files, the Human Interface assigns your user ID as the owner ID of the file. Being the owner of a file gives you complete control over the file; you can read it, delete it, write it, update it, and select the access that you wish to grant to other users. Your own ability to access files created by other users depends on the access they grant you.

Once the interactive job has been created by the Operating System, an initial program begins execution. The initial program that runs in your interactive job (at your terminal) may be different from one that runs at another terminal. (A configuration option specifies which initial programs are associated with which user IDs.) Initial programs are command line interpreters (CLIs), which read and parse command input and start programs running based on that input. The Human Interface supplies a standard CLI, which this manual assumes you are using. The standard CLI begins running by displaying the following (configurable) header message and prompt:

```
    iRMX 86 HI CLI, V<x.y>: user = <user ID>
    Copyright <year> Intel Corporation
    -
```

where:

V<x.y>	The version number of the Human Interface.
user = user ID	A display of your user ID. The Human Interface uses this ID to determine the type of access you have to files and devices. Most single-access systems are set up to give you an ID of WORLD (65535 decimal), but some may differ. The user ID WORLD is compatible with multi-access systems (if transferring files is necessary), because every multi-access user has read and write access to files created by WORLD.
- (hyphen)	The Human Interface prompt. This prompt implies that the CLI is ready to accept command input.

If the information that appears at your terminal is different from this, contact the person who configured your iRMX 86 Operating System to determine the differences between your initial program and the standard CLI.

Next, the standard CLI searches for the logon file, a file whose pathname is :PROG:R?LOGON (later sections of this chapter discuss pathnames of files). There can be a file with this name for each user of the system.

The CLI expects to find command invocation lines in this file. When it finds this file, the CLI automatically invokes the SUBMIT command to process all the commands in the file (refer to Chapter 3 for more information about SUBMIT). You can modify the information in your :PROG:R?LOGON file to change the amount of processing that occurs automatically when the Operating System recognizes your terminal. The Operating System does not have a default R?LOGON file. If the Human Interface does not find a R?LOGON file, it returns an error message. You can ignore the error message.

As supplied with the Start-Up versions of the Operating System, the R?LOGON file for each user contains the DATE and TIME commands which ask you for the correct date and time as follows:

```
-date query

DATE:
```

In response, enter the date. Any of the following formats is acceptable:

```
5/13/83
1 OCT 83
25 OCTOBER 83
```

If you use an improper format, the DATE discards your entry and prompts you for another date. For more information, refer to the description of the DATE command in Chapter 3. After you enter the date correctly, DATE responds by displaying the date. Then the following display occurs:

```
-time query

TIME:
```

In response, enter the correct time in the format:

```
hours:minutes:seconds
```

You can omit the last field or the last two fields. TIME sets the omitted fields to zero. The following are all valid times:

```
13:02:45
8:34
17
```

For more information, refer to the description of the TIME command in Chapter 3. TIME responds by displaying the date and time.

After processing all the commands in the logon file, the CLI issues its prompt (-) and returns control to you. At this point you can enter Human Interface commands and invoke programs.

FILE STRUCTURE

One of the primary uses of Human Interface commands is manipulating files. Before you can use the Human Interface commands described in Chapter 3, you should have an understanding of the kinds of files that exist in an iRMX 86 environment and how to access those files.

TYPES OF FILES

There are three basic types of files in an iRMX 86 environment: named files, physical files, and stream files. These files are used as follows:

Named files	Named files divide the data on mass storage devices into individually-accessible units. Users and programs refer to these files by name when they want to access information stored in them. Terminal operators access named files more often than any other file type.
Physical files	Physical files are mechanisms by which the Operating System can access an entire I/O device as a single file. The Human Interface accesses backup volumes and devices such as line printers and terminals in this manner. It also accesses secondary storage devices (such as disk drives) as physical devices when formatting them. When terminal operators access physical files, it is usually in a manner that is transparent to them (such as copying a named file to the line printer or formatting a disk).
Stream files	Stream files are mechanisms for communicating between programs. Two programs can use a stream file for communication if one program writes information to the stream file while another program reads the information. Terminal operators seldom use stream files directly.

When manipulating data with Human Interface commands, you are most often dealing with named files. Therefore it is important that you know about the hierarchy of named files and file-naming conventions. The next sections discuss these topics in detail.

NAMED FILE HIERARCHY

The iRMX 86 Operating System allows you to organize named files into structures called file trees, as shown in Figure 2-2. Figure 2-3 shows the actual file structure you recieve in a Start-Up system. The file structure in Figure 2-3 is what you would see if you kept the original file hierarchy intact in your system.

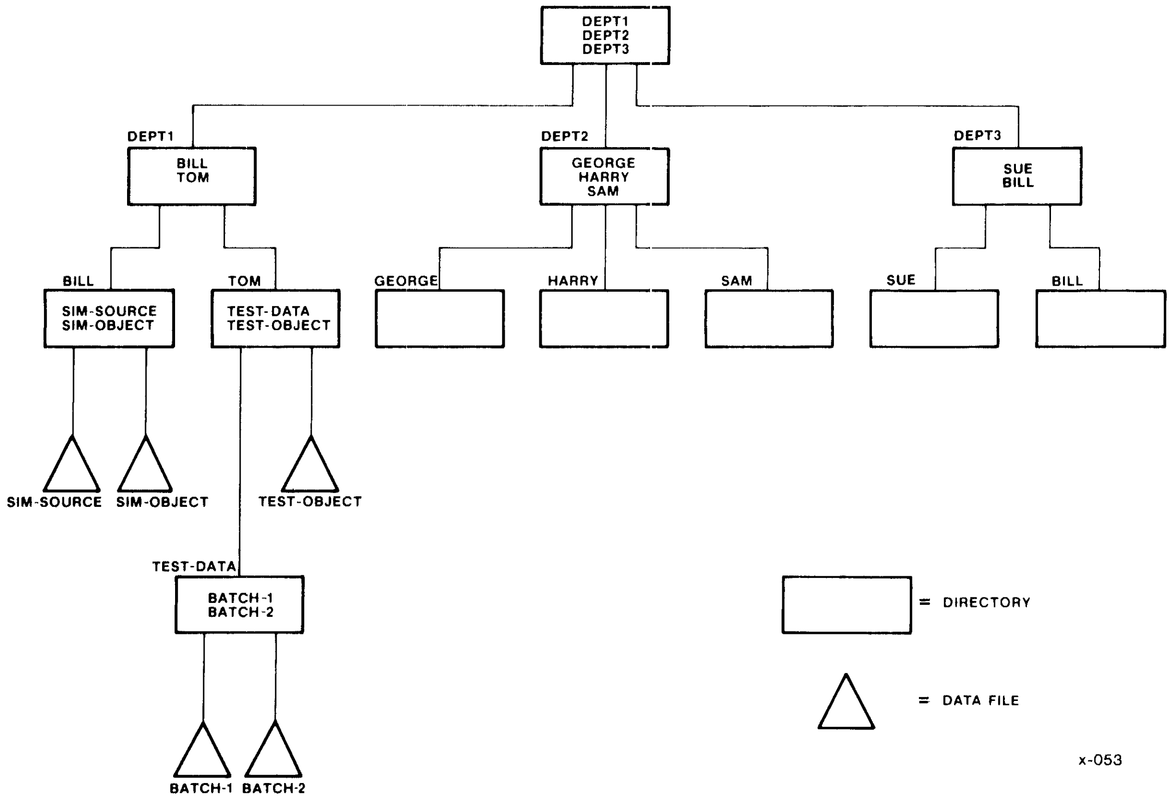


Figure 2-2. Example of a Named-File Tree

As you can see from the figure, there are two kinds of files in the file tree: data files and directories. Data files, (shown as triangles in Figure 2-2) contain the information that you manipulate in the course of your terminal session (for example, inventory, accounts payable, text, source code, and object code). Directories (shown as rectangles in Figure 2-2) contain only pointers to other files (either named files or directories). The iRMX 86 Operating System allows you to have multiple directories in a hierarchical structure so that instead of having a single directory containing an enormous number of files, you can organize your files into logical groupings under several directories. You can display the list of files in any directory by invoking the DIR command for that directory (refer to Chapter 3 for more information).

Another advantage of hierarchical file structure is that duplicate file names are permitted unless the files reside in the same directory. Notice in Figure 2-2 that the file tree contains two directories named BILL. (These directories are on the extreme left and extreme right of the figure.) However, the Operating System recognizes them as unique files because each resides in a different directory.

Each file tree resides on a secondary storage volume -- the storage medium that contains the data. Examples of volumes include flexible diskettes, hard disks, and bubble memories. Before you can place named files on a volume, you must format the volume to accept named files. The formatting process writes a number of data structures on the volume to aid the Operating System in creating and maintaining files. You can use the FORMAT command (described in Chapter 3) to format your volumes.

The uppermost point of each file tree is a directory called the root directory. When formatted for named files, each secondary storage volume has one and only one root directory. For these reasons:

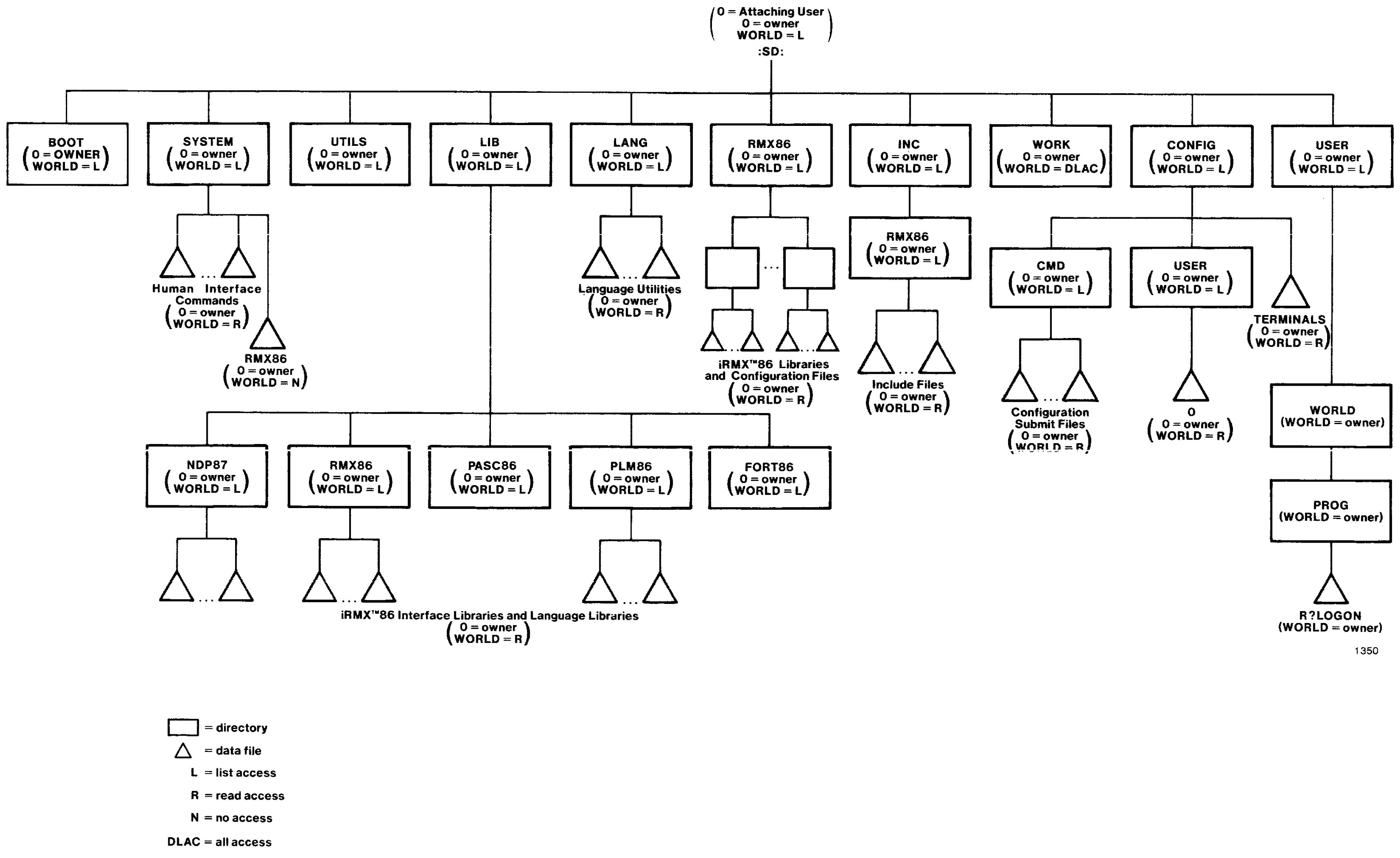
- There can be only one file tree per secondary storage volume.
- A file tree cannot extend to more than one volume.

PATHNAMES

This section describes how to specify a particular file in a named-file tree. For simplification, it assumes that all files reside in the same file tree, and thus in the same volume. To identify the volume as well as the file, you must include a logical name for the device as the first portion of the file specification. Refer to the "Logical Names" section, later in this chapter, for more information about logical names.

In a file tree, each file (data or directory) has a unique shortest path connecting it to the root directory. For example, in Figure 2-2, the shortest path from the root directory to file BATCH-2 goes through directory DEPT1, through directory TOM, through directory TEST-DATA, and finally stops at data file BATCH-2. When you want to perform an operation on a file (for example, using the COPY command to copy one file to another), you must specify not only the file's name, but the path through the file tree to the file. This description is called the file's pathname. For file BATCH-2 in Figure 2-2, the pathname is:

```
DEPT1/TOM/TEST-DATA/BATCH-2
```



(0 = Attaching User 0 = owner WORLD = L)
:SD:
BOOT (0 = OWNER WORLD = L)
SYSTEM (0 = owner WORLD = L)
UTILS (0 = owner WORLD = L)
LIB (0 = owner WORLD = L)
LANG (0 = owner WORLD = L)
RMX86 (0 = owner WORLD = L)
INC (0 = owner WORLD = L)
WORK (0 = owner WORLD = DLAC)
CONFIG (0 = owner WORLD = L)
USER (0 = owner WORLD = L)
Human Interface Commands (0 = owner WORLD = R)
RMX86 (0 = owner WORLD = N)
Language Utilities (0 = owner WORLD = R)
iRMX™86 Libraries and Configuration Files (0 = owner WORLD = R)
RMX86 (0 = owner WORLD = L)
Include Files (0 = owner WORLD = R)
CMD (0 = owner WORLD = L)
USER (0 = owner WORLD = L)
TERMINALS (0 = owner WORLD = R)
Configuration Submit Files (0 = owner WORLD = R)
0 (0 = owner WORLD = R)
WORLD (WORLD = owner)
PROG (WORLD = owner)
R?LOGON (WORLD = owner)
NDP87 (0 = owner WORLD = L)
RMX86 (0 = owner WORLD = L)
PASC86 (0 = owner WORLD = L)
PLM86 (0 = owner WORLD = L)
FORT86 (0 = owner WORLD = L)
iRMX™86 Interface Libraries and Language Libraries (0 = owner WORLD = R)
1350
= directory
= data file
L = list access
R = read access
N = no access
DLAC = all access

Figure 2-3. File Structure on an Intel Supplied Start-Up System

This pathname consists of the names of files (in uppercase or lowercase characters; the Operating System treats them as the same) and separators. In this case, slashes (/) separate the individual components of the pathname and tell the Operating System that the next component resides down one level in the file tree. You can use another separator, the circumflex or up-arrow (^), btween path components. Each circumflex tells the Operating System that the next path component resides up one level in the file tree. The following pathname, although not the shortest possible pathname, indicates another path to file BATCH-2:

```
DEPT1/BILL^TOM/TEST-DATA/BATCH-2
```

If you always start at the root directory, the circumflex separator is not very useful, since you usually want to traverse down the file tree. However, in some systems, your starting point in the file tree may be a directory other than the root directory. In such cases the circumflex separator is useful in accessing files in other branches of the file tree. Your default prefix (discussed later in the "Logical Names" section of this chapter) determines your starting point in the file tree.

For example, suppose your starting point in the file tree is the directory TOM shown in Figure 2-2. In order for you to access a file in directory BILL from this starting point, you must use the circumflex in the pathname. To indicate file SIM-SOURCE in directory BILL, you could enter the pathname:

```
^BILL/SIM-SOURCE
```

This path tells the Operating System to go up one level in the file tree from the starting point (to directory DEPT1 from directory TOM), search in that directory for directory BILL, and search in directory BILL for file SIM-SOURCE.

More than one circumflex allows you to go up any number of levels within the file structrue. For example, if your starting point is TOM, then you can go up to the root directory by using two circumflexes.

Another way to specify files in different branches of the file tree is by including the slash separator as the first character in the pathname. The slash tells the Operating System to ignore your normal starting point and begin the path from the root directory. Using the previous example where the starting point is directory TOM, another way to specify SIM-SOURCE is with the pathname:

```
/DEPT1/BILL/SIM-SOURCE
```

The initial slash causes the Operating System to search in the root directory for directory DEPT1 instead of in the normal starting directory (TOM).

LOGICAL NAMES

Although the Operating System allows you to use pathnames to refer to files, it also allows you to create symbolic names that correspond to files or devices. These symbolic names are called logical names. You can create logical names that represent devices, data files, or directories. After creating a logical name, you can refer to the entity it represents by specifying the logical name. The rules for logical names are:

- Each logical name must contain 1 to 12 ASCII characters.
- The hexadecimal representation of each character must be between 021h and 07Fh inclusive (printable characters).
- The logical name cannot include the characters colon (:), slash (/), up-arrow or circumflex (^), asterisk (*), and question mark (?).
- When you specify a logical name, you must surround it with colons.

When referring to logical names, this manual always lists the surrounding colons.

For an example of how to use logical names, refer again to Figure 2-2. Suppose you have created a logical name called :ME: that represents the pathname DEPT1/TOM/TEST-DATA (a later paragraph in this section discusses how to create this logical name). If you want to refer to the directory TEST-DATA, you can either specify its pathname as before, or you can specify the logical name :ME:. If you want to refer to the file BATCH-1 under directory TEST-DATA, you can do this in either of the following ways:

```
    DEPT1/TOM/TEST-DATA/BATCH-1
```

or

```
    :ME:BATCH-1
```

The second line shows that you can use a logical name as a beginning portion (or prefix) of a pathname. The logical name tells the Operating System where to begin in its search for the file. However, you cannot use a logical name in the middle or at the end of a pathname. If you use a logical name, you must specify it at the beginning.

Notice that you must not include a slash or circumflex between the logical name and the next path component if you want the Operating System to search down one level. If you include the slash, the Operating System ignores the normal starting point (the directory TEST-DATA) and searches for the file BATCH-1 in the root directory of the volume. If you include the circumflex, the Operating System searches up one level from the starting point.

As a Human Interface user, you deal with two general classes of logical names: logical names for devices and logical names for files.

Logical Names for Devices

Device logical names allow you to refer to specific devices by name. The Operating System can establish logical names for devices during system initialization. You can establish other logical names for new or existing devices by invoking the ATTACHDEVICE command (see Chapter 3 for details).

By using device logical names as the prefix portion of your pathname specifications, you can refer to any file on any device. For example, suppose your system contains two flexible disk drives for which you have established logical names :F0: and :F1:. (You used the ATTACHDEVICE command to attach the devices as :F0: and :F1:.) If you have a diskette containing the file DEPT2/HARRY, you could place the diskette in drive :F0: and access the file with the pathname:

```
    :F0:DEPT2/HARRY
```

If you put the same diskette in drive :F1:, you could access the file by specifying the pathname:

```
    :F1:DEPT2/HARRY
```

You can see that for devices containing named files, the device logical name is actually a logical name for the root directory on that device. If you enter the DIR command (described in Chapter 3) to list the directory of device :F1:, as follows:

```
    DIR :F1:
```

Logical Names for Files

A logical name for a file provides a shorthand way of accessing that file. For example, suppose you have a file that resides several levels down in the file tree, such as:

```
    :F1:DEPT1/TOM/TEST-DATA/BATCH-2
```

where :F1: is logical name for the device that contains the file. After entering this pathname a few times, you might find it inconvenient to continually enter so many characters. If so, you can establish a logical name for this pathname, such as :BATCH:. (You could also say that you attached the file with the logical name :BATCH:.) Then, whenever you want to refer to the file in a command, you can specify the logical name instead of the pathname.

If your logical names refer to directories instead of data files, you can use the logical names in the prefix portion of a pathname. For example, consider the same pathname:

```
:F1:DEPT1/TOM/TEST-DATA/BATCH-2
```

Suppose you have attached the pathname :F1:DEPT1/TOM/TEST-DATA as logical name :TEST:; therefore it is a logical name for the directory TEST-DATA. To refer to file BATCH-2, you could enter:

```
:TEST:BATCH-2
```

Logical names for files come into existence in two ways. One way is for you to invoke the ATTACHFILE command (refer to Chapter 3 for details). The other way is for the Operating System to create them. The Operating System establishes a number of logical names for files during system initialization. A later section lists these logical names.

Where Logical Names are Stored

When the Operating System creates logical names, at initialization time or as a result of ATTACHFILE or ATTACHDEVICE commands, it does so by placing the logical name, along with a token for a connection to the file or device, into an object directory. This process is referred to as cataloging the logical name (refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information about this process). The object directory that receives this information determines the scope of the logical name (that is, who can use the logical name). There are three possibilities:

Root object directory	Logical Names cataloged in the object directory of the root job can be accessed by every user. When you use ATTACHDEVICE to create logical names for devices, the Operating System catalogs the logical names in the root directory. Logical names cataloged in the root object directory remain valid until deleted or until the system is reinitialized.
Global object directory	Logical names can be cataloged in the object directory of a job that is designated as a global job (refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information about global jobs). Each interactive job (user session) is a global job. When you use ATTACHFILE to create logical names for files, the Operating System catalogs the logical names in your global job. Likewise, if you invoke any commands that issue ATTACHFILE commands

	(such as a SUBMIT command), the Operating System catalogs the logical names in your global job. You (and any commands that you invoke) can use the logical names cataloged in your interactive job. However, other users have no access to these logical names. Logical names cataloged in your interactive job remain valid for the life of your interactive job or until deleted.
Local object directory	Logical names can be cataloged in the object directory of the job itself. When you invoke a command (such as DIR), the Operating System creates a job for that command and catalogs certain objects in its object directory. A command that you create and invoke might also use iRMX 86 system calls to catalog logical names in its own object directory. Logical names cataloged in a local job remain valid only for the life of the job or until deleted.

Whenever you (or one of the commands you invoke) use a logical name, the Operating System searches for that logical name in as many as three different object directories. It first looks in the local object directory. If the logical name is not defined there, it next looks in the global object directory and finally, if necessary, the root object directory. It uses the first such logical name it finds.

Because of this order of search, you can override the system logical names (those cataloged in the root object directory) by cataloging the same logical names (but representing different files or devices) in the object directory of your interactive job. For example, suppose you used the ATTACHFILE command to attach a file with the logical name :SYSTEM:. Then, whenever you specify :SYSTEM:, the Operating System refers to your file and not the one represented by the same logical name in the root object directory.

Logical Names Created by the Operating System

The Operating System establishes several logical names that you can use without first having to create them. It catalogs some of these logical names in the root object directory (where they are available to all users). It catalogs others in global object directories (these are specific to each interactive job). It catalogs others in local object directories (these are specific to each interactive job and to each command invoked).

The Human Interface catalogs system-wide logical names in the root object directory. These logical names are available to all users and they represent the same file or device for all users. The number of logical names created and their identities depend on the configuration of your Operating System. However, the following logical names are available on most systems.

:BB:	A device that is treated as an infinite sink (byte bucket). Anything written to :BB: disappears, and anything read from :BB: returns an end-of-file.
:LANG:	A directory used to store language products, such as assemblers, compilers, and linkers.
:SD:	The system device. If you used the Bootstrap Loader to load your system, this logical name refers to the device from which the Bootstrap Loader read the Operating System file.
:STREAM:	The prototype stream file connection. To create a connection to a stream file, you must use this logical name as the prefix portion of the pathname.
:SYSTEM:	The directory containing the Human Interface commands.
:UTILS:	A directory used to store utility programs created by users.
:WORK:	A directory that Intel language translators and utilities use to store their temporary and work files.

The following logical names are cataloged in each user's global object directory. Although each user has access to these names, the names represent different files or devices for each user.

:$:	Your default prefix. This is the path to your default directory. If you do not specify a logical name (a prefix) at the beginning of a pathname, the Operating System automatically uses :$: as the prefix. In this case, the Operating System assumes that the file resides in the directory corresponding to :$:. During an interactive session, you can use the ATTACHFILE command to change the directory corresponding to :$:.
:HOME:	Your default prefix when you first start using the Human Interface. Initially, :HOME: and :$: represent the same directory. This logical name provides you with the ability to re-establish your original :$: logical name if you become lost in the hierarchical file structure. You should not use ATTACHFILE to change the directory corresponding to :HOME:.
:PROG:	A directory in which to store your programs.

The following logical names are cataloged in the local object directory of each user and each command that a user invokes. These logical names can have different meanings for each user and each command.

:CI:	The terminal keyboard (or command input). As the name implies, each user's :CI: refers to the terminal associated with that user.
:CO:	The terminal screen (or command output). As the name implies, each user's :CO: refers to the terminal associated with that user.

Upon initialization, your Human Interface may create additional logical names. These logical names are configuration parameters. Contact the person who configured your system for more information about the logical names initially available to you. The iRMX 86 CONFIGURATION GUIDE discusses this subject in more detail.

Removing Volumes from Devices

Removing volumes from devices (such as removing flexible diskettes from drives) destroys any connections that may have existed to files on that device. Therefore, any logical names that represent files on the volume are no longer valid once you remove the volume. The names remain cataloged in the directories, but they do not represent valid connections. Therefore, before removing volumes, you should invoke DETACHFILE commands to detach the files.

WILD CARDS

Wild cards provide a shorthand notation for specifying several files in a single reference when entering commands. You can use either of two special wild card characters in the last component of a pathname to replace some or all characters in that component. The wild card characters are:

? The question mark matches any single character. The Human Interface allows any character to appear in that character position. It selects every file that meets this requirement. For example, the name "FILE?" could imply all of the following files:

```
FILE1
FILE2
FILEA
```

* The asterisk matches any number of characters (including zero characters). The Human Interface allows any number of characters to appear in that character position. It selects every file that meets this requirement. For example, the name "FILE*" could imply all of the following files:

```
FILE1
FILE.OBJ
FILE
FILECHANGE
```

You can use multiple wild cards in a single pathname. For example, the name:

```
?PIF?.*
```

matches every file whose second through fourth characters are "PIF" and whose sixth character is a period. These files could include all of the following names (or more):

```
RPIFC.LIB
EPIFL.TXT
HPIFC.
```

You can use wild cards in both input pathnames (files that commands read for information) and output pathnames (files into which commands write information). For example, in the command:

```
COPY A* TO B*
```

the A* represents the input pathname and B* represents the output pathname. In this command (which copies information from one file to another), the Human Interface searches the appropriate directory for all files that begin with the "A" character. Then it copies each file to a file of the same name, but beginning with the "B" character. If the directory contains the files:

```
ALPHA
A112
A
```

the previous command would copy files in the following manner:

```
ALPHA TO BLPHA
A112  TO B112
A     TO B
```

There are several operational characteristics that you should be aware of when using wild cards:

- Wild cards are valid in the last component of the pathname only. Therefore, :F1:SYSTEM/APP1/FILE* is a valid pathname, but :F1:SYSTEM/APP*/FILE1 is not valid.

- You can negate the meaning of a wild card character by enclosing it in quotes, either single (') or double ("). For example, if you have a file named F*123, you can refer to it alone in a command by specifying F'*'123 or 'F*123'.

- When you specify input and output pathnames in commands, you can specify lists of pathnames, separated by commas. For example:

  ```
  COPY A,B,C TO D,E,F
  ```

  copies A to D, B to E, and C to F. If you use a wild cards in any one of the output pathnames, you must use the same wild cards in the same order in the corresponding input pathname. The term "same order" means that if you use both the "*" and the "?" characters, their ordering must be the same in both the input and output pathnames. For example, the following is valid:

  ```
  COPY A*B?C* TO *DE?FGH*I
  ```

  However, the following is invalid because the wild cards are out of order:

  ```
  COPY A*B?C* TO *DE*FGH?I
  ```

- If you use wild cards in an input pathname, you can omit all wild cards from the corresponding output pathname to cause the Human Interface to perform file concatenation. For example, suppose a directory contains files A1, B1, and C1. The following command is valid:

  ```
  COPY *1 TO X
  ```

  It copies files in the following manner:

  ```
  A1 TO X
  B1 AFTER X
  C1 AFTER X
  ```

  But if X is a directory, concatenation is not performed by this series of commands. Instead, the Human Interface copies each file over to the new directory. Refer to the "Command Syntax" section later in this chapter for more information about the prepositions TO and AFTER.

- The "*" character matches as close to the end of the pathname as possible. For example, suppose the directory contains the file "ABXCDEFXGH", and you enter the command:

  ```
  COPY *X* TO *1*
  ```

  This command copies:

  ```
  ABXCDEFXGH TO ABXCDEF1GH
  ```

  The first asterisk matches the characters "ABXCDEF", and the second asterisk matches the characters "GH".

- You can also use wild cards for finding invisible files. An invisible file is a file which will not appear in any normal directory listing. Thus, the file is "invisible" to the user. The name of any invisible file must begin with the prefix R?. Since the wildcard function does not automatically match invisible files, you must explicitly state the R? prefix. You do this procedure by using quotes. Thus to find all the invisible files within a directory, you use the format "R'?'*". You need to include the quotes so that "?" is not interpreted as a wildcard symbol.

DEVICES

The iRMX 86 Operating System allows you to communicate with various types of devices (disks, terminals, etc.). Each device supports one or more types of files (named or physical). The following paragraphs identify, in general, the kinds of devices with which you can communicate and the types of files supported on those devices.

Terminals	An operator needs the terminal to communicate with the Human Interface. You can also write programs that read from and write to terminals.
Disks	Disks provide permanent storage for programs and data. The Operating System allows you to communicate with a variety of disk devices: Winchester disks, other hard disks, and flexible diskettes.
Bubble Memory	You can reconfigure the Operating System to include support Bubble Memory. Once you have done this, you can treat the bubble memory as a physical device, or you can use it to store named files.

AUTOMATIC DEVICE CHARACTERISTICS RECOGNITION

Automatic device recognition gives the Operating System the ability to recognize and access named disks of different formats without requiring you to reattach the device. This feature does not work with physical files. Refer to the iRMX 86 CONFIGURATION GUIDE for more information on automatic device characteristics recognition.

HOW AUTOMATIC DEVICE CHARACTERISTICS RECOGNITION WORKS

The Basic I/O System, the Extended I/O System, and the formatting utility (either the FORMAT command or the Files Utility) combine to provide the automatic device characteristics recognition feature. They do this procedure as follows:

1. When the formatting utility formats a disk as a named volume, it formats with the same characteristics you specified when you used ATTACHDEVICE to attach the device. However, it formats track 0 with a fixed density (single density) and a fixed sector size (128 bytes) regardless of the way it formats the rest of the disk. On track 0, the formatting program places the iRMX 86 volume label, a table that describes the characteristics of the remainder of the volume (granularity, density, number of sides, etc.). Refer to the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for a description of the iRMX 86 volume label.

   Because track 0 is formatted the same way for all named disks, the Basic I/O System can access the information on track 0 without knowing the format of the remainder of the disk.

2. When you attach a device to the system as a named device (using the ATTACHDEVICE command), you specify the name of a DUIB (device-unit information block) as one of the ATTACHDEVICE parameters. The DUIB names you can use are the ones that you specified as input to the "Device-Unit Information" screen of the ICU.

   The DUIB tells the Basic I/O System which device-unit (disk drive) to attach and which characteristics (granularity, density, number of sides, etc.) to assume about the disk drive.

3. During the attach process, the Basic I/O System reads the information from track 0 (the volume label) and compares it with the information in the DUIB you specified when attaching the device. If the information does not match, the Basic I/O System performs the following operations:

   a. It compares the information in the volume label with all the other DUIBs defined for that device-unit. If it finds a match, it "switches" DUIBs and uses the matching one as the current DUIB.

   b. If none of the DUIBs defined for that device-unit match the information in the volume label, the Basic I/O System creates a temporary DUIB that does match. It uses the information from the DUIB that you specified when attaching the device and modifies it with information from the volume label. As a name for the temporary DUIB, the Basic I/O System appends a question mark (?) to the beginning of the old DUIB name.

4. Whenever you remove a disk from a drive, the Operating System automatically detaches the device. If it was accessing the device through a temporary DUIB (as opposed to the one you specified as an ATTACHDEVICE parameter), it destroys the DUIB. However, it remembers the name of the DUIB that you specified as input to ATTACHDEVICE.

5. When you insert a new disk into the drive and attempt to access it as a named volume (by invoking the DIR command, for example), the Operating System automatically reattaches the device using the same process listed in step 3.

From these steps, you can see that you can continue to change diskettes without having to detach and reattach the device. The Operating System does this change for you automatically. However, this process occurs only for named-file operations. Whenever the Operating System performs physical-file operations, it cannot use the temporary or "switched" DUIBs. Instead, it must use the DUIB you specified as a parameter to ATTACHDEVICE.

COMMANDS THAT CANNOT RECOGNIZE DEVICE CHARACTERISTICS

Because the automatic device characteristics recognition feature does not apply to physical-file operations, some Human Interface commands cannot make use of this feature. They are:

FORMAT
BACKUP
RESTORE
DISKVERIFY

Each of these commands must detach the device and re-attach it again as a physical device. This process cancels the ability of the Basic I/O System to recognize the characteristics of the volume. Therefore, these commands assume that the device characteristics are those listed in the DUIB you specified as an ATTACHDEVICE parameter. Consequently, if you do not include, for example, a DUIB for a double-sided, double-density diskette in your configuration, you cannot format such a diskette. Neither can you create a backup volume in this format nor restore information from one.

If you plan to use one of these commands and you are not sure how your device was attached, use DETACHDEVICE and ATTACHDEVICE to reattach the device with the characteristics you require.

OPERATIONAL CONSIDERATIONS FOR iSBC® 215/iSBX™ 218 DEVICES

If your system contains an iSBC 215/iSBX 218 controller, you may receive error messages that are not appropriate when switching diskettes. For example, if you attach your device as a double-sided/double-density device and insert a single-sided/single-density diskette, you will receive an I/O error message when attempting an I/O operation. In this situation, the message does not indicate a problem. If you try the I/O operation again, it will usually succeed.

COMMAND SYNTAX

This section describes the general syntax rules that apply when entering Human Interface commands at a terminal. These rules apply equally to both the supplied Human Interface commands and any user-created commands that may have been added to your system. The individual command descriptions in Chapter 3 contain additional and more specific information about each supplied Human Interface command.

The elements that form a standard command entry include a command name, required input parameters (if any), and optional parameters. The general structure of a command line is as follows (brackets [] indicate optional portions):

```
    command-name [inpath-list [preposition outpath-list]]  [parameters] cr
```

where:

command-name	Pathname of the file containing the command's executable object code.
inpath-list	One or more pathnames, separated by commas, of files to be read as input during command execution.
preposition	A word that tells the executing command how to handle the output. The four prepositions used in Intel-supplied commands are TO, OVER, AFTER, and AS.
outpath-list	One or more pathnames, separated by commas, of files that are to receive the output during command execution.
parameters	Parameters that cause the command to perform additional or extended services during command execution.
cr	A line terminator character. This character terminates the current line and causes the cursor to go to a new line. This character also causes a command to be loaded and executed if the cr character is not preceded by the ampersand (&) symbol. The RETURN (or CARRIAGE RETURN) key and NEW LINE (or LINE FEED) key are both line terminators.

You can enter all elements of a command line in uppercase characters, lowercase characters, or a mix of both. The Human Interface makes no distinction between cases when it reads command line items. In addition, you can include the following optional command line entries:

continuation mark	An ampersand character (&) indicates that the command continues on the next line. When you include the ampersand character, the Human Interface displays two asterisks (**) on the next line to prompt for the continuation line. All characters appearing after the continuation mark but before the line terminator are interpreted as comments.

Within available memory limits, you can use as many continuation lines for a given command as you desire. After you enter the line terminator without a preceding ampersand character, the invoked command receives the entire command string as a single command.

comment character	A semicolon (;) character indicates that all text following it on the current line is a non-executable comment. You can also enter comments after a continuation mark (&) but before the line terminator. A common use of comments in commands is in a SUBMIT file list of commands (see the SUBMIT command in Chapter 3).
quoting characters	Two single-quote (') or double-quote (") characters remove the semantics of special characters they surround. For example, if you surround an ampersand character (&) with single quotes, the ampersand is not recognized as a continuation character. The same holds for other characters such as asterisk (*), question mark (?), equals (=), semicolon (;), and others. The only special characters not affected by the quoting characters are the pathname separators, semicolon (;), and dollar sign ($).

Although you can use either single quotes or double quotes as quoting characters, you must use the same quoting character at the beginning and at the end of your quoted string. If you want to include the quoting character inside your quoted string, you must specify the character twice. For example:

```
'can''t'
```

You can accomplish the same effect by using the other quoting character, for example:

```
"can't"
```

Although the Human Interface places no restriction on the number of characters in a command, each terminal line can have a maximum of 255 characters, including any punctuation, embedded blanks, continuation mark, non-executable comments, and carriage return. If your command requires more characters, use continuation lines.

The following sections discuss the individual elements of the command syntax in more detail.

COMMAND NAME

Each Human Interface command is a file of executable code that resides in secondary storage. When you specify a command name, you actually specify the name of the file containing the command's code. If you write your own command (refer to the iRMX 86 HUMAN INTERFACE REFERENCE MANUAL for information), you invoke it by entering the name of the file that contains it. After you invoke a command, the Operating System loads it from secondary storage into memory and executes it in conformance with parameters you specify.

When you enter a command name, you can enter the complete pathname of the command, or, in many cases, you can enter just the last component of the pathname.

- If you enter the complete pathname of the command (that is, if you include a logical name as the prefix portion of the pathname), the Operating System searches only the device and directory you specify for the command. If it cannot find the command there, it returns an error message.

- If you enter only the last component of the pathname (such as COPY instead of :F1:SYSTEM/COPY), the Operating System automatically searches a certain number of directories for the command. It does not return an error message until it has searched each of the directories. The number of directories searched and the order of search are Human Interface configuration parameters. However, in the default case, the Operating System searches the following directories, in order, for commands:

```
    :PROG:
    :UTILS:
    :SYSTEM:
    :LANG:
    :$:
```

When writing your own commands, you can take advantage of the order in which the Operating System searches directories. For example, suppose you write your own copy command, one that provides more or different functions than the Human Interface COPY command. If you want to invoke your own program whenever you type the command "COPY", you can simply place your copy program in a file called COPY in your :PROG: directory. Since the Operating System searches the :PROG: directory before searching the :SYSTEM: directory (the directory that normally contains Human Interface commands), it will invoke your copy program when you enter the command "COPY".

If you still want to be able to invoke the Human Interface COPY command, you can do so by entering its complete pathname, that is, by entering the following:

```
:SYSTEM:COPY
```

PREPOSITIONS

Preposition parameters in a command line tell the command how you want it to process the output file or files. The Human Interface commands usually provide three options in the choice of a preposition: TO, OVER, and AFTER. The preposition AS is also available for use in the ATTACHDEVICE and ATTACHFILE commands. The TO preposition and :CO: (console screen) will be used by default if you do not specify a preposition and an output file. The prepositions have the following meaning:

TO — Causes the command to send the processed output to new files; that is, to files that do not already exist in the given directory. If a listed output file already exists, the command displays the following query at the console screen:

```
<pathname>, already exists, OVERWRITE?
```

Enter a Y or y if you wish to write over the existing file. Enter any other character if you do not wish the file to be overwritten. In the latter case, the command does not process the corresponding input file but rather goes to the next input file in the command line. Commands process input files and write to output files on a one-for-one basis. For example:

```
COPY A,B TO C,D
```

copies file A to file C and file B to file D.

OVER — Causes the command to write your input files to the output files in sequence, destroying any information currently contained in the output files. It creates new output files if they do not exist already. For example:

```
COPY SAMP1,SAMP2 OVER OUT1,OUT2
```

copies the data from file SAMP1 over the present contents of file OUT1, and copies the data of SAMP2 over the contents of file OUT2.

AFTER — Causes the command to append the contents of one or more files to the end of one or more new or existing files (file concatenation). For example:

```
COPY IN1,IN2 AFTER DEST1,DEST2
```

appends the contents of file IN1 to the the end of file DEST1, and appends the contents of IN2 to the end of DEST2.

AS — A special preposition used with the ATTACHDEVICE and ATTACHFILE commands. When you use the AS preposition, the Operating System does not assume that the command contains input pathnames and output pathnames. Rather, it sees the parameters as entities that it must associate (for example, ATTACHFILE associates a pathname with a logical name).INPATH-LIST AND OUTPATH-LIST

An inpath-list specifies the files on which a command is to operate. An outpath-list defines the destination or destinations of the processed output. Each inpath-list or outpath-list consists of a pathname (or logical name) or list of pathnames. If you specify multiple pathnames, you must separate the individual pathnames with commas. Embedded blanks between pathnames are optional. You can also use wild cards to indicate multiple pathnames (refer to the "Wild Cards" section of this chapter). Usually when you specify multiple pathnames, each pathname in the inpath-list has a corresponding pathname in the outpath-list. For example, the command:

```
COPY A, B  TO  C, D
```

copies file A to file C and also copies file B to file D. Therefore, A and C are corresponding pathnames, and so are B and D. However, there are some instances when the number of input pathnames you enter differs from the number of output pathnames. The validity of the operation depends on whether the pathname lists contain single pathnames,

Table 2-1. Input Pathname and Output Pathname Combinations

Inpath-list	Outpath-list	Human Interface Action
single pathname	single pathname	one-for-one match
single pathname	list of pathnames	error
single pathname	wild-card pathname	error
single pathname	list of wild cards	error
single pathname	pathname to directory	one-for-one match
list of pathnames	single pathname	concatenate
list of pathnames	list of pathnames	one-for-one match
list of pathnames	wild-card pathname	error
list of pathnames	list of wild cards	error
list of pathnames	pathname to directory	one-for-one match
wild-card pathname	single pathname	concatenate
wild-card pathname	list of pathnames	error
wild-card pathname	wild-card pathname	one-for-one match
wild-card pathname	pathname to directory	one-for-one match
wild-card pathname	list of wild cards	error
list of wild cards	single pathname	concatenate
list of wild cards	list of pathnames	concatenate
list of wild cards	wild-card pathname	concatenate
list of wild cards	list of wild cards	one-for-one match
list of wild cards	pathname to directory	one-for-one match

lists of pathnames, a wild-card pathname, or lists of wild-card pathnames. Table 2-1 lists the possibilities and describes the Human Interface's action in each instance. The following sections discuss the Human Interface's actions in more detail.

One-For-One Match

The combinations in Table 2-1 that are marked "one-for-one match" are those in which each element in the inpath-list is matched with an element of the outpath-list. An example of this is the command:

```
    COPY A*, B*  TO  C*, D*
```

In this case, the Human Interface copies all files beginning with the character "A" to corresponding files beginning with the character "C". When it finishes this operation, it advances past the comma to the next set of pathnames (copies all files beginning with "B" to corresponding files beginning with "D").

Concatenate

The combinations in Table 2-1 that are marked "concatenate" are those in which there are multiple input pathnames that correspond to a single output pathname. In this situation, the Operating System automatically appends the remaining input files to the end of the specified output file, regardless of the preposition you specify.

This allows you to combine one-for-one file operations (as in TO or OVER preposition) with file concatenation (as in the AFTER preposition) in a single command, and thus avoid entering an extra command to perform a separate concatenation operation. The following example explains this situation.

Assume that in a COPY command, you use the TO preposition and specify the following input and output pathnames:

```
        COPY A,B,C  TO  D
```

When the Human Interface processes the command line, it copies file "A" to file "D" and appends files "B" and "C" to the end of file "D" as follows:

```
        A  TO  D
        B  AFTER  D
        C  AFTER  D
```

Notice that this concatenation occurs only when there are multiple elements in the inpath-list that correspond to a single element of the outpath-list. This means that the following commands are invalid:

```
    COPY A, B, C     TO  D, E      ; INVALID COMMAND

    COPY A*, B*, C*  TO  D*, E*    ; INVALID COMMAND
```

Error Conditions

The combinations in Table 2-1 that are marked "error" indicate invalid operations. For these combinations, the Human Interface returns an error message without performing the requested operation.

OTHER PARAMETERS

Most commands allow you to enter parameters other than inpath-lists, outpath-lists, and prepositions. These other parameters are known as keyword parameters, because you must enter a particular word, called a keyword, to obtain the additional or extended services provided by the parameter.

For example, the DIR command (described in Chapter 3) lists the contents of a directory. You can enter several different keyword parameters to specify the amount of information displayed and the format of the display. A command such as:

```
    DIR :SYSTEM: EXTENDED
```

displays the contents of the :SYSTEM: directory in extended format. You could substitute other keywords such as SHORT or LONG to obtain different formats.

The command descriptions in Chapter 3 list the keyword parameters available with each command. However, the descriptions list the complete names for the keywords. When you use keywords, you can enter their complete names or you can enter only as many characters as are necessary to uniquely identify the keyword. For example, you could enter the previous command as:

```
    DIR :SYSTEM: E
```

For the DIR command, the character E uniquely identifies the EXTENDED parameter. Other keywords might require additional characters to make them unique.

Some keyword parameters also require an associated value. An example of this is the FORMAT command (described in Chapter 3), which prepares secondary storage volumes for iRMX 86 use. A command such as:

```
    FORMAT :F1:TEST  FILES = 60
```

formats a volume on device :F1: and sets up the volume to contain at most 60 files. The keyword in this command (FILES) has an associated value (60). Although this example and the descriptions in Chapter 3 use the equal sign (=) to associate keywords and values, there are actually two ways to do this. They are:

```
keyword = value
keyword (value)
```

The blanks are optional. You can use either method when entering Human Interface commands.

SYSTEM MANAGER

The multi-access Human Interface supports a user called the system manager. The system manager's primary purpose is to maintain the multi-access configuration files. The system manager can modify these files to add or delete user IDs, add or delete terminals, and change terminal or user characteristics (refer to the iRMX 86 CONFIGURATION GUIDE for more information). For security reasons, no user other than the system manager can access these files.

In addition, the system manager has a special user ID which gives that user privileges that other users do not have. The system manager:

- Has read access to all data files and list access to all directories.
- Can change the access rights of any file, regardless of the file's owner.
- Can detach devices attached by any user.
- Can delete any user from the system.

Any operator can become the system manager by invoking a Human Interface command called SUPER. This command (which requires entering a password) changes the operator's user ID from its normal value to that of the system manager. Once an operator invokes SUPER, that operator has all the powers of the system manager. Refer to Chapter 3 for more information about the SUPER command.

***

CHAPTER 3 HUMAN INTERFACE COMMANDS

This chapter presents the commands in alphabetical sequence without regard for functional organization. The Human Interface Command Dictionary (Table 3-1) also lists a functional grouping of the commands for fast reference.

The commands described in this chapter are supplied by Intel for iRMX 86 Operating Systems that are configured with the Human Interface. If you are a new user of the Human Interface, it is suggested that you review the information on file-naming conventions and invocation considerations in Chapter 2 before reading this chapter.

This chapter does not describe how to specify the names of the devices and directories that contain the Human Interface commands. This is because during the Human Interface configuration process you can specify a number of directories that the Human Interface automatically searches for commands. If you place your Human Interface commands in one of these directories (normally the :SYSTEM: directory), you can invoke the commands by entering only their names. However, if your commands reside in a directory that the Human Interface does not search automatically, or if you have multiple commands with the same name in different directories, you can use the complete pathname for the command. For example, if the DIR command resides in directory COMMANDS on device :F6: (a directory not normally searched by the Human Interface), you can invoke the command by entering:

```
:F6:COMMANDS/DIR
```

Refer to the iRMX 86 CONFIGURATION GUIDE for more information about Human Interface Configuration.

ERROR MESSAGES

Each command can generate a number of error messages which indicate errors in the way you specified the command. The messages that apply to a specific command are listed with that command. However, the following are general error messages that can appear with many of the commands:

- command not found

  There is no file whose pathname is the same as the command name you specified, nor can the Human Interface find the file in any of the directories it automatically searches.

- <logical name>, device does not belong to you

  The device you specified was originally attached by a user other than WORLD or you.

- <pathname>, file does not exist

  The pathname you specified does not represent an existing file.

- <pathname>, invalid file type

  You specified a data file for an operation that required a directory, or vice versa.

- <logical name>, invalid logical name

  The logical name you specified contains unmatched colons, is longer than 12 characters, or contains invalid characters.

- <pathname>, invalid pathname

  The pathname you specified contains invalid characters or a component of the pathname (other than the last one) does not exist or does not represent a directory.

- <logical name>, is not a device connection

  The logical name you specified does not represent a connection to a physical device.

- <logical name>, logical name does not exist

  The logical name you specified does not exist.

- parameters required

  The command you specified cannot be entered without parameters.

- program version incompatible with system

  The command and the Operating System are not compatible. The command expects to obtain information from internal tables that are not present. Therefore the command cannot run successfully.

- <control>, unrecognized control

  The parameter you entered is not valid for the specified command.

- <exception value> : <exception mnemonic>, while loading command

  The Operating System encountered an exceptional condition while attempting to load the command into memory from secondary storage. The message lists the exception code encountered.

- <exception value> : <exception mnemonic>

  An operational error occurred during the execution of the command. The <exception value> and <exception mnemonic> portions of the message indicate the exception code encountered.

- <parameter>, <exception value> : <exception mnemonic>

  The command encountered an exceptional condition while attempting to process the <parameter> portion of the command. The <exception value> and <exception mnemonic> portions of the message indicate the exception code encountered.

COMMAND SYNTAX SCHEMATICS

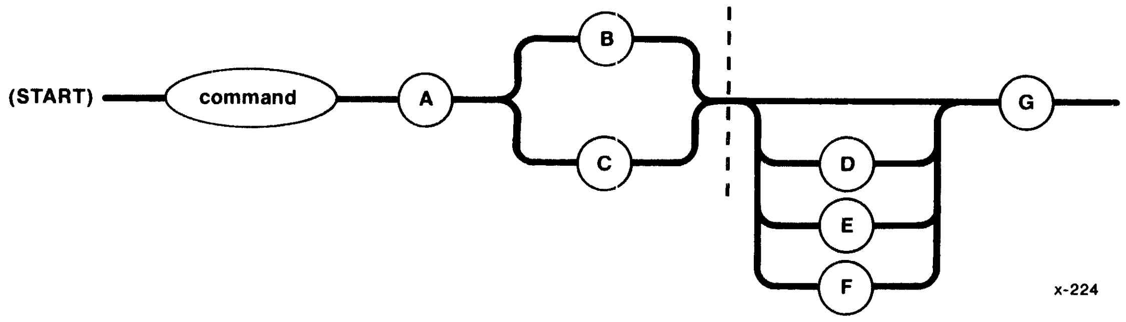
The syntax for each command described in this chapter is presented by means of a "railroad track" schematic, with syntactic elements scattered along the track. Your entrance to any given schematic is always from left to right, beginning with some command name entry.

Elements shown in uppercase characters must be typed in a command line exactly as shown in the command schematics except that you can type them either in uppercase or lowercase characters; the Human Interface makes no distinction between cases in alphabetic characters. Syntactic elements shown in lowercase characters are generic terms, which means that you supply the specific item, such as the pathname for a file.

The vertical dotted line separates the position-dependent parameters from those that are position-independent. Parameters to the left of the dotted line must be entered in the order listed (from left to right). Parameters to the right of the dotted line can be entered in any order (as long as they obey the rest of the syntax).

The example that follows shows all the possible paths through a railroad track schematic. Notice that the main track goes through required elements in a given command.

"Railroad sidings" go through optional parameter elements. In some cases, you have a choice of going through one of several possible sidings before returning to the main track. In still other cases, the main track itself diverges into two separate tracks, which means that you must select one parameter or the other but not both.



In this example:

- A is a required element. It is position-dependent; it must be entered first.

- Either B or C is required but not both. These elements are also position-dependent. Whichever element you enter must follow A immediately.

- D, E, or F are all optional but only one can be selected. These are position-independent elements. If you select one of these elements, you can enter it before or after G.

- G is required. It is a position-independent parameter. You can enter it before or after D, E, or F.

Table 3-1. Human Interface Command Dictionary

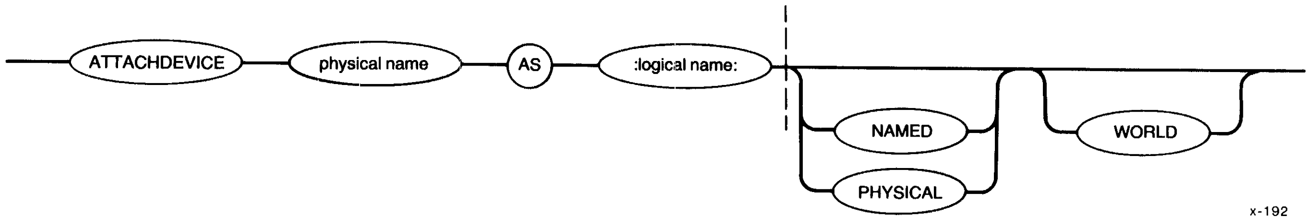
Command	Synopsis	Page
File Management Commands		
ATTACHFILE	Associates a logical name with an existing file.	13
COPY	Creates new data files, or copies files to other pathnames.	24
CREATEDIR	Creates one or more new directories.	28
DELETE	Deletes data files and empty directories from a volume on secondary storage.	33
DETACHFILE	Removes the association of a logical name with a file.	38
DIR	Lists a directory's filenames (and optionally, file attributes).	40
DOWNCOPY	Copies files and directories from an iRMX 86 volume mounted on a secondary storage device to an ISIS-II secondary storage device.	53
PERMIT	Grants or rescinds user access to a file.	83
RENAME	Renames files or directories.	88
UPCOPY	Copies files and directories from an ISIS-II secondary storage device to an iRMX 86 volume mounted on a secondary storage device.	107
Volume Management Commands		
ATTACHDEVICE	Attaches a new physical device to the system and catalogs its logical name in the root job's object directory.	7
BACKUP	Copies named files to a backup volume.	16
DETACHDEVICE	Removes a physical device from system use and deletes its logical name from the root job's object directory.	35
DISKVERIFY	Verifies the data structures of named and physical volumes.	48
FORMAT	Formats an iRMX 86 volume.	56

Table 3-1. Human Interface Command Dictionary (continued)

Command	Synopsis	Page
Volume Management Commands (continued)		
LOCDATA	Puts relocatable programs in absolute locations.	71
RESTORE	Copies files from a backup volume to a named volume.	91
Multi-Access Commands		
INITSTATUS	Displays the initialization status of Human Interface terminals.	67
JOBDELETE	Deletes a running interactive job.	69
LOCK	Prevents the Human Interface from automatically creating an interactive job.	75
SUPER	Changes the operator's user ID into that of the system manager (user ID 0).	102
General Utility Commands		
DATE	Sets or resets the system date, or displays the current date.	29
DEBUG	Transfers control to the iSDM 86 or iSDM 286 monitor to debug an iRMX 86 application program.	31
LOGICALNAMES	Lists all the logical names within the system	77
MEMORY	Displays the memory available to the user.	80
PATH	Shows the pathname for a file.	81
SUBMIT	Reads, loads, and executes a string of commands from secondary storage instead of the keyboard.	98
TIME	Sets or resets the system clock, or displays the current system time.	105
VERSION	Displays the version numbers of commands.	110
WHOAMI	Displays the current ID associated with the user.	112

ATTACHDEVICE

This command attaches a physical device to the Operating System and associates a logical name with the device. The command catalogs the logical name in the root object directory, making the logical name accessible to all users. The format of the command is as follows:



INPUT PARAMETERS

physical name	Physical device name of the device to be attached to the system. This name must be the name used in one of the Basic I/O System's Device Unit Information Blocks (DUIB), as defined at system configuration time (see Table 3-2).
AS	Preposition; required for the command.
:logical name:	A 1- to 12-character name, that represents the logical name to be associated with the device. Colons surrounding the logical name are optional; however, if you use colons, you must use matching colons.
NAMED	Specifies that the volume mounted on the device is already formatted for NAMED files. Examples of volumes that can contain named files are diskettes or hard disk platters. If neither NAMED nor PHYSICAL are specified, NAMED is the default. See the FORMAT command in this chapter for a further description of NAMED files.
PHYSICAL	Specifies that the volume mounted on the logical device is considered to be a single, large file. Examples include line printers and terminals. See the FORMAT command in this chapter for a further description of PHYSICAL volumes.

WORLD	Specifies that user ID WORLD (65535 decimal) is the owner of the device. This implies that any user can detach the device. If you omit this parameter, your user ID is listed as the owner of the device. In this case, only you and the system manager can detach the device.

DESCRIPTION

ATTACHDEVICE attaches a device to the system and catalogs a logical name for it in the root job's object directory. The logical name is the means by which all users can access the device. Devices must have their characteristics listed in the Basic I/O System's Device Unit Information Block (DUIB) at configuration time before they can be attached with the ATTACHDEVICE command.

Table 3-2A and Table 3-3B list the physical device names normally used with the Basic I/O System. Your system might support a subset of these devices or it might support devices not listed. If it supports the devices listed, it might support them under different names. Therefore, consult the person who configured your system to determine the correct device names for your system.

One frequent use of the ATTACHDEVICE command is to attach a new device, such as a new disk drive or a line printer, without having to reconfigure portions of the Operating System. (See the DETACHDEVICE command in this chapter for a description of how to detach a device from the system without reconfiguring.)

Unless you have a user ID of WORLD (65535) or specify the WORLD parameter, once you attach a device, only you and the system manager can detach the device. This limitation prevents users from detaching devices belonging to other users and prevents you from accidentally detaching system volumes. However, if you have a user ID of WORLD or specify the WORLD parameter, any device that you attach can be detached by any other user. Refer to the DETACHDEVICE command for more information.

When the device attachment is completed, the ATTACHDEVICE command displays the following message:

```
    <physical name>, attached as <logical name>, id = <user id>
```

where <physical name> and <logical name> are as specified in the ATTACHDEVICE command and <user id> is your user ID (or WORLD, if you specify the WORLD parameter).

Table 3-2. Suggested Physical Device Names

Physical Device Names	Controller	Device Type	Unit Number	Sides	Density	Bytes per Sector
		Flexible Disk Drives: 8 Inch Drives				
F0	204	Shugart SA800	0	1	Single	128
F1	204	Shugart SA800	1	1	Single	128
FX0	204	Shugart SA800	0	1	Single	512
FX1	204	Shugart SA800	1	1	Single	512
AF0	208	Shugart SA800	0	1	Single	128
AF1	208	Shugart SA800	1	1	Single	128
AFD0	208	Shugart SA800	0	1	Double	256
AFD1	208	Shugart SA800	1	1	Double	256
AMF0	208	Shugart SA410	0	1	Double	256
AMF1	208	Shugart SA410	1	1	Double	256
AFDD0	208	Shugart SA850/SA851	0	2	Double	256
AFDD1	208	Shugart SA850/SA851	1	2	Double	256
AFDX0	208	Shugart SA850/SA851	0	2	Double	1024
AFDX1	208	Shugart SA850/SA851	1	2	Double	1024
WF0	218(A)	Shugart SA800	0	1	Single	128
WF1	218(A)	Shugart SA800	1	1	Single	128
WFD0	218(A)	Shugart SA800	0	1	Double	256
WFD1	218(A)	Shugart SA800	1	1	Double	256
WMF0	218(A)	Shugart SA410	0	1	Double	256
WMF1	218(A)	Shugart SA410	1	1	Double	256
WFDD0	218(A)	Shugart SA850/SA851	0	2	Double	256
WFDD1	218(A)	Shugart SA850/SA851	1	2	Double	256
WFDX0	218(A)	Shugart SA850/SA851	0	2	Double	1024
WFDX1	218(A)	Shugart SA850/SA851	1	2	Double	1024
		Flexible Disk Drives: 5 1/4 Inch Drives				
AMFDX0	208	Shugart 450	0	2	Double	512
AMFDX1	208	Shugart 450	1	2	Double	512
AMFDY0	208	Shugart 460	0	2	Double	512
AMFDY1	208	Shugart 460	1	2	Double	512
PMF0*	218A	Shugart 460	0	2	Double	512
PMFDX0*	218A	Shugart 450	0	2	Double	512
PMFDX1*	218A	Shugart 450	1	2	Double	512
PMFY0*	218A	Shugart 460	0	2	Double	512
PMFY1*	218A	Shugart 460	1	2	Double	512
WMFDX0	218(A)	Shugart 450	0	2	Double	512
WMFDX1	218(A)	Shugart 450	1	2	Double	512
WMFDY0	218(A)	Shugart 460	0	2	Double	512
WMFDY1	218(A)	Shugart 460	1	2	Double	512

* Mounted on processor board. (A) = either 218 or 218A controller.

Table 3-2. Suggested Physical Device Names (continued)

Physical Device Names	Controller	Device Type	Unit Number	Bytes per Sector
Hard Disk Drives				
D0	206		0	512
D1	206		1	512
DS0	206		0	128
DS1	206		1	128
Winchester Disk Drives				
W0	215	generic drive	0	1024
W1	215	generic drive	1	1024
IW0	215	Priam 3450 (8")	0	1024
MW0	215	Memorex 101 (8")	0	1024
PW0	215	Pertec D8000 (8")	0	1024
SW0	215	Shugart SA1002 (8")	0	1024
CM0	215	CMI 5412 (5 1/4")	0	1024
CM1	215	CMI 5412 (5 1/4")	1	1024
CMB0	215	CMI 5419 (5 1/4")	0	1024
CMB1	215	CMI 5419 (5 1/4")	1	1024
Storage Module Disk (SMD) Drives				
SMD0	220		0	1024
SMD1	220		1	1024
Bubble Memory Device				
BX0	251		0	256
B0	254		0	256
Others				
BB		Byte bucket (already attached)		
STREAM		Stream file device (already attached)		
T(n)*		terminal		
LP		line printer		

* Physical device names for terminals begin with the letter °T° and are followed by a single digit.

Table 3-3. Controllers Connected to the iSBC® 186/03 SASI/SCSI Interface

Device Name	Manufacturer And Model	Unit Number	Bytes Per Sector
SA0 *	generic controller	0	512
SC0 *	generic controller	0	512
ATS0 **	Adaptec ACB-4000	0	512
XES0 **	Xebec S1410	0	512
FJS0	Fujitsu M2312K	0	512
SHS0 **	Shugart SA1610-2	0	512

* SA0 is the SASI generic device name. SC0 is the SCSI generic device name.
** These controllers support the ST506 Winchester Interface.

ERROR MESSAGES

- <device name>, cannot attach device

  There is a hardware problem or for SCSI an incorrect configuration.

- <device name>, cannot be ATTACHED as <type> device

  The device specified by <device name> cannot support the type of files specified by <type> (NAMED or PHYSICAL). ATTACHDEVICE does not attach the device. For example, the NAMED option is not valid for a device such as a line printer.

- <device name>, device already attached

  The specified device has already been attached. ATTACHDEVICE does not attach the device.

- <device name>, device does not exist

  The physical device name you specified does not correspond to a name the Basic I/O System recognizes. That is, the person who configured your application system did not specify <device name> as the name of a device-unit during configuration of the Basic I/O System. ATTACHDEVICE does not attach the device.

- <logical name>, logical name already exists

  The specified logical name is already cataloged in the root job's object directory. ATTACHDEVICE does not attach the device.

- 0085 : E$LIST, too many device names

  You tried to attach more than one physical device with a single ATTACHDEVICE command. ATTACHDEVICE does not attach more than one device.

- <logical name>, volume is not a NAMED volume

  ATTACHDEVICE attempted to attach a device as a named device and discovered a physical volume on the device. However, ATTACHDEVICE does attach the device. You can use the device after formatting the volume as a named volume or after inserting a named volume in the device.

- <logical name>, volume not formatted
  <logical name>, <exception value> : <exception mnemonic>

  ATTACHDEVICE attempted to attach a device as a named device and encountered an I/O error while searching for the volume's root directory. This usually indicates that the volume is not formatted. However, ATTACHDEVICE does attach the device.

- <logical name>, volume not mounted

  The specified device does not contain a volume. However, ATTACHDEVICE does attach the device.

- <exception value> : <exception mnemonic>, while collecting device name

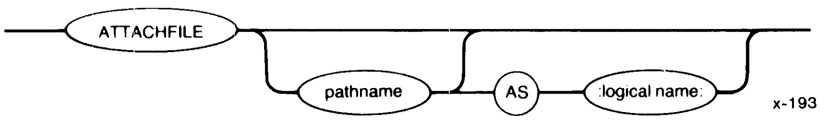
  ATTACHDEVICE encountered an exceptional condition while parsing the device name from the command line. This message lists the resulting exception code. ATTACHDEVICE does not attach the device.

- <exception value> : <exception mnemonic>, while collecting logical name

  ATTACHDEVICE encountered an exceptional condition while parsing the logical name from the command line. This message lists the resulting exception code.

ATTACHFILE

This command allows you to associate a logical name with an existing file. The command catalogs the logical name in your global object directory. The format of this command is as follows:



INPUT PARAMETERS

pathname	Pathname of the file to which the Human Interface associates a logical name.
:logical name:	1- to 12-character name that represents the logical name to be associated with the file. Colons surrounding the logical name are optional; however, if you use colons, you must use matching colons. If you omit this parameter, the default logical name is :$:.

If you enter the ATTACHFILE command without parameters, the default is:

```
ATTACHFILE :HOME: AS :$:
```

DESCRIPTION

The ATTACHFILE command allows you to associate a logical name with an existing file. After making this association, you can use the logical name, instead of the entire pathname, to refer to the file.

When the attachment is complete, ATTACHFILE displays the following message:

```
<pathname>, attached AS <logical name>
```

where <pathname> and <logical name> are as specified in the ATTACHFILE command.

ATTACHFILE makes the association between a file and a logical name by cataloging a connection to the file in your global object directory (this is normally the object directory of your interactive job). It catalogs the connection under the name specified as the logical name. If there is another connection cataloged in the object directory under the same logical name, ATTACHFILE uncatalogs and deletes the previous connection before cataloging the new one. If an object other than a connection is cataloged in the directory under the specified logical name, ATTACHFILE leaves the previous object as is, does not catalog the new connection, and displays an error message to describe the situation.

Because ATTACHFILE catalogs the connection in your global object directory, the logical name has effect only within your interactive job. Therefore, several users can specify the same logical name without affecting the others.

If you specify a pathname for a file but omit the logical name, ATTACHFILE attaches the file as :$:. This allows you to change your default prefix. Changing your default prefix can be useful when you want to manipulate files that reside in a directory other than the one specified by your original default prefix. For example, suppose you have a file that you normally refer to as:

```
:PROG:SOURCE/PLM/INTERRUPT/TEST.P86
```

You can change your default prefix with the command:

```
ATTACHFILE SOURCE/PLM/INTERRUPT
```

Then, you can refer to the file as simply:

```
TEST.P86
```

When you finish using the files in directory :PROG:SOURCE/PLM/INTERRUPT, you can return your default prefix to its original setting by entering:

```
ATTACHFILE
```

This is the same as entering:

```
ATTACHFILE :HOME: AS :$:
```

:HOME: is a logical name that refers to the same directory as your original default prefix. Therefore, you can change your default prefix as much as you like with ATTACHFILE and return to the original setting by making reference to :HOME:. However, you cannot use ATTACHFILE to change the meaning of :HOME:. (Also, you cannot use ATTACHFILE to change the meaning of :CI: and :CO:.)

The logical name created with ATTACHFILE remains valid until one of the following situations occur:

- A DETACHFILE command (described later in this chapter) dissolves the association between file and logical name.
- The interactive session that specified the ATTACHFILE command terminates processing. This event occurs when a user, in response to the Human Interface prompt, enters a Control-Z character to reinitialize the interactive job. In this case, the Operating System deletes the interactive job and then recreates it.

- A task deletes the connection to the file via a Basic I/O System or Extended I/O System call (refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL or the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information about connections). In this instance, the logical name remains cataloged in the global directory, but the connection to which it refers does not exist.

- A user forcibly detaches the volume containing the file via the DETACHDEVICE command (described later in this chapter).

- A user removes the volume from the drive.

ERROR MESSAGES

- <logical name>, list of logical names not allowed

  You entered more than one logical name as input to ATTACHFILE.

- <pathname>, list of pathnames not allowed

  You entered more than one pathname as input to ATTACHFILE.

- <logical name>, logical name not allowed

  You attempted to attach a file using a logical name :HOME:, :CI:, or :CO:. You cannot change the meaning of these logical names.

- <logical name>, not a file connection

  The logical name you specified, <logical name>, is already cataloged in object directory of the session and does not represent a connection object.

- <pathname>, not allowed as default prefix

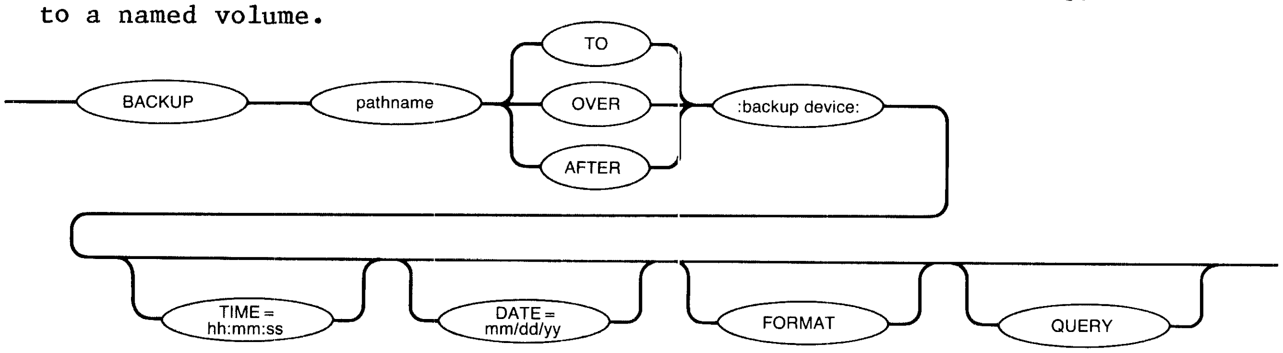
  You attempted to attach a physical or stream file as your default prefix (:$:). Only named files are valid.

- <logical name>, too many logical names

  Your global object directory is full. Therefore ATTACHFILE is unable to catalog the file's name in the object directory.

BACKUP

This command saves files on a named volume by copying them to a physical volume which serves as a backup storage device. This command provides a way of saving a large volume (a winchester disk, for example) onto a number of smaller volumes such as diskettes or onto another mass storage device such as a tape drive. Later, you can use the RESTORE command (described later in this chapter) to retrieve these files and copy them to a named volume.



x-667

INPUT PARAMETERS

pathname	Pathname of a file on the source volume. BACKUP saves all the files starting from this point on the file tree. If you specify the logical name of the device only, BACKUP saves all files in the volume, beginning with the root directory. If you specify a file and not a directory, then only the specified file is saved.
DATE	BACKUP saves all files created or modified on or after the date and time specified with the DATE/TIME parameters. If the DATE parameter is omitted, the date defaults to the current system date. If both date and time parameters are omitted (DATE/TIME), then the date and time default to 1/1/78 and 00:00:00.
mm/dd/yy	Form used to specify the DATE.

mm	Numerical designation for the month. (For example: 1 represents January, 2 represents February, etc.). Must be a digit.
dd	Numerical designation for the day of the month. Value must be in digits.
year	Designation for the year. You enter this as a two digit number, as follows:

entered year	actual year
0 through 77	2000 through 2077
78 through 99	1978 through 1999
100 through 1977	error
1978 through 2099	1978 through 2099
2100 and up	error

TIME	TIME is used in conjunction with the DATE parameter to determine which files to save. If TIME is omitted, the default is 00:00:00. BACKUP saves only those files modified since the specified date or time.
hh:mm:ss	Format for TIME parameter: hh Hours specified as 0-24 mm Minutes specified as 0-59 ss Seconds specified as 0-59
<name>=name	Name that is given to the backup volume. If you have a set of physical volumes, this name applies to the set as a whole.
FORMAT	Causes BACKUP to format each volume before writing to it. Interleave is set to one. FORMAT should be inserted whenever a new volume is used.
QUERY	Causes the Human Interface to prompt for permission to save each file. The Human Interface prompts with one of the following queries:

```
pathname, BACKUP Data File?
                or
pathname, BACKUP Directory?
```

Enter one of the following responses to the query:

ENTRY	Action
Y or y	Save the file.
E or e	Exit from BACKUP.
R or r	Continue saving files without further query.
N or n	If data file, do not save the file; if directory do not save the directory or any file in in that part of the directory tree. Query for the next file, if any.

	Other	Error message and reprompt.

OUTPUT PARAMETERS

TO	Causes BACKUP to send the processed output to new backup volume. This preposition also causes BACKUP to read the volume label from each newly mounted physical volume in an attempt to determine the volume type. This is an attempt to ensure that the volume is compatible with any previously mounted volumes in the backup set.
OVER	Causes BACKUP to begin writing on each fresh volume without checking the label for compatibility. BACKUP writes over any previous files or directories on the backup volume.
AFTER	Causes BACKUP to search the mounted volume looking for the end of a previous backup operation. BACKUP then appends the file or directory after the previous backup operation. If more volumes are needed to complete the backup operation, then BACKUP behaves as if the TO preposition had been specified for subsequent volumes. If FORMAT was specified, BACKUP formats any new volumes required to finish the backup operation.
:backup device:	The logical name of the device to which BACKUP copies the files.

DESCRIPTION

BACKUP is a utility which saves named files on backup volumes such as tapes or diskettes. For BACKUP to save files from a named volume, you must have read access to the files and to the directories that contain them.

BACKUP saves the following information for each file:

- File name
- Access list, including owner
- Extension data
- File granularity
- Contents of the file

When you enter BACKUP, the command displays the following sign-on message:

```
        iRMX 86 BACKUP, Vx.y
        Copyright <year> Intel Corporation
```

where Vx.y is the version number of the utility.

Once the command line has been scanned the following message is displayed to indicate what TIME and DATE has been used to save files:

```
     All Files Modified After <date> , <time>  Will Be Saved
```

where <date> and <time> are the values you specified in the date and time parameters (or defaults). BACKUP then prompts you to mount the backup volume.

When you use the BACKUP command, you do not have to format a volume previous to issuing the command. BACKUP has a FORMAT parameter which you can use to format any volume while a backup operation is occurring.

Whenever BACKUP requires a new backup volume, the command displays the following message:

```
     <device>, Mount Backup Volume [(name) #<nn>], Enter Y to Continue:
```

where <device> is the logical name of the backup device, (name) is the name of the physical volume set, and nn is the identifying number of the requested volume. In response to this message, you place a volume in the backup device and enter one of the following responses:

Entry	Action
Y, y, R or r	Continue the backup process.
E or e	Exit from the BACKUP command.
Any other character	Invalid entry; reprompt for entry.

BACKUP continues prompting for a backup volume until you supply one that it can access.

If BACKUP detects that a volume cannot be read, that a volume is named volume, or that the volume is a physical volume containing data, the command informs you with one the following messages:

```
    <device>, Recognized Volume
    <device>, Volume Not Correctly Formatted
    <device>. Backup Volume <name> #<nn>, <date>, <time> Mounted
    <device>, Named Volume, <name> Mounted
```

where <device> is the logical name of the backup device, <name> is the volume name as recorded in the label, <nn> is the volume number of the backup volume, and <date> and <time> are the date and times when the last backup operation was performed. If the situation is appropriate, then the command may prompt you by a request to FORMAT or to OVERWRITE the mounted volume in the following way:

```
<device>, Enter Y to Overwrite/Format:
```

In response to this prompt, you enter one of the following:

Entry	Action
Y or y	Use the volume as a backup volume.
R or r	Use the volume and do not query for permission again. This is equivalent to specifying 'OVER' on the command line for the rest of the BACKUP operation.
E or e	Exit from the BACKUP command.
N or n	Reprompt for another volume.
Other	Invalid Response--reprompt for entry.

When BACKUP has finished has finished a backup routine, the command prints the following message:

```
Physical Volume (name), #nn, Complete
```

After the backup operation is complete, the number of data files and directories which were saved are displayed for you in the following format:

```
nn Data File[s] Saved
nn Director[y] [ies] Saved

BACKUP [Not] Complete
```

ERROR MESSAGES

If the error message requires a response, enter one of the following:

Entry	Action
Y, y, R or r	Continue the backup process.
E or e	Exit from the BACKUP command.
Any other character	Invalid entry; reprompt for entry.

- <backup device>, backup operation not completed

  When BACKUP requested a new backup volume, you specified an "E" to exit BACKUP. This message is a reminder that the backup operation is not complete. The last file on the last backup volume may be incomplete.

- <backup device>, backup volume #<nn>, <date>, <time>, mounted
  <backup device>, enter Y to overwrite:

  The backup volume you supplied already contains backup information. BACKUP lists the logical name of the backup device, the volume number, and the date on which the original backup occurred. It overwrites this volume if you enter Y, y, R, or r.

- <backup device>, cannot attach volume
  <backup device>, <exception value> : <exception mnemonic>
  <backup device>, mount backup volume #<nn>, enter Y to continue:

  BACKUP cannot access the backup volume. This could be because there is no volume in the backup device or because of a hardware problem with the device. The second line of the message indicates the iRMX 86 exception code encountered. BACKUP continues to issue this message until you supply a volume that BACKUP can access.

- <pathname>, <exception value> : <exception mnemonic>, cannot back up file

  For some reason BACKUP could not copy a file from the named volume, possibly because you do not have read access to the file or because there is a faulty area on the named volume. The message lists the pathname of the file and the exception code encountered. BACKUP copies as much of the file as possible and continues with the next file.

- <backup device>, device in use
  <backup device>, <exception value> : <exception mnemonic>

  The device you specified for the backup device is the same device that contains your input pathname. Continuing would result in damage to the files on the input volume.

- <backup device>, error writing volume label
  <backup device>, <exception value> : <exception mnemonic>

- <backup device>, input and output are on same device

  The device you specified for the backup device is the same device that contains your input pathname. Continuing would result in damage to the files on the input volume.

- <backup device>, invalid backup device

  The logical name you specified for the backup device was not a logical name for a device. Examples of invalid names are :CI:, :CO:, and :HOME:.

- <exception value> : <exception mnemonic>, invalid DATE or TIME

  For either the DATE or TIME parameter, you entered a value that is out of range (such as 31/02/81 or 26:03:62). The message lists the exception code encountered as a result of this entry.

- invalid output specification

  You did not supply the logical name of the backup device when you entered the BACKUP command.

- <backup device>, mount backup volume #<nn>, enter Y to continue:

  When BACKUP attempted to write a label on the backup volume, it encountered an error condition, possibly because of a faulty area on the volume, or because the volume is write-protected. The second line of the message indicates the iRMX 86 exception code encountered. BACKUP reprompts for a different backup volume.

- <backup device>, named volume, <volume name>, enter Y to overwrite:

  The backup volume you supplied is a named volume. BACKUP lists the logical name of the device containing the volume and the volume name. It overwrites this volume if you enter Y, y, R, or r.

- <backup device> not correctly formatted, enter Y to format:

  The backup volume was not correctly formatted.

- <exception value> : <exception mnemonic>, requested date/time later than system date/time

  Either the date and time you specified in the BACKUP command are in error or you did not set the system date and time.

- <pathname>, too many input pathnames

  You attempted to enter a list of pathnames or use a wild-carded pathname as the input pathname. You can enter only one pathname per invocation of BACKUP.

- <pathname>, too many output pathnames

  You attempted to enter a list of logical names for the backup device. You can enter only one output logical name per invocation of BACKUP.

- <pathname>, unable to complete directory

  BACKUP encountered an error when accessing a file in the <pathname> directory. It skips the rest of the files in the directory and goes on to the next directory. This error could occur if you do not have list access to the directory.

- <backup device>, unrecognized volume, enter Y to overwrite:

  The backup volume you supplied is a formatted volume, but it has a label that is not readable. BACKUP will overwrite this volume if you enter Y, y, R, or r.

- <backup device>, volume not formatted

  <backup device>, mount backup volume #<nn>, enter Y to continue:

  The backup volume you supplied was not formatted. BACKUP continues to issue this message until you supply a formatted backup volume.

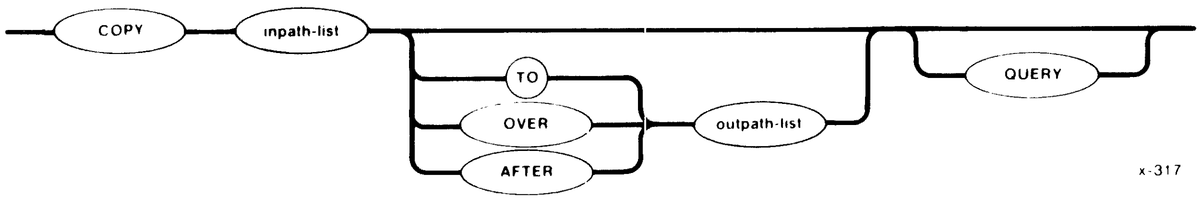
- <backup device>, write error on backup volume
  <backup device>, <exception value> : <exception mnemonic>

  BACKUP encountered an error condition when writing information to the backup volume. The second line of the message lists the exception code encountered. This error is probably the result of a faulty area on the volume.

COPY

This command reads data from the specified input source or sources and writes the output to the specified destination file or files.

The format of the command is as follows:



INPUT PARAMETERS

inpath-list	One or more pathnames for the files to be copied. Multiple pathnames must be separated by commas. Separating blanks are optional. To copy files on a one-for-one basis, you must specify the same number of files in the inpath-list as in the outpath-list.
QUERY	Causes the Human Interface to prompt for permission to copy each file. Depending on the specified preposition (TO, OVER, or AFTER), the Human Interface prompts with one of the following queries:

```
<pathname>, copy TO <out-pathname>?

<pathname>, copy OVER <out-pathname>?

<pathname>, copy AFTER <out-pathname>?
```

Enter one of the following (followed by a carriage return) in response to the query:

Entry	Action
Y or y	Copy the file.
E or e	Exit from COPY command.
R or r	Continue copying files without further query.
Any other character	Do not copy this file; go to the next file in the input list.

OUTPUT PARAMETERS

TO	Writes the listed input files to named new output files. The specified output file or files should not already exist. If they do, COPY displays the following message: `<pathname>, already exists, OVERWRITE?` Enter Y, y, R, or r if you wish to write over the existing file. Enter an "N" (upper or lower case) or a carriage return alone if you do not wish to overwrite the existing file. In the latter case, the COPY command will pass over the corresponding input file without copying it, and will attempt to copy the next input file to its corresponding output file. If you specify multiple input files and a single output file, COPY appends the remaining input files to the end of the output file.
OVER	Writes the input files over (replaces) the existing output files on a one-for-one basis, regardless of file size. If an output file does not already exist, its corresponding input file is written to a new file with the corresponding output file name. If you specify multiple input files and a single output file, COPY appends the remaining input files to the end of the output file.
AFTER	Appends the input file or files to the current data in the existing output file or files. If the output file does not already exist, all listed input files will be concatenated into a new file with the listed output file name.
outpath-list	One or more pathnames for the output files. Multiple pathnames must be separated by commas. Separating blanks are optional. If you omit the preposition and outpath-list parameters, COPY displays the output at your console screen (TO :CO:).

DESCRIPTION

The COPY command can be used to perform several different operations. Some of these include:

- Creating new files (TO preposition).
- Copying over existing files or creating new files (OVER preposition).
- Adding data to the end of existing files (AFTER preposition).
- Copying a list of files to another list of files on a one-for-one basis.
- Concatenating two or more files into a single output file.

As each file is copied, the COPY command displays one of the following messages:

```
<pathname>, copied TO <out-pathname>

<pathname>, copied OVER <out-pathname>

<pathname>, copied AFTER <out-pathname>
```

When you copy files, the number of input pathnames you specify must equal the number of output pathnames, unless you specify only one output pathname. In the latter case, COPY appends the remainder of the input files to the end of the ouput file. As each file is appended, the following message is displayed on the console screen:

```
<pathname>, copied AFTER <output-file>
```

If you specify multiple output files, and there are more input files than output files, or if you specify fewer input files than output files, COPY returns an error message.

Also, if you specify a wild-card character in an output pathname, you must specify the same wild-card character in the corresponding input pathname. Other combinations result in error conditions.

You cannot successfully use COPY to copy a directory to a data file or to another directory. Although a directory can be copied, the attributes of the directory are lost. That is, the directory can no longer be used as a directory. However, a file listed under one directory can be copied to another directory. For example:

```
COPY SAMP/TEST/A TO :F1:/ALPHA/BETA
```

This would copy the A data file to a different volume, directory, and filename, where the new file's pathname would be :F1:/ALPHA/BETA.

The user ID of the user who invokes the COPY command is considered the owner of new files created by COPY. Only the owner can change the access rights associated with the file (refer to the PERMIT command later in this chapter).

When COPY creates new files, it sets the access rights and list of accessors as follows:

- It sets the file for ALL access (delete, read, append, and change).
- It sets the owner as the only accessor to the file.

Refer to the PERMIT command for more information about access rights and the list of accessors.

ERROR MESSAGES

- <pathname>, output file same as input file

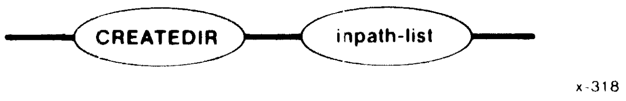
  You attempted to copy a file to itself.

- <pathname>, UPDATE or ADD access required

  Either you cannot overwrite the information in a file because you do not have update access to it, or you cannot copy information to a new file because you do not have add entry access to the file's parent directory.

CREATEDIR

This command creates one or more iRMX 86 user directories. The format is as follows:



INPUT PARAMETER

inpath-list	One or more pathnames of the iRMX 86 directories to be created. Multiple pathnames must be separated by commas. Embedded blanks between commas and pathnames are optional.

DESCRIPTION

CREATEDIR creates a directory with all access rights available to you, the owner. That is, you can delete, list, add, and change the contents of the directory you created with CREATEDIR. Other users (except the system manager) have no access to the directory unless you use the PERMIT command (described later in this chapter) to change the access rights and list of accessors.

The following message is displayed if a directory is successfully created:

```
    <directory-name>, directory created
```

You can create new directories that are subordinate to other directories. For example:

```
    CREATEDIR AB/DC/EF/GH
```

causes the newly-created directory GH to be nested within existing directory EF, which in turn, is nested within directory DC, and so on. The directories AB, DC, and EF must already exist before entering this command.

You can check the contents of the directory at any time by using the DIR command to list the directory (see the DIR command in this chapter).

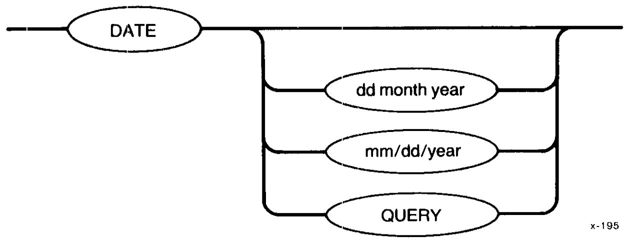
ERROR MESSAGE

- <directory-name>, file already exists

  The pathname of the directory to be created already exists.

DATE

This command sets a new system date or displays the current date. The format is as follows:



INPUT PARAMETERS

dd	Two-digit number that specifies the day of the month. Both digits are not required to set this parameter.
month	Designation for the month. You can enter the whole name (such as AUGUST) or enough characters to distinguish one month from another (for example, AU, to distinguish AUGUST from APRIL). You can use this form for specifying the month only when using the "dd month year" format.
mm	Numerical designation for the month (for example: 1 represents January, 2 represents February, etc.). You can use this form for specifying the month only when using the "mm/dd/year" format. Both digits are not required to set this parameter.
year	Designation for the year. You can enter this as a two- or four-digit number, as follows:

entered year	actual year
0 through 77	2000 through 2077
78 through 99	1978 through 1999
100 through 1977	error
1978 through 2099	1978 through 2099
2100 and up	error

QUERY — Causes DATE to prompt for the date by issuing the following message:

```
DATE:
```

DATE continues to issue this prompt until you enter a valid date.

DESCRIPTION

If you set one date parameter, you must set all three; there are no default settings for individual date parameters. You must separate the dd, month, and year entries with single blanks.

If you omit the date parameters, DATE displays the current date and time in the following form:

```
dd mm yy, hh:mm:ss
```

When the Operating System displays the date, it displays only the first three characters of the month and the last two digits of the year. It separates the hours, minutes, and seconds of the time with colons.

When you start up or reset the Operating System, the date is automatically set to the last time you accessed the :SYSTEM: directory. You may then reset the DATE setting to any acceptable value.

ERROR MESSAGES

- <date>, invalid date

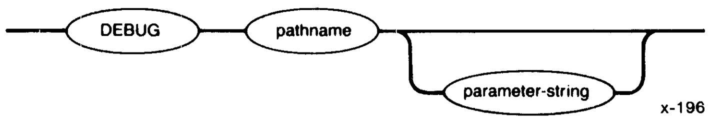
  You entered an invalid date. This error could result from specifying a day that is invalid for the month you specified (such as 31 FEB 82), entering characters for the year parameter that do not fall into the legitimate ranges listed under the year parameter, entering a month parameter that does not uniquely identify the month, or entering invalid characters.

- <parameter>, invalid syntax

  You specified both a date and the QUERY parameter in the DATE command.

DEBUG

This command allows you to debug your iRMX 86 application jobs if your system is configured with the iSDM 86, iSDM 286, or iSBC 957B monitors.



INPUT PARAMETERS

pathname	Pathname of the file containing the application program to be debugged.
parameter-string	String of required, optional, and default parameters that can be used in the command line to load and execute the application program.

DESCRIPTION

DEBUG loads your specified application program into main memory and transfers control to the system monitor. You can then use the monitor to single-step, display registers, and set breakpoints within the program. Refer to the iSDM 86 SYSTEM DEBUG MONITOR REFERENCE MANUAL and the iSDM 286 SYSTEM DEBUG MONITOR REFERENCE MANUAL for more information.

When you invoke the DEBUG command, it displays the following message:

```
DEBUG file, <pathname>
```

where <pathname> is the pathname of the file containing the application job to debug. Then DEBUG loads the application job and displays information about the location of the job's segments and groups. Figure 3-1 shows an example of this output.

As Figure 3-1 shows, the first line of the display lists the token for the job that was created. The remaining lines list the base portions of all segments and groups assigned by LINK86 when the code was linked. The S(n) and G(n) values are the same as those that appear on the link map. Therefore, you can match the base values shown in this display with the offset values shown in the link map to determine the exact location of a symbol listed in the link map. Refer to the iAPX 86, 88 FAMILY UTILITIES USER'S GUIDE for information about LINK86 and the link map.

```
    SEGMENT AND GROUP MAP FOR JOB:  A88F

NAME   BASE     NAME   BASE     NAME   BASE     NAME   BASE     NAME   BASE

S(1)   9E4E     S(2)   9E32     S(3)   9CFF     S(5)   9CEC     S(6)   A863
S(7)   A229     S(8)   A84D     S(9)   A152     S(13)  9C91     S(15)  9C85
S(17)  9C67     S(18)  9C5C

G(1)   A229     G(2)   A152
```

Figure 3-1. Sample DEBUG Display

When DEBUG executes, the monitor in your system disables interrupts. This causes the time-keeping function to stop when code is not executing. This slowing of the timing function:

- Affects the ability of the Nucleus to execute time-out tasks that have provided time limits to system calls, such as RECEIVE$UNITS and RECEIVE$MESSAGE.

- Affects the ability of the Basic I/O System to keep track of the time-of-day and write its data structures to secondary storage.

Unless you use the monitor's NQ command to single-step through code, the system monitor cannot tolerate interrupts while single-stepping. The NQ command disables interrupts while single-stepping, allowing you to single-step through code without being interrupted by the system clock.

When DEBUG is invoked to debug an application program, it loads the application program into its own dynamic memory. As a result of this process, the application program obtains dynamic memory from the memory pool of DEBUG, not from the memory pool of the user session. Because DEBUG uses a different set of default values than the CLI, it is possible that the program may behave differently than when it is run independently.

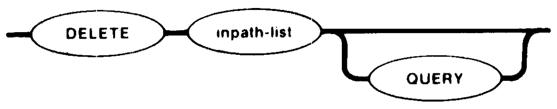
ERROR MESSAGE

- <exception value> : <exception mnemonic> command aborted by EH

  While processing, the DEBUG command encountered an exceptional condition. Therefore, the Human Interface's exception handler aborted the command. The message lists the exception code that occurred.

DELETE

This command removes data files and empty directories from secondary storage. The format is as follows:



x-319

INPUT PARAMETERS

inpath-list	One or more pathnames for the named data files or empty directories to be deleted. Multiple pathname entries must be separated by commas. Separating blanks are optional.
QUERY	Causes the DELETE command to ask for your permission to delete each file in the list. Prior to deleting a file, the DELETE command displays the following query:

<pathname>, DELETE?

Enter one of the following (followed by a carriage return) in response to the query:

Entry	Action
Y or y	Delete the file.
E or e	Exit from DELETE command.
R or r	Continue deleting without further query.
Any other character	Do not delete file; query for next file in sequence.

DESCRIPTION

The DELETE command allows you to release unused secondary storage space for new uses by removing empty directories and unneeded data files. To delete a file, you need not be the owner of the file; however you must have delete access to the file. If a user or program is accessing the file (has a connection to the file) when you enter the DELETE command, DELETE marks the file for deletion and deletes it when all connections to the file are gone.

Non-empty directories cannot be deleted. If you wish to delete a directory that contains files, you must first delete all its contents. For example, if you wish to delete a directory named ALPHA whose entire contents consist of a directory BETA containing a data file SAMP, you would enter the following command:

```
    DELETE ALPHA/BETA/SAMP, ALPHA/BETA, ALPHA
```

This command sequence would delete all the files contained under ALPHA before deleting the directory itself.

DELETE displays the following message as it deletes each file or marks the file for deletion:

```
    <pathname>, DELETED
```

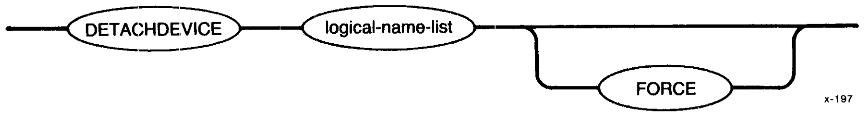
ERROR MESSAGE

- `<pathname>, DELETE access required`

  You do not have permission to delete the specified file.

DETACHDEVICE

This command detaches the specified devices and deletes their logical names from the root job's object directory. The format of this command is as follows:



INPUT PARAMETER

logical-name-list	One or more logical names of the physical devices that are to be detached. Colons surrounding each logical name are optional; however, if you use colons, you must use matching colons. Multiple logical names must be separated by commas.
FORCE	Causes DETACHDEVICE to detach the device even if connections to files on the device currently exist.

DESCRIPTION

The DETACHDEVICE command allows you to detach a device without having to reconfigure the system. After a device is detached, no volume mounted on that device is accessible for system use.

Unless you are the system manager (user ID 0), you can detach only the following devices:

- Devices that are configured with your user ID as the owner ID
- Devices you originally attached using the ATTACHDEVICE command
- Devices originally attached using the WORLD parameter of ATTACHDEVICE
- Devices originally attached by user WORLD (user ID 65535)

DETACHDEVICE returns an error message if you attempt to detach devices originally attached by other users. This error prevents users from detaching devices belonging to other users and from accidentally detaching system volumes. However, the system manager can detach all devices.

Unless you specify the FORCE parameter, you cannot detach a device if any connections exist to files on the device (that is, if other users are currently accessing the device). However, the FORCE parameter causes DETACHDEVICE to delete all connections to files on the device before detaching the device.

After detaching the device and deleting its logical name from the root job's object directory, the DETACHDEVICE command displays the following message:

```
    <logical-name>, detached
```

NOTE

Using the DETACHDEVICE command to detach the device containing your Human Interface commands causes loss of access to Human Interface functions until the system is restarted.

ERROR MESSAGES

- <logical name>, can't detach device
  <logical name>, <exception value> : <exception mnemonic>

  An exceptional condition occurred which prevented DETACHDEVICE from detaching the device. This message lists the resulting exception code.

- <logical name>, device does not belong to you

  The device was originally attached by a user other than WORLD or you. Thus you cannot detach the device.

- <logical name>, device has outstanding file connections

  There are existing connections to files on the device. Because you did not specify the FORCE parameter, DETACHDEVICE does not detach the device.

- <logical name>, device is in use

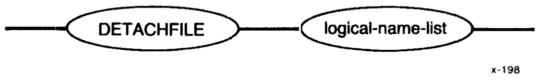
  Another user or program is accessing the device (has a connection to a file). Therefore, you must specify the FORCE parameter in order to detach the device.

- <logical name>, outstanding connections to device have been deleted

  There were outstanding connections to files on the volume. However, because you specified the FORCE parameter, DETACHDEVICE deleted those connections. This is a warning message that does not prevent DETACHDEVICE from detaching the device.

DETACHFILE

This command allows you to terminate the association of a logical name with a file. The format of this command is as follows:



PARAMETERS

logical-name-list	List of logical names, separated by commas, that represent the files to be detached. Each logical name must be contain 1 to 12 characters. Colons surrounding each logical name are optional; however, if you use colons, you must use matching colons.

DESCRIPTION

You establish an association between a file and a logical name by entering the ATTACHFILE command. DETACHFILE breaks this association. It does this by uncataloging the logical name from your interactive job's global object directory. When DETACHFILE detaches a file in this manner, it displays the following message:

```
    <logical name>, detached
```

where <logical name> is the name you specified.

You cannot use DETACHFILE to detach logical names that do not represent files. DETACHFILE returns an error message if you make such an attempt. In particular, you cannot use DETACHFILE to detach devices.

You cannot use DETACHFILE to detach logical names originally created by other users. DETACHFILE searches for logical names in the global object directory of your interactive job only. It does not search the root job's object directory nor the object directories of any other interactive jobs.

ERROR MESSAGES

- <exception value> : <exception mnemonic> invalid global job

  The Human Interface encountered an internal system problem when it attempted to remove the logical name from the global job's object directory. The message lists the resulting exception code.

- <logical name>, logical name does not exist

  The logical name is not cataloged in the global object directory of your interactive job.

- <logical name>, logical name not allowed

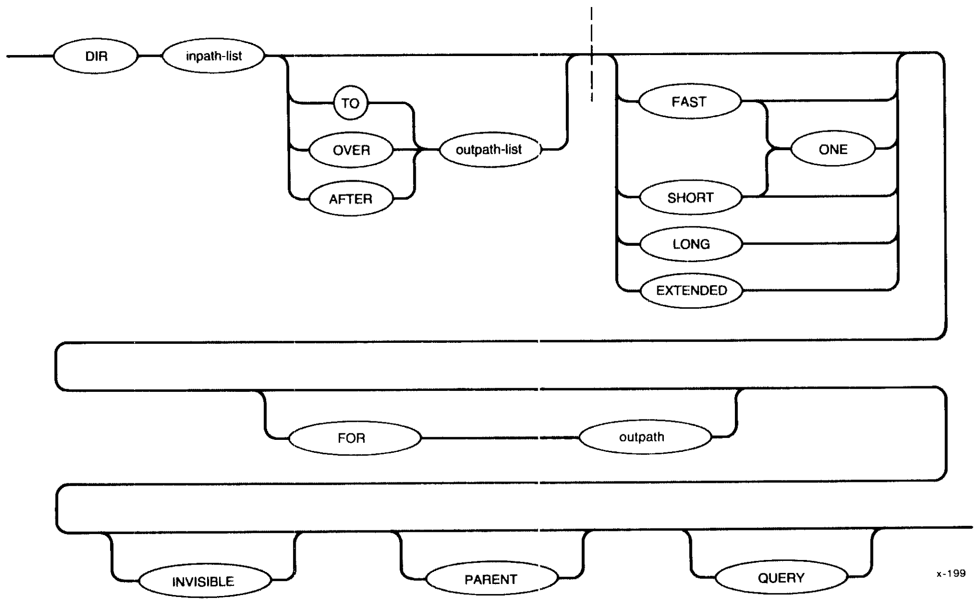
  The logical name you specified was either :$:, :HOME:, :CI:, or :CO:. You cannot detach the files associated with these logical names.

- <logical name>, not a file connection

  The logical name you specified is cataloged in the global object directory of your interactive job,but it is not the logical name of a file.

DIR

This command lists the names and attributes of the data and directory files contained in a given directory. The format of the command is as follows:



INPUT PARAMETERS

inpath-list	One or more pathnames of the directories to be listed (the pathnames can represent data files if the PARENT parameter is also specified). Multiple directory pathname entries must be separated by commas. Separating blanks are optional. If no pathname is specified, the user's default directory is listed.
FAST	Lists only the filenames and directory names in the directory. The output format contains five columns of filenames unless you also specify the ONE parameter (see Figure 3-2 at the end of this command description). FAST is the default if you omit the listing format.
SHORT	Lists the file information in a two-column format (see Figure 3-3 at the end of this command description).
ONE	Lists the output of a FAST or SHORT listing in single-column format. ONE is the default number of columns for EXTENDED or LONG listings.

LONG	Lists file information in a one-line format (see Figure 3-4 at the end of this command description).
EXTENDED	Lists all available information for each data file or directory file in the directory. The first line for each file is the same as for the LONG form. The second line contains the last access date, creation date, and the accessor list. The listing is in a double-column format (see Figure 3-5 at the end of this command description).
FREE	Lists the amount of free space available on the volume containing the given directory. The listing shows the number of free files, free volume blocks, and free bytes.
INVISIBLE	Lists the invisible files (those beginning with the characters "R?" or "r?") in addition to the rest of the files in the directory. If you omit this parameter, DIR does not display invisible files.
PARENT	Causes DIR to display an entry for the directory specified in the inpath-list in addition to the files contained in the directory. This parameter is useful for obtaining information about the root directory of a volume when using the LONG or EXTENDED parameters.
QUERY	Causes the DIR command to prompt you for permission to list a directory by issuing the following message:

```
<pathname>, DIR?
```

Enter one of the following (followed by a carriage return) in response to the query:

Entry	Action
Y or y	List the directory.
E or e	Exit from DIR command.
R or r	Continue listing directories without further query.
Any other character	Do not list directory; query for the next directory, if any.

FOR	Selects only those files within a specific directory in the path-list. FOR can be used with wildcard file designators.

OUTPUT PARAMETERS

TO	Copies the directory listing to the specified destination data file. If the destination file already exists, DIR displays the following information: `<pathname>, already exists, OVERWRITE?` Enter Y, y, R, or r if you wish to delete the existing file. Enter any other character if you do not wish to delete the file. If you omit the TO/OVER/AFTER preposition and the output pathname, TO :CO: is the default.
OVER	Copies the directory listing to the specified output file and writes over (replaces) the previous contents.
AFTER	Appends the directory listing to the current contents of the specified output file.
outpath-list	One or more pathnames of the files to receive the directory listing. Multiple pathname entries must be separated by commas. Separating blanks are optional. If you omit the preposition and the outpath-list, the default destination is the user's console screen (TO :CO:).

DESCRIPTION

You do not need to be the owner of a directory to list its contents with DIR; however, you must have list access to the directory.

The amount of information listed for each file depends upon what listing format you specify (FAST, SHORT, LONG, or EXTENDED). The end of the SHORT, LONG, and EXTENDED DIR listings show the amount of space used (first line) by the files and the amount of free space left over (second line).

An example of each type of listing format is provided at the end of the DIR command description in Figures 3-2 through 3-5 respectively. Table 3-3, which follows the figures, provides an explanation of the illustrated headings.

If you want to list the default user directory but also wish to specify a listing format other than FAST, use the default directory name explicitly. For example:

```
    DIR :$: EXTENDED
```

displays a listing of the default directory in the EXTENDED format. Note that the identity of your default directory is a configuration option.

Figures 3-2, 3-3, 3-4, and 3-5 show output examples for FAST, SHORT, LONG, and EXTENDED listing formats respectively. Table 3-3 defines the displayed column headings.

If a file name begins with the characters "R?" or "r?", it is an invisible file. Normally DIR does not display invisible files. However, you can specify the INVISIBLE parameter to display these files.

```
-DIR alpha

03 MAR 82  04:25:40
  DIRECTORY OF alpha  ON mvol
fname1  fname2  fname3  fname4  fname5
fname6  fname7  fname8  fname9  fname10
fname11 . . .
.
.
.
```

Figure 3-2. FAST Directory Listing Example (Default Listing Format)

```
-DIR mydirectory2 S

03 MAR 82   21:55:24
DIRECTORY OF mydirectory2 ON myvol
  NAME          AT   ACC   BLKS  LENGTH     NAME            AT   ACC   BLKS  LENGTH
append              -R--     02  1425       alpha.obj            DRAU     3    2871
REFERENCE       DR  -L--      1    10       DATA            DR   DLAC     1       4
LEMONADEIT          DRAU   123456789 123456789
time                DRAU      6  5374       detachdevice         DRAU     4    3414
test                -R--      5  4415       schedule             ---U     7    6976
testprog.a86        -RA-      2  2040       DATABASE.LST         -RAU    11   10336
EXPERIMENTAL    DR  -LAC      1    20       BACKUP          DR   DLAC     1      10

     13 FILES     44 BLOCKS         36895 BYTES
     33 FILES  3,000 BLOCKS     3,072,000 BYTES FREE
```

Figure 3-3. SHORT Directory Listing Example

```
-DIR mydirectory1 L

03 MAR 82   21:55:24
DIRECTORY OF mydirectory1 ON myvol
                                                  GRAN
   NAME         AT   ACC   BLKS      LENGTH    VOL FIL OWNER           LAST MOD
ed                   -R--    11        1057    1024   1 # 47           02 MAR 82
programs        DR   DL--    30       30185    1024   1 # 47           03 MAR 82
fmat                 DRAU     1          39    1024   1 # 655535       08 NOV 81
OBJFILE              ---U     3        2895    1024   1 # 47           18 DEC 81
ALPHA1.P86           DLAC     2        1304    1024   1 # 50           22 OCT 81
ALPHA1.MP1           DLAC     6        5397    1024   1 # 50           22 OCT 81
manuals         DR   -L--     1         304    1024   1 # 47           02 JUL 80

     7 FILES     54 BLOCKS    41181 BYTES
    33 FILES  3,000 BLOCKS    3,072,000 BYTES FREE
```

Figure 3-4. LONG Directory Listing Example

```
-DIR mydir E

03 MAR 82   21:55:24
DIRECTORY OF mydir ON myvol
                                                  GRAN
   NAME         AT   ACC   BLKS      LENGTH    VOL FIL OWNER           LAST MOD
programs        DR   DL--    30       30185    1024   1 # 47           03 MAR 82
                          CREATION: 01 JAN 81  04:05:44    ACCESSORS       ACC
                          LAST ACC: 03 MAR 82  05:52:33    # 47           DL--
                          LAST MOD: 03 MAR 82  05:52:33    # 50           -LA-
                                                           # 82           -L--
ed                   -R--    11        1057    1024   1 # 47           02 MAR 82
                          CREATION: 11 NOV 81  12:24:05    ACCESSORS       ACC
                          LAST ACC: 02 MAR 82  14:22:16    # 47           -R--
                          LAST MOD: 02 MAR 82  14:22:16
fmat                 DRAU     1          39    1024   1 # 65535        08 NOV 81
                          CREATION: 01 NOV 81  08:54:39    ACCESSORS       ACC
                          LAST ACC: 03 MAR 82  14:56:59    # 65535        DRAU
                          LAST MOD: 08 NOV 81  20:44:01
testdir         DR   DLAC     1          32    1024   1 # 47           01 MAR 82
                          CREATION: 02 FEB 82  15:02:42    ACCESSORS       ACC
                          LAST ACC: 03 MAR 82  09:32:53    # 47           DLAC
                          LAST MOD: 01 MAR 82  13:13:07    # 50           -LA-
                                                           # 65535        -L--

        4 FILES     43 BLOCKS        32213 BYTES
       33 FILES  3,000 BLOCKS    3,072,000 BYTES FREE
```

Figure 3-5. EXTENDED Directory Listing Example

Table 3-4. Directory Listing Headings

Heading	Meaning
NAME	14-character file name.

AT	File attribute, where:
	DR = Directory
	MP = Bit map file
	blank = Data file

ACC	File access rights of the user who entered the DIR command, where:
	
	`┌------- Delete`
	`│┌------ List`
	Directories: `││┌----- Add`
	`│││┌---- Change`
	`DLAC`
	
	`DRAU`
	`│││└---- Update`
	Data Files: `││└----- Append`
	`│└------ Read`
	`└------- Delete`

BLKS	Nine-digit number (five digits on SHORT listing) giving the volume-granularity units allocated to the file. On the SHORT display, if the number of digits exceeds five, DIR displays the file in the nine-digit form (see the LEMONADEIT file in Figure 3-5).
LENGTH	10-digit number (7 digits on SHORT listing) giving the length of the file in bytes. On the SHORT form, if the number of digits exceeds 7, the file is displayed in the 10-digit form (see the LEMONADIT file in Figure 3-5).
VOL	Five-digit number giving the volume granularity in bytes.
FIL	Three-digit number giving the granularity of the file in multiples of volume granularity.
OWNER	14-character, alphanumeric owner name.
LAST MOD	Date of last file modification.
LAST ACC	Date of last file access.
CREATION	Date of file creation.

Table 3-4. Directory Listing Headings (continued)

Heading	Meaning
ACCESSORS	User IDs of users who have access to the file.
ACC	Access rights of the corresponding user. The format of this field is identical to ACC as described previously.

ERROR MESSAGES

- no directory files found

  None of the files you specified were directories.

- <pathname>, READ access required

  You do not have read (list) access to the directory.

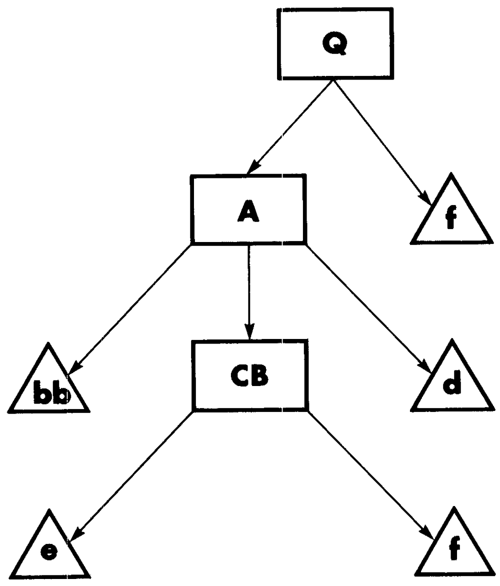
- <pathname>, UPDATE or ADD access required

  Either you cannot overwrite the information in a file because you do not have update access to it, or you cannot copy information to a new file because you do not have add entry access to the file's parent directory. The outpath-list is most likely to be at fault.

EXAMPLES

The examples that follow show how a directory's files are listed when you use your default prefix in a directory's pathname. In the examples, directory names are enclosed in triangles; data file names are enclosed in rectangles.

Assume you have the following directory structure for your files:



x-324

Example 1:

Suppose your default prefix is :F0:Q. This example shows the files that would be listed in response to various DIR commands. It shows the pathnames that you could enter and the resulting files that DIR would list.

Pathname	Files Listed
omitted	A, f
f	not allowed because f is a data file
A	bb, CB, d
A/d	not allowed because d is a data file
A/CB	e, f
A/CB/e	not allowed because e is a data file

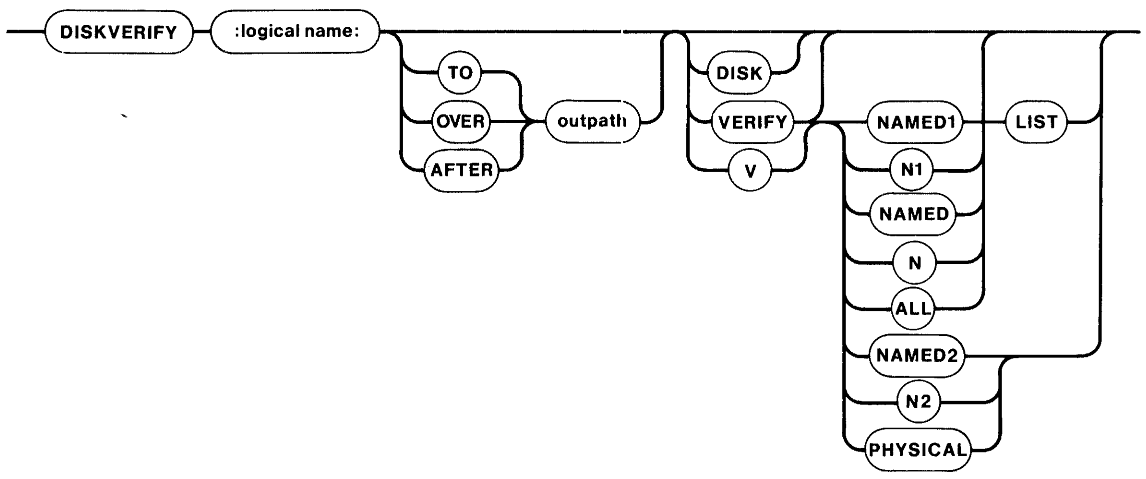
Example 2:

Suppose your default prefix is :F0:Q/A. This example also shows the files that would be listed in response to various DIR commands.

Pathname	Files Listed
omitted	bb, CB, d
A	not allowed because directory A does not contain an entry A
CB	e, f

DISKVERIFY

This command invokes the disk verification utility which verifies the data structures of iRMX 86 physical and named volumes. This utility can also be used to reconstruct portions of the volume and perform absolute editing on the volume. The format of the DISKVERIFY command is as follows:



1120

INPUT PARAMETERS

:logical-name:	Logical name of the secondary storage device containing the volume.
DISK	Displays the attributes of the volume (such as type of volume, device granularity, block size, number of blocks, interleave factor, extension size, volume size, and number of fnodes) and returns control to you at the Human Interface level. You can then enter any Human Interface command. If you omit this parameter (and the VERIFY parameter), the utility displays a sign-on message and the utility prompt (*). You can then enter individual disk verification commands. These commands are described in the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL.

VERIFY or V	Performs a verification of the volume. If you specify this parameter and omit the options, the utility performs the NAMED verification. If you specify this parameter, the utility performs the verification function and returns control to you at the Human Interface level. You can then enter any Human Interface command. If you omit this parameter (and the DISK parameter), the utility displays a sign-on message and the utility prompt (*). You can then enter individual disk verification commands. These commands are described in the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL.
NAMED1 or N1	VERIFY option that applies to named volumes only. This option checks the fnodes of the volume to ensure that they match the directories in terms of file type and file hierarchy. (Refer to the description of the FORMAT command for more information about fnodes.) This option also checks the information in each fnode to ensure that it is consistent. As a result of this option, DISKVERIFY displays a list of all files on the volume that are in error, with information about each file. Refer to the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information.
NAMED or N	VERIFY option that performs both the NAMED1 and NAMED2 verification functions on a named volume. If you omit the VERIFY option, NAMED is the default option.
ALL	VERIFY option that applies to both named and physical volumes. For named volumes, this option performs both the NAMED and PHYSICAL verification functions. For physical volumes, this option performs only the PHYSICAL verification function.
NAMED2 or N2	VERIFY option that applies to named volumes only. This option checks the allocation of fnodes on the volume, checks the allocation of space on the volume, and verifies that the fnodes point to the correct locations on the volume. Refer to the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information.
PHYSICAL	VERIFY option that applies to both named and physical volumes. This option reads all blocks on the volume and checks for I/O errors.

LIST	VERIFY option that you can use with other VERIFY options that, either explicitly or implicitly, specify the NAMED1 option. When you use this option, the file information generated by VERIFY is displayed for every file on the volume, even if the file contains no errors. Refer to the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information.

OUTPUT PARAMETERS

TO	Copies the output from the disk verification utility to the specified file. If the file already exists, DISKVERIFY displays the following information: `<pathname>, already exists, OVERWRITE?` Enter Y, y, R, or r to write over the existing file. Enter any other character if you do not wish to overwrite the file. If no preposition is specified, TO :CO: is the default.
OVER	Copies the output from the disk verification utility over the specified file.
AFTER	Appends the output from the disk verification utility to the end of the specified file.
outpath	Pathname of the file to receive the output from the disk verification utility. If you omit this parameter and the TO/OVER/AFTER preposition, the utility copies the output to the console screen (TO :CO:). You cannot direct the output to a file on the volume being verified. If you attempt this, the utility returns an E$NOT_CONNECTED error message.

DESCRIPTION

When you enter the DISKVERIFY command, the utility responds by displaying the following line:

```
iRMX 86 DISK VERIFY UTILITY, Vx.y
Copyright <year> Intel Corporation
```

where Vx.y is the version number of the utility. If you specify the VERIFY or DISK parameter in the DISKVERIFY command, the utility performs the operation specified in the parameter and copies the output to the console (or to the file specified by the outpath parameter).

Refer to the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for a description of the output. After generating the output, the utility returns control to the Human Interface, which prompts you for more Human Interface commands. The following is an example of a DISKVERIFY command that uses the VERIFY option:

```
-DISKVERIFY :F1: VERIFY NAMED2
iRMX 86 DISK VERIFY UTILITY ,  Vx.y
Copyright <year> Intel Corporation
DEVICE NAME = F1         : DEVICE SIZE = 0003E900 : BLOCK SIZE = 0080

'NAMED2'  VERIFICATION
    BIT MAPS O.K.
-
```

The following is an example of a DISKVERIFY command that uses the DISK option:

```
-DISKVERIFY :F2: DISK
iRMX 86 DISK VERIFY UTILITY, Vx.y
Copyright <year> Intel Corporation

Device name = WF0
   Named disk, Volume name = UTILS
      Device gran = 0080
       Block size = 0080
     No of blocks = 0000072D : No of Free blocks = 00000408
      Volume size = 0003E900
       Interleave = 0005
   Extension size = 03
     No of fnodes = 0038      : No of Free fnodes = 0022
-
```

However, if you omit the VERIFY and DISK parameters from the DISKVERIFY command, the utility does not return control to the Human Interface. Instead, it issues an asterisk (*) as a prompt and waits for you to enter individual DISKVERIFY commands. The following is an example of such a DISKVERIFY command:

```
-DISKVERIFY :F1:
iRMX 86 DISK VERIFY UTILITY, Vx.y
Copyright <year> Intel Corporation
*
```

After you receive the asterisk prompt, you can enter any of the DISKVERIFY commands listed in the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL.

ERROR MESSAGES

- argument error

  The VERIFY option you specified is not valid.

- command syntax error

  You made a syntax error when entering the command.

- device size inconsistent
      size in volume label = <value1> : computed size = <value2>

  When the disk verification utility computed the size of the volume, the size it computed did not match the information recorded in the iRMX 86 volume label. It is likely that the volume label contains invalid or corrupted information. This error is not a fatal error, but it is an indication that further error conditions may result during the verification session. You may have to reformat the volume or use the disk verification utility to modify the volume label. Refer to the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information about the disk verification utility commands.

- not a named disk

  You tried to perform a NAMED, NAMED1, or NAMED2 verification on a physical volume.

The NAMED1, NAMED2, and PHYSICAL verification options can also produce error messages. Refer to the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information about these messages.

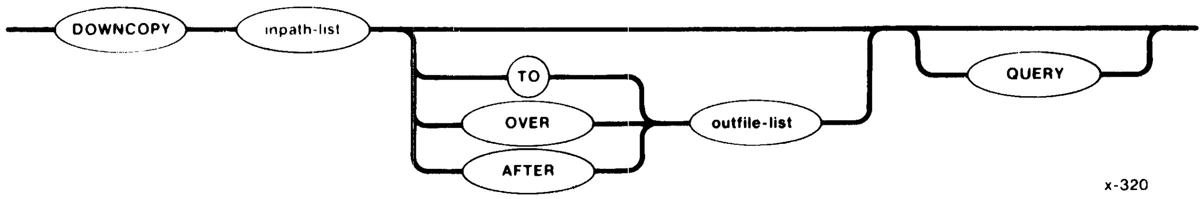
EXAMPLE

The following command performs both named and physical verification of a named volume.

```
-DISKVERIFY :F1: VERIFY ALL

iRMX 86 DISK VERIFY UTILITY, Vx.y
Copyright <year> Intel Corporation

DEVICE NAME = F1           : DEVICE SIZE = 0003E900 : BLK SIZE = 0080

'NAMED1' VERIFICATION

'NAMED2' VERIFICATION
   BIT MAPS O.K.
'PHYSICAL' VERIFICATION
   NO ERRORS

-
```

DOWNCOPY

This command copies files from a volume on an iRMX 86 secondary storage device to a volume on an ISIS-II secondary storage device via the monitor. The format is as follows:



INPUT PARAMETERS

inpath-list	One or more iRMX 86 pathnames for files, separated by commas, that are to be copied to ISIS-II secondary storage. Separating blanks between pathnames are optional. The files may be copied in the listed sequence either on a one-for-one basis or concatenated into one or more files.
QUERY	Causes the Human Interface to prompt for permission to copy each iRMX 86 file to the listed ISIS-II destination file. Depending on which preposition you specify (TO, OVER, or AFTER), the Human Interface prompts with one of the following queries:

```
<pathname>, copy down TO <outfile>?

<pathname>, copy down OVER <outfile>?

<pathname>, copy down AFTER <outfile>?
```

Enter one of the following in response to the query:

Entry	Action
Y or y	Copy the file.
E or e	Exit from the DOWNCOPY command.
R or r	Continue copying files without further query.
Any other character	Do not copy this file; query for the next file in sequence.

OUTPUT PARAMETERS

TO	Reads iRMX 86 files and copies them TO new ISIS-II files in the listed sequence. If the specified output files already exist in the ISIS-II directory when the TO parameter is used, DOWNCOPY displays the following message: `<filename>, already exists, OVERWRITE?` Enter Y, y, R, or r if you wish to delete the existing file. Enter any other character if you do not wish the existing file to be deleted. If no preposition is specified, TO :CO: (ISIS-II console screen) is the default. If more input files than output files are specified, the remaining input files are appended to the end of the last-specified ISIS-II file.
OVER	Copies the iRMX 86 input files OVER the existing ISIS-II destination files in the specified sequence. If you specify multiple input files and one output file, DOWNCOPY appends the remaining input files to the end of the output file.
AFTER	Copies the iRMX 86 input files, in sequence, AFTER the end of data on the existing ISIS-II destination files.
outfile-list	One or more ISIS-II filenames for the output files. Multiple filenames must be separated by commas. Separating blanks are optional. If the preposition and output file defaults are not used in the command line, the output goes to the ISIS-II console screen.

DESCRIPTION

The DOWNCOPY command cannot be used to copy directories from an iRMX 86 system to a Series III Microcomputer Development System; only files can be copied.

Before you enter a DOWNCOPY command on the iRMX 86 console keyboard, your target system must be connected to a Series III system via the iSBC 957B package, the iSDM 86 monitor, and the iSDM 286 monitor. To do this, you must start your iRMX 86 system from the Series III terminal (either by loading the software into the target system and using the monitor G command to start execution, or by using the monitor B command to bootstrap load the software). DOWNCOPY does not function if you start up your system from the iRMX 86 terminal or if you establish the link between the Series III system and target system after starting up your iRMX 86 system.

When DOWNCOPY copies files to the development system, it turns off all ISIS-II file attributes.

As each file in the input list is copied, one of the following messages will be displayed on the Human Interface console output device (:CO:):

```
<pathname>, copied down TO <out-filename>

<pathname>, copied down OVER <out-filename>

<pathname>, copied down AFTER <out-filename>
```

When the DOWNCOPY command is executing, the monitor disables interrupts. This event affects services such as the time-of-day clock. Also, the Operating System is unable to receive any characters that you type-ahead while the monitor is disabling interrupts.

ERROR MESSAGES

- <pathname>, DELETE access required

  DOWNCOPY could not replace an existing ISIS-II file because the file is write-protected.

- <pathname>, ISIS ERROR: <nnn>

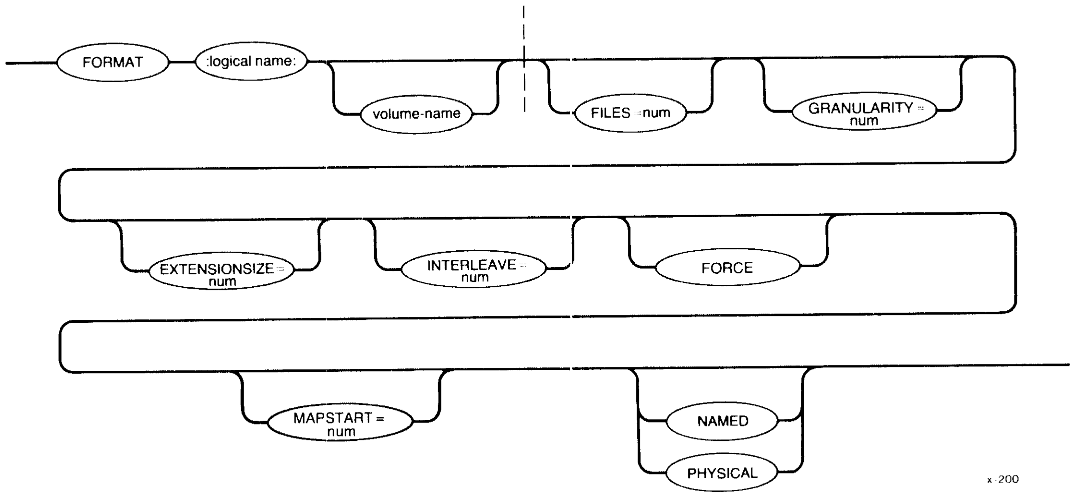
  An ISIS-II Operating System error occurred when DOWNCOPY tried to transfer the file to the Microcomputer Development System. Refer to the INTELLEC SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM CONSOLE OPERATING INSTRUCTIONS for a description of the resulting error code.

- ISIS link not present

  The the iRMX 86 system is not connected to the development system via the monitor.

FORMAT

This command formats or reformats a volume on an iRMX 86 secondary storage device, such as a diskette, tape drive, hard disk, or bubble memory. The format is as follows:



INPUT PARAMETERS

:logical-name:	Logical name of the physical device-unit to be formatted. You must surround the logical name with colons. Also, you must not leave space between the logical name and the succeeding volume name parameter.
volume-name	Six-character, alphanumeric ASCII name, without embedded blanks, to be assigned to the volume. If you include this parameter, you must not leave spaces between the logical name and the volume name.
FILES=num	Defines the maximum decimal number of user files that can be created on a NAMED volume. (This parameter is not meaningful when formatting a PHYSICAL volume and is ignored if specified for such volumes.) FORMAT uses the information specified in this parameter to determine how many structures (called fnodes) to create on the NAMED volume. The range for the FILES parameter is 1 through 32,761, although the maximum number of user files you can define depends on the settings of the GRANULARITY and EXTENSIONSIZE parameters (as explained in the "Description" portion of this command write-up). When you use this parameter, FORMAT creates six additional fnodes for internal system files. If not specified, the default is 200 user files.

FORCE	Forcibly deletes any existing connections to files on the volume before formatting the volume. If you do not specify FORCE, you cannot format the volume if any connections to files on the volume still exist.
MAPSTART=num	Gives the volume block number where the fnodes file, bit map files, and the root directory should start. The size of the block is set by the GRANULARITY parameter. If no number is given, the Operating System puts the fnodes file in the center of the volume. If the number is too low, the Operating System places the map files at the lowest available space on the volume.
GRANULARITY=num	Volume granularity; the minimum number of bytes to be allocated for each increment of file size on a NAMED volume. (This parameter is not meaningful for PHYSICAL volumes, and is ignored if specified for such volumes.) FORMAT rounds the value you specify up to the next multiple of the device granularity. Then it places the decimal number in the header of the volume, where it becomes the default file granularity when a file is created on the volume. The range is 1 through 65,535 (decimal) bytes, although the maximum allowable volume granularity depends on the settings of the FILES and EXTENSIONSIZE parameters (as explained in the "Description" portion of this write-up). If not specified, the default granularity is the device granularity. Once the volume granularity is defined, it applies to every file created on that volume.

NOTE

Using a large volume granularity (in excess of 1024), might cause users to exceed their memory limits when executing programs that reside on the volume. This error can occur because the Operating System uses the volume granularity as a minimum buffer size when reading and writing files.

EXTENSIONSIZE=num	Size, in bytes, of the extension data portion of each file. (This parameter is not meaningful for PHYSICAL volumes, and is ignored if specified for such volumes.) The range is 0 through 255 (decimal), although the maximum allowable extension size depends on the settings of the FILES and GRANULARITY parameters (as explained in the "Description" portion of this write-up). If not specified, the default extension size is 3 bytes.

INTERLEAVE=num	Interleave factor for a NAMED or PHYSICAL volume. Acceptable values are 1 through 255 decimal. If not specified, the default value is 5. See the interleave discussion under "Description" in this section.
NAMED	The volume can store only named files; that is, the volume can hold many files (up to the number of fnodes allocated), each of which can be accessed by its pathname. A diskette or hard disk surface are examples of devices that would be formatted for named files. If neither NAMED nor PHYSICAL is specified, the volume is formatted for the file specified when you attached the device (with the ATTACHDEVICE command).
PHYSICAL	The volume can be used only as a single, physical file. The GRANULARITY and FILES parameters are not meaningful when PHYSICAL is specified for the volume. If neither NAMED nor PHYSICAL is specified, the volume is formatted for the file type specified when you attached the device (with the ATTACHDEVICE command).
QUERY	Prompts the user for permission to format the volume. The Human Interface displays the following: `<volume name>, FORMAT?` If the user replies with a 'Y', 'y', 'R', or 'r', then the volume is formatted. Any other response is considered by the Human Interface be a 'no'.

DESCRIPTION

Every physical device-unit used for secondary storage must be formatted before it can be used for storing and then accessing its files. For example, every time you mount a previously unused diskette into a drive, you must enter a FORMAT command to format that diskette as a new volume before you can create, store and access files on it.

Once a volume is formatted, its name becomes a volume identifier when you display any directory of the volume, and the name appears in the directory's heading. Although the Human Interface uses the volume name in its own internal processing when you access the volume, you need not specify the volume name in any subsequent command after the volume is formatted. You must specify only the logical name of the secondary storage device that contains the volume.

Volume Name

The Human Interface requires a volume name for its own internal processing of your read/write accesses to the volume. Once the volume is formatted, you need never specify the volume name in a command; you only specify the logical name for the device on which you later mount the volume.

For diskettes, a volume name gives you a method for identifying a volume in case the stick-on label on the diskette gets lost or destroyed. You need only mount the disk on a drive and enter a DIR command for that drive to get a directory listing that specifies the volume name.

Fnodes

The number of fnodes on a volume defines the number of files that can exist on the volume. You can specify the number of fnodes reserved for user files with the FILES parameter. Each fnode is a data structure that contains information about a file. Each time you create a file on the volume, the Operating System records information about the file in an unused fnode. Later, it uses the fnode to determine the location of the file on the volume. You can locate fnodes anywhere you wish on a volume.

Internal Files

When you format a named volume, FORMAT creates six internal system files. It names three of these files and lists their names in the root directory of the volume. The files are invisible. The files are:

file	description
R?SPACEMAP	Volume free space map
R?FNODEMAP	Free fnodes map
R?BADBLOCKMAP	Bad blocks map
R?VOLUMELABEL	Volume label

The Operating System grants the user WORLD read access to these files. Refer to the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information about these files.

Root Directory

FORMAT also uses one of the fnodes for the root directory. It lists the user who formats the volume as the owner, giving that user all access rights. No other user has access to the root directory until the owner explicitly grants access. The owner can grant other users access to the volume via the PERMIT command described later in this chapter. However, because the owner has all access rights to the root directory, the owner can obtain exclusive access to the volume, and can obtain delete access to any file created on the volume, even files created by other users.

Extension Data

Each fnode contains a field that stores extension data for its associated file. An operating system extension can access and modify this extension data by invoking the A$GET$EXTENSION$DATA and A$SET$EXTENSION$DATA system calls (refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for more information). When you format a volume, you can use the EXTENSIONSIZE parameter to set the size of the extension data field in each fnode. Although you can specify any size from 0 to 255 bytes, the Human Interface requires all fnodes to have at least 3 bytes of extension data.

Volume Granularity

The default volume granularity is always the granularity of the physical device for the volume. For example, if the default granularity for a device is 128 bytes of secondary storage, the I/O System will automatically allocate permanent storage to each new file you create on that volume in multiples of 128 bytes, regardless of whether the file requires the full amount.

Relationship between FILES, GRANULARITY, and EXTENSIONSIZE

Although the FILES, GRANULARITY, and EXTENSIONSIZE parameters have maximum values which are listed in the parameter descriptions, the combination of these parameters must also satisfy the following formula:

$$(87 + \text{EXTENSIONSIZE}) \times (\text{FILES} + 6) / \text{GRANULARITY} \leq 65535$$

where all numbers are decimal. FORMAT displays an error message if the combination of parameter values exceeds the limit.

Interleave Factor

The interleave factor applies to volumes formatted either for NAMED or

PHYSICAL files. The interleave factor specifies the logical sector sequence. If the consecutively-accessed sectors of a disk are staggered (that is, if they are not consecutive physical sectors), disk access time can decrease considerably. The reason for this decrease is that although a controller cannot read a sector and issue another read command in the time it takes for the next sector to be positioned under the head, the controller can perform this operation in less time than it takes for the disk to revolve once. Therefore, if the consecutively-accessed sectors are staggered correctly, the next accessed sector will be positioned under the read head just as the controller becomes ready to read it.

The amount of staggering is called the interleave factor. An interleave factor of two means that as the disk rotates, the controller consecutively accesses every second sector. An interleave factor of five means that the controller consecutively accesses every fifth sector. The following diagram illustrates how a controller accesses sectors on a 12-sector disk with an interleave factor of two.

Sector Number	Access Number	Rotation Number
Sector 0	0	1
Sector 1	6	2
Sector 2	1	1
Sector 3	7	2
Sector 4	2	1
Sector 5	8	2
Sector 6	3	1
Sector 7	9	2
Sector 8	4	1
Sector 9	10	2
Sector 10	5	1
Sector 11	11	2

Note that the interleave factor also implies the number of disk rotations necessary to access all the sectors on a given track. Thus from the previous diagram you can see that an interleave factor of two implies that it takes two rotations of the disk to access all the sectors on a track.

When The Interleave Factor Is Important

The interleave factor is important when large transfers of consecutive data take place at speeds that approach the maximum transfer rate of the disk. This type of transfer occurs in the following cases:

- When you bootstrap load the Operating System from disk.
- When you use the Application Loader to load an application program from disk.
- When you invoke programs that perform large transfers of consecutive data, such as the Human Interface COPY command.

How To Select An Interleave Factor

Suitable interleave factors depend on the turnaround time of the software that controls the I/O operations; that is, the time between reading a sector and becoming ready to read the next sector. In the cases listed in the previous paragraph, the turnaround time between sector accesses is different. Therefore the ideal interleave factors could be different. The differences are:

- The Bootstrap Loader instructs the disk controller to read one sector at a time. Thus, the turnaround time depends on the execution overhead of the Bootstrap Loader and is comparatively long. A large interleave factor is optimal for flexible disks that you use with the Bootstrap Loader. For hard disks however, the Bootstrap Loader has no effect on the turnaround time because revolution speed is so great that more than one disk revolution occurs between sector reads.
- The Application Loader reads several sectors at a time into its internal buffer. Then it takes a relatively long time to process the object records in this buffer. The ideal interleave factor here is one that optimizes for the object record processing time between disk accesses. For flexible diskettes, this interleave factor is somewhat smaller than that for the Bootstrap Loader. However, hard disks, as in the previous paragraph, are not affected by the Application Loader.
- Applications which transfer large amounts of consecutive data (such as the COPY command) can initiate data transfers involving many sequential sectors. Thus the controller accesses sectors on a given track as fast as possible. Here, the ideal interleave factor is one that optimizes for the turnaround speed of the disk controller.

The ideal interleave factor depends heavily on the application. However, because the revolution speed of hard disks is so high, you should format them with interleave factors that are optimized for the turnaround speed of the disk controller.

The value to use for flexible diskettes depends on how you are going to use the diskettes. For flexible diskettes that contain bootstrap-loadable information (system disks), you should select an interleave factor that optimizes for Bootstrap Loader performance. This ensures that the bootstrap loading process completes in a reasonable amount of time, despite using a device that is relatively slow-turning. For non-system diskettes that contain loadable files (such as Human Interface commands), select an interleave factor that optimizes for Application Loader performance. Otherwise, select a value that optimizes for copying.

If you do not know the optimal value for an interleave factor, it is better to specify an interleave factor that is too large rather than one that is too small. An interleave factor that is slightly larger than optimal causes the disk to move only an extra sector or two before reaching the correct sector. However, an interleave factor that is less than optimal causes the disk to make nearly a complete revolution before reaching the sector.

Optimal Interleave Factors For Intel Devices

This section lists the optimum values for some devices that Intel has tested.

Table 3-5. Optimal Interleave Factor for Hard Disk Controllers

Device	Optimal Interleave Factor
iSBC 206 device	4
iSBC 215 device	
Priam	3
ANSI	4
Fujitsu/Memorex	3
Shugart SA 1004/Quantum	2
CMI	4
RMS	4
iSBC 220 device	4
iSBC 254 device	(Not applicable to bubble memory devices)

Table 3-6. Flexible Disk Controllers (using 8" disks)

Device	Optimal Interleave Factor For		
	Application Loader	Bootstrap Loader	COPY
iSBC 204 device	5	7	1
iSBC 208 device	5	7	1
iSBX 218 device	5	7	2

Output Display

The FORMAT command displays one of the following message when volume formatting is completed. For physical volumes:

```
volume (<volume name>) will be formatted as a PHYSICAL volume
    device gran.        = <number>
    interleave          = <number>
    volume size         = <k-number> K (or M)

TTTTTTTTTTTTTTTTTT...

volume formatted
```

While the storage device is being formatted, FORMAT displays on the console the letter "T" for every 100 tracks formatted. For example, if you see three T's on the screen, the Operating System has finished formatting at least 300 tracks. Displaying the T's on the screen is useful when you format large capacity disks. A continuous stream of T's lets you know that the system hasn°t failed diring the FORMAT operation.

For named volumes:

```
volume (<volume name>) will be formatted as a NAMED volume
    granularity        = <number>     map start = <number>
    interleave         = <number>     sides = <sides>
    files              = <number>     density = <density>
    extensionsize      = <number>     disk size = <d-size>
    volume size        = <k-number> K (or M)

TTTTTTTTTTTT...

volume formatted
```

where:

<number>	Position where the fnodes start.
<volume name>	Volume name specified in the FORMAT command.
<number>	Decimal number as specified in the command (or the default)
<k-number>	Volume size in K (1024-byte units) or M (1048576-byte units). FORMAT displays the volume size in Kbyte units unless the size is greater than 25 Mbytes.
<sides>	Number of sides of the volume that will be formatted (1 or 2). This field is displayed only for flexible diskettes in which FORMAT can recognize this characteristic.
<density>	Density at which the volume will be formatted (single or double). This field is displayed only for flexible diskettes in which FORMAT can recognize this characteristic.
<d-size>	Size of the volume (8 or 5.25). This field is displayed only for flexible diskettes in which FORMAT can recognize this characteristic.

ERROR MESSAGES

- <logical name>, can't attach device
  <logical name>, <exception value> : <exception mnemonic>

  FORMAT cannot attach the device for formatting, or it cannot re-attach the device (that is, restore it to its original condition) after formatting takes place.

- <logical name>, can't detach device
  <logical name>, <exception value> : <exception mnemonic>

  FORMAT cannot detach the device for formatting, which means that the volume does not exist, the volume is busy, or the device on which the volume is mounted is not currently attached to the system.

- <logical name>, device is in use

  You cannot format the volume because there are outstanding connections to files on the volume and you did not specify the FORCE parameter.

- <vol-name>, fnode file size exceeds 65535 volume blocks

  The values you specified for fnode size, granularity, and extension data size cause the formula listed in the "Description" section to exceed its limit.

- <number>, invalid number

  You specified an out-of-range number for any of the FILES, GRANULARITY, EXTENSIONSIZE, or INTERLEAVE parameters.

- <logical-name>, map files do not fit

  The volume is too small for the map files or the map start block is too high to allow room for the map files.

- <logical name>, outstanding connections to device have been deleted

  There were outstanding connections to files on the volume. However, because you specified the FORCE parameter, FORMAT deleted those connections. This is a warning message that does not prevent FORMAT from formatting the volume.

- 0085 : E$LIST, too many values

  You entered multiple logical-name/volume-name combinations separated by commas. FORMAT can format only one volume per invocation.

- <logical-name>: <exception code> unit status <xx>

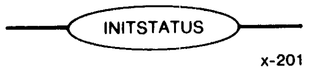
  An I/O error occurred while physically formatting the volume. <exception code> informs you of the type of error.

- <volume name>, volume name is too long

  FORMAT requires the volume name you specify to be 6 characters or less.

INITSTATUS

This command displays the initialization status of Human Interface terminals. The format of this command is as follows:



DESCRIPTION

INITSTATUS displays at the user terminal the initialization status of all Human Interface terminals. Figure 3-6 illustrates the format of the INITSTATUS display.

```
   TERMINAL        CONFIG     DEVICE     INIT       USER      JOB     USER
 DEVICE NAME       EXCEP      EXCEP      EXCEP      STATE     ID      ID
 .T0.               0000       0000       0000       LE       1       65535
 .T1.               0000       0000       0000       -E       2       1
 .T3.               0000       0002                  --
 .T4.               0021                             --
```

Figure 3-6. INITSTATUS Display

The columns listed in Figure 3-6 contain the following information.

TERMINAL DEVICE NAME	The physical name of the terminal, as defined during the configuration of the Basic I/O System and as attached by the Human Interface. Periods surround each name.
CONFIG EXCEP	Hexadecimal condition code that the Human Interface received when it attempted to interpret the terminal definition and user definition files (refer to the iRMX 86 CONFIGURATION GUIDE for more information). A zero value indicates a normal condition. Nonzero values indicate exceptional conditions. Refer to Appendix B for a list of exception codes.
DEVICE EXCEP	Hexadecimal condition code that the Human Interface received when it originally attached the terminal as a physical device.

INIT EXCEP	Condition code that the Human Interface received when it created a job for the interactive session.
USER STATE	Two characters that indicate the current state of the terminal. The first character can be either: L The terminal is locked and cannot be reinitialized (refer to the LOCK command later in this chapter). - The terminal is unlocked. The second character can be either: E The Human Interface created the interactive job associated with this terminal and the job exists. - The interactive job does not exist.
JOB ID	A sequential number that the Human Interface assigns to the interactive job during initialization. You must specify this number as a parameter in the JOBDELETE command in order to delete the corresponding interactive job.
USER ID	User ID associated with the interactive job. This ID is the identification of the user that the Human Interface associates with the job when the user begins a Human Interface session.

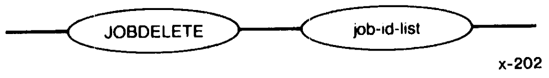
ERROR MESSAGE

- not a multi-access system

  The Human Interface cannot return information about terminals because it is not configured for multi-access.

JOBDELETE

This command deletes a running interactive job. The system manager can use this command to delete any interactive job. Other users can delete only those interactive jobs that have the same user ID that they have. The format of this command is as follows:



where:

job-id-list	One or more job IDs, separated by commas, of the interactive jobs to be deleted. You can obtain the IDs of jobs by invoking the INITSTATUS command (described earlier in this chapter).

DESCRIPTION

The JOBDELETE command allows users to delete interactive jobs. Deleting an interactive job causes the Human Interface to terminate the corresponding user session.

When JOBDELETE attempts to delete a job, it first attempts to delete the job's offspring jobs (for example, a SUBMIT file or a program invoked as a result of an RQ$CREATE$IO$JOB system call). It deletes multiple levels of offspring jobs. However, JOBDELETE cannot delete any interactive job (or offspring) that contains extension objects. Refer to the iRMX 86 NUCLEUS REFERENCE MANUAL for more information about deleting jobs containing extension objects.

Normally, when a user's interactive job is deleted, the Human Interface recreates the interactive job, thus restarting the user session. However, if the LOCK command (described later in this chapter) has been specified for the user's terminal, the Human Interface does not automatically recreate the user's interactive job after a JOBDELETE command. Therefore, the system manager can use the combination of LOCK and JOBDELETE to remove users from the system prior to a system shutdown.

As JOBDELETE deletes each job, it displays the following message at the user terminal (:CO:):

```
    <job-ID>, deleted
```

where <job-ID> is the identifier of the deleted job.

ERROR MESSAGES

- <job-ID>, does not exist

  The interactive job associated with the identifier <job-ID> does not exist. It has already been deleted.

- <job-ID>, invalid job id

  The number <job-ID> is not a job ID that is associated with any terminal managed by the Human Interface.

- <job-ID>, job does not belong to you

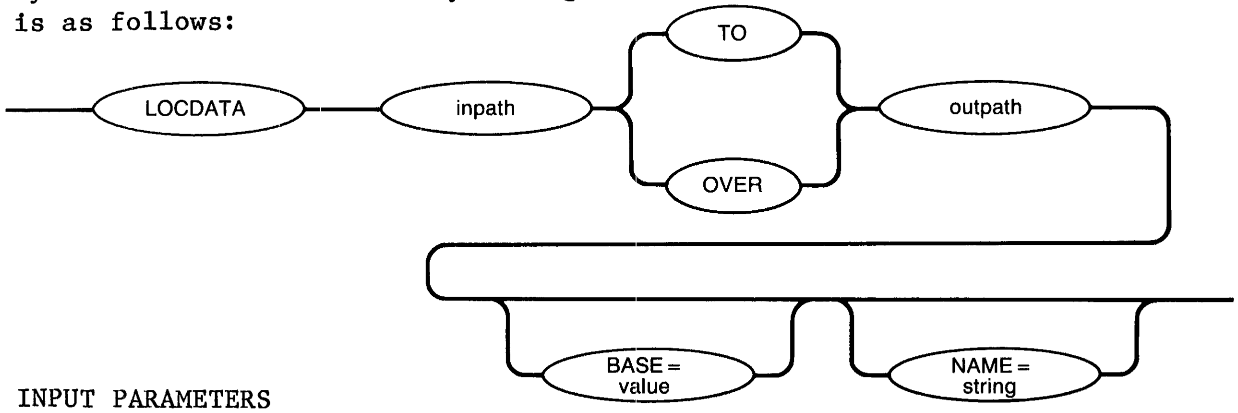
  The user who attempted to delete the interactive job does not have the same user ID as the interactive job or is not the system manager.

- <job-ID>, not deleted
  <job-ID>, <exception value> : <exception mnemonic>

  An exceptional condition occurred, preventing JOBDELETE from deleting the job <job-ID>. JOBDELETE displays the exception code that resulted.

LOCDATA

This command locates a data stream and transforms it into an object module that the iAPX 86, 88 utilities (LINK86, LOC86, LIB86, etc.) can process and the iRMX 86 Bootstrap Loader can load. By locating the data stream, it sets the absolute address at which the Bootstrap Loader loads the data. You normally use this command when creating an application system that includes a RAM-disk (an area of memory that the Operating System treats as a secondary storage device). The format of this command is as follows:



x-668

INPUT PARAMETERS

inpath	Pathname of the file to be processed. Multiple or wild-card pathnames are not allowed.
BASE=value	Base portion of the address at which LOCDATA locates the data stream. The data stream can be located only on 16-byte (paragraph) boundaries. Therefore LOCDATA always uses a 0 value for the offset portion of the address (that is, value:0). You can specify a radix character of "O" or "H" at the end of the value to indicate octal or hexadecimal, respectively. If you omit the radix character, decimal is the default. If you omit this parameter, LOCDATA assumes a value of 0. If you are using LOCDATA to set up the information to be bootstrap loaded into a RAM DISK, you must set this parameter to correspond to the beginning address of the memory device. Refer to the iRMX 86 CONFIGURATION GUIDE for more information about setting up a RAM DISK.
NAME=string	Module name which LOCDATA associates with the output module. Whenever you use LINK86 or LOC86 to process the module, this name appears in the map files. You should use a valid PL/M-86 identifier for this parameter. If you omit this parameter, LOCDATA uses a default value of @LOCDATA.

OUTPUT PARAMETERS

TO	Writes the processed output to a named file. The specified output file should not already exist. If it does, LOCDATA displays the following message: `<pathname>, already exists, OVERWRITE?` Enter Y, y, R, or r if you wish to write over the existing file. Enter an "N" (upper or lower case) or a carriage return alone if you do not wish to overwrite the existing file. In the latter case, the LOCDATA command will exit without processing the data.
OVER	Writes over (replaces) the existing output file, regardless of file size. If the output file does not already exist, LOCDATA creates a new file.
outpath	Pathname of the file to receive the output of LOCDATA. Multiple or wild-card pathnames are not allowed.

DESCRIPTION

LOCDATA transforms an arbitrary string of data into a module that you can add to a library using LIB86 and load into memory using the iRMX 86 Bootstrap Loader. LOCDATA creates an L-module to contain the data. The L-module consists of an L-module header record (LHEADR), a SEGDEF record, a number of physically-enumerated data records (PEDATA), and a module end record (MODEND). LOCDATA places the module name into the LHEADR record. It sets the appropriate fields of the MODEND record to indicate that the module is a non-main module with no start address. Refer to the iAPX 86, 88 FAMILY UTILITIES USER'S GUIDE for more information about L-modules.

After processing the data, LOCDATA displays one of the following messages:

```
<inpath>, located TO <outpath>

<inpath>, located OVER <outpath>
```

Two of the LOCDATA parameters allow you to specify information about the L-module that LOCDATA creates. The BASE parameter allows you to specify the base address of the module. When the Bootstrap Loader loads the module, it places the data at the specified base address with an offset of 0. The NAME parameter allows you to specify a module name. If you process the module with LINK86 or LOC86, the resulting map files list the name you specify as the only symbol name in the module.

LOCDATA is a valuable tool for configuring an application system that includes a RAM DISK (an area of memory that acts like a secondary storage device). LOCDATA allows you to process an entire volume of Human Interface commands (and any other files you desire) so that you can include a copy of that volume in a library that the Bootstrap Loader loads. If you do this correctly, when you bootstrap load the system your RAM DISK will be formatted automatically, and it will contain the commands and files you need. This feature is useful in many applications and is necessary when installing the Operating System on a system that contains a tape drive and an unformatted Winchester disk.

To create an application system that contains a RAM DISK that receives data via the Bootstrap Loader, perform the following steps:

1. Configure a version of the Operating System that includes a RAM DISK. Refer to the iRMX 86 CONFIGURATION GUIDE for more information. Make a special note of the address to which you assign the device.

2. Bootstrap load this new version of the Operating System.

3. Attach the RAM DISK as a named device. For example, you could enter the following command:

   ```
   ATTACHDEVICE RAM AS :RAM:
   ```

4. Format the RAM DISK for named files. For example, you could enter the following command:

   ```
   FORMAT :RAM:
   ```

5. Copy Human Interface commands (and other files that you require) to the RAM DISK. An error message will occur when you run out of room in the RAM DISK.

6. Detach the RAM DISK. For example:

   ```
   DETACHDEVICE :RAM:
   ```

7. Attach the RAM DISK as a physical device. For example:

   ```
   ATTACHDEVICE RAM AS :RAM: PHYSICAL
   ```

   This allows you to access all the data in the device, including the formatting information.

8. Use LOCDATA to process the information from the RAM DISK and place the output in another file. Use the address of the RAM DISK as the value for the BASE parameter. For example, if you configured your RAM DISK to have a base address of 5000H, you could enter the following command:

   ```
   LOCDATA :RAM: TO COMMANDS  BASE=5000H  NAME=COMMAND5000
   ```

9. Use LIB86 to add the processed output (in this case, the file COMMANDS) to the library that contains the bootstrap loadable version of the Operating System.

Now, whenever you bootstrap load this version of the Operating System, the RAM DISK will be formatted, and it will contain the commands and files that LOCDATA placed into the COMMANDS file.

The iRMX 86 CONFIGURATION GUIDE contains more information about the configuration of the RAM DISK. It also describes how to configure a system that you can bootstrap load from a tape device.

ERROR MESSAGES

- invalid input specification

  The input pathname you specified contains invalid characters.

- <value>, invalid number

  The value you entered with the BASE parameter was not a valid hexadecimal, decimal, or octal number.

- invalid output specification

  The preposition or the output pathname you specified contains invalid characters.

- <pathname>, list of pathnames not allowed

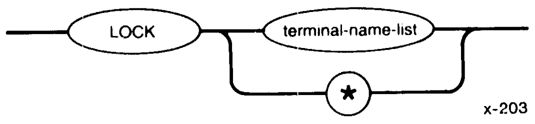
  You entered more than one pathname for either the inpath or outpath parameters.

- <pathname>, output file same as input file

  You attempted to place the output into the input file. LOCDATA does not allow this.

LOCK

This command prevents the Human Interface from automatically recreating the interactive job for a terminal once that interactive job has been deleted. As a result of disallowing recreation of the interactive job, users cannot access that specific terminal. This process is called locking the terminal. The system manager can use this command to lock any terminal. Other users can lock only those terminals whose interactive jobs have the same user ID that they have. The format of this command is as follows:



where:

terminal-name-list	One or more terminal device names, separated by commas, of the terminals to be locked. You can obtain the terminal device names by invoking the INITSTATUS command (described earlier in this chapter).
*	A special character indicating that all configured terminals should be locked.

DESCRIPTION

The system manager can use the LOCK command in conjunction with the JOBDELETE command either to selectively delete users from the system or to shut down the entire system. LOCK prevents the Human Interface from recreating a user's interactive job once that job has been deleted. Interactive jobs can be deleted in any of the following ways:

- As a result of the JOBDELETE command (described earlier in this chapter)
- By shutting off the terminal
- By entering an end-of-file character (CTRL/z) at the terminal

As LOCK locks each terminal, it displays the following message to the user terminal (:CO:):

```
<terminal-name>, locked
```

where <terminal-name> is the terminal device name of the locked terminal.

ERROR MESSAGES

- lock not allowed

  You attempted to lock your own terminal. Only system managers can lock their own terminals.

- <terminal-name>, not found

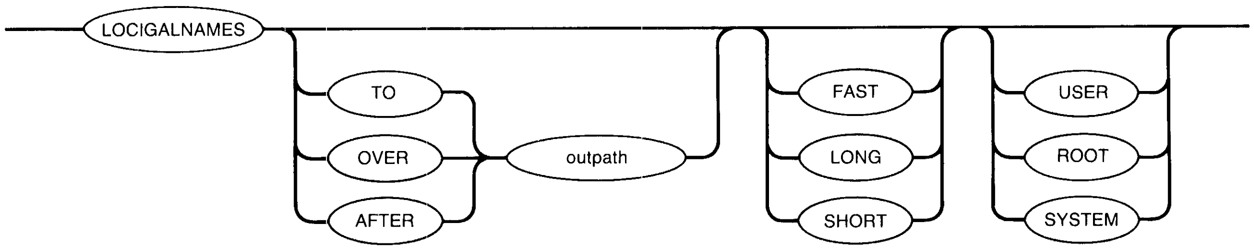
  A terminal with device name <terminal-name> is not configured into your application system.

- not a multi-access system

  The LOCK command does not function if the Human Interface is configured for single-access only.

LOGICALNAMES

This command lists all the current logical names available to the user.



x-662

INPUT PARAMETERS

FAST	Lists the logical names in a system without providing any additional information beyond the name itself. FAST is the default parameter.
SHORT	Lists all the logical names with the following additional information: type of logical name, the physical device name, owner of the logical name, and the current connections to the file or device.
LONG	Like the SHORT parameter, but also adds the complete pathname associated with a logical name.
ROOT	If ROOT is specified with the LONG parameter, then the pathname associated with the logicalname is displayed back to the root device.
USER	Displays all the logical names associated with the current user.
SYSTEM	Displays the logical names of system defined files and devices.

OUTPUT PARAMETERS

outpath-list	The pathname of the file to receive the output of command.

DESCRIPTION

The following is an example of the listing you get when you invoke the command with a FAST control (FAST is also the default parameter):

```
    --LOGICALNAMES FAST

    USER LOGICAL NAMES:

    $      CI      HOME      CO      PROG
    TERM   ACCOUNTS

    SYSTEM LOGICAL NAMES

    SYSTEM     WORK     *SD     *BB     *STREAM
```

When an asterisk preceeds a name, the logical name refers to a logical device.

The following example shows the output listing when you use the SHORT parameter:

```
    --LOGICALNAMES SHORT

    USER LOGICAL NAMES:

      name          type       fdr        con        dev name      owner
    $                dir       NAM        3        smd0            WORLD
    CI              file       PHY        5        T1
    CO              file       PHY        5        T1
    HOME             dir       NAM        3        smd0            WORLD
    PROG             dir       NAM        2        smd0            WORLD
    TERM            file       PHY        5        T1

    SYSTEM LOGICAL NAMES:
      name          type       fdr        con        dev name      owner
    SYSTEM           dir       NAM        1        smd0             0
    WORK             dir       NAM        1        smd0            WORLD
    SD              ldev       NAM        1        smd0             0
    BB              ldev       PHY                 BB               0
    STREAM          ldev       PHY        1        STREAM           0
```

In the listing, TYPE refers to the kind of logical name: file, directory, map (system file), or logical device (ldev). fdr (file driver) indicates whether the connection is to a named or physical file. The number of connections which a file or device has is under the CON heading. The dev name heading shows the physical device associated with the logical name. In the case of a directory or file, the name shows on what the device the file or directory exists. The originator of the connection to the logical name is shown under the owner heading.

The use of the LONG parameter produces the same type of listing as the SHORT parameter with the addition of the complete pathname of the logical name. Following is an example of the LONG listing:

```
--LOGICALNAMES LONG

USER LOGICAL NAMES:

  name    type     fdr      con       dev name    owner    pathname
$          dir     nam      3        smd0         WORLD    :sd:user/world
CI        file     PHY      5        T1             0      :ci:
CO        file     PHY      5        T1             0      :co:
HOME       dir     NAM      3        smd0         WORLD    :sd:user/world
PROG       dir     NAM      2        smd0         WORLD    :$:prog
TERM      file     PHY      5        T1             0      :term:

SYSTEM LOGICAL NAMES:

  name    type     fdr      con      dev name     owner    pathname
SYSTEM     dir     NAM      1        smd0           0      :sd:system
WORK       dir     NAM      1        smd0          WORLD   :sd:work
SD        ldev     NAM      1        smd0           0      :sd:
BB        ldev     PHY               BB             0      :bb:
STREAM    ldev     PHY      1        STREAM         0      :stream:
```

The ROOT parameter produces the same type of listing as the LONG parameter except that the pathname starts at the root directory. Thus, you get a complete description of the pathname associated with the logical name.

If the pathname has elipses before it (.../user/dirl/dir2/dir3/filename), LOGICALNAMES truncated the pathname because it was too long to fit in its column. The pathname only shows the last elements of the pathname which describes a file or directory.

MEMORY

This command displays the available memory in the user's memory pool.

x-665

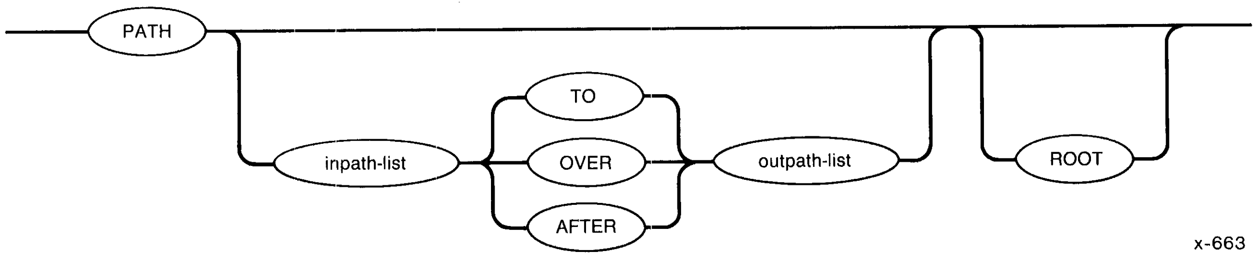
DESCRIPTION

This command requires no parameters. The following is an example of the listing produced by this command:

```
    --MEMORY
      AVAILABLE MEMORY:  123.4 K BYTES
```

PATH

This command lists the pathname of a data file or directory.



INPUT PARAMETERS

inpath-list	The list of files whose pathnames you want to know. The default file is your home directory.
ROOT	Specifies that the pathname should start from the root directory of whatever device holds the file or directory.

OUTPUT PARAMETERS

TO

Writes the listed input files to named new output files. The specified output file or files should not already exist. If they do, PATH displays the following message:

```
<pathname>, already exists, OVERWRITE?
```

Enter Y, y, R, or r if you wish to write over the existing file. Enter an N (upper or lower case) or a carriage return alone if you do not wish to overwrite the existing file. In the latter case, the PATH command will pass over the corresponding input file without copying it, and will attempt to copy the next input file to its corresponding output file.

If you specify multiple input files and a single output file, PATH appends the remaining input files to the end of the output file.

OVER

Writes the input files over (replaces) the existing output files on a one-for-one basis, regardless of file size. If an output file does not already exist, its corresponding input file is written to a new file with the corresponding output file name. If you specify multiple input files and a single output file, PATH appends the remaining input files to the end of the output file.

AFTER	Appends the input file or files to the current data in the existing output file or files. If the output file does not already exist, all listed input files will be concatenated into a new file with the listed output file name.
outpath-list	One or more pathnames for the output files.

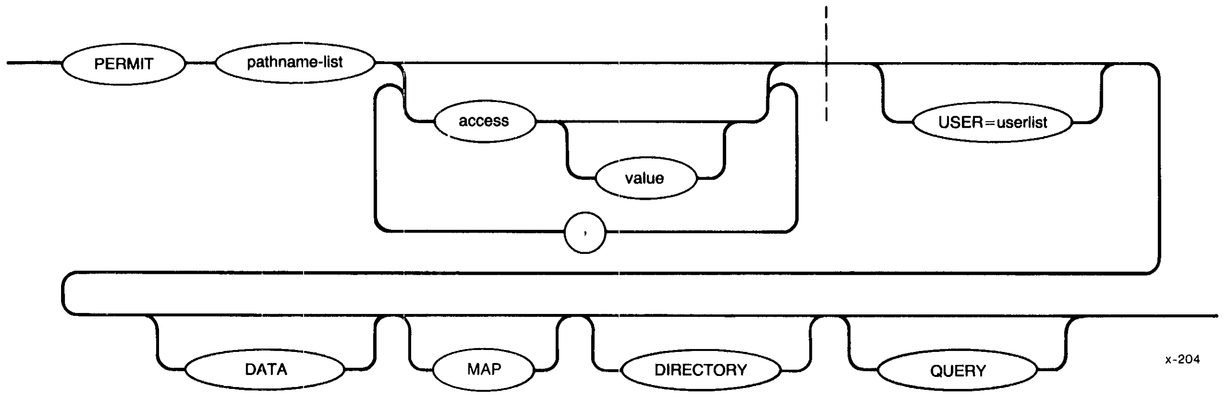
DESCRIPTION

This command is useful for finding where you may be located within the file structure. The command gives the following listing when it is invoked with no input file listing:

```
    --PATH
    :sd:user/world
```

PERMIT

This command allows you to grant or revoke user access to files that you own. The format of this command is as follows:



INPUT PARAMETERS

pathname-list	One or more pathnames, separated by commas, of the files that are to have their access rights or list of accessors changed.
access	Access characters that grant or rescind the corresponding access to the file, depending on the value parameter that follows. The possible values include:

value	access
D	Delete
L or R	List (for directories) and read (for data files)
A	Add entry (for directories) and append (for data files)
C or U	Change (for directories) and Update (for data files)
N	Rescinds all access not explicitly granted (used without an accompanying value)

If specified without an accompanying value, each access character grants the specified access. Specifying N alone rescinds all access and removes the users specified with the USER parameter from the file's access list. Specifying N with other characters grants the access specified by those characters and rescinds all other access. You can use L and R interchangeably for both data files and directories; likewise C and U.

value	Value which specifies whether to grant or rescind the associated access right. Possible values include: value / meaning 0 — Rescind the access right 1 — Grant the access right The default value is 1. That is, specifying an access character without a value grants the corresponding access.
user-list	User IDs for whom the previously-specified access rights apply. Two special values are also acceptable for this parameter. They are: WORLD — Special user ID (0FFFFh) giving all users access to the file. * — Designator indicating that the access rights apply to all users currently in the file's access list. The Operating System limits each file to three user IDs in the access list. If you omit this parameter, PERMIT assumes the user ID associated with your interactive job.
DATA	Specifies that the access information applies to the data files in the pathname list. If you omit both the DATA and DIRECTORY parameters, PERMIT assumes both.
DIRECTORY	Specifies that the access information applies to the directories in the pathname list. If you omit both the DATA and DIRECTORY parameters, PERMIT assumes both.
MAP	Specifies that access information also applies to the map and volume label files in the pathname list.

QUERY — Causes PERMIT to prompt for permission to modify the access rights associated with each file. It does this prompting by displaying the following message:

```
<pathname>,
     accessor  = <new id>, <new access>,  PERMIT?--
```

Enter one of the following (followed by a carriage return) in response to the query:

Entry	Action
Y or y	Change the access.
E or e	Exit from the PERMIT command.
R or r	Change the access and continue with the command without further query.
Any other character	Do not change access; continue with PERMIT command and query for next access change, if any.

DESCRIPTION

You can use the PERMIT command to update the access information for the following files:

- Files for which you are listed as the owner.
- Files for which you have change-entry access to the file's parent directory.

You cannot change the access information for other files. PERMIT can perform the following functions:

- Adding or subtracting users from a file's list of accessors. This list determines which users have access to the file.
- Setting the type of access (access rights) granted to the users in the accessor list.

Currently the Operating System allows only three user IDs in the list of accessors, but one of these IDs can be the special ID WORLD, which grants access to all users.

You specify the type of access to be granted or rescinded by means of access characters and values. You can concatenate access characters and values together or you can separate the individual access specifications with commas. For example, if you want to grant delete access and rescind add and update access, you could enter any of the following combinations:

```
    A0DU0
    A0,D,U0
    A0D1U0
    A0,D1,U0
```

As you can see from the previous lines, D is equivalent to D1. Also, the order in which you specify access characters is not important.

If there are multiple occurrences of an access character in the PERMIT command, PERMIT uses the last such character to determine the access. For example, the combination:

```
    D0,A1,R1,D1
```

is the same as the combination:

```
    A1,R1,D1
```

In the first combination, the D1 overrides the D0.

You can use the N character to rescind all access to the file. If specified alone, it removes user IDs from the accessor list. However, the N character can also be useful when changing access rights, if you don't remember the specified user's current access rights. In this case you can specify the N character first, to clear all the access rights, and follow it with other characters to grant the desired access. For example, if you want to grant list access only, instead of specifying:

```
    D0A0C0L
```

you could specify:

```
    NL
```

After changing the access information for a file, PERMIT displays the following information:

```
    <pathname>,
        accessor  = <accessor ID>, <access>
                  .
                  .
                  .
```

where <pathname> is the pathname of the specified file, <accessor ID> is the user ID of one of the files accessors, and <access> indicates the access rights that the corresponding user has. PERMIT displays the access rights as access characters: DLAC for directories and DRAU for data files. If a particular access right is not allowed, the display replaces the corresponding character with a dash (-). For example, the display:

```
    -L-C
```

indicates that the corresponding user has list and change access.

ERROR MESSAGES

- <pathname>, accessor limit reached

  The Operating System permits only three IDs in the accessor list of a file. Before you can add another accessor, you must remove one of the current accessors by setting its access rights to N.

- <pathname>, directory CHANGE access required

  Either you are not the owner of the file specified by <pathname>, or you do not have change access to the file's parent directory. You must satisfy one of these two conditions in order to use the PERMIT command.

- <user ID>, duplicate USER control

  You must specify the keyword and parameter combination USER = userlist only once during the PERMIT command. However, you can specify multiple user IDs by separating them with commas in the userlist. PERMIT exits without updating the access rights.

- <character>, invalid access switch

  The character you entered to indicate the access rights for the file was not a valid access character. PERMIT exits without updating the access rights.

- <invalid id>, invalid user id

  The user IDs you supply with the USER parameter must consist of decimal or hexadecimal characters, the characters WORLD, or the character *. PERMIT exits if supplied other characters.

- missing access switches

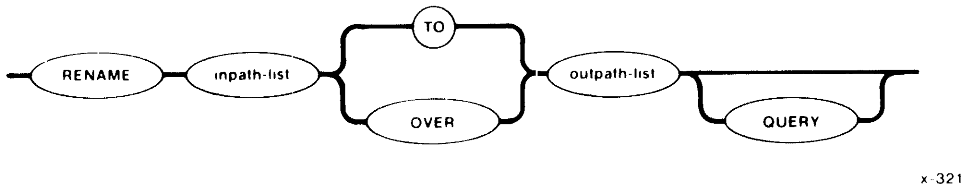
  You must specify one or more access characters with the PERMIT command. PERMIT exits without updating the access rights.

- no files found

  There were no files of the type you specified (data, directory, or both) in the pathname list.

RENAME

This command allows you to change the pathname of one or more data files or directories. RENAME is effective across directory boundaries on the same volume. The format is as follows:



INPUT PARAMETERS

inpath-list	One or more pathnames, separated by commas, of files or directories that are to be renamed. Blanks between pathnames are optional separators.
QUERY	Causes the Human Interface to prompt for permission to rename each pathname in the input list by issuing one of the following messages:

```
<oldname>, rename TO <newname>?
<oldname>, rename OVER <newname>?
```

Enter one of the following (followed by a carriage return) in response to the query:

Entry	Action
Y or y	Rename the file.
E or e	Exit from RENAME command.
R or r	Continue renaming without further query.
Any other character	Do not rename file; query for the next file in sequence.

OUTPUT PARAMETERS

TO	Moves the data to the new pathnames in the output list. A new pathname in the output list should not already exist. If the output pathname already exists, RENAME displays the following message:

<pathname>, already exists, DELETE?

	Enter Y, y, R, or r to delete the existing file. Enter any other character if you do not wish tp delete the file. In the later case, RENAME skips over the specified file without changing it and attempts to rename the next pathname in the list.
OVER	Changes each old pathname in a list to the corresponding new pathname, even if the new pathname already exists. OVER cannot be used to rename a non-empty directory over another non-empty directory.
outpath-list	List of new pathnames. Multiple pathnames must be separated by commas. Separating blanks are optional.

DESCRIPTION

The primary distinction between the RENAME command and the COPY command is that, as the RENAME command runs, it releases the pathnames of the input files for new uses without performing any further operation on the files.

Another distinction between RENAME and COPY is that RENAME cannot be used across volume boundaries; that is, you cannot use the RENAME command to rename a file or move data from a volume located on one secondary storage device to a volume located on another secondary storage device (for example, from one diskette to another). An attempt to do so causes an error message. Use the COPY command or a combination of COPY and DELETE commands if you wish to rename files or move data across volume boundaries.

To use RENAME, you must have delete access to the current file and add-entry access to the destination directory. If you rename a file OVER an existing file, you must also have delete access to the second file.

Although RENAME can be used to rename an existing directory pathname TO a new pathname, it cannot be used to rename an existing directory OVER another existing directory. For example:

```
-RENAME ALPHA TO DELTA         ;allowed
-RENAME ALPHA OVER BETA        ;not allowed (unless BETA is empty)
-RENAME ALPHA/SAMP1 OVER BETA/TEST1       ;allowed
```

NOTE

Changing the name of a directory also changes the path of all files listed in that directory. All subsequent accesses to those files must specify the new pathnames for the files.

As each file in a pathname list is renamed, the RENAME command displays one of the following messages, as appropriate:

```
<old pathname>, renamed TO <new pathname>
                     or
<old pathname>, renamed OVER <new pathname>
```

ERROR MESSAGES

- <old pathname>, DELETE access required

  You cannot rename a file unless you have delete access to that file.

- <new pathname>, directory ADD ENTRY access required

  You cannot rename a file unless you have add-entry access to the destination directory.

- <new pathname>, new pathname same as old pathname

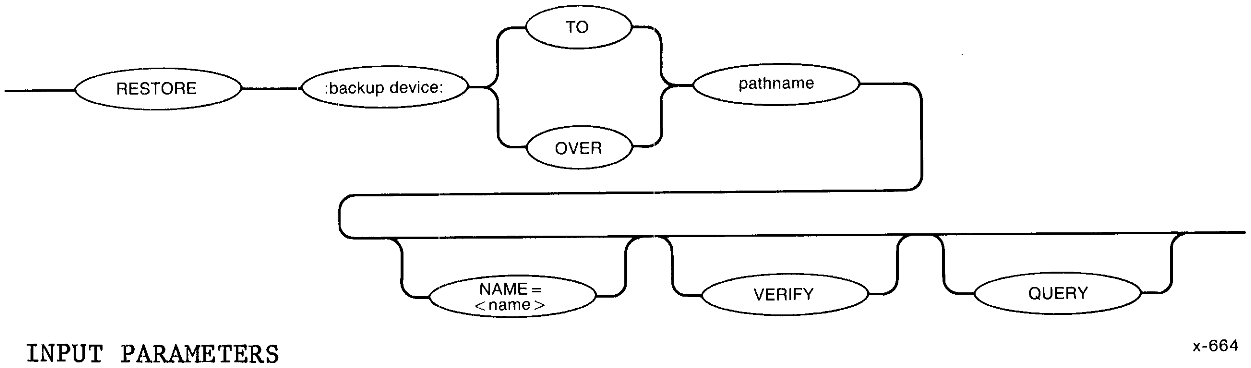
  You specified the same name for the input pathname as you did for the output pathname.

- TO or OVER preposition expected

  Either you used the AFTER preposition with the RENAME command or the number of files in your inpath-list did not match the number in your outpath-list.

RESTORE

This command transfers files from a backup volume to a named volume. The format of this command is as follows:



INPUT PARAMETERS

:backup device:	Logical name of the backup device from which RESTORE retrieves files.
QUERY	Causes the Human Interface to prompt for permission to restore each file. The Human Interface prompts with one of the following queries:

<pathname>, RESTORE data file?

or

<pathname>, RESTORE directory?

Enter one of the following responses to the query:

Entry	Action
Y or y	Restore the file.
E or e	Exit from the RESTORE command.
R or r	Continue restoring files without further query.
Any other character	If data file, do not restore the file; if directory file, do not restore the directory or any file in that portion of the directory tree. Query for the next file, if any.

VERIFY	Verifies that RESTORE has produced a restorable set of volumes. RESTORE produces one of the following messages: <pathname>, Restored or <dir>, Directory Restored
NAME=name	Begins restoration from a specific volume. If no name is given, RESTORE only writes back the first volume it encounters.

OUTPUT PARAMETERS

TO

Restores the files from the backup volume to new files on the named volume, if the files do not already exist. If a file being restored already exists on the named volume, RESTORE displays the following message:

```
<pathname>, already exists, OVERWRITE?
```

Enter one of the following in response to the query:

Entry	Action
Y, y, R, or r	Delete the file and replace it with the one from the backup volume.
E or e	Exit from the RESTORE command.
Any other character	Do not restore the file; go on to the next file.

OVER

Restores the files from the backup volume over (replaces) the files on the named volume. If a file does not exist on the named volume, RESTORE creates a new file on the named volume. When you specify the OVER preposition, RESTORE does not prompt you for permission to overwrite existing files.

pathname

Pathname of a file which receives the restored files (you must specify a directory pathname when restoring more than one file). If you specify a logical name for a device, RESTORE places the files under the root directory for that device. However, the device must contain a volume formatted as a named volume. If you wish to restore files to the directory in which they originated, you should specify the same pathname parameter as you used with the BACKUP command.

DESCRIPTION

RESTORE is a utility which copies files from backup volumes (where the BACKUP command originally saved them) to named volumes. RESTORE copies the files to any directory you specify, maintaining the hierarchical relationships between the backed-up files. RESTORE allows the transfer operation to begin at any physical or logical volume in a backup volume set.

Normally, when RESTORE copies files, it copies only those files to which you have access. When it copies these files to the named volume, it establishes your user ID as the owner ID (regardless of what the previous owner ID was). However, if you are the system manager (user ID 0), RESTORE restores all files from the backup volume and leaves the owner ID the same as it was.

When copying files, RESTORE reconstructs the following information:

- File name
- Access list
- Extension data
- File granularity
- Contents of the file

Each backup volume which is used as input to the RESTORE command must contain files placed there by the BACKUP command. In addition, if the backup operation required multiple backup volumes, you must restore these volumes in the same order as they were backed up.

The output volume which receives the restored files must be a named volume. You must have sufficient access rights to the files in that volume to allow RESTORE to perform all necessary operations. For RESTORE to create new files on a named volume, you must have add entry access to directories on that volume. For RESTORE to restore files over existing files, you must have add entry and change entry access to the files in that volume and delete, append, and update access to data files.

When you enter the RESTORE command, RESTORE displays the following sign-on message:

```
iRMX 86 DISK RESTORE UTILITY Vx.y
Copyright <year> Intel Corporation
```

where Vx.y is the version number of the utility. Then the command prompts you for a backup volume.

Whenever RESTORE requires a new backup volume, it issues the following message:

```
<backup device>, mount backup volume #<nn>, enter Y to continue:
```

where <backup device> indicates the logical name of the backup device and <nn> the number of the requested volume. (RESTORE in some cases displays additional information to indicate problems with the current volume.) In response to this message, place the backup volume in the backup device (make sure that the volume number is correct if the backup operation involved multiple volumes). Enter one of the following:

Entry	Action
Y, y, R, or r	Continue the restore process.
E or e	Exit from the RESTORE command.
N or n	Reprompt for a new volume.
Any other character	Invalid entry; reprompt for entry.

RESTORE continues prompting you until you supply the correct backup volume If a data file with the same pathname already exists when you use the TO proposition, RESTORE displays the following message:

```
pathname  , already exists, OVERWRITE?
```

Enter one of the following prompts:

Entry	Action
Y, y, R, or r	Continue the restore process.
E or e	Exit from the RESTORE command.
N or n	Reprompt for a new volume.
Any other character	Invalid entry; reprompt for entry.

As it restores each file, RESTORE displays one of the following messages at the Human Interface console output device (:CO:):

```
<pathname>, restored/verified
```

or

```
<pathname>, directory restored
```

If a "not restored" prompt is displayed, then a more detailed error message is printed.

ERROR MESSAGES

- <pathname>, access to directory or file denied

  RESTORE could not restore a file, either because you did not have add entry access to the file's parent directory or because you did not have update access to the file. RESTORE continues with the next file.

- <backup device>, backup volume #<nn>, <date>, mounted
  <backup device>, backup volume #<nn>, <date>, required

  <backup device>, mount backup volume #<nn>, enter Y to continue:

  RESTORE cannot continue because the backup volume you supplied is not the one that RESTORE expected. Either you supplied a volume out of order or you supplied a volume from a different backup session. RESTORE reprompts for the correct backup volume.

- <backup device>, cannot attach volume
  <backup device>, <exception value> : <exception mnemonic>

  <backup device>, mount backup volume #<nn>, enter Y to continue:

  RESTORE cannot access the backup volume. This could be because there is no volume in the backup device or because of a hardware problem with the device. The second line of the message indicates the iRMX 86 exception code encountered. RESTORE continues to issue this message until you supply a volume that RESTORE can access.

- <pathname>, <exception value> : <exception mnemonic>, error during BACKUP, file not restored

  When the BACKUP utility saved files, it encountered an error when attempting to save the file indicated by this pathname. RESTORE is unable to restore this file. The message lists the iRMX 86 exception code encountered.

- <pathname>, <exception value> : <exception mnemonic>, error during BACKUP, restore incomplete

  When the BACKUP utility saved the files, it encountered an error when attempting to save the file indicated by this pathname. RESTORE restores as much of the file as possible to the named volume. The message lists the iRMX 86 exception code encountered.

- <backup device>, error reading backup volume
  <backup device>, <exception value> : <exception mnemonic>

  RESTORE tried to read the backup volume but encountered an error condition, possibly because of a faulty area on the volume. The second line of the message indicates the iRMX 86 exception code encountered.

- <pathname>, <exception value> : <exception mnemonic>, error writing output file, restore incomplete

  RESTORE encountered an error while writing a file to the named volume. This message lists the iRMX 86 exception code encountered. RESTORE writes as much of the file as possible to the named volume.

- <pathname>, extension data not restored, <nn> bytes required

  The amount of space available on the named volume for extension data is not sufficient to contain all the extension data associated with the specified file. The value <nn> indicates the number of bytes required to contain all the extension data. This message indicates that the named volume on which RESTORE is restoring files is formatted differently than the named volume which originally contained the files. To ensure that you restore all the extension data from the backup volume, you should restore the files to a volume formatted with an extension size set equal to the largest value reported in any message of this kind. Refer to the description of the FORMAT command for information about setting the extension size.

- <backup device>, invalid backup device

  The logical name you specified for the backup device was not a logical name for a device.

- <backup device>, not a backup volume

  <backup device>, mount backup volume #<nn>, enter Y to continue:

  The volume you supplied on the backup device was not a backup volume. RESTORE continues to issue this message until you supply a backup volume.

- <pathname>, not restored

  For some reason, RESTORE was unable to restore a file from the backup volume. RESTORE continues with the next file. Another message usually precedes this message to indicate the reason for not restoring the file.

- output specification missing

  You did not specify a pathname to indicate the destination of the restored files.

- <pathname>, READ access required

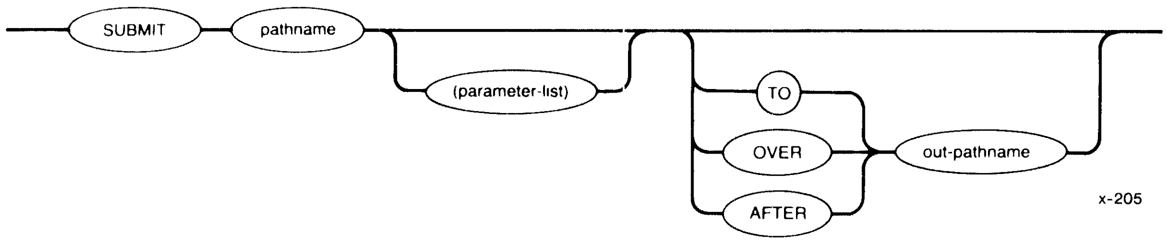
  You do not have read access to a file on the backup volume; therefore RESTORE cannot restore the file.

- <pathname>, too many input pathnames

  You attempted to enter a list of logical names for the backup devices. You can enter only one input logical name per invocation of RESTORE.

SUBMIT

This command reads and executes a set of commands from a file in secondary storage instead of from the console keyboard.



INPUT PARAMETERS

pathname	Name of the file from which the commands will be read. This file may contain nested SUBMIT commands.
parameter-list	Actual parameters that are to replace the formal parameters in the SUBMIT file. You must surround this parameter list with parentheses. You can specify as many as 10 parameters, separated by commas, in the SUBMIT command. If you omit a parameter, you must reserve its position by entering a comma. If a parameter contains a comma, space, or parenthesis, you must enclose the parameter in single quotes. The sum of all characters in the parameter list must not exceed 512 characters.

OUTPUT PARAMETERS

TO	Causes the output from each command in the SUBMIT file to be written to the specified new file instead of the console screen. If the output file already exists, the SUBMIT command displays the following message:

```
<pathname>, already exists OVERWRITE?
```

	Enter Y, y, R, or r if you wish the existing output file to be deleted. Enter any other character if you do not wish the existing file to be deleted. A response other than Y or y causes the SUBMIT command to be terminated and you will be prompted for a new command entry.
OVER	Causes the output for each command in the SUBMIT file to be written over the specified existing file instead of the console screen.
AFTER	Causes the output from each command in the SUBMIT file to be written to the end of an existing file instead of the console screen.
out-pathname	Pathname of the file to receive the processed output from each command executed from the SUBMIT file. If no preposition or output file is specified, TO :CO: is the default.
ECHO	ECHO causes the a copy of the data read from the first level of a SUBMIT file to be sent to the CRT. This parameter lets you know which action specified within a SUBMIT file is currently executing. Nested SUBMIT commands do not have their contents sent to the console.

DESCRIPTION

To use the SUBMIT command you must first create a data file that defines the command sequence and formal parameters (if any). The Operating System first looks for the pathname ending in "CSD". If no such file is found, then the Operating System looks for the specified file in the pathname.

Any program that reads its commands from the console input (:CI:) can be executed from a SUBMIT file. If another SUBMIT command is itself used in a SUBMIT file, it causes another SUBMIT file to be invoked. You can nest SUBMIT files to any level of nesting until memory is exhausted (each level of SUBMIT requires approximately 10K of dynamic memory). When one nested SUBMIT file completes execution, it returns control to the next higher level of SUBMIT file.

If, during the execution of SUBMIT (or any nested SUBMIT), you enter the CTRL/c character to abort processing, all SUBMIT processing exits and control returns to your user session.

When you create a SUBMIT file, you indicate formal parameters by specifying the characters %n, where n ranges from 0 through 9. When SUBMIT executes the file, it replaces the formal parameters with the actual parameters listed in the SUBMIT command (the first parameter replaces all instances of %0, the second parameter replaces all instances of %1, and so forth). If the actual parameter is surrounded by quotes, SUBMIT removes the quotes before performing the substitution. If there is no actual parameter that corresponds to a formal parameter, SUBMIT replaces the formal parameter with a null string.

When you specify a preposition and output file (other than :CO:) in a SUBMIT command, only your SUBMIT command entry will be echoed on the console screen; the individual command entries in the submit file are not displayed on the screen as they are loaded and executed.

The SUBMIT command will display the following message when all commands in the submit file have been executed:

```
END SUBMIT <pathname>
```

A SUBMIT can contain SUPER subcommands. But before you invoke a SUBMIT file with SUPER subcommands, you must first invoke the SUPER command. The following steps show you the general procedure for using the a SUBMIT command which contains SUPER directives:

- Invoke SUPER. Give the appropriate password.
- Invoke the SUBMIT file.
- SUBMIT functions are performed by the Operating System.
- SUBMIT command finishes.
- EXIT from SUPER.

ERROR MESSAGES

- <pathname>, end of file reached before end of command

  The last command in the input file was not specified completely. For example, the last line might contain a continuation character.

- <parameter>, incorrectly formed parameter

  You separated the individual parameters in the parameter list with a separator character other than a comma.

- <pathname>, output file same as input file

  You attempted to place the output from SUBMIT into the input file.

- <pathname>, too many input files

  You specified more than one pathname as input to SUBMIT. SUBMIT can process only one file per invocation.
- <parameter>, too many parameters

You specified more than 10 parameters in your parameter list.

- <pathname>, UPDATE or ADD access required

  SUBMIT cannot write its output to the output file because you do not have update access to the file (if it already exists) or because you do not have add access to the file's parent directory (if the file does not currently exist).

EXAMPLE

This example shows a SUBMIT file that uses formal parameters and the command that you can enter to invoke this SUBMIT file. The SUBMIT file, which resides on file :F1:MOVE$FILE, contains the following lines:

```
ATTACHDEVICE F1 AS %0
CREATEDIR %0/%1
UPCOPY :F1:%2  TO  %0%1/%2
```

The SUBMIT file contains three formal parameters, indicated by %0, %1, and %2. The %0 indicates the logical name of an iRMX 86 device; the %1 indicates the name of a directory on that device; the %2 indicates the name of a file which will be copied from an ISIS-II disk to the iRMX 86 device.

The SUBMIT command used to invoke this file is as follows:

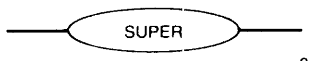
```
-SUBMIT :F0:MOVE$FILE (:F1:, PROG, FILE1)
```

The command sequence created and executed by SUBMIT is shown as it would be echoed on the console output device.

```
-ATTACHDEVICE F1 AS :F1:
F1, attached as :F1:
-CREATEDIR :F1:/PROG
:F1:PROG, directory created
-UPCOPY :F1:FILE1  TO  :F1:PROG/FILE1
:F1:FILE1 upcopied TO :F1:PROG/FILE1
END SUBMIT :F0:MOVE$FILE
-
```

SUPER

This command allows operators who are designated as system managers to change their user IDs to the system manager user ID (user ID 0). Having entered the SUPER command, these users can invoke a sub-command to change to any other user ID. The format of this command is as follows:



x-206

DESCRIPTION

SUPER allows you to change your user ID to that of the system manager. It has two sub-commands (CHANGEID and EXIT) that are available only after you have invoked SUPER. CHANGEID allows you to change your user ID to any possible value. EXIT exits the SUPER utility.

To invoke SUPER, you must know a password associated with the system manager. This password is stored in the user definition file for user ID 0 (refer to the iRMX 86 CONFIGURATION GUIDE for more information). After you enter the SUPER command, SUPER prompts for the password by displaying:

```
    ENTER PASSWORD:
```

You must then enter the correct password. (SUPER does not echo your input at the terminal.) After you enter the correct password, SUPER changes your user ID to user ID 0 and issues the following prompt.

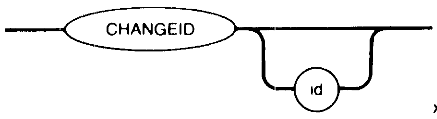
```
    super-
```

This prompt is a new system prompt (replacing the "-") that appears whenever the Human Interface is ready to accept input. At this point, you can enter any Human Interface commands and access any files available to the system manager. If you create new files, they will be listed as owned by user ID 0. You can also invoke the sub-commands available with SUPER.

SUBCOMMANDS

There are two sub-commands available with SUPER: CHANGEID and EXIT. You can invoke these sub-commands only after first invoking the SUPER command.

The CHANGEID sub-command allows you to change your current user ID to any value between 0 and 65535 decimal. The format of the CHANGEID sub-command is as follows:



x-207

where:

id	Value to which you want to change your user ID. This integer can be any numeric value from 0 to 65535 decimal, or the characters "WORLD" which specifies ID 65535 decimal. If you omit this value, CHANGEID sets your user ID to that of the system manager (user ID 0).

If you change your user ID to anything other than that of the system manager (user ID 0), the system prompt changes to the following:

```
super(id)-
```

where id is the decimal equivalent of your new user ID (or the characters "WORLD").

The EXIT sub-command exits from the SUPER utility. The format of this sub-command is as follows:

x-208

After you enter this sub-command, the Human Interface changes your user ID back to the ID you had before entering the SUPER command. It also changes the system prompt back to the "-" value. To change your user ID again, you must invoke the SUPER command.

ERROR MESSAGES

- <exception value> : <exception mnenonic> cannot set default user

  An internal system problem prevented the Human Interface from changing your user ID.

- <user-id>, invalid user id

  The user ID you specified contained invalid characters or was not in the range 0 to 65535 decimal.

- invalid password

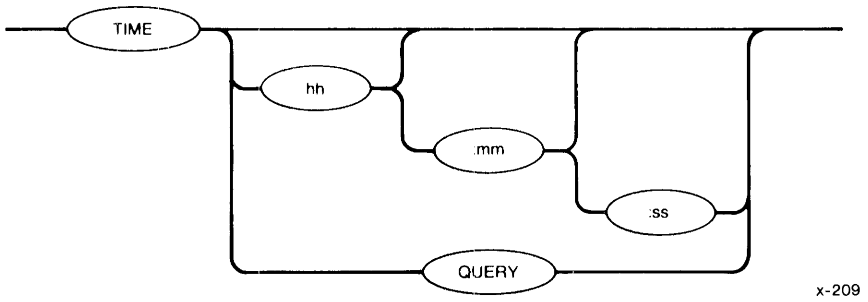
  The password you entered does not match the password associated with the system manager that is listed in the user definition file.

- <exception value> : <exception mnemonic>, SUPER is unavailable

  The Human Interface encountered an error while reading the password you entered or while accessing the system manager's user definition file (to determine if the password is correct). This message lists the exception code that occurs.

TIME

This command sets the system clock. If no new time is entered, the TIME displays the current system time. The format is as follows:



INPUT PARAMETERS

hh	Hours specified as 0 through 24.
mm	Minutes specified as 0 through 59. If you omit this parameter, 0 is assumed.
ss	Seconds specified as 0 through 59. If you omit this parameter, 0 is assumed.
QUERY	Causes TIME to prompt you for the time by issuing the following message: TIME: TIME continues to issue this message until you enter a valid time.

DESCRIPTION

You must separate the individual time parameters with colons.

If you omit the time parameters, TIME displays the current date and time in the following format:

```
dd mmm yy, hh:mm:ss
```

where dd mmm yy indicates the date and hh:mm:ss indicates the time.

To obtain the correct time when you enter the TIME command without parameters, you must initially set the time. If you request the time on a system in which you haven't already set the time, TIME command displays the last time (and date) you accessed the :SYSTEM: directory.

ERROR MESSAGES

- <time>, invalid time

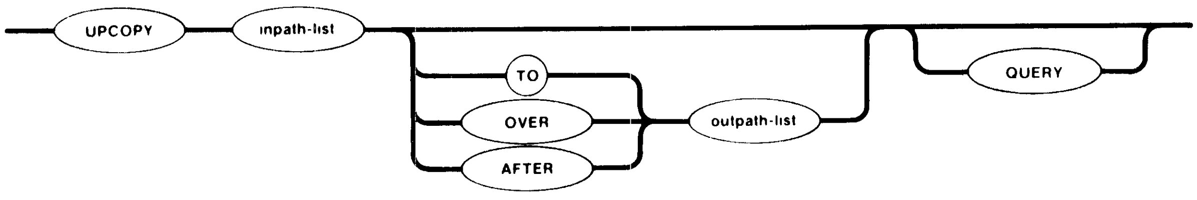
  You specified an invalid or out-of-range entry for one or more of the time parameters.

- <parameter>, invalid syntax

  You specified both a time and the QUERY parameter in the TIME command.

UPCOPY

This command copies files from a volume on ISIS-II secondary storage to a volume on iRMX 86 secondary storage.



x-323

INPUT PARAMETERS

inpath-list	List of one or more filenames of the ISIS-II files that are to be copied to iRMX 86 secondary storage, either on a one-for-one basis or concatenated into one or more iRMX 86 output files.
QUERY	Causes the Human Interface to prompt for permission to copy each ISIS-II file to the listed iRMX 86 output file. Depending on which preposition you specify (TO, OVER, or AFTER), the Human Interface prompts with one of the following queries:

```
<in-pathname>, copy up TO <out-pathname>?

<in-pathname>, copy up OVER <out-pathname>?

<in-pathname>, copy up AFTER <out-pathname>?
```

Enter one of the following (followed by a carriage return) in response to the query:

Entry	Action
Y or y	Copy the file.
E or e	Exit from the UPCOPY command.
R or r	Continue copying files without further query.
Any other character	Do not copy this file; go to the next file in sequence.

OUTPUT PARAMETERS

TO	Copies the ISIS-II file or files TO a new iRMX 86 file or files in the listed sequence. If the output file already exists, UPCOPY displays the the following message: `<pathname>, already exists, OVERWRITE?` Enter Y, y, R, or r if you wish to write over the existing file. Enter any other character if you do not wish the file to be overwritten. If no preposition is specified, TO :CO: is the default. If more input files than output files are specified in the command line, the remaining input files will be appended to the end of the last listed output file.
OVER	Copies the listed ISIS-II input file or files OVER existing iRMX 86 destination files in the listed sequence. If more input files than output files are listed in the command line, the remaining input files will be appended to the end of the last listed output file.
AFTER	Appends the listed ISIS-II input file or files AFTER the end-of-data on an existing iRMX 86 output file or files in the listed sequence.
outpath-list	One or more pathnames of the iRMX 86 destination files. Multiple pathnames must be separated by commas. Separating blanks are optional. If the preposition and output parameter defaults are used in the command line, the output will go to the iRMX 86 console screen.

DESCRIPTION

Before you enter an UPCOPY command on the iRMX 86 console keyboard, you must have your target system connected to a development system via the monitor. To do this, you must start your iRMX 86 system from the development system terminal (either by loading the software into the target system and using the monitor G command to start execution, or by using the monitor B command to bootstrap load the software). UPCOPY does not function if you start up your system from the iRMX 86 terminal or if you establish the link between development system and target system after starting up your iRMX 86 system.

The user ID of the user who invokes the UPCOPY command is considered the owner of new files created by UPCOPY. Only the owner can change the access rights associated with the file (refer to the PERMIT command).

As it copies each ISIS-II file in the input list, UPCOPY displays one of the following messages at the terminal, as appropriate:

```
<in-pathname>, copied up TO <out-pathname>

<in-pathname>, copied up OVER <out-pathname>

<in-pathname>, copied up AFTER <out-pathname>
```

When the UPCOPY command is executing, the monitor disables interrupts. This action affects services such as the time-of-day clock. Also, the Operating System is unable to receive any characters that you type-ahead while the UPCOPY command is executing.

ERROR MESSAGES

- <pathname>, ISIS ERROR: <nnn>

  An ISIS-II Operating System error occurred when UPCOPY tried to transfer the file to the Microcomputer Development System. Refer to the INTELLEC SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM CONSOLE OPERATING INSTRUCTIONS for a description of the resulting error code.

- ISIS link not present

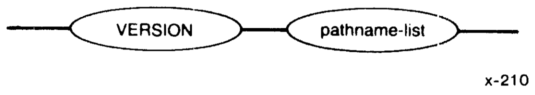
  The the iRMX 86 system is not connected to the development system via the monitor.

- <pathname>, UPDATE or ADD access required

  Either you cannot overwrite the information in a file because you do not have update access to it, or you cannot copy information to a new file because you do not have add entry access to the file's parent directory.

VERSION

This command displays the version number of a file if that file has a version number. The file can be an object file or library. The format of this command is as follows:



INPUT PARAMETER

pathname-list	One or more pathnames, separated by commas, of commands for which a version number is desired.

DESCRIPTION

When you enter the VERSION command, it displays the version number of each file, if there is one, in the following format:

```
<pathname>, <command-name> version is x.y
```

where:

<pathname>	Pathname of the file containing the command.
<command-name>	Name of the specified command; Intel-supplied commands have names as listed in this manual.
x.y	Version number of the command.

You can use VERSION to determine the version number of any Human Interface command. You can also use it to determine the version numbers of commands that you write. If the file is a library, the command shows the current and previous version numbers. However, for VERSION to work on your commands, you must include a literal string in the command's source code to specify the name of the command and its version. The string must contain the following information:

```
'program_version_number=xxxx',
'program_name=yyyy...yyy',0
```

where:

program_version_number=	You must specify this portion exactly as shown (lower case, underscore separating the words, no spaces).

xxxx	Version number of the product. This can be any four characters, but it must be exactly four characters long.
program_name=	This portion is optional. However, if you want VERSION to recognize and display the program name, you must specify this portion exactly as shown.
yyyy...yyy	Name of the command. This name can be any number of characters.
0	The literal string must be terminated with a byte of binary zero.

An example of such a literal string is:

```
DECLARE version (*)  BYTE  DATA('program_version_number=V8.5',
                                'program_name=MYPROGRAM',0);
```

If your program includes this declaration, when you invoke VERSION, it will display the following information:

```
<pathname>, MYPROG version is V8.5
```

A literal string that does not include the program name is:

```
DECLARE vers2(*)    BYTE  DATA('program_version_number=1983',0);
```

If your program includes this declaration, when you invoke VERSION, it will display the following information:

```
<pathname>, version is 1983
```

ERROR MESSAGES

- <pathname>, does not contain a program version number.
- <pathname>, is not an object module.

WHOAMI

This command lists the currents user's identification and access rights.

x-666

DESCRIPTION

This example shows the output from WHOAMI:

```
--WHOAMI
USER ID: 5
ACCESS ID'S: 5, WORLD
```

The number after USER is the user's ID number. The numbers after ACCESS are the ID'S of people who have given the user access to their files. See the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL for more information on USER groups.

***

CHAPTER 4
HUMAN INTERFACE EXAMPLES

This chapter shows you how to use some of the Human Interface commands. Its primary intent is to introduce you to basic techniques by presenting a series of examples that illustrate typical command entries.

COMMAND EXAMPLES FORMAT

To make it easier to follow the interactive dialog between the user and Human Interface in the examples, the user keyboard entries are underscored. All other items displayed in the examples are Human Interface command output. For instance, in the example:

```
-copy samp to test
samp copied TO test
-copy test
aaaaa
bbbbb
test copied TO :CO:
-
```

the underscored items are user command entries; all other characters and lines are output by the Human Interface or the supplied commands.

Control characters, such as (CTRL/z), are enclosed in parentheses in the examples to indicate that such entries are not echoed at the console screen as they are entered. Do not actually enclose control key entries in parentheses.

HOW TO BEGIN A CONSOLE SESSION

You can begin an interactive dialog with the Human Interface after the initial program displays a sign-on message at your console screen. Although the sign-on message is a system configuration option, the message supplied with the default initial program of the Human Interface is as follows:

```
iRMX 86 HI CLI Vx.y: USER = <userid>
COPYRIGHT <year> Intel Corporation
-
```

This message tells you the Human Interface is running; it also tells you your user ID. The hyphen (-) is a Human Interface prompt to indicate that it is ready to accept your first command line. Begin entering a command immediately after and on the same line as the prompt. For example:

```
-copy :ci: to test1
```

HOW TO CREATE A SIMPLE DATA FILE

You can use the COPY command to create data files during a console session. Assume you wish to create a file called ALPHA and write two lines of data into the file. Also assume you wish the data file to be listed under your default directory. Enter the following command and data:

```
-copy :ci: to alpha
aaaaa
bbbbb
cr (carriage return)
(CTRL/z)

:ci: copied TO alpha
-
```

In this example, the :ci: in the COPY command line tells the command to read data from the keyboard (:ci: = console input) and write the data (aaaaa and bbbbb) to a new file named ALPHA. Because you did not preface the file name with a directory name, COPY places the file ALPHA in your default directory.

The command does not prompt you for the data lines; you simply begin entering data after you press RETURN at the end of the command line. Your CTRL/z entry writes an end-of-file mark at the end of your data to inform the COPY command that there is no more data to be copied.

Note that after you enter the last line of data, you must press the RETURN key before you enter a CTRL/z to insert an end-of-file. Otherwise, none of the characters entered after you press the RETURN key and before you enter a CTRL/z are written to the file. For example:

```
-copy :ci: to alpha
ccccc
ddddd (CTRL/z) (then press RETURN)
```

would only write the data ccccc to the new file named ALPHA.

Since control characters are not echoed on the screen as you enter them, (such as a RETURN or CTRL function), the above file creation sequence would be displayed on the screen as follows:

```
-copy :ci: to alpha
ccccc
ddddd


:ci: copied TO alpha
-
```

Now, assume that when you entered the COPY command line, the Human Interface sent you the following message and query:

```
-copy :ci: to alpha
alpha, already exists, OVERWRITE?
```

Whenever you create a new data file, the COPY command expects a new pathname rather than one already listed in the directory file. If your entry to the query is:

```
    alpha, already exists, OVERWRITE? y
```

the COPY command deletes the data in the existing file and waits for you to enter new data under that pathname.

If your response to the query is:

```
    alpha, already exists, OVERWRITE? n   (or any other character except y)
```

your COPY command is ignored and the Human Interface prompts for a new command entry by issuing a hyphen (-).

HOW TO COPY FILES

COPY command options provide a number of different ways for you to copy existing files. You exercise these options either by specifying one of the TO/OVER/AFTER prepositions, by the way in which you specify your input file and output file pathname lists, or by a combination of both techniques. The services of the COPY command include:

- Copying files on a one-for-one basis.
- Displaying the contents of files at the console screen.
- Creating multiple copies of the same file.
- Copying data from multiple files to a new or existing file.
- Replacing data in one file with data from another file.
- Adding data from one or more files to the end of the data in another file.
- Combining one-for-one file copying with file concatenation in a single COPY command.

The examples that follow show you how to use these services. They also call your attention to certain file handling considerations when using the COPY command.

HOW TO COPY TO NEW FILES

Copying existing files to new files is most frequently done on a one-for-one basis; that is, you list a number of existing files to be copied and a matching list of files to receive the copies. The files are copied in the same sequence you specify in the input list and output list on the command line. For example, assume you wished to copy files A1, A2, and A3 to files B1, B2, and B3 respectively. Enter the following command:

```
-copy a1, a2, a3 to b1, b2, b3
a1 copied TO b1
a2 copied TO b2
a3 copied TO b3
-
```

You could also make use of the wild card feature when copying files. If the files A1, A2, and A3 are the only files in the directory that begin with the character "A", you can use the following command to perform the same operation:

```
-copy a* to b*
a1 copied TO b1
a2 copied TO b2
a3 copied TO b3
-
```

The asterisks in the command are the wild card characters. In this instance, the command copies all files in the default directory that start with the character "A" to new files starting with the character "B." If files other than A1, A2, and A3 also begin with the character "A", this command will copy them also.

When you copy files, you can specify wild card characters (as in the previous example), lists of file names (as in the example before that), or a combination of both. However, some of the possible combinations are invalid. When copying files, remember the following rules:

- If you specify multiple input pathnames and a single output pathname, file concatenation takes place. If the output parameter is simply a directory with no wild card in its pathname, then the Human Interface copies all the files listed in the input parameter into the directory. Each file keeps its original name in the new directory (such as alpha).

- If you specify multiple output pathnames, you must specify the same number of input pathnames as output pathnames. Specifying more input pathnames than output pathnames results in an error message. For example, the command:

  ```
  -copy a,b,c to d,e    (invalid)
  ```

  returns an error message. The command:

  ```
  -copy a,b to c,d,e    (invalid)
  ```

  also returns an error message. Refer to the "Inpath-List and Outpath-List" section of Chapter 2 for more information.

HOW TO DISPLAY THE CONTENTS OF FILES

When you perform a number of file manipulations during a single session, it is occasionally advisable to display a file's contents at the terminal before proceeding further. Assume you wish to display the contents of a file named ALPHA that is contained in your default directory. Simply enter the command:

```
    -copy alpha
    aaaaa
    aaaaa

    alpha copied TO :CO:
    -
```

This COPY command example uses the default preposition (TO) and default output file (:CO:), which means that the command copies the output to the console screen.

You can halt the scrolling of a displayed list to examine the data more closely. Press the following CTRL keys to control scrolling of the output:

CTRL/s	Stops the data from scrolling off the screen until you press a CNTRL/q.
CTRL/q	Resumes scrolling of listed data until the end-of-file is reached or you enter a CTRL/c.
CTRL/c	Cancels listing the data and returns control to the Human Interface, which prompts for a new command.

HOW TO REPLACE EXISTING FILES

There may be occasions when you wish to update the contents of an existing file. One way to do this updating is to create a new file and then replace the contents of the old file with the new data. Although you can use the RENAME command to perform this operation, this section shows how to replace the contents of a file with the COPY command's OVER preposition.

Assume the following conditions:

- You have a file named ALPHA that is accessed under that name by a number of different programs. ALPHA has outmoded data.
- Since you cannot change the name without also modifying the programs that access ALPHA, you must retain the name but update the outmoded file contents.

Enter the following command sequence:

```
    -copy :ci: to temp
    nu nu nu nu
    nu nu nu nu
    (CTRL/z)

    :ci: copied TO temp
    -copy temp over alpha
    temp copied OVER alpha
    -copy alpha
    nu nu nu nu
    nu nu nu nu

    alpha copied TO :CO:
    -
```

The last COPY ALPHA command lists the file at the terminal to show that the old file contents have been successfully replaced.

You could have used the TO preposition in the COPY command to write TEMP over ALPHA; but since the Human Interface always expects a new output file when the TO preposition is used, this would have caused unnecessary keystrokes, as shown in the following:

```
    -copy temp to alpha
    alpha, already exists, OVERWRITE? y
    temp copied TO alpha
    -
```

Note that you now have two copies of the same new data; one in the TEMP file and one in the ALPHA file. If you had used the OVER preposition in a RENAME command instead of the COPY command, file TEMP would have been deleted automatically when RENAME was executed. However, if you did not want two existing copies of the same data, you could update the existing file directly from the keyboard. Enter the following command:

```
    -copy :ci: over alpha
    newnewnew
    (CTRL/z)
    :ci: copied OVER alpha
    -
```

HOW TO CONCATENATE FILES

Concatenation is the process of combining a number of files by appending them in sequence into a single file. You can use the COPY command in several ways to concatenate files:

- by specifying the AFTER preposition in the command line
- by specifying multiple input pathnames and a single output pathname (if the output pathname is a directory, concatenation does not occur)

- by using a combination of both techniques

Assume you have four existing files named A, B, C, D respectively, and want to append the contents of B, C, and D to the end of file A. Although you could specify the TO preposition in the COPY command line, the TO preposition would force you to enter extra keystrokes because your listed output file (A) already exists. It would also force you to delete the previous contents of A, which is not always desirable. Therefore, use the AFTER preposition, as follows:

```
-copy b,c,d after a
b copied AFTER a
c copied AFTER a
d copied AFTER a
-
```

Now, assume you wish to concatenate all four files into a new file called ALL. You can still use the AFTER preposition, or you can use the TO parameter, as follows:

```
-copy a,b,c,d to all
a copied TO all
b copied AFTER all
c copied AFTER all
d copied AFTER all
-
```

In this example, file A is copied to ALL and the remaining input files are automatically appended to the end of ALL.

You can save keystrokes when listing a series of files on the screen by using this automatic concatenation in a single command line. Assume you wish to list files named ALPHA, BETA, and GAMMA. Enter the following command, using the default TO preposition and default output file (:CO:):

```
-copy alpha,beta,gamma
aaaaa
aaaaa
alpha copied TO :CO:
bbbbb
bbbbb
beta copied AFTER :CO:
ggggg
ggggg
gamma copied AFTER :CO:
-
```

When data sequence and/or data format are important in a concatenated file, remember that all copy operations are performed in the sequence you specify in the command line.

Assume you have formatted data in a group of files named A, B, C, D, and E, and you wish to concatenate their contents into a new file named SQUARE in that sequence. However, if you list the input files on the command line in a haphazard sequence, as follows:

```
-copy b,a,d,c,e to square
```

the format of the total data block is destroyed, as can be seen in the following incorrect and correct versions of the listed output. Although the data block of Latin words shown in the left-hand example seems correct when read horizontally, the intent and meaning of the vertical columns has been lost. The right-hand example shows the corrected file sequence:

```
b,a,d,c,e                           a,b,c,d,e
sequence                            sequence

A R E P O                           S A T O R
S A T O R                           A R E P O
O P E R A                           T E N E T
T E N E T                           O P E R A
R O T A S                           R O T A S
```

In the right-hand example, the Latin "magic square" now reads the same both horizontally and vertically, which was the intended operation.

HOW TO DELETE FILES

It is vital to good file housekeeping that you routinely delete obsolete or unused files and empty directories. (Deleting unused directories is described later in this chapter.) In addition to the obvious benefit of recovering unused secondary storage, deleting your obsolete files reduces confusion and file manipulation errors.

Assume that you want to delete files ALPHA and BETA from the system. Enter the following command:

```
-delete alpha,beta
alpha, deleted
beta, deleted
-
```

Now, assume that you entered the following command line and received the following error message:

```
-delete ay,bee,key
ay, deleted
bee, deleted
key, does not exist
-
```

The error message for the KEY file tells you one of three things:

- There is a syntax error in the spelling of the KEY file.
- The file does not exist.
- The file exists in a directory other than the one you are currently accessing (see the directory examples later in this chapter).

HOW TO USE DIRECTORIES

A directory is a kind of file under which you assign and maintain other files or directories. It is distinguished from a data file by a directory heading that is automatically created when you create a new directory. Under that heading, the directory maintains a formatted list of its containing files and directories. This heading is updated whenever you assign new files to the directory. Directories provide you with a convenient and efficient technique for organizing large numbers of files into logical groupings. Creating your own directories aids you in two ways:

- It allows you to organize your files into logical groupings. This capability eases the task of maintaining large numbers of files on the system.
- It reduces the possibility of accidental destruction of files, either by yourself or other system users.

A directory contains a list of all files assigned under its name, which you can display by using the DIR command (described later). Optional DIR command parameters also allow you to access and display other pertinent information about each file, such as file size and other file attributes.

Previous command examples in this chapter, when creating and accessing files, have used the default directory configured for your user ID. The following examples show you how to create and use your own directories for easier file management.

HOW TO CREATE A NEW DIRECTORY

Whenever you wish to group a series of files under a single topical structure, you normally create a new directory in which to assign them before creating the files themselves. (You can also move existing files under a new directory name by using the RENAME command, as described later.)

You create new directories by using the CREATEDIR command to specify a list of directory names for the new directories. You will find it easier to keep track of both your directories and files if you use directory names that give some hint of a directory's topical structure.

Assume you wish to create two directories named MYTEST and NUTEST under which you will assign several practice files. Enter the following command:

```
-createdir MYTEST,NUTEST
MYTEST, directory created
NUTEST, directory created
-
```

This example specified the directory pathnames as uppercase characters. It is suggested that you also capitalize all directory pathnames in a CREATEDIR command and use lowercase characters for data pathnames when you create new files with the COPY command. This practice is recommended because, when you subsequently list a directory by using the DIR command (described later), it will be much easier for you to distinguish between data file names and directory names.

Once you create directories and data files, you can enter their pathnames in either lowercase or uppercase characters in subsequent commands; the Human Interface commands make no distinction in interpretation.

HOW TO REFER TO A DIRECTORY

After you create a new directory, all named files or directories that you assign to that directory will have a hierarchical relationship to this "parent" directory. This relationship to the parent is called a path. When you wish to access any file or other directory assigned to the parent, you must specifically identify the path in the form of a pathname in your command.

For example, assume your default directory has a directory named NUTEST under which you have another directory named SAMP. SAMP, in turn, has a data file named TEST. NUTEST is then the parent directory for the SAMP directory and SAMP, in turn, is the parent for the TEST data file. In a command, the pathname for the SAMP directory would be NUTEST/SAMP, where the slash characters separate the individual hierarchical components of the pathname. The pathname for the TEST data file would be NUTEST/SAMP/TEST.

If the files are contained in your default directory, you can refer to them without specifying a logical name as a prefix. When you enter the pathname:

```
    NUTEST/SAMP/TEST
```

the Human Interface automatically appends the prefix :$: to the beginning. However, if the files are contained in a directory other than your default directory, you must enter the complete pathname for the file. For example, if the files reside on a device whose logical name is :AD3:, you must include this logical name as the prefix portion of the pathname, as follows:

```
    :AD3:NUTEST/SAMP/TEST
```

If you omit the :AD3: portion, the Human Interface assumes the files reside in the default directory.

HOW TO ADD NEW ENTRIES TO A DIRECTORY

Previous data file examples in this chapter used the default directory (as configured for your system) for all file creation and access. Consequently, each example that created a new file or accessed an existing file specified only the last component of the file's pathname; it did not need to specify a logical name or intermediate pathname components. However, whenever you wish to create a new data file to be assigned to a specific directory, you must precede the filename with the directory name and separate the two names with a slash (/) in the COPY command, as described in the previous subsection. You might also need to specify a logical name, if the directories do not reside in your default directory.

For example, assume you wish to create files named SAMP1 and SAMP2 and assign them to the MYTEST directory (MYTEST resides in your default directory). Enter the following commands:

```
    -copy :ci: to mytest/samp1
    aaaaa
    (CTRL/z)

    :ci: copied TO mytest/samp1
    -copy :ci: to mytest/samp2
    bbbbb
    (CTRL/z)

    :ci: copied TO mytest/samp2
    -
```

Remember that once you have added files to a specific directory, every subsequent operation involving those files must specify a preceding directory name and the slash separator (unless you change your default directory, as described in a later section). For example, assume you want to delete files SAMP1 and SAMP2 from the MYTEST directory. You might enter the following command:

```
    -delete mytest/samp1,samp2
    mytest/samp1, deleted
    samp2, does not exist
    -
```

The Human Interface issues the "does not exist" message for SAMP2 because it looked for the file in your default directory instead of the MYTEST directory. The correct command line entry should have been:

```
    -delete mytest/samp1,mytest/samp2
```

so that the Human Interface would search the correct directory for each listed file.

HOW TO CREATE A DIRECTORY WITHIN A DIRECTORY

In the same manner that you create new directories in your default directory, you can also create new directories in other directories, therefore expanding the file hierarchy.

For example, assume you have data files ALPHA, BETA, and GAMMA assigned to the MYTEST directory and now wish to add a new directory file named URTEST to the directory. Enter a CREATEDIR command, as follows:

```
-createdir mytest/URTEST
mytest/URTEST, directory created
-
```

Now, assume you wish to create a new data file named NOMOR and assign it to the URTEST directory. Enter the following COPY command:

```
-copy :ci: to mytest/urtest/nomor
nononon
nononon
(CONTROL/z)

:ci: copied TO mytest/urtest/nomor
-
```

The "MYTEST/URTEST" sequence is the path from your default directory to the URTEST directory, and the "MYTEST/URTEST/NOMOR" sequence is the path from your default directory to the NOMOR file. When you use file-handling commands, you must always specify a path to the file, either a path from your default directory to the file, or a path from some other known point (such as from the root directory for another device). For example, assume you have another data file in URTEST named SUMOR and wish to list both NOMOR and SUMOR on the console screen. Enter the following command and specify the pathname for each file:

```
-copy mytest/urtest/nomor,mytest/urtest/sumor
nononon
nononon
mytest/urtest/nomor copied TO :CO:
sumsumsum
sumsumsum
mytest/urtest/sumor copied TO :CO:
-
```

If the directory MYTEST resides on a device (for example, :F6:) other than your default device, you would specify the previous command as follows:

```
-copy :f6:mytest/urtest/nomor,:f6:mytest/urtest/sumor
nononon
nononon
:f6:mytest/urtest/nomor copied TO :CO:
sumsumsum
sumsumsum
:f6:mytest/urtest/sumor copied TO :CO:
-
```

You can also specify file operations involving two or more different directories, and these directories need not be on the same path. Assume you wish to list the ALPHA file from MYEST and a file named DIFF on a directory path ONE/MOR. Enter the following command:

```
-copy mytest/alpha,one/mor/diff
aaaaa
aaaaa
mytest/alpha copied TO :CO:
yyyyy
yyyyy
one/more/diff copied TO :CO:
-
```

HOW TO LIST DIRECTORIES

Previous examples have shown you how to list the contents of data files by specifying a directory pathname in a COPY command. However, you should not use the COPY command to list the contents of directories, because COPY lists the directory as though it were a data file. For example, if you enter the COPY command to list the MYTEST directory on the screen, you obtain:

```
-copy mytest
alphabetagammaurtest copied to :CO:
-
```

The resulting output is almost unreadable. Instead, use the DIR command to list the directory's catalog of files as follows:

```
-dir mytest

01  JAN 78  00:00:00
  DIRECTORY OF MYTEST  ON VOLUME disk2
alpha          beta              gamma
URTEST

-
```

This example used the DIR command's default TO preposition and FAST format for the listing. You could have sent the directory listing to another output file and specified either the OVER preposition, to write the listing over the file's previous contents, or the AFTER preposition, to append the directory listing to other data. If you want to list more information about each file, specify the EXTENDED parameter. See the DIR description in Chapter 3 for examples of the available listing formats.

HOW TO MOVE FILES BETWEEN DIRECTORIES

There may be situations when you wish to reorganize a large group of existing files under new headings (directories). You can copy files from one directory to another by using the COPY command. For example, assume you wish to copy files ALPHA, BETA, and GAMMA from your default directory to the existing directory MYTEST. Enter the following command line, using the QUERY parameter (optional):

```
-COPY alpha,beta,gamma, to MYTEST QUERY
alpha, COPY TO MYTEST/alpha? y
alpha COPIED TO MYTEST/alpha
beta, COPY TO MYTEST/beta? y
beta COPIED TO MYTEST/beta
gamma, COPY TO MYTEST/gamma? y
gamma COPIED TO MYTEST/gamma
-
```

Assume you later decide to move file ALPHA back to your default directory. You need not specify the default directory in the new pathname for ALPHA. Enter the following command:

```
-rename mytest/alpha to alpha
mytest/alpha renamed TO alpha
-
```

Any subsequent operations involving file ALPHA would only require the file name. For example:

```
-copy alpha
aaaaa
aaaaa
alpha copied TO :CO:
-
```

HOW TO DELETE A DIRECTORY

You delete unused directories from secondary storage by using the DELETE command. However, the Human Interface protects you from accidently destroying valuable files by refusing to delete a directory that contains one or more files. For example, assume you wish to delete directory MYTEST and do not realize it contains a data file named ALPHA and a directory named DED that itself contains a data file named LIV. You enter the following command:

```
-delete mytest
mytest, directory not empty
-
```

At this point you should list the MYTEST directory by using the DIR command to determine the contents of MYTEST, as follows:

```
-dir mytest

01  JAN 78  00:00:00
  DIRECTORY OF MYTEST  ON VOLUME disk2
alpha         DED

-
```

You now have two options. You can use the RENAME command to move any files to be saved to a different directory on the same volume, or you can use the DELETE command to delete the entire contents of MYTEST before deleting the directory.

Assume you wish to move ALPHA to the NUTEST directory and delete the rest of the directory's contents so that MYTEST itself can be deleted. Enter the following commands:

```
-rename mytest/alpha to nutest/alpha
mytest/alpha renamed TO nutest/alpha
-delete mytest/ded/liv,mytest/ded,mytest
mytest/ded/liv deleted
mytest/ded deleted
mytest deleted
-
```

The RENAME command automatically deleted the MYTEST/ALPHA pathname from the MYTEST directory. Note how the pathname sequence in the DELETE command travelled upward through the hierarchical structure, with the MYTEST directory being the last item to be deleted.

HOW TO CHANGE YOUR DEFAULT DIRECTORY

Suppose your default directory contains a directory called MYTEST which contains another directory called URTEST which in turn contains several data files called MOR, SUMOR, STILMOR, and NOMOR. If you plan to manipulate these data files extensively, your Human Interface commands can become very cumbersome, due to the length of the pathnames involved. For example, suppose you wish to copy the data files to files called ALPHA, BETA, DELTA, and GAMMA in the same directory. The command to do this is:

```
-copy mytest/urtest/mor, mytest/urtest/sumor, mytest/urtest/stilmor, &
**mytest/urtest/nomor  to mytest/urtest/alpha, mytest/urtest/beta, &
**mytest/urtest/delta, mytest/urtest/gamma
```

If there are more levels in the directory structure, your commands can become even longer.

To eliminate some of these long pathnames, you can use the ATTACHFILE command to change your default directory to be a directory closer to the level of the files with which you are working. To make the previous command shorter, you could change your default directory to the URTEST directory, as follows:

```
-attachfile mytest/urtest as :$:
mytest/urtest attached AS :$:
-
```

Now, when you make references to files without specifying the entire pathname, the Human Interface assumes that they reside in the URTEST directory, not your previous default directory. Therefore, to perform the same operation as in the previous COPY command, you could now enter the following command:

```
-copy mor, sumor, stilmor, nomor  to  alpha, beta, delta, gamma
```

You can use the ATTACHFILE command to change your default directory to any directory that you wish, so that you can manipulate the files in that directory more easily. To return to your original default directory, enter the following command:

```
-attachfile
```

This command uses the default parameters and has the same effect as "ATTACHFILE :HOME: AS :$:". The :HOME: logical name represents your original default directory; therefore the command returns :$: to its original value.

HOW TO RENAME FILES AND DIRECTORIES

The most direct method to save the contents of a file or directory but change its pathname is to use the the RENAME command. To make the process easier to follow, this section discusses the renaming of files and directories separately.

HOW TO RENAME FILES

Assume you wish to change the name of file ALPHA to a new name of OMEGA, where OMEGA does not already exist. Enter the following command:

```
-rename alpha to omega
alpha renamed TO omega
-
```

The ALPHA pathname is automatically deleted from the system when the RENAME command is executed. You can also rename lists of files to new pathnames In this case, it is useful to include the QUERY parameter in your command line to make certain that your old pathnames and new pathnames are matched up in the way you intend.

Assume you wish to rename files ALPHA, BETA, and GAMMA to TOM, DICK, and HARRY respectively. Enter the following command sequence:

```
-rename alpha,beta,gamma to tom,dick,harry
alpha renamed TO tom
beta renamed TO dick
gamma renamed TO harry
-
```

Remember that when using the RENAME command, you must always have a one-for-one match of pathnames between the new list and the old file list. For example, more old pathnames than new pathnames would cause the following exchange at the terminal:

```
-rename alpha,beta to tom
alpha renamed TO tom
TO or OVER preposition expected
-
```

Similarly, specifying fewer old pathnames than new pathnames would cause the following exchange:

```
-rename alpha to beta,tom
008B: E$UNMATCHED_LISTS
-
```

So far, these RENAME examples have used the TO parameter to give new names to existing files. However, you can also use the OVER preposition with RENAME. The primary purpose of OVER is to move data from one named file over the data in another existing file. This use of the OVER preposition matches the action of the OVER preposition in the COPY command with one important distinction: RENAME automatically deletes the input file when the command is executed.

Exercise a little care here! It's easy to get into semantic confusion when using the OVER preposition in a RENAME command. Just remember a few simple rules:

- Use the pathname of the data to be moved to a different but existing pathname as the input parameter; that is, on the left-hand side of the OVER preposition. This pathname will be deleted when the command is executed.

- Use the pathname that receives the input data as the ouput parameter; that is, on the right-hand side of the OVER preposition. The previous contents of this file will be replaced when the command is executed.

For example, assume you have a file named ABLE whose contents consist of the data line aaaaa, and another file named BAKER whose contents consist of the data line bbbbb. You wish to rename ABLE with the name BAKER. Enter the following command:

```
-rename able over baker
able renamed OVER baker
-
```

Now display the contents of the file previously named ABLE but now named BAKER:

```
-copy baker
aaaaa

baker copied TO :CO:
-
```

The previous contents of BAKER have been deleted, and pathname ABLE has been deleted from its directory. You can also use the TO preposition to rename files with other existing pathnames. Using TO might be slightly less confusing but you must enter extra keystrokes. For example, assume you wish to rename ALPHA and BETA with the existing file names GAMMA and DELTA. Enter the following command:

```
-rename alpha,beta to gamma,delta
gamma, already exists, OVERWRITE? y
alpha renamed TO gamma
delta, already exists, OVERWRITE? y
beta renamed TO DELTA
-
```

HOW TO RENAME DIRECTORIES

A directory can be renamed to new pathname on the same volume (but not to an existing pathname). Assume you have a directory whose pathname is ALPHA/BETA and you wish to rename it with a new pathname of AY/BEE. Enter the following command:

```
-rename alpha/beta to AY/BEE
alpha/beta renamed TO AY/BEE
-dir alpha/beta
alpha/beta, does not exist
-
```

Be careful when renaming directories! The last message explains the consequences of renaming a directory to a new pathname. ONCE YOU RENAME A DIRECTORY, ALL FILES LISTED UNDER THAT DIRECTORY WILL ALSO HAVE THIER PATHNAMES CHANGED. If your system has other programs that use data files that are listed under the old directory name, those programs will never find the files. In such a case, you must either rename the directories to their original names or modify the programs.

In summary, the distinctions between using the RENAME and COPY commands are as follows:

- When you use COPY to move the contents of an existing file TO a new file or OVER an existing file, the input file still exists.

- When you use RENAME to move the contents of an input file TO a named new file or OVER an existing file, the input pathname is automatically released for new uses.

HOW TO MOVE FILES ACROSS VOLUME BOUNDARIES

You can use all Human Interface file-handling commands except RENAME to manipulate files across volume boundaries. That is, you can copy files or directories from one diskette or disk platter to another one mounted on a different drive. The restriction against using RENAME across volume boundaries is intended for the protection of files against accidental deletion.

You access a different volume by entering the logical name for the device (the drive on which the volume is mounted) as the first item in the pathname. For example, assume you have a volume mounted on a drive whose logical name is :fl:. Further assume you wish to list the root directory for that volume to see what directories and data files you have on the volume. Enter the following command:

```
-dir :fl:

01 JAN 81  00:00:00
  DIRECTORY OF  :fl:  ON VOLUME disk2
able           baker          chuck           OMNI          samp
BUS            nusamp         STATS

-
```

Assume you wish to copy file ABLE from this volume mounted on :fl: to the MYTEST directory (which resides in your default directory). Enter the following command:

```
-copy :fl:/able to mytest/able
:fl:/able copied TO mytest/able
-
```

If you then wish to delete files ABLE and BAKER from the :fl: volume, simply enter the command:

```
-delete :fl:/able,:fl:/baker
:fl:/able, deleted
:fl:/baker, deleted
-
```

Now, assume the following conditions:

- You have two data files on the :fl: volume with the pathnames STATS/SALES/FEB and STATS/SALES/MAR.
- You wish to merge both files to a new file with the pathname MYTEST/PEEK/SUBTOT on your system's default volume.

Enter the following command:

```
-copy :fl:/stats/sales/feb, :fl:/stats/sales/mar to mytest/peek/subtot
:fl:/stats/sales/feb copied TO mytest/peek/subtot
:fl:/stats/sales/mar copied AFTER mytest/peek/subtot
-
```

Note that a volume prefix must be specified for each pathname in any command that crosses volume boundaries. A volume uses the prefix of the drive on which it is mounted.

HOW TO FORMAT A NEW VOLUME

Whenever you wish to use a new volume on a secondary storage device (such a diskette, disk platter, or bubble memory), you must format the volume before you can write any information in it. Assume you are going to mount a new diskette on a disk drive with the prefix :fl: that you have attached (with the ATTACHDEVICE command) as a named device.

Enter the following command:

```
-format :fl:
volume () will be formatted as a NAMED volume
    granularity   = 128          map start = 938
    interleave    = 5            sides     = 1
    files         = 200          density   = single
    extensionsize = 3            disk size = standard (8")
    volume size   = 250K

TTTTTTTTTTTTTTTTT...

volume formatted
-
```

This formatting example exercised all the default options.

This example did not specify a volume name as parameter of FORMAT. A volume name is not required; however, for diskettes, a volume name gives you a method for identifying a volume in case the stick-on label on the diskette gets lost or destroyed. You need only mount the disk on a drive and enter a DIR command for that drive to get a directory listing that specifies the volume name.

The GRANULARITY, INTERLEAVE, EXTENSIONSIZE, MAPSTART, and FILES parameters tell the FORMAT command how you want the physical space (for instance, disk surface space) on the volume allocated and accessed for maximum efficiency. The default parameters caused the NEWVOL example to be formatted with the following attributes:

- Since the device is attached as a named device, the NAMED parameter is the default with FORMAT. It specifies that you will be using the volume only to handle named files and directories. If you specified the PHYSICAL parameter, the entire volume would be treated as a single, large physical file. Once you you define the volume as NAMED or PHYSICAL, you can only use it for that purpose.

- The GRANULARITY parameter specifies the minimum number of bytes to be allocated for each increment of file size on the volume. The default granularity is the granularity of the physical device. Once the volume granularity is defined, it is applied to every file you create on the volume.

  For example, assume the default volume granularity for your device is 1024 bytes. Each time you create a new file on the volume, the I/O System automatically allocates 1024 bytes of primary storage to that file, whether or not the file requires the full 1024 bytes. If the size of your file exceeds 1024 bytes, the I/O System will increment your file size by still another block of 1024 bytes, and so on, until the end-of-file is reached.

- The INTERLEAVE default specifies that you want an interleave factor of 5. The interleave factor defines the number of physical sectors that occur between sequential logical sectors. This value maximizes access speed for the files on a given volume, depending upon the intent of the volume and the device configuration of your system.

  For example, an interleave value of 5 for a flexible disk system means that, for each file, the I/O System will read every fifth sector on the diskette, starting from an index of 1 (other, hard disk systems may be different, depending on your hardware configuration). Therefore, the I/O System does not need to wait for the disk to make a complete revolution before it accesses the next sector; the next sector by an increment of 5 is ready to be accessed for read/write by the time the first accessed sector has been processed.

  Note that the INTERLEAVE is the only optional parameter that is meaningful for volumes formatted for PHYSICAL files; the FILES, EXTENSIONSIZE, and GRANULARITY options are ignored in FORMAT commands that specify a PHYSICAL file format for the volume.

- The default FILES parameter specifies that you wish create a maximum of 50 user files on the volume. Although the actual number of files you can specify is 1 through 32,761, at a practical level, one of your determining factors will be the incremental file size you specify in the GRANULARITY parameter.

  The default EXTENSIONSIZE parameter specifies that you wish to create three bytes of extension data for each file. The Human Interface requires that at least three bytes of extension data be available. Other system programs included in your system may require larger values.

- The MAP START gives the volume block number where the fnode and map files start. If you do not specify a number, the Human Interface places the fnode and map files in the center of the volume.

DISKETTE SWITCHING PROCEDURES

If your system is configured with the iSBC 204 flexible disk controller and you are using single-density diskettes to perform file management functions, a special procedure is required to switch the diskettes. Perform the following steps:

1. Remove the old diskette and mount the new one into the drive.
2. Enter a DIR command for the root directory of the new diskette to force physical access. The root directory "name" is actually the prefix (logical name) for the drive on which the diskette is mounted. For example:

   ```
   -dir :fl:
   ```

   The following exception message will be displayed:

   ```
   E$IO
   -
   ```

3. Ignore the error message and begin entering Human Interface commands that access the volume.

***

CHAPTER 5
PATCHING UTILITY

The iAPX 86, 88 Patching Utility is a utility that runs on both iRMX 86 application systems and Series III Microcomputer Development Systems. It modifies combine-type attributes of object modules, permitting them to be written over (or repaired) with replacement modules. This capability provides you with a method of modifying relocatable object modules with software updates or repair code. The process requires that the replacement code first be generated with the ASM86 Assembler.

The Patching Utility also enables you to list the translator header records of the modules in a library. This listing allows you to see which patches have been installed in a module or a library.

In previous releases of the iRMX 86 Operating System, the Patching Utility served as a means of applying Intel-supplied software updates to your system. This method of applying software updates is no longer in effect. Software updates are now being distributed by Intel through the iRMX 86 Update Package. Fixes to software problems (referred to now by the general term "ZAP's") are applied to your system automatically when the update is installed. You will not have to invoke the patch utility to do this. Instructions for installing the update are included in the customer letter that accompanies each update package. The update package is describe in detail in the Installation Guide.

TYPES OF PATCHES

You can update a module to a newer version or add repair code in two ways:

- As a patch that generates a jump instruction to the replacement code and appends the replacement code to the end of the original module.
- As an in-place patch that directly overlays the replacement code on that of the original module.

An example of each technique is provided later in this chapter.

KINDS OF CODES THAT CAN BE PATCHED

The Patching Utility requires you to relink your object modules after you install the patches. Therefore you cannot patch modules which have been located at absolute addresses (run through LOC86) or those which have been linked using the BIND control of LINK86. Absolute and bound modules cannot be relinked; therefore they cannot be patched.
For example:

Object Module	Patch
iRMX 86 System Libraries	Yes
Human Interface Commands	No
An iRMX 86 Application System	No

VERSIONS OF THE PATCHING UTILITY

There is one version of the Patching Utility. The Patching Utility runs on an iRMX 86 application system and can be invoked with a Human Interface command. The Patching Utility can also run on a Series III Microcomputer Development System and can be invoked using the Series III RUN command. The name of the file containing the Patching Utility is called PTCH86.86.

If you are using a Series III system and your release diskette is in ISIS format (media type B), use the COPY command to copy PTCH86.86 from the Release Diskette onto an Intellec disk. If your release diskette is in iRMX 86 format (either media type E or media type J), use the Human Interface DOWNCOPY command (refer to Chapter 3) or the Files Utility (refer to Chapter 6) to transfer a copy of PTCH86.86 to the Intellec disk.

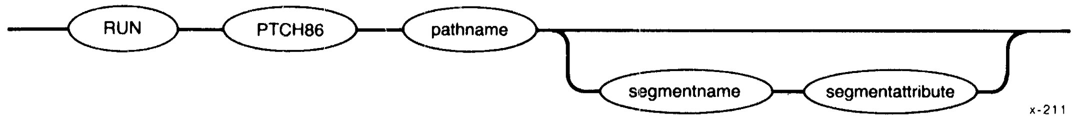
If you are using an iRMX 86 system and your release diskette is in iRMX 86 format (either media type E or media type J), execute the "instal.csd" command file (using the SUBMIT command) residing on the Release Diskette. The instal.csd file copies PTCH86.86 from the diskette into the appropriate iRMX directory. In the process PTCH86.86 is automatically renamed to PTCH86. (Your system must conform to the standard iRMX 86 directory structure in order for the instal.csd command file to work.) If your Release Diskette is in ISIS-II format (media type B), use the Human Interface UPCOPY command (refer to Chapter 3) or the Files Utility (refer to Chapter 6) to transfer a copy of PTCH86.86 into the RMX system. Since iRMX 86 does not require the ".86" suffix in order to run the Patch Utility, you may want to change the PTCH86.86 name (e.g. from PTCH86.86 to PTCH86).

INVOKING THE PATCHING UTILITY

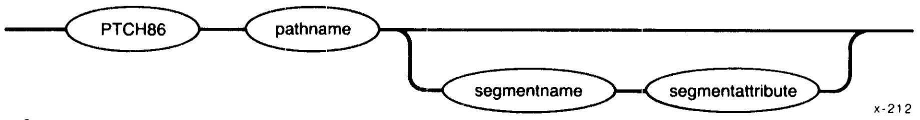
Before invoking the Patching Utility, ensure that the file containing it resides in the proper place. If you are using the Series III version, ensure that the PTCH86.86 resides on drive 0 of your development system.

If you are using the iRMX 86 version, ensure that PTCH86 resides in your default directory or in one of the directories that the Operating System automatically searches (usually :PROG: and :SYSTEM:). Then you can invoke the Patching Utility by entering one of the following commands:

<u>Series III Invocation</u>



<u>iRMX 86 Invocation</u>



where:

pathname	Pathname of the file containing an iAPX 86, 88 object module produced by PL/M-86, ASM86, or LINK86. If you are listing the translator header records, the object module can be one produced by LOC86 or LIB86.
segmentname	Name of the segment whose combine-type attribute is to be modified. The name must be a valid segment name (usually CODE). If you omit this parameter (and the segmentattribute parameter), the utility does not modify the attributes of any segment. Instead, it displays the translator header records contained in the module.
segmentattribute	Combine-type attribute to be given to the named segment. You must specify one of the following values for this attribute:
COMMON	Allows patch code to be overlaid on the segment.
PUBLIC	Returns the segment to the combination mode normally given by the PL/M-86 compiler.

Refer to the ASM86 LANGUAGE REFERENCE MANUAL for a more detailed explanation of combine-type attributes.

In response, the Patching Utility displays the following header message:

```
<operating system> iAPX 86, 88 OBJECT PATCHING UTILITY, Vx.y
Copyright <year> Intel Corporation
```

where:

<operating system>	Name of the operating system, either iRMX 86 or SERIES III.
x.y	Version number of the Patching Utility.

If you exclude the optional segmentname and segmentattribute parameters, the utility next displays the translator header records contained in the object file. These header records serve to identify the patches that have been made to a module (described later in this chapter). This displays allows you to determine the update status of the file.

If you specify the segmentname and segmentattribute parameters, the utility omits the display of the translator header records. Instead, it changes the combine-type attribute as specified and displays the following message:

```
ATTRIBUTE MODIFIED
```

Any other message indicates an error condition.

ERROR MESSAGES

When the Patching Utility encounters an error condition during a module repair session, it displays one of the following error messages:

- ERROR nnn

  When using the Patching Utility on a Series III development system, the operating system returned an error message. Refer to the INTELLEC SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM CONSOLE OPERATING INSTRUCTIONS for a description of numbered messages of this form.

- <exception code value> : <exception code mnemonic>

  When using the Patching Utility on an iRMX 86 application system, the operating system returned an exceptional condition code. Refer to Appendix A of this manual for summary of these condition codes. Refer to the appropriate iRMX 86 manual for detailed descriptions of iRMX 86 exception codes.

- INVALID MODULE TYPE

  The file specified is not suitable for the segment attribute modification and subsequent patching.

- INVALID RECORD TYPE

  The object file contains an invalid record type for the object module format. Perhaps you entered the wrong file name or specified a file that contains code other than object code.

- INVALID SYNTAX

  The command line contains an error that was caused by a missing file name or a missing or misspelled keyword.

- SEGMENT NOT FOUND

  The desired record was not found before the end of the module.

PATCHING PROCEDURES

Repair modules that you insert into existing modules must be generated with the ASM86 Assembler. To patch an independent object module containing errors (patching library modules is described later in this chapter), you must first invoke the Patching Utility to modify the combine-type attribute in the desired module segment to COMMON. This step allows you to use LINK86 to overlay the repair module on the segment to be patched. After linking with the repair module, you then use the Patching Utility to restore the PUBLIC attribute to the segment. The following example illustrates the steps for repairing independent object module files:

1. Enter the PTCH86 command to set the CODE segment combine-type attribute to COMMON. For example, on a Series III development system, enter:

   ```
   RUN PTCH86 badmodule CODE COMMON
   ```

2. Enter the LINK86 command to overlay the repair object module on the original version. (When entering the LINK86 input parameter list, the name of the original module <u>MUST</u> precede the name of the repair module) For example:

   ```
   RUN LINK86 badmodule, repairmodule TO newmodule
   ```

3. Enter the PTCH86 command to restore the CODE segment to PUBLIC, for example:

   ```
   RUN PTCH86 newmodule CODE PUBLIC
   ```

Typical examples of jump instruction overlays, in-place patch overlays, library module patching, and listing module header records are given in the following sections. A submit file, PATCH.CSD, is provided on the Release Diskette as an example of how to use the Patching Utility.

JUMP INSTRUCTION PATCH

In the following example, the module generates a patch that overlays a three-byte jump instruction on offset 0100H through 0102H of the original module. The jump transfers control to repair code at offset 0500H. The repair code is appended to the end of the module.

EXAMPLE:

```
    NAME     REPAIR_V00001         ; Identifying module name.

    CODE     SEGMENT     WORD COMMON    'CODE'
    CGROUP   GROUP       CODE
             ASSUME      CS : CGROUP

             ORG         0100H     ; Offset of area in original
                                   ; module to be patched.

             JMP         REPAIRCODE

    RETURN   LABEL       NEAR      ; Return here from repair area.

             ORG         0500H     ; Offset of end of original
                                     module.

    REPAIRCODE:
    ;
    ;   (Repair goes here)
    ;
             JMP         RETURN    ; Return control to original
                                     module.
    CODE     ENDS
    END
```

When making a jump instruction patch similar to the one listed previously, you must overlay the three-byte jump instruction on one of the following:

- A three-byte instruction
- A two-byte instruction and a one-byte instruction
- Three one-byte instructions

Otherwise, you must place NOP instructions after the JMP instruction and before the RETURN label so that the repair code returns to the start of the next instruction (not to the middle of a previous instruction).

IN-PLACE PATCH

The following example generates an in-place patch that directly overlays repair code on a module's previous code.

EXAMPLE:

```
NAME     REPAIR_V00002         ; Module name identification.

CODE     SEGMENT    WORD COMMON    'CODE'
CGROUP   GROUP      CODE
         ASSUME     CS : CGROUP

         ORG        0200H      ; Offset of the original operand.

         ADD        AX, 3      ; Replaces the original value with a "3"
                               ; (the new instruction must be the same
                               ; size as the original instruction).

CODE     ENDS

END
```

LISTING TRANSLATOR HEADER RECORDS

If you want the Patching Utility to list an object module's translator header records on the console screen, enter the PTCH86 command without specifying the segment name or segment attribute. The listed records allow you to identify the patches that have been made to the module. A typical PTCH86 command entry (from an iRMX 86 system) and resulting header record display is as follows:

```
-PTCH86 FILE.OBJ
iRMX 86 iAPX 86, 88 OBJECT PATCHING UTILITY, Vx.y
    ORIGINALMODULE
    ORIGINALMODULE_REPAIR_V030-01
    ORIGINALMODULE_REPAIR_V030-02
```

The "030" stands for version 3.0 of the software being patched, and "01" and "02" are the patch numbers of the Intel-supplied patches that have been made to the module. The Patching Utility can perform this listing operation on both object modules and libraries.

***

CHAPTER 6
FILES UTILITY SYSTEM

Because INTELLEC Microcomputer Development Systems do not recognize iRMX 86 diskette files, you cannot read, write, or format iRMX 86 diskettes directly from the ISIS-II operating system. However, you can perform these operations indirectly from the Development System by using the iRMX 86 Files Utility System. The iRMX 86 Files Utility System is an iRMX 86 application system that allows you to perform the following operations:

- Format an iRMX 86 diskette.
- Copy a file from an ISIS-II diskette to an iRMX 86 diskette.
- Copy a file from an iRMX 86 diskette to an ISIS-II diskette.
- Delete a file from an iRMX 86 diskette.
- Create a directory on an iRMX 86 diskette.
- Display, on the Development System terminal, the contents of a directory of an iRMX 86 diskette.

If you cannot use the startup system (described in the iRMX 86 INSTALLATION GUIDE) to format your first iRMX 86 secondary storage volumes and transfer necessary files (such as Human Interface commands) to these volumes, you can use the Files Utility for this purpose. The Files Utility System also gives you the ability to build and maintain secondary storage volumes for iRMX 86 application systems that do not include the Human Interface.

HARDWARE REQUIRED

The Files Utility System requires the following hardware:

- A Series III Microcomputer Development System having at least 64k bytes of memory and at least one disk drive (hard or flexible). The Files Utility is not supported on Series IV Microcomputer Development Systems.
- A target system consisting of an iAPX 86, 88, 186, or 188-based Single Board Computer, at least 192k bytes of memory, and at least one disk drive (hard or flexible). THE FILES UTILITY DOES NOT RUN ON iAPX 286 MICROPROCESSOR BASED SYSTEMS.
- The iSDM 86 MONITOR.

STARTING THE FILES UTILITY

Before you can enter commands to the Files Utility, you must start it up. This involves connecting certain hardware modules and then entering appropriate commands at the Series III terminal.

After you have assembled your hardware, perform the following steps:

1. Place an ISIS-II system diskette containing the iSDM 86 monitor software into drive 0 of your INTELLEC Microcomputer Development System and the Utilities release diskette into any other drive.

2. Load the ISIS-II system.

3. Enter the following:

```
ISDM 86

.R :fx:fs86

or

.R :fx:fs186
```

where:

fx	Identifier of the disk drive containing the Utility release diskette.
fs86	Identifies the monitor as iAPX 86 microprocessor based.
fs186	Identifies the monitor as iAPX 186 microprocessor based.

These commands instruct the monitor to load the Files Utility System from a diskette on the INTELLEC system into RAM on the target system. The screen of your INTELLEC system should display the heading:

```
iRMX 86 FILES UTILITY Vx.y
Copyright <year> Intel Corporation
```

The Files Utility signals that it is ready to accept your next command by displaying an asterisk (*) at the screen of the INTELLEC system.

USING THE FILES UTILITY

The Files Utility provides 10 file management commands, as follows:

ATTACHDEV	DIR
BREAK	DOWNCOPY
CREATEDIR	FORMAT
DELETE	HELP
DETACH	UPCOPY

The commands are described in alphabetical sequence later in this chapter. However, before actually using the commands, you should understand the diskette handling procedures and how the Files Utility System handles errors.

CHANGING DISKETTES

When the Files Utility is running and you have already performed an operation on a particular diskette, you cannot simply remove that diskette from the drive and replace it with another. The Utility System is not aware of diskette changes and treats the second diskette as if it were the first, and thereby possibly writes over or destroys valuable information. To change diskettes in a drive, you must enter a DETACH command to logically detach the drive from the system, change diskettes, and then (with one exception) enter an ATTACHDEV command to again logically attach the device.

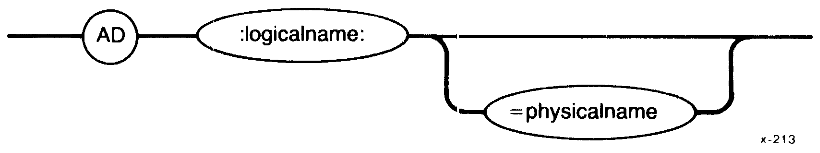
The one exception to this command entry sequence is the FORMAT command. As described later in this chapter, this command writes iRMX 86 formatting information on blank diskettes. Since the FORMAT command always expects a blank diskette and a detached drive, you can replace diskettes in a drive any number of times if you use only the FORMAT command before entering the ATTACHDEV command. The FORMAT command destroys the information, if any, previously contained on the diskette.

COMMANDS

This section provides descriptions of the Files Utility commands and their parameters in alphabetical sequence. Each command has a two-character abbreviation. You can use either the full name or its abbreviation when entering a command.

ATTACHDEV (AD)

This command attaches a physical device to the system and associates a logical name with the device. The command can also be used to display the current attachment of a logical name. The format is as follows:



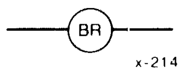
where:

:logical-name:	A 1-to 12-character ASCII name, surrounded by colons.
=physical-name	If used, there must be no spaces surrounding the equal sign. This specifies the physical device name as configured in the I/O System (see Table 3-2). If physical name is omitted, the current attachment is displayed by default; for example:

```
AD :F0:    (command entry)
:F0: = FX0 (displayed output)
```

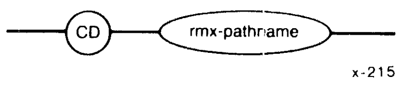
BREAK (BR)

This command causes an exit from the Files Utility System to the monitor. The format is as follows:



CREATEDIR (CD)

This command creates an iRMX 86 directory file on the attached device. The format is as follows:

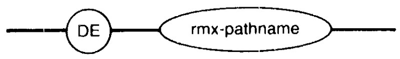


where:

rmx-pathname	Pathname of the iRMX 86 directory file to be created.

DELETE (DE)

This command removes the specified iRMX 86 file from the directory where it is listed. The format command is as follows:



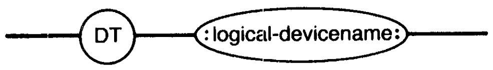
x-216

where:

rmx-pathname	Pathname of the iRMX 86 file to be deleted.

DETACH (DT)

This command detaches a logical name from the system. The command is used for changing diskettes, prior to entering a FORMAT command, or to reconfigure a device to a different sector size. The format is as follows:



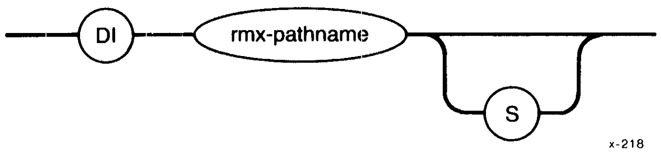
x-217

where:

:logical-name:	The logical name you assigned to a physical device via an ATTACHDEV command.

DIR (DI)

This command lists an iRMX 86 directory file at the Development System console. The format is as follows:



x-218

where:

rmx-pathname	Pathname of the iRMX 86 directory file to be listed.

S	Switch that causes a "long" or expanded display of directory file that includes: file type (a "DR" heading for a directory file, a "MP" heading for the bit map file, or a blank heading for a data file), number of blocks, and number of bytes in file. If S is not specified, a "fast" format will be displayed, consisting of file names only.

The directory file listing includes a line that lists the size of the directory. This line appears as:

    <n> FILES

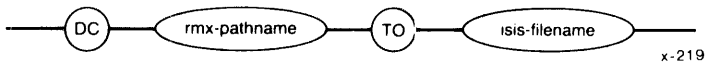
In this line, <n> specifies the number of entries currently present in the directory. If you specify the S parameter, this command also lists the following information about the directory:

    <numblks> BLOCKS   <numbytes> BYTES

In this line, <numblks> specifies the number of volume-granularity blocks allocated to files in the directory and <numbytes> specifies the number of bytes allocated to files in the directory.

DOWNCOPY (DC)

This command creates an ISIS-II file and copies the specified iRMX 86 file to it. If the ISIS-II file already exists, it is written over. The format is as follows:



where:

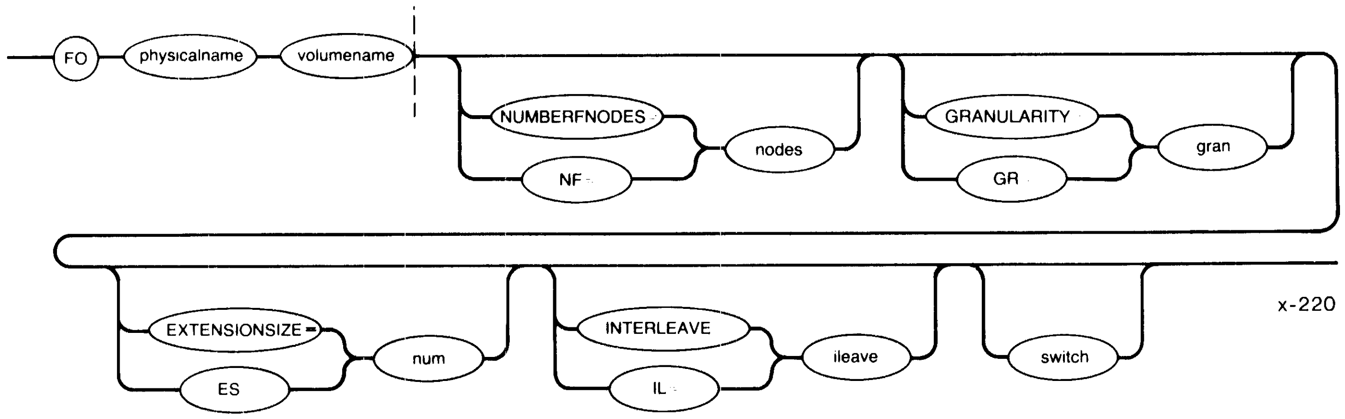
rmx-pathname	Pathname of the iRMX 86 file to be copied.
isis-filename	Name of the ISIS-II file to be created.

FORMAT (FO)

This command writes iRMX 86 formatting information on a secondary storage device. It performs the same kind of operations as the Human Interface FORMAT command described in Chapter 3. All information previously contained on the device is destroyed by the formatting operation. Each device must be formatted before it can be used by the iRMX 86 Operating System.

The FORMAT command expects an unattached device. The device can either be unattached at system start up, or you can detach it by entering a DETACH command prior to entering the FORMAT command. Since the device remains unattached after FORMAT completes execution, you must attach the device by entering an ATTACHDEV command before entering any other Utility command except another FORMAT command. (See also the "Changing Diskettes" section in this chapter, and the ATTACHDEV and DETACH command descriptions.)

The FORMAT command contains parameters that are specified in the form "keyword=value". When entering parameters of this type, you must not place any spaces around the equal sign. Also, you can abbreviate each of these keywords as shown. The abbreviations and the format of this command are as follows:



x-220

where:

physicalname	Physical device name for the drive, as configured in the I/O System, that denotes the iRMX 86 drive on which the diskette resides. Possible values are itemized in Table 3-2.
volumename	A 1- to 10-character volume name that identifies the diskette. Decimal digits, uppercase and lowercase letters, and the following special characters can be used in the volume names:

```
! & , * ;
" ' ( + /
% . ) : = ?
```

nodes	The number of files (including internal system files) that can be created on this volume. If you omit this parameter, a default value of 57 is assumed.

gran	The granularity, in bytes, for this volume. The granularity is the number of bytes obtained during each diskette access. If you omit this parameter, the default volume granularity is the device granularity (the number of bytes in a physical sector). Specifying any value less than the device granularity causes the default to be used. Any non-multiple of device granularity (such as 128 or 512) is rounded upward to the next higher multiple of device granularity.
num	Size, in bytes, of the extension data associated with each file. This data is used by A$GET$EXTENSION$DATA and A$SET$EXTENSION$DATA system calls (refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL). The Human Interface requires files it accesses to have three bytes of extension data. The range is 0 through 255 (decimal). If not specified, the default is three bytes.
ileave	The interleave factor for the volume, or the number of physical sectors between logical sectors. You can specify any integer from 1 to 13 for this value. If you omit this parameter, a default value of 5 is assumed.
switch	A switch that indicates the support option for this volume. One value can be entered for the switch: NAMED — The volume is created to contain named files. The ROOT directory is initialized. If you omit this switch, the volume is created as a single physical file. In this case, FORMAT records the interleave information on the diskette but does not initialize any of the iRMX 86 file structures.

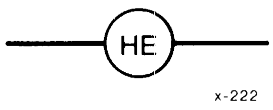
When it formats a named volume, the FORMAT command creates six internal system files. It names four of these files and lists their names in the root directory of the volume. The files are:

file	description
R?SPACEMAP	Volume free space map
R?FNODEMAP	Free fnodes map
R?BADBLOCKMAP	Bad blocks map
R?VOLUMELABEL	Volume label

The command assumes that the user WORLD is the owner of these files. Refer to the iRMX 86 DISK VERIFICATION UTILITY REFERENCE MANUAL for more information about these files.

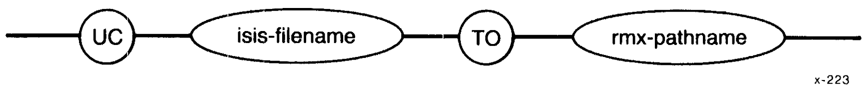
HELP (HE)

This command displays a list of the available Files Utility commands and their syntax on the console screen. The format is as follows:



UPCOPY (UC)

This command creates an iRMX 86 file and copies the specified ISIS-II file to it. If the iRMX 86 file already exists, it is written over. The format is as follows:



where:

isis-filename	Name of the ISIS-II file to be copied.
rmx-pathname	Pathname of the iRMX 86 file to be created.

ERROR MESSAGES

The Files Utility displays all error messages on the screen of the INTELLEC Microcomputer Development System. These messages can be in any of three forms. They are:

- UNRECOGNIZED COMMAND

  The Files Utility does not recognize the spelling of your command. It prompts for another command.

- ISIS ERROR # <nnn>

  The Files Utility actually uses the ISIS-II operating system to read and write diskettes attached to the INTELLEC Microcomputer Development System. If the ISIS-II system detects any errors, it returns an error code to the Files Utility. To interpret this error message, refer to the INTELLEC SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM CONSOLE OPERATING INSTRUCTIONS. Fatal errors require you to restart the Files Utility System by using the FILES.CSD file, as described earlier in this chapter.

- RMX EXCEPTION # <mmmm>

  When reading or writing on drives attached to the target system, the Files Utility System uses the iRMX 86 Nucleus and the iRMX 86 I/O System. If either of these layers returns an exceptional condition code, the Files Utility displays the condition code in this format, where mmmm is in hexadecimal. For a brief explanation of such an error message, refer to Appendix A. After displaying this message, the Files Utility prompts for the next command.

***

APPENDIX A
CONDITION CODES SUMMARY

Table A-1 provides a list of the iRMX 86 condition codes that can occur during system operations. This table provides a minimum of information about each condition code. In most cases, the condition code must be considered in terms of the unique circumstances that caused the condition. Table A-1 is provided to guide you to the most appropriate manual. The appropriate iRMX 86 manuals have more detailed descriptions of the meanings. The appropriate manual is listed in the column marked "Manuals".

Table A-1. iRMX™ 86 Condition Codes

Hex. Value	Mnemonic	Manuals N B E L H	Meaning
0H	E$OK	* * * * *	No exceptional conditions (normal)
		Environmental Conditions	
1H	E$TIME	* * * * *	A time limit (possibly a limit of zero time) expired without a task's request being satisfied.
2H	E$MEM	* * * * *	Insufficient available memory to satisfy a task's request.
3H	E$BUSY	*	Another task currently has access to data protected by a region.
4H	E$LIMIT	* * * * *	A task attempted an operation which, if it had been successful, would have violated a Nucleus-enforced limit.
5H	E$CONTEXT	* * * * *	A system call was issued out of proper context.
6H	E$EXIST	* * * * *	A token parameter has a value which is not the token of an existing object.

N	Nucleus Reference Manual	L	Loader Reference Manual
B	Basic I/O System Ref Manual	H	Human Interface Reference Manual
E	Extended I/O Sys Ref Manual		

Table A-1. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals					Meaning
		N	B	E	L	H	
							Environmental Conditions (continued)
7H	E$STATE	*					A task attempted an operation which would have caused an impossible transition of a task's state.
8H	E$NOT$CON-FIGURED	*	*	*	*	*	This system call is not part of the present configuration.
9H	E$INTER-RUPT$SAT-URATION	*					An interrupt task has accumulated the maximum allowable amount of SIGNAL$INTERRUPT requests.
0AH	E$INTER-RUPT$-OVERFLOW	*					An interrupt task has accumulated more than the maximum allowable amount of SIGNAL$INTERRUPT requests.
20H	E$FEXIST		*	*			File already exists.
21H	E$FNEXIST		*	*	*	*	File does not exist.
22H	E$DEVFD		*	*		*	Device and file driver are incompatible.
23H	E$SUPPORT		*	*	*	*	Combination of parameters not supported.
24H	E$EMPTY$-ENTRY		*	*			The specified slot in a directory file is empty.
25H	E$DIR$END		*	*			The specified slot is beyond the end of a directory file.
26H	E$FACCESS		*	*	*	*	File access not granted.
27H	E$FTYPE		*	*		*	Incompatible file type.
28H	E$SHARE		*	*	*	*	Improper file sharing requested.
29H	E$SPACE		*	*			No space left.

N	Nucleus Reference Manual	L	Loader Reference Manual
B	Basic I/O System Ref Manual	H	Human Interface Reference Manual
E	Extended I/O Sys Ref Manual		

Table A-1. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals					Meaning
		N	B	E	L	H	
							Environmental Conditions (continued)
2AH	E$IDDR		*	*			Invalid device driver request.
2BH	E$IO		*	*	*	*	An I/O error occurred.
2CH	E$FLUSHING		*	*	*	*	Connection specified in call was deleted before the operation was completed.
2DH	E$ILLVOL		*	*		*	Invalid volume name.
2EH	E$DEV$OFF-LINE		*				The device being accessed is now offline.
2FH	E$IFDR		*	*			Invalid file driver request.
30H	E$FRAGMENT-ATION	*					The file is too fragmented to be extended.
31H	E$DIR$NOT$-EMPTY	*	*				The call attempted to delete a directory with files.
32H	E$NOT$FILE$-CONN		*				The connection parameter is not a file connection.
33H	ENOTDEV-ICE$CONN		*				The connection parameter is not a device connection.
34H	E$CONN$NOT$-OPEN		*				The connection is closed or open but not compatible with current request.
35H	E$CONN$OPEN		*				The task is trying to open a connection which is already open.
36H	E$BUFFERED$-CONN		*				The connection was opened with one or more buffers.
37H	E$OUTSTAND-ING$CONN		*				A soft detach was specified, but connections to the device still exist.

N	Nucleus Reference Manual	L	Loader Reference Manual
B	Basic I/O System Ref Manual	H	Human Interface Reference Manual
E	Extended I/O Sys Ref Manual		

Table A-1. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals					Meaning
		N	B	E	L	H	
							Environmental Conditions (continued)
38H	E$ALREADY$-ATTACHED		*				The specified device is already attached.
39H	EDEV-DETACHING		*				The file is on device that is being detached.
3AH	E$NOT$SAME$-DEVICE		*				The existing pathname and the new pathname refer to different devices. You cannot simultaneously rename a a file and move it.
3BH	E$ILLOGICAL$-RENAME		*				The call attempted to rename a directory to a new path containing itself.
3CH	E$STREAM$-SPECIAL		*				A stream file is out of context.
3DH	E$INVALID$-FNODE		*				The connection refers to a file with an invalid fnode.
3EH	E$PATHNAME$-SYNTAX		*				The specified pathname contains invalid characters.
3FH	E$FNODE$-LIMIT		*				The volume already contains the maximum number of files.
40H	E$LOG$NAME$-SYNTAX			*		*	The specified path starts with a colon (:) but does not contain a second, matching colon.
42H	E$IOMEM			*		*	The Basic I/O System has insufficient memory to process a request.
44H	E$MEDIA			*		*	The device containing a specified file is not online.
45H	E$LOG$NAME$-NEXIST			*		*	The Extended I/O System was unable to find a specified logical name in the object directories that it checks.

N	Nucleus Reference Manual	L	Loader Reference Manual
B	Basic I/O System Ref Manual	H	Human Interface Reference Manual
E	Extended I/O Sys Ref Manual		

Table A-1. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals					Meaning
		N	B	E	L	H	
Environmental Conditions (continued)							
46H	E$NOT$OWNER			*			The user who attempted to detach the device is not the owner of the device.
47H	E$IO$JOB			*			The Extended I/O System cannot create an I/O job because the size specified for the object directory is too small.
50H	E$IO$UNCLASS			*			An unknown type of I/O error occurred.
51H	E$IO$SOFT		*	*			A soft I/O error occurred. A retry might be successful.
52H	E$IO$HARD		*	*			A hard I/O error occurred. A retry is probably useless.
53H	E$IO$OPRINT		*	*			The device was off-line. Operator intervention is required.
54H	E$IO$WRPROT		*	*			The volume is write-protected.
55H	E$IO$NO$DATA		*	*			A tape drive attempted to read the next record, but found no data.
56H	E$IO$MODE		*	*			A tape drive attempted a read (write) operation before the previous write (read) operation completed.
61H	E$BAD$GROUP				*	*	Invalid group component in the a group definition record.
62H	EBAD-HEADER				*	*	Invalid header record in the object file.
63H	E$BAD$SEG-DEF				*	*	Invalid segment definition record.
64H	E$CHECKSUM				*	*	A checksum error occurred while reading an object record.

N	Nucleus Reference Manual	L	Loader Reference Manual
B	Basic I/O System Ref Manual	H	Human Interface Reference Manual
E	Extended I/O Sys Ref Manual		

Table A-1. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals N	B	E	L	H	Meaning
Environmental Conditions (continued)							
65H	E$EOF				*	*	Unexpected end of file encountered while reading object records.
66H	E$FIXUP				*	*	Invalid fixup record in the object file.
67H	E$NO$LOADER $MEM				*	*	Insufficient memory to satisfy loader dynamic memory requirements.
68H	E$NO$MEM				*	*	Insufficient memory to create PIC/LTL segments.
69H	E$REC$FOR-MAT				*	*	Invalid record format encountered.
6AH	EREC-LENGTH				*	*	Record length of an object record exceeds configured loader-buffer size.
6BH	E$REC$TYPE				*	*	Invalid record type encountered in the object file.
6CH	E$NO$START				*	*	Start address not found.
6DH	E$JOB$SIZE				*	*	Maximum job-size specified is less than the memory requirement specified in the object file.
6EH	E$OVERLAY				*		Overlay name does not match with any of the overlay module names.
6FH	E$LOADER $SUPPORT				*	*	The object file being loaded requires features not supported by the configured loader.
70H	ESEG BOUNDS				*		One of the data records in a module loaded by the Application Loader referred to an address outside the segment created for it.

N	Nucleus Reference Manual	L	Loader Reference Manual
B	Basic I/O System Ref Manual	H	Human Interface Reference Manual
E	Extended I/O Sys Ref Manual		

Table A-1. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals					Meaning
		N	B	E	L	H	
Environmental Conditions (continued)							
80H	E$LITERAL					*	The parse buffer contains a literal with no closing quote.
81H	E$STRING$-BUFFER					*	The string to be returned as the parameter name exceeds the size of the buffer the user provided in the call.
82H	E$SEPARA-TOR					*	The parse buffer contains a command separator.
83H	E$CONTINUED					*	The parse buffer contains a continuation character.
84H	E$INVALID$-NUMERIC					*	A numeric value contains invalid characters.
85H	E$LIST					*	The last value of the value list is missing.
86H	E$WILDCARD					*	A wild-card character appears in an invalid context, such as an intermediate component of a pathname.
87H	E$PREPOSI-TION					*	The same preposition as on the the command line was indicated, but can not be used.
88H	E$PATH					*	The command line specifies an invalid pathname.
89H	E$CONTROL$C					*	The user typed CONTROL-C while the command was being loaded.
8AH	E$CONTROL					*	The command line contains an invalid control.
8BH	E$UNMATCHED $LISTS					*	There were no more input pathnames although the output pathname list was not empty.

N	Nucleus Reference Manual	L	Loader Reference Manual
B	Basic I/O System Ref Manual	H	Human Interface Reference Manual
E	Extended I/O Sys Ref Manual		

Table A-1. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals N	B	E	L	H	Meaning
							Programmer Errors
8CH	E$DATE	*					The operator entered an invalid date.
8DH	E$NO$PARAM-ETERS					*	A command expected parameters, but the operator didn't supply any.
8EH	E$VERSION					*	The Human Interface is not compatible with the version of the command the operator invoked.
8FH	E$GET$PATH$-ORDER					*	A command called C$GET$OUTPUT$PATHNAME before calling C$GET$INPUT$PATHNAME.
00C0H	E$UNKNOWN$-EXIT						The program exited normally.
00C1H	E$WARNING$ EXIT						The program issued warning messages.
00C2H	E$ERROR$EXIT						The program detected errors.
00C3H	E$FATAL$EXIT						A fatal error occurred in the program.
00C4H	E$ABORT$EXIT						The system aborted the program.
00C5H	E$UDI$IN-TERNAL						A UDI internal error occurred.
8000H	E$ZERO$-DIVIDE	*					A task attempted to divide by zero.
8001H	E$OVERFLOW	*					An overflow interrupt occurred.
8002H	E$TYPE	*	*	*	*	*	A token parameter referred to an existing object that is not of the required type.

N	Nucleus Reference Manual	L	Loader Reference Manual
B	Basic I/O System Ref Manual	H	Human Interface Reference Manual
E	Extended I/O Sys Ref Manual		

Table A-1. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals N B E L H	Meaning
Programmer Errors (continued)			
8004H	E$PARAM	* * * * *	A parameter which is neither a token nor an offset has an invalid value.
8005H	E$BAD$CALL	* *	The I/O System code has been damaged, probably due to a bug in an application task. Recovery is not possible.
8006H	E$ARRAY$-BOUNDS	*	Hardware or software has detected an array overflow.
8007H	ENDP-STATUS	*	An 8087 Numeric Processor Extension error has been detected; Operating System extensions can return the status of the 8087 to the exception handler.
8008H	E$CHECK$EX-CEPTION	*	A software interrupt 17 has occurred.
8009H	E$EMULATOR$-TRAP	*	The iAPX 186 or 286 processor tried to execute an ESC instruction with the "emulator" bit set in the relocation register (iAPX 186) or the machine status word (iAPX 286).
800AH	E$INTERRUPT$-TABLE$LIMIT	*	An iAPX 286 LIDT instruction changed the interrupt table limit to a value between 20H and 42H.
800BH	E$CPUXFER$-DATA$LIMIT	*	For an iAPX 286 processor, the processor extension data transfer exceeded the offset of 0FFFFH in a segment.
800CH	E$SEG$WRAP$-AROUND	*	For an iAPX 286 processor, either a word operation attempted a segment wraparound at offset 0FFFFH; or a PUSH, CALL, or INT instruction attempted to execute while SP=1.

N	Nucleus Reference Manual	L	Loader Reference Manual
B	Basic I/O System Ref Manual	H	Human Interface Reference Manual
E	Extended I/O Sys Ref Manual		

Table A-1. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals N	B	E	L	H	Meaning
							Programmer Errors (continued)
8017H	E$CHECK$EX-CEPTION	*					A Pascal task has exceeded the bounds of a CASE statement.
8021H	E$NOUSER		*	*		*	No default user.
8022H	E$NOPREFIX		*	*		*	No default prefix.
8040H	E$NOT$LOG$-NAME			*		*	Specified object is not a device connection or file connection.
8041H	ENOT-DEVICE			*			A token parameter referred to an existing object that is not, but should be, a device connection.
8042H	E$NOT$CON-NECTION			*			A token parameter referred to an existing object that is not, but should be, a file connection.
8060H	E$JOB$PARAM				*	*	The maximum job-size specified is less than the minimum job-size.
8080H	E$PARSE$-TABLES					*	There is an error in the internal parse tables.
8081H	EJOB-TABLES					*	An internal Human Interface table was overwritten, causing it to contain an invalid value.
8085H	E$ERROR$-OUTPUT					*	The command invoked by C$SEND$COMMAND includes a call to C$SEND$EO$RESPONSE, but the command connection does not permit C$SEND$EO$RESPONSE calls.

N	Nucleus Reference Manual	L	Loader Reference Manual
B	Basic I/O System Ref Manual	H	Human Interface Reference Manual
E	Extended I/O Sys Ref Manual		

Table A-1. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals					Meaning
		N	B	E	L	H	
							Programmer Errors (continued)
8083H	E$DEFAULT$SO					*	The default output name STRING is invalid.
8084H	E$STRING					*	The pathname to be returned exceeds 255 characters in length.
80C6H	E$RESERVE$-PARAM						The calling program tried to reserve memory for more than 12 files or buffers.
80C7H	E$OPEN$PARM						The calling program requested more than two buffers when opening a file.

N	Nucleus Reference Manual	L	Loader Reference Manual
B	Basic I/O System Ref Manual	H	Human Interface Reference Manual
E	Extended I/O Sys Ref Manual		

***

INDEX

Primary references are underscored.

***

iRMX™ 86 DISK VERIFICATION UTILITY REFERENCE MANUAL

PREFACE

This manual documents the Disk Verification Utility, a software tool that runs as a Human Interface command, verifying and modifying the data structures of iRMX 86 named and physical volumes. The manual describes the utility invocation and contains detailed descriptions of all utility commands. Also, because users must be familiar with the structure of iRMX 86 volumes to use the Disk Verification Utility features intelligently, the manual contains an appendix that describes the structure of iRMX 86 named volumes.

READER LEVEL

This manual is intended for system programmers who have had experience in examining actual volume information. It does not attempt to teach the user the proper procedures for examining and editing volume information.

NOTATIONAL CONVENTIONS

This manual uses the following conventions to illustrate syntax.

UPPERCASE	Uppercase information must be entered exactly as shown. You can, however, enter this information in uppercase or lowercase.
lowercase	Lowercase fields contain variable information. You must enter the appropriate value or symbol for variable fields.
<u>underscore</u>	In examples of dialog at the terminal, user input is underscored to distinguish it from system output.
<variable>	Whenever an error message or the output resulting from a DISKVERIFY command contains a variable part, that variable part is enclosed in angle brackets < >.

Also, this manual uses the "railroad track" schematic to illustrate the syntax of the disk verification commands. This syntax consists of what looks like an aerial view of a model railroad setup, with syntactic elements scattered along the track. To interpret the command syntax, you start at the left side of the schematic, follow the track through all the syntactic elements you desire (sharp turns and backing up are not allowed), and exit at the right side of the schematic. The syntactic elements that you encounter, separated by spaces, comprise a valid command. For example, a command that consists of a command name and two optional parameters would have the following schematic representation:

You could enter this command in any of the following forms:

```
    COMMAND
    COMMAND param1
    COMMAND param2
    COMMAND param1 param2
```

The arrows indicate the possible flow through the tracks; they are omitted in the remainder of the manual.

CONTENTS

PAGE

***

CHAPTER 1
INVOKING THE DISK VERIFICATION UTILITY

In the process of using an iRMX 86 application system, you may have occasion to store data on secondary storage devices, sometimes large amounts of data. Due to the nature of secondary storage devices, unforseen circumstances such as power irregularities or accidental reset may destroy information on these devices, causing them to be inaccessible to your iRMX 86 system. In some cases, the loss of only a small amount of data can render an entire volume, such as a disk, useless.

In such cases, it is desirable to have a mechanism to examine and modify the damaged volume. This mechanism would allow you to determine how much of the information on the volume was damaged. It would also allow you to recreate file structures on the damaged volume so that you could salvage some of the valid data. The iRMX 86 disk verification utility is a tool that allows you to perform these functions.

The disk verification utility verifies the data structures of iRMX 86 physical and named volumes. It can also be used to reconstruct the free fnodes map, the volume free space map, and the bad blocks map of the volume and perform absolute editing.

You can use the disk verification utility in one of two ways:

- As a single command which verifies the structures of a volume and returns control to the Human Interface.

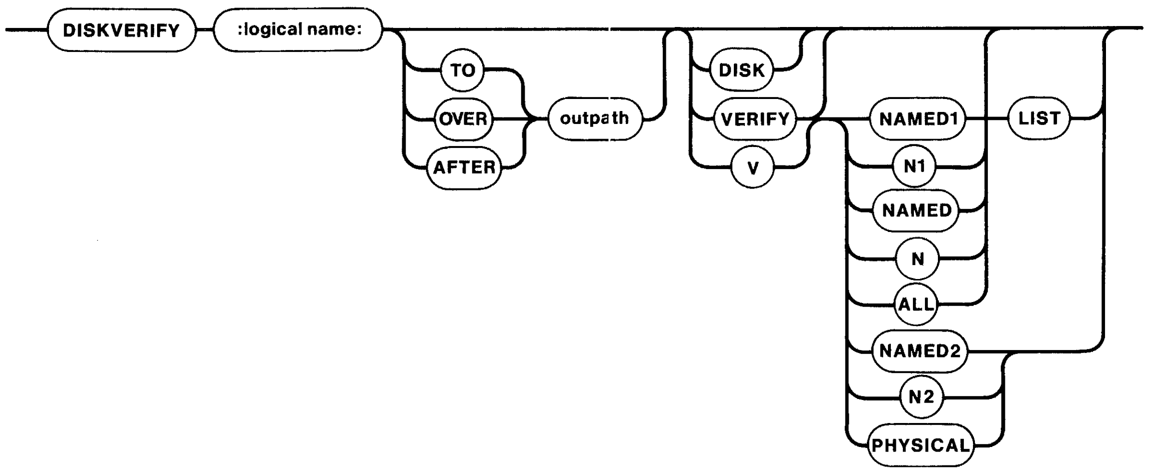
- As an interactive program which allows you to check and modify information on the volume by entering individual disk verification commands.

To take full advantage of the capabilities of the disk verification utility, you must be familiar with the structure of iRMX 86 named volumes. Appendix A contains detailed information about the volume structure. If you are unfamiliar with the iRMX 86 volume structure, you should avoid using the individual disk verification commands. When used carelessly, these commands can make your volumes unusable.

However, even if you know nothing about iRMX 86 volume structures, you can still use the utility as a single command to verify that the data structures on an iRMX 86 volume are valid.

INVOCATION

The format of the Human Interface command used to invoke the disk verification utility is as follows:



1120

where:

:logical-name:	Logical name of the secondary storage device containing the volume.
TO	Copies the output from the disk verification utility to the specified file. If no preposition is specified, TO :CO: is the default.
OVER	Copies the output from the disk verification utility over the specified file.
AFTER	Appends the output from the disk verification utility to the end of the specified file.
outpath	Pathname of the file to receive the output from the disk verification utility. If you omit this parameter and the TO/OVER/AFTER preposition, the utility copies the output to the console screen (TO :CO:). You cannot direct the output to a file on the volume being verified. If you attempt this, the utility returns an E$NOT_CONNECTION error message.

DISK	Displays the attributes of the volume being verified. If you specify this parameter, the utility performs the disk function and returns control to you at the Human Interface level. You can then enter any Human Interface command provided that the device verified is not the system device. Refer to the description of the DISK command in Chapter 2 for more information. Any parameter after this one is ignored.
VERIFY or V	Performs a verification of the volume. This verification function and the associated options are described in detail in the "VERIFY Command" section of Chapter 2. If you specify this parameter and omit the options, the utility performs the NAMED verification. If you specify this parameter, the utility performs the verification function and returns control to you at the Human Interface level. You can then enter any Human Interface command if the device is not the system device (:sd:). If you omit this parameter and the DISK parameter, the utility displays a header message and the utility prompt (*). You can then enter any of the disk verification commands listed in Chapter 2.
NAMED1 or N1	VERIFY option that applies to named volumes only. This option checks the fnodes of the volume to ensure that they match the directories in terms of file type and file heirarchy. This option also checks the information in each fnode to ensure that it is consistent. Refer to the description of the VERIFY command in Chapter 2 for more information.
NAMED or N	VERIFY option that performs both the NAMED1 and NAMED2 verification functions on a named volume. If you omit the VERIFY option, NAMED is the default option.
ALL	VERIFY option that applies to both named and physical volumes. For named volumes, this option performs both the NAMED and PHYSICAL verification functions. For physical volumes, this option performs the PHYSICAL verification function.
NAMED2 or N2	VERIFY option that applies to named volumes only. This option checks the allocation of fnodes on the volume, checks the allocation of space on the volume, and verifies that the fnodes point to the correct locations on the volume. Refer to the description of the VERIFY command in Chapter 2 for more information.

PHYSICAL	VERIFY option that applies to both named and physical volumes. This option reads all blocks on the volume and checks for I/O errors.
LIST	VERIFY option that you can use with those VERIFY parameters that, either explicitly or implicitly, specify the NAMED1 parameter. When you use this option, the file information generated by VERIFY is displayed for every file on the volume, even if the file contains no errors. Refer to the description of the VERIFY command in Chapter 2 for more information.

OUTPUT

When you enter the DISKVERIFY command, the utility responds by displaying the following line:

```
iRMX 86 DISK VERIFY UTILITY, Vx.x
Copyright <year> Intel Corporation
```

where Vx.x is the version number of the utility. If you specify the VERIFY or V parameter in the DISKVERIFY command, the utility performs a verification of the volume and copies the verification information to the console (or to the file specified by the outpath parameter). The verification information is the same as that produced by the VERIFY utility command. Refer to the description of the VERIFY command in Chapter 2 for a description of the verification output. After generating the verification output, the utility returns control to the Human Interface, which prompts you for more Human Interface commands. The following is an example of such a DISKVERIFY command:

```
-DISKVERIFY :F1: VERIFY NAMED2
iRMX 86 DISK VERIFY UTILITY ,  Vx.x
DEVICE NAME = F1          : DEVICE SIZE = 0003E900 : BLOCK SIZE = 0080

'NAMED2'  VERIFICATION

    BIT MAPS O.K.

-
```

However, if you omit the VERIFY (or V) parameter from the DISKVERIFY command, the utility does not return control to the Human Interface. Instead, it issues an asterisk (*) as a prompt and waits for you to enter individual DISKVERIFY commands. The following is an example of such a DISKVERIFY command:

```
-DISKVERIFY :F1:
iRMX 86 DISK VERIFY UTILITY ,  Vx.x
*
```

After you receive the asterisk prompt, you can enter any of the DISKVERIFY commands listed in the next section. If you enter anything else, the utility will display an error message.

NOTE

Although DISKVERIFY many be used to verify the system device (:sd:), you must be aware that all connections to the device are deleted by the Operating System. After you exit, you will have to reboot the Operating System.

INVOCATION ERROR MESSAGES

Message	Description
argument error	The VERIFY option you specified is not valid.
0021 : E$FILE_NOT_EXIST or 0040 : E$LOGICAL_NAME_SYNTAX	The logical name you specified was not surrounded by colons (:), was longer than 12 characters, or contained invalid characters.
0045 : E$LOG_NAME_NEXIST or <logical name>, logical name does not exist	You specified a nonexistent <logical name> in either the :logical name: parameter or the outpath parameter.
8042 : E$NOT_CONNECTION	You attempted to direct output to a file on the volume being verified.
command syntax error	You made a syntax error when entering the command.
device size inconsistent size in volume label = <value1> : computed size = <value2>	When the disk verification utility computed the size of the volume, the size it computed did not match the information recorded in the iRMX 86 volume label. It is likely that the volume label contains invalid or corrupted information. This error is not a fatal error, but it is an indication that further error conditions may result during the verification session. You may have to reformat the volume or use the disk verification utility to modify the volume label.
not a named disk	You tried to perform a NAMED, NAMED1, or NAMED2 verification on a physical volume.

***

CHAPTER 2
DISKVERIFY COMMANDS

When the disk verification utility issues the asterisk prompt, you can enter individual DISKVERIFY commands to examine or change system information on the volume. This process usually involves reading a portion of the volume into a buffer, modifying that buffer, and writing the information back to the volume. This chapter describes the commands that allow you to perform these operations.

The commands in this chapter are presented in alphabetical order, without regard to function. Before describing the individual commands, this chapter discusses command names, parameters, input radices, and error messages. It also provides a command dictionary.

COMMAND NAMES

When you enter a DISKVERIFY command, you can enter the command name or command name abbreviation as listed in this chapter, or you can enter any portion of the command name that uniquely identifies the command from all other DISKVERIFY commands.

For example, when specifying the DISPLAYFNODE command, you can enter the command name as:

```
DISPLAYFNODE
DF
DISPLAYF
```

or any other partial form of the word DISPLAYFNODE that contains at least the characters DISPLAYF.

PARAMETERS

Several DISKVERIFY commands have parameters which this chapter describes as being in the form:

```
keyword = value
```

Even though the individual command descriptions do not mention this, you can also enter these parameters in the form:

```
keyword (value)
```

For example, the following are two acceptable ways of specifying a FREE command:

```
FREE FNODE = 10

FREE FNODE (10)
```

INPUT RADICES

DISKVERIFY always produces numerical output in hexadecimal format. However, when you provide input to DISKVERIFY, you can specify the radix of numerical quantities by including a radix character immediately after the number. The valid radix characters include:

radix	character	example
hexadecimal	h or H	16h, 7CH
decimal	t or T	23t, 100T
octal	o, O, q, or Q	27o, 33Q

If you omit the radix character, DISKVERIFY assumes the number is hexadecimal.

ABORTING DISKVERIFY COMMANDS

You can abort the following DISKVERIFY commands by typing a CONTROL-C (press the CONTROL key, and while holding it down, press the C key) at your keyboard.

```
DISK
DISPLAYBYTE
DISPLAYDIRECTORY
DISPLAYFNODE
DISPLAYNEXTBLOCK
DISPLAYPREVIOUSBLOCK
DISPLAYWORD
LISTBADBLOCKS
SUBSTITUTEBYTE
SUBSTITUTEWORD
VERIFY
```

CONTROL-C terminates the command and returns control to the disk verification utility.

COMMAND ERROR MESSAGES

Each DISKVERIFY command can generate a number of error messages which indicate errors in the way you specified the command or problems with the volume itself. The messages for each command are listed with the command itself. However, the following messages can also occur with many of the commands:

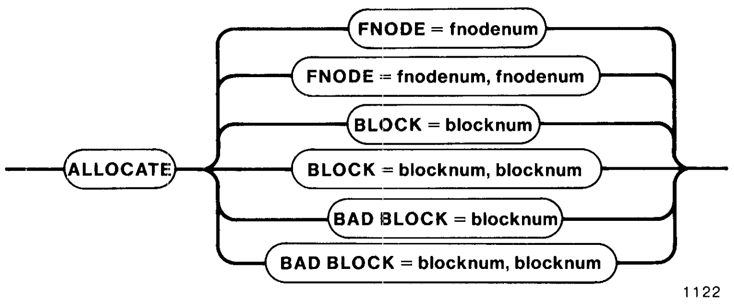
Message	Description
block I/O error	The utility attempted to read or write a block on the volume and found that the block was physically flawed. Thus it cannot complete the requested command.
command syntax error	You made a syntax error in a command.
illegal command	The command you specified is not a valid DISKVERIFY command.
fnodes/space map fnode fnode data inconsistent	The command cannot find the initial fnode files. See Appendix A for more information concerning the initial fnode files.
not a named disk	When the disk verification utility begins processing, it obtains some information from the iRMX 86 volume label. If the label contains invalid information, the utility (in some cases) can assume that a named volume is a physical volume. If this occurs, the commands that apply to named volumes <u>only</u> (such as DISPLAYFNODE, DISPLAYDIRECTORY, and VERIFY NAMED) issue this message. If you are convinced that your volume is indeed a named volume, this message may indicate that the iRMX 86 volume label is corrupted.
seek error	The utility unsuccessfully attempted to seek to a location on the volume. This error normally results from invalid information in the iRMX 86 volume label or in the fnodes.

COMMAND DICTIONARY

Command	Synopsis	Page
ALLOCATE	Marks a particular fnode or volume block as allocated	2-5
DISK	Lists the attributes of the volume	2-8
DISPLAYBYTE	Displays the working buffer in byte format	2-10
DISPLAYDIRECTORY	Displays directory contents	2-13
DISPLAYFNODE	Displays fnode information	2-15
DISPLAYNEXTBLOCK	Displays the "next" volume block	2-20
DISPLAYPREVIOUSBLOCK	Displays the "previous" volume block	2-21
DISPLAYWORD	Displays the working buffer in word format	2-22
EXIT	Exits the disk verification utility	2-25
FREE	Marks a particular fnode or volume block as free	2-26
HELP	Lists the DISKVERIFY commands	2-28
LISTBADBLOCKS	Displays all the bad blocks on the volume	2-29
miscellaneous commands	Perform useful arithmetic and conversion functions; the commands include ADD, SUB, MUL, DIV, MOD, HEX, DEC, ADDRESS, and BLOCK.	2-30
QUIT	Exits the disk verification utility	2-36
READ	Reads a volume block into the working buffer	2-37
SAVE	Writes the updated fnode, free space maps, and bad block maps to the volume	2-38
SUBSTITUTEBYTE	Modifies the contents of the working buffer in byte format	2-40
SUBSTITUTEWORD	Modifies the contents of the working buffer in word format	2-43
VERIFY	Verifies the volume	2-46
WRITE	Writes the working buffer to the volume	2-55

ALLOCATE COMMAND

This command designates file descriptor nodes (fnodes) and volume blocks as allocated. You can also use this command to designate one or a range of volume blocks as "bad." The format of the ALLOCATE command is as follows:


ALLOCATE
FNODE = fnodenum
FNODE = fnodenum, fnodenum
BLOCK = blocknum
BLOCK = blocknum, blocknum
BAD BLOCK = blocknum
BAD BLOCK = blocknum, blocknum
1122

INPUT PARAMETERS

fnodenum	Number of the fnode to allocate. This number can range from 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted. Two fnode values seperated by a comma signifies a range of fnodes.
blocknum	Number of the volume block to allocate. This number can range from 0 through (max blocks - 1), where max blocks is the number of volume blocks in the volume. Two block numbers seperated by a comma signifies a range of fnodes.

OUTPUT

If you are using ALLOCATE to allocate fnodes, ALLOCATE displays the following message:

```
<fnodenum>, fnode marked allocated
```

where <fnodenum> is the number of the fnode that the utility designated as allocated.

If you are using ALLOCATE to allocate volume blocks, ALLOCATE displays the following message:

```
<blocknum>, block marked allocated
```

where <blocknum> is the number of the volume block that the utility designated as allocated.

If you are using ALLOCATE to designate one or more volume blocks as "bad," ALLOCATE displays the following message:

```
<blocknum>, block marked bad
```

where <blocknum> is the number of the volume block that the utility designated as "bad." If this block was not allocated before you attempt to designate it as "bad", ALLOCATE also displays:

```
<blocknum>, block marked allocated
```

ALLOCATE checks the allocation status of fnodes or blocks before allocating them. Therefore, if you specify ALLOCATE for a block or fnode that is already allocated, ALLOCATE returns one of the following messages:

```
<fnodenum>, fnode already marked allocated

<blocknum>, block already marked allocated

<blocknum>, block already marked bad
```

DESCRIPTION

Fnodes are data structures on the volume that describe the files on the volume. They are created when the volume is formatted. An allocated fnode is one that represents an actual file. ALLOCATE designates fnodes as allocated by updating the FLAGS field of the fnode and free fnodes map file with this information.

An allocated volume block is a block of data storage that is part of a file; it is not available to be assigned to a new file. ALLOCATE designates volume blocks as allocated by updating the volume free space map with this information.

When you use ALLOCATE to designate bad blocks, it not only updates the volume free space map, but it also marks the bit as "bad" in the bad blocks file.

ERROR MESSAGES

Message	Description
argument error	You made a syntax error in the command or specified a nonnumeric character in the blocknum or fnodenum parameter.
<blocknum>, block out of range	The block number that you specified was larger than the largest block number in the volume.

<fnodenum>, fnode out of range	The fnode number that you specified was larger than the largest fnode number in the volume.
no badblocks file	Your system does not have a bad blocks file. This message could appear because you used a Release 4 or earlier version of the Human Interface command, FORMAT, when you formatted your disk.

DISK COMMAND

This command displays the attributes of the volume being verified. You can abort this command by typing a CONTROL-C (press the CONTROL key, and while holding it down, press the C key). The format of the DISK command is as follows:

x-225

OUTPUT

The output of the DISK command depends on whether the volume is formatted as a physical or named volume. For a physical volume, the DISK command displays the following information:

```
    Device name = <devname>
       Physical disk
          Device gran = <devgran>
           Block size = <devgran>
         No of blocks = <numblocks>
          Volume size = <size>
```

where:

<devname>	Name of the device containing the volume. This is the physical name of the device, as specified in the ATTACHDEVICE Human Interface command.
<devgran>	Granularity of the device, as defined in the device unit information block (DUIB) for the device. Refer to the iRMX 86 CONFIGURATION GUIDE for more information about DUIBs. For physical devices, this is also the volume block size.
<numblocks>	Number of volume blocks in the volume.
<size>	Size of the volume, in bytes.

For a named volume, the DISK command displays the following information:

```
    Device name = <devname>
       Named disk, Volume name = <volname>
          Device gran = <devgran>
           Block size = <volgran>
         No of blocks = <numblocks>  : No of Free blocks = <numfreeblocks>
          Volume size = <size>
           Interleave = <inleave>
       Extension Size = <xsize>
         No of fnodes = <numfnodes>  : No of Free fnodes = <numfreefnodes>
```

The <devname>, <devgran>, <numblocks>, and <size> fields are the same as for physical files. The remaining fields are as follows:

<volname>	Name of the volume, as specified when the volume was formatted.
<volgran>	Volume granularity, as specified when the volume was formatted.
<numfreeblocks>	Number of available volume blocks in the volume.
<inleave>	The interleave factor for a named volume.
<xsize>	Size, in bytes, of the extension data portion of each file descriptor node (fnode).
<numfnodes>	Number of fnodes in the volume. The fnodes were created when the volume was formatted.
<numfreefnodes>	Number of available fnodes in the named volume.

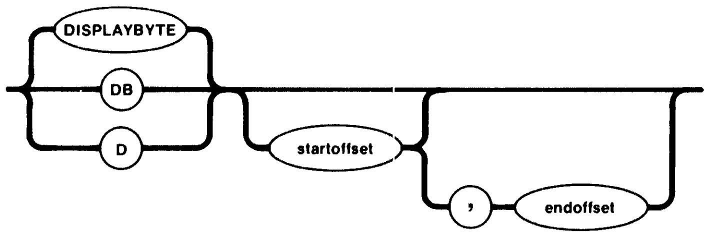
Refer to Appendix A or to the description of the FORMAT command in the iRMX 86 OPERATOR'S MANUAL for more information about the named disk fields.

DESCRIPTION

The DISK command displays the attributes of the volume. The format of the output from DISK depends on whether the volume is formatted as a named or physical volume.

DISPLAYBYTE COMMAND

This command displays the specified portion of the working buffer in byte format. It displays the buffer in 16-byte rows. You can abort this command by typing a CONTROL-C (press the CONTROL key, and while holding it down, press the C key). The format of the DISPLAYBYTE command is as follows:



x-226

INPUT PARAMETERS

startoffset	Number of the byte, relative to the start of the buffer, which begins the display. DISPLAYBYTE starts the display with the row containing the specified offset. If you omit this parameter, DISPLAYBYTE starts displaying from the beginning of the working buffer.
endoffset	Number of the byte, relative to the start of the buffer, which ends the display. If you omit this parameter, DISPLAYBYTE displays only the row indicated by startoffset. However, if you omit both startoffset and endoffset, DISPLAYBYTE displays the entire working buffer.

OUTPUT

In response to the command, DISPLAYBYTE displays the specified portion of the working buffer in rows, with 16 bytes displayed in each row. Figure 2-1 illustrates the format of the display.

As Figure 2-1 shows, DISPLAYBYTE begins by listing the block number of the data from the contents of the buffer. It then lists the specified portion of the buffer, providing the column numbers as a header and beginning each row with the relative address of the first byte in the row. It also includes, at the right of the listing, the ASCII equivalents of the bytes, if the ASCII equivalents are printable characters. (If a byte is not a printable character, DISPLAYBYTE displays a period in the corresponding position.)

```
BLOCK NUMBER = blocknum

offset  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   ASCII STRING
 0000  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   ................
 0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
 0020  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF   ................
  .                .                      .                     .
  .                .                      .                     .
  .                .                      .                     .
```

Figure 2-1. DISPLAYBYTE Format

DESCRIPTION

DISKVERIFY manintains a working buffer for READ and WRITE commands. The size of the buffer is equal to the volume's granularity value. After you read a volume block of memory into the working buffer with the READ command, you can display part or all of that memory, in byte format, by entering the DISPLAYBYTE command. DISPLAYBYTE displays the hexadecimal value for each byte in the specified portion of the buffer.

If you omit all parameters, DISPLAYBYTE displays the entire block stored in the working buffer.

ERROR MESSAGES

Message	Description
argument error	You made a syntax error in the command or specified a nonnumeric character in one of the offset parameters.
invalid offset	You either specified a larger value for startoffset than for endoffset or you specified an offset value that was larger than the number of bytes in the block.

EXAMPLES

Assuming that the volume granularity is 128 bytes and assuming that you have read block 20h into the working buffer with the READ command, the following command displays that block.

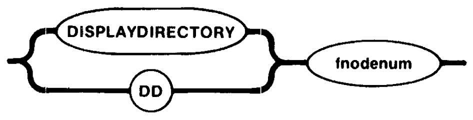
```
  *DISPLAYBYTE

  BLOCK NUMBER = 20

  offset  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  ASCII STRING
   0000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
   0010  00 00 08 00 00 00 00 00 00 00 01 00 0F FF FF 00  ................
   0020  00 00 00 00 00 05 00 00 00 00 25 00 08 01 FF FF  ..........%.....
   0030  25 1F 00 00 2E 00 00 00 25 1F 00 00 2B 00 00 00  %.......%...+...
   0040  01 00 00 00 01 00 80 00 00 00 00 00 00 00 00 00  ................
   0050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
   0060  00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  ................
   0070  00 00 00 00 01 00 0F FF FF 00 00 00 00 00 00 05  ................
  *
```

The following command displays the portion of the block containing the offsets 31h through 45h.

```
  *D 31, 45

  BLOCK NUMBER = 20

  offset  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  ASCII STRING
   0030  25 1F 00 00 2E 00 00 00 25 1F 00 00 2B 00 00 00  %.......%...+...
   0040  01 00 00 00 01 00 80 00 00 00 00 00 00 00 00 00  ................
  *
```

DISPLAYDIRECTORY COMMAND

This command lists all the files contained in a directory. You can abort this command by typing a CONTROL-C (press the CONTROL key, and while holding it down, press the C key). The format of the DISPLAYDIRECTORY command is as follows:



x-227

INPUT PARAMETER

fnodenum	Number of the fnode that corresponds to a directory file. This number can range from 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted. DISPLAYDIRECTORY lists all files contained in this directory.

OUTPUT

In response to the command, DISPLAYDIRECTORY lists information about all files contained in the specified directory. The format of this display is as follows:

```
FILE NAME FNODE   TYPE    FILE NAME FNODE   TYPE    FILE NAME FNODE TYPE

<filenam> <fnode> <type>  <filenam> <fnode> <type>  <filenam> <fnode> <type>
<filenam> <fnode> <type>  <filenam> <fnode> <type>  <filenam> <fnode> <type>
         .                         .                          .
         .                         .                          .
         .                         .                          .
```

where:

<filenam>	Name of the file contained in the directory.
<fnode>	Number of the fnode that describes the file.

<type> Type of the file. The <type> can be:

Type of file	Description
DATA	data files
DIR	directory files
SMAP	volume free space map
FMAP	free fnodes map
BMAP	bad blocks map
VLAB	volume label file

DESCRIPTION

DISPLAYDIRECTORY displays a list of files contained in the specified directory, along with their fnode numbers and types. With this information you can use other disk verification commands to examine the individual files.

ERROR MESSAGES

Message	Description
argument error	You specified a nonnumeric character in the fnodenum parameter.
<fnodenum>, fnode not allocated	The number you specified for the fnodenum parameter does not correspond to an allocated fnode. This fnode does not represent an actual file.
<fnodenum>, not a directory fnode	The number you specified for the fnodenum parameter is not an fnode for a directory file.
<fnodenum>, fnode out of range	The number you specified for the fnodenum parameter is larger than the largest fnode number on the volume.

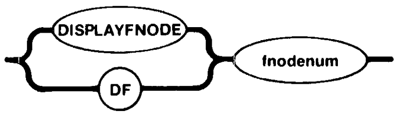
EXAMPLE

The following command lists the files contained in the directory whose fnode is fnode 9.

```
*DISPLAYDIRECTORY 5

FILE NAME  FNODE TYPE    FILE NAME  FNODE TYPE    FILE NAME FNODE TYPE

change.p86  0006 DATA    samp.txt    0007 DATA    NAMES      0008  DIR
PLACES      0009 DIR     change.plm  000A DATA

*
```

DISPLAYFNODE COMMAND

This command displays the fields associated with an fnode. You can abort this command by typing a CONTROL-C (press the CONTROL key, and while holding it down, press the C key). The format of the DISPLAYFNODE command is as follows:



x-228

INPUT PARAMETER

fnodenum	Number of the fnode to be displayed. This number can range from 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted.

OUTPUT

In response to this command, DISPLAYFNODE displays the fields of the specified fnode. The format of the display is as follows:

```
Fnode number = <fnodenum>
                    flags : <flgs>
                     type : <typ>
       file gran/vol gran : <gran>
                    owner : <own>
 create,access,mod times : <crtime>, <acctime>, <modtime>
               total size : <totsize>
             total blocks : <totblks>
         block pointer(1) : <blks>, <blkptr>
         block pointer(2) : <blks>, <blkptr>
         block pointer(3) : <blks>, <blkptr>
         block pointer(4) : <blks>, <blkptr>
         block pointer(5) : <blks>, <blkptr>
         block pointer(6) : <blks>, <blkptr>
         block pointer(7) : <blks>, <blkptr>
         block pointer(8) : <blks>, <blkptr>
                this size : <thissize>
                 id count : <count>
              accessor(1) : <access>, <id>
              accessor(2) : <access>, <id>
              accessor(3) : <access>, <id>
                   parent : <prnt>
                   aux(*) : <auxbytes>
```

where:

<fnodenum>	Number of the fnode being displayed. If the fnode does not describe an actual file (that is, if it is not allocated), the following message appears next to this field:

*** ALLOCATION STATUS BIT IN THIS FNODE NOT SET ***

In this case, the fnode fields are normally set to zero.

<flgs> A word defining the attributes of the file. Significant bits of this word are:

Bit	Meaning
0	Allocation status. This bit is set to 1 for allocated fnodes and set to 0 for free fnodes.
1	Long or short file attribute. This bit is set to 1 for long files and set to 0 for short files.
5	Modification attribute. This bit is set to 1 whenever a file is modified.
6	Deletion attribute. This bit is set to 1 to indicate a temporary file or a file that is going to be deleted.

The DISPLAYFNODE command displays a message next to this field to indicate whether the file is a long or short file.

<typ> Type of file. This field contains a value and a message. The possible values and messages are:

Value	Message
00	fnode file
01	volume map file
02	fnode map file
03	account file
04	bad block file
06	directory file
08	data file
09	volume label file

<gran> File granularity, specified as a multiple of the volume granularity.

<own> User ID of the owner of the file.

<crtime> <acctime> <modtime>	Time and date of file creation, last access, and last modification. These values are expressed as the time since January 1, 1978.
<totsize>	Total size, in bytes, of the actual data in the file.
<totblks>	Total number of volume blocks used by the file, including indirect block overhead.
<blks>, <blkptr>	Values which identify the data blocks of the file. For short files, each <blks> parameter indicates the number of volume blocks in the data block and each <blkptr> is the number of the first such volume block. For long files, each <blks> parameter indicates the number of volume blocks pointed to by an indirect block and each <blkptr> is the block number of the indirect block.
<thissize>	Size in bytes of the total data space allocated to the file, minus any space used for indirect blocks.
<count>	Number of user IDs associated with the file.
<access>, <id>	Each pair of fields indicate the access rights for the file (access) and the ID of the user who has that access ID. Bits in the <access> field are set to indicate the following access rights:

Bit	Data File Operation	Directory Operation
0	delete	delete
1	read	display
2	append	add entry
3	update	change entry

The first ID listed is the owning user's ID.

<prnt>	Fnode number of the directory file which contains the file.
<auxbytes>	Auxiliary bytes associated with the file.

Appendix A contains a more detailed description of the fnode fields.

DESCRIPTION

Fnodes are system data structures on the volume that describe the files on the volume. The fnode structures are created when the volume is formatted. Each time a file is created on the volume, the iRMX 86 Basic I/O System allocates an fnode for the file and fills in the fnode fields to describe the file. The DISPLAYFNODE command allows you to examine these fnodes and determine where the data for each file resides.

ERROR MESSAGES

Message	Description
argument error	You entered a value for the fnodenum parameter that was not a legitimate fnode number.
<fnodenum>, fnode out of range	The number you specified for the fnodenum parameter is larger than the largest fnode number on the volume.

EXAMPLE

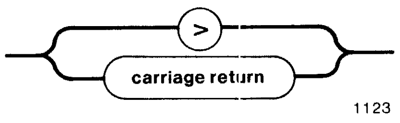
The following example displays fnode 6 (root directory) of a volume. Notice that the parent of the root directory is the root directory itself.

```
*DISPLAYFNODE 6

Fnode number = 6
                   flags : 0025  =>  short file
                    type : 06  =>  directory file
      file gran/vol gran : 01
                   owner : 0000
 create,access,mod times : 00000017, 00000158, 00000018
              total size : 00000400
            total blocks : 00000001
        block pointer(1) : 0001, 000050
        block pointer(2) : 0000, 000000
        block pointer(3) : 0000, 000000
        block pointer(4) : 0000, 000000
        block pointer(5) : 0000, 000000
        block pointer(6) : 0000, 000000
        block pointer(7) : 0000, 000000
        block pointer(8) : 0000, 000000
               this size : 00000400
                id count : 0001
             accessor(1) : 0F, 0000
             accessor(2) : 00, 0000
```

```
                  accessor(3) : 00, 0000
                       parent : 0006
                       aux(*) : 000000

      *
```

DISPLAYNEXTBLOCK

This command displays the "next" volume block. (The "next" volume block is the block which immediately follows the block currently in the working buffer.) The display format can be either WORD or BYTE. The utility remembers the mode in which you displayed the volume block currently in the working buffer and it displays the next block in that format. So, if you used DISPLAYBYTE to display the current volume block, the next volume block appears in BYTE format; if you used DISPLAYWORD, the next volume block appears in WORD format. DISPLAYNEXTBLOCK uses the BYTE format as a default if you have not yet displayed a volume block. You can abort this command by typing a CONTROL-C (press the CONTROL key, and while holding it down, press the C key).



OUTPUT

In response to the command, DISPLAYNEXTBLOCK reads the "next" volume block into the working buffer and displays it on the screen. The format (WORD or BYTE) of the new volume block depends upon the command you used to display the current volume block.

DESCRIPTION

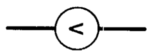
The DISPLAYNEXTBLOCK command copies the "next" volume block from the volume to the working buffer and displays it at your terminal. It destroys any data currently in the working buffer. Once the block is in the working buffer, you can use SUBSTITUTEBYTE and SUBSTITUTEWORD to change the data in the block. Finally, you can use the WRITE command to write the modified block back out to the volume.

NOTE

If you specify the DISPLAYNEXTBLOCK command at the end of the volume, the utility "wraps around" and displays the first block in the volume.

DISPLAYPREVIOUSBLOCK

This command displays the "previous" volume block. (The "previous" volume block is the block which immediately precedes the block currently in the working buffer.) The display format can be either WORD or BYTE. The utility remembers the mode in which you displayed the volume block currently in the working buffer and it displays the previous block in that format. So, if you used DISPLAYBYTE to display the current volume block, the previous volume block appears in BYTE format; if you used DISPLAYWORD, the previous volume block appears in WORD format. DISPLAYPREVIOUSBLOCK uses the BYTE format as a default if you have not yet displayed a volume block. You can abort this command by typing a CONTROL-C (press the CONTROL key, and while holding it down, press the C key).



1121

OUTPUT

In response to the command, DISPLAYPREVIOUSBLOCK reads the "previous" volume block into the working buffer and displays it on the screen. The format (WORD or BYTE) of the new volume block depends upon the command you used to display the current volume block.

DESCRIPTION

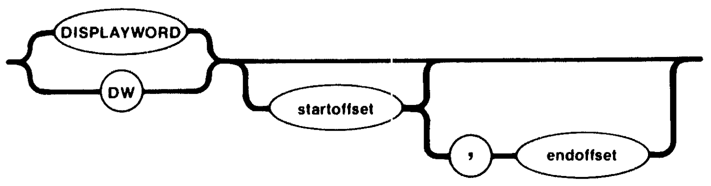
The DISPLAYPREVIOUSBLOCK command copies the "previous" volume block from the volume to the working buffer and displays it at your terminal. It destroys any data currently in the working buffer. Once the block is in the working buffer, you can use SUBSTITUTEBYTE and SUBSTITUTEWORD to change the data in the block. Finally, you can use the WRITE command to write the modified block back out to the volume.

NOTE

> If you specify the DISPLAYPREVIOUSBLOCK command at the beginning of the volume, the utility "wraps around" and displays the last block in the volume.

DISPLAYWORD COMMAND

This command displays the specified portion of the working buffer in word format. It displays the buffer in 8-word rows. You can abort this command by typing a CONTROL-C (press the CONTROL key, and while holding it down, press the C key). The format of the DISPLAYWORD command is as follows:



x-229

INPUT PARAMETERS

startoffset	Number of the byte, relative to the start of the buffer, which begins the display. DISPLAYWORD starts the display with the row containing the specified offset. If you omit this parameter, DISPLAYWORD starts displaying from the beginning of the working buffer.
endoffset	Number of the byte, relative to the start of the buffer, which ends the display. If you omit this parameter, DISPLAYWORD displays only the row indicated by startoffset. However, if you omit both startoffset and endoffset, DISPLAYWORD displays the entire working buffer.

OUTPUT

In response to the command, DISPLAYWORD displays the specified portion of the working buffer in rows, with 8 words displayed in each row. Figure 2-2 illustrates the format of the display.

As Figure 2-2 shows, DISPLAYWORD begins by listing the block number of the data being displayed. It then lists the specified portion of the buffer, providing the column numbers as a header and beginning each row with the relative address of the first word in the row.

```
BLOCK NUMBER = blocknum

offset     0     2     4     6     8     A     C     E
 0000    0100  0302  0504  0706  0908  0B0A  0D0C  0F0E
 0010    0000  0000  0000  0000  0000  0000  0000  0000
 0020    FFFF  FFFF  FFFF  FFFF  FFFF  FFFF  FFFF  FFFF
  .                 .                    .
  .                 .                    .
  .                 .                    .
```

Figure 2-2. DISPLAYWORD Format

DESCRIPTION

After you read a block of memory into the working buffer with the READ command, you can display part or all of that memory, in word format, by entering the DISPLAYWORD command. DISPLAYWORD displays the hexadecimal value for each word in the specified portion of the buffer.

If you omit all parameters, DISPLAYWORD displays the entire block stored in the working buffer.

ERROR MESSAGES

Message	Description
argument error	You made a syntax error in the command or specified a nonnumeric character in one of the offset parameters.
invalid offset	You either specified a larger value for startoffset than for endoffset or you specified an offset value that was larger than the number of bytes in the block.

EXAMPLES

Assuming that the volume granularity is 128 bytes and that you have read block 20h into the working buffer with the READ command, the following command displays that block in word format.

```
*DISPLAYWORD

BLOCK NUMBER = 20

offset    0     2     4     6     8     A     C     E
 0000    0000  0000  0000  0000  0000  0000  0000  0000
 0010    0000  0080  0000  0000  0000  0001  FF0F  00FF
 0020    0000  0000  0500  0000  0000  0025  0108  FFFF
 0030    1F25  0000  002E  0000  1F25  0000  002B  0000
 0040    0001  0000  0001  0080  0000  0000  0000  0000
 0050    0000  0000  0000  0000  0000  0000  0000  0000
 0060    0000  0000  0000  0000  0000  0000  0080  0000
 0070    0000  0000  0001  FF0F  00FF  0000  0000  0500
*
```

The following command displays the portion of the block that contains the offsets 31h through 45h (words beginning at odd addresses).

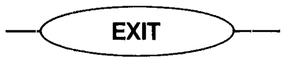
```
*DW 31, 45

BLOCK NUMBER = 20

offset    0     2     4     6     8     A     C     E
 0031    001F  2E00  0000  2500  001F  2B00  0000  0100
 0041    0000  0100  8000  0000  0000  0000  0000  0000
*
```

The following command displays the portion of the block that contains the offsets 30h through 45h (words beginning at even addresses).

```
*DISPLAYWORD 30, 45

BLOCK NUMBER = 20

offset    0     2     4     6     8     A     C     E
 0030    1F25  0000  002E  0000  1F25  0000  002B  0000
 0040    0001  0000  0001  0080  0000  0000  0000  0000
*
```

EXIT COMMAND

This command exits the disk verification utility and returns control to the Human Interface command level. The format of the EXIT command is as follows:

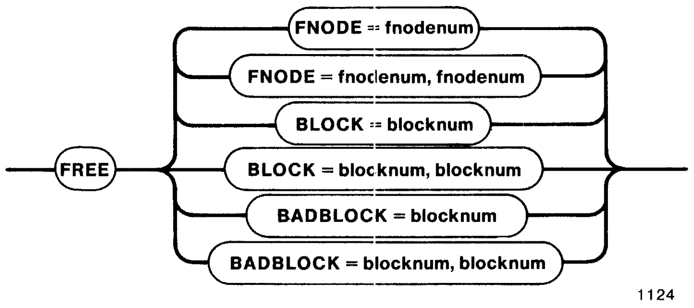


x-286

This command is identical to the QUIT command.

FREE COMMAND

This command designates fnodes and volume blocks as free (unallocated). It also removes volume blocks from the bad blocks file. The format of the FREE command is as follows:


FREE
FNODE = fnodenum
FNODE = fnodenum, fnodenum
BLOCK = blocknum
BLOCK = blocknum, blocknum
BADBLOCK = blocknum
BADBLOCK = blocknum, blocknum
1124

INPUT PARAMETERS

fnodenum	Number of the fnode to free. This number can range from 0 through (max fnodes - 1), where max fnodes is the number of fnodes defined when the volume was originally formatted.
blocknum	Number of the volume block to free. This number can range from 0 through (max blocks - 1), where max blocks is the number of volume blocks in the volume.

OUTPUT

If you are using FREE to deallocate fnodes, FREE displays the following message:

```
    <fnodenum>, fnode marked free
```

where <fnodenum> is the number of the fnode that the utility designated as free.

If you are using FREE to deallocate volume blocks, FREE displays the following message:

```
    <blocknum>, block marked free
```

where <blocknum> is the number of the volume block that the utility designated as free.

If you are using FREE to designate one or more "bad" blocks as "good", FREE displays the following message:

```
    <blocknum>, block marked good
```

where <blocknum> is the number of the volume block that the utility designated as "good."

FREE checks the allocation status of fnodes or blocks before freeing them. Therefore, if you specify FREE for a block or fnode that is already allocated, FREE returns one of the following messages:

```
<fnodenum>, fnode already marked free

<blocknum>, block already marked free

<blocknum>, block already marked good
```

DESCRIPTION

Free fnodes are fnodes for which no actual files exist. FREE designates fnodes as free by updating both the FLAGS field of the fnode and the free fnodes map file.

Free volume blocks are blocks that are not part of any file; they are available to be assigned to any new or current file. FREE designates volume blocks as free by updating the volume free space map.

When you use the FREE command to designate one or more bad blocks as "good", it removes the block number from the bad blocks file. However, it does not update the volume free space map which designates the block as available for use. Therefore, DISKVERIFY does not know that the block is free.

ERROR MESSAGES

Message	Description
argument error	You made a syntax error in the command or specified a nonnumeric character in the blocknum or fnodenum parameter.
<blocknum>, block out of range	The block number that you specified was larger than the largest block number in the volume.
<fnodenum>, fnode out of range	The fnode number that you specified was larger than the largest fnode number in the volume.
no badblocks file	Your system does not have a bad blocks file. This message could appear because you used an old version of the Human Interface command, FORMAT, when you formatted your disk.

HELP COMMAND

This command lists all available DISKVERIFY commands and provides a short description of each command. The format of this command is:

x-287

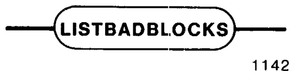
OUTPUT

In response to this command, HELP displays the following information:

```
                  read : read a disk block into the buffer
     display byte/word : display the buffer (byte/word format)
    display next block : read and display 'next' volume block
display previous block : read and display 'previous' volume block
  substitute byte/word : modify the buffer (byte/word format)
                 write : write to the disk block from the buffer
                verify : verify the disk
                  save : save free fnodes, free space, and bad blocks maps
         allocate/free : allocate/free fnodes, space blocks, or bad blocks
         listbadblocks : list bad blocks on the volume
                  disk : display disk attributes
             exit/quit : quit disk verify
             Control-C : abort the command in progress
     display directory : display the directory contents
         display fnode : display fnode information
miscellaneous commands :
               address : convert block number to absolute address
               block   : convert absolute address to block number
               hex/dec : display number as hexadecimal/decimal number
   add/sub/mul/div/mod : arithmetic operations on unsigned numbers
```

LISTBADBLOCKS

This command displays all the bad blocks on a named volume. You can abort this command by typing a CONTROL-C (press the CONTROL key, and while holding it down, press the C key). The format of the LISTBADBLOCKS command is as follows:



OUTPUT

In response to this command, LISTBADBLOCKS displays up to eight columns of block numbers that you specified as "bad." Figure 2-3 illustrates the format of the display.

```
Bad Blocks on Volume:  volumenum

<blocknum> <blocknum> <blocknum> <blocknum> <blocknum> <blocknum>
<blocknum> <blocknum> <blocknum> <blocknum> <blocknum> <blocknum>
     .          .          .          .          .          .
     .          .          .          .          .          .
     .          .          .          .          .          .
<blocknum> <blocknum> <blocknum> <blocknum> <blocknum> <blocknum>
```

Figure 2-3. LISTBADBLOCKS Format

If you have not specified any bad blocks, LISTBADBLOCKS displays the following message:

```
no badblocks.
```

ERROR MESSAGES

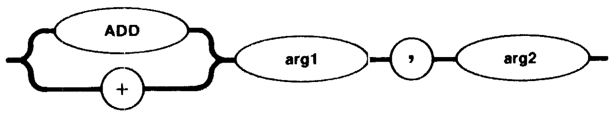
Message	Description
no badblocks file	Your system does not have a bad blocks file. This message could appear because you used an old version of the Human Interface command, FORMAT, when you formatted your disk or because the disk is a physical volume.

MISCELLANEOUS COMMANDS

The following commands provide you with the ability to perform arithmetic and conversion operations within the disk verification utility. The commands perform the operations on unsigned numbers only and do not report any overflow conditions.

ADD

This command adds two numbers together. Its format is:



x-230

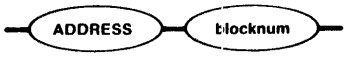
where:

arg1 and arg2	Numbers which the command adds together.

In response, the command displays the unsigned sum of the two numbers in both hexadecimal and decimal format.

ADDRESS

All memory in a volume is divided into volume blocks, which are areas of memory the same size as the volume granularity. Volume blocks are numbered sequentially in the volume, starting with the block containing the smallest addresses (block 0). The ADDRESS command converts a block number into an absolute address on the volume, so that you don't have to perform this conversion by hand. The format of this command is:



x-231

where:

blocknum	Volume block number which ADDRESS converts into an absolute address. This parameter can range from 0 through (max blocks - 1), where max blocks is the number of volume blocks in the volume.

ADDRESS (continued)

In response, ADDRESS displays the following information:

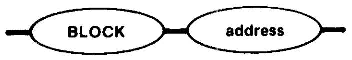
```
    absolute address = <addr>
```

where:

<addr>	Absolute address (in hexadecimal) that corresponds to the specified block number. This address represents the number of the byte that begins the block and can range from 0 through (volume size - 1), where volume size is the size, in bytes, of the volume.

BLOCK

The BLOCK command is the inverse of the address command. It converts a 32-bit absolute address into a volume block number, so that you don't have to perform this conversion by hand. The format of this command is:



x-232

where:

address	Absolute address, which BLOCK converts into a block number. This parameter can range from 0 through (volume size - 1), where volume size is the size, in bytes, of the volume.

In response, BLOCK displays the following information:

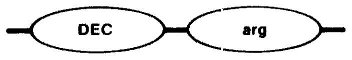
```
    block number = <blocknum>
```

where:

<blocknum>	Number of the volume block that contains the specified absolute address. The BLOCK command determines this value by dividing the absolute address by the volume block size and truncating the result.

DEC

This command finds the decimal equivalent of a number. Its format is:



x-233

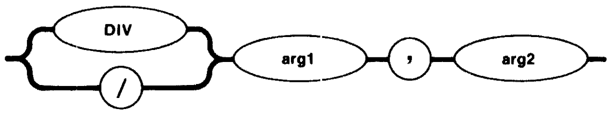
where:

arg	Number for which the command finds the decimal equivalent. This number can be no longer than 10 digits. The default base is in hexadecimal.

In response, the command displays the decimal equivalent of the specified number.

DIV

This command divides one number by another. Its format is:



x-234

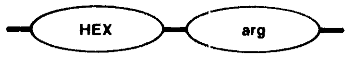
where:

arg1 and arg2	Numbers on which the command operates. It divides arg1 by arg2. These arguments can be no longer than 10 digits decimal or 8 digits hexadecimal.

In response, the command displays the unsigned, integer quotient in both hexadecimal and decimal format.

HEX

This command finds the hexadecimal equivalent of a number. Its format is:



x-235

HEX (continued)

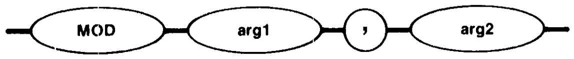
where:

arg	Number for which the command finds the hexadecimal equivalent. This number can be no longer than 10 digits decimal.

In response, the command displays equivalent of the specified number.

MOD

This command finds the remainder of one number divided by another. Its format is:



x-236

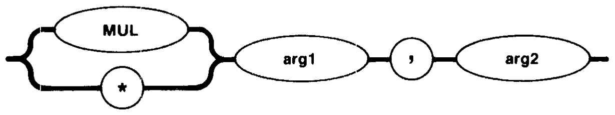
where:

arg1 and arg2	Numbers on which the command operates. It performs the operation arg1 modulo arg2. These arguments can be no longer than 10 digits decimal or 8 digits hexadecimal..

In response, the command displays the value arg1 modulo arg2 in both hexadecimal and decimal format.

MUL

This command multiplies two numbers together. Its format is:



x-237

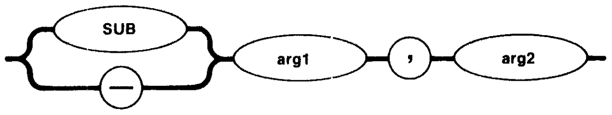
where:

arg1 and arg2	Numbers which the command multiplies together. These arguments can be no longer than 10 digits decimal or 8 digits hexadecimal.

In response, the command displays the unsigned product of the two numbers in both hexadecimal and decimal format.

SUB

This command subtracts one number from another. Its format is:



x-238

where:

arg1 and arg2	Numbers on which the command operates. The command subtracts arg2 from arg1. These arguments can be no longer than 10 digits decimal or 8 digits hexadecimal.

In response, the command displays the unsigned difference in both hexadecimal and decimal format.

ERROR MESSAGES

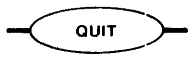
Message	Description
argument error	You made a syntax error in the command, specified a nonnumeric value for one of the arguments, or specified a value for a block number parameter that was not a valid block number.
<blocknum>, block out of range	If the command was an ADDRESS command, the block number you entered was greater than the number of blocks in the volume.
<address>, address not on the disk	If the command was a BLOCK command, BLOCK converted the address to a volume block number, but the block number was greater than the number of blocks in the volume.

EXAMPLES

```
    *MUL 134T, 13T
     6CE ( 1742T)
    *+ 8, 4
      0C (   12T)
    *SUB 8884, 256
    862E (34350T)
    *MOD 1225, 256T
      25 (   37T)
    *HEX 155T
     9B
    *ADDRESS 15
    absolute address = 0A80
    *
    *BLOCK 2236
      block number = 44
```

QUIT COMMAND

This command exits the disk verification utility and returns control to the Human Interface command level. The format of the QUIT command is as follows:

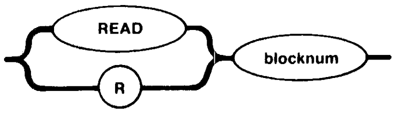


x-239

This command is identical to the EXIT command.

READ COMMAND

This command reads a volume block from the disk into the working buffer. The format of the READ command is:



x-240

INPUT PARAMETER

blocknum	Number of the volume block to read. This number can range from 0 through (max blocks - 1), where max blocks is the number of volume blocks in the volume.

OUTPUT

In response to the command, READ reads the block into the working buffer and displays the following message:

```
read block number:  <blocknum>
```

where <blocknum> is the number of the block.

DESCRIPTION

The READ command copies a specified volume block from the volume to the working buffer. It destroys any data currently in the working buffer. Once the block is in the working buffer, you can use DISPLAYBYTE and DISPLAYWORD to display the block and you can use SUBSTITUTEBYTE and SUBSTITUTEWORD to change the data in the block. Finally, you can use the WRITE command to write the modified block back out to the volume to repair damaged system data.

ERROR MESSAGES

Message	Description
argument error	You specified a nonnumeric character in the blocknum parameter.
<blocknum>, block out of range	The block number that you specified was larger than the largest block number in the volume.

SAVE COMMAND

This command writes the reconstructed free fnodes map, volume free space map, and the bad blocks map to the volume being verified. (The NAMED2 and PHYSICAL options of the VERIFY command originally created the maps.) The format of the SAVE command is:

x-241

OUTPUT

In response to this command, SAVE displays the following message:

```
    save fnode map?
```

If you want to write the reconstructed free fnodes map to the volume, enter Y or YES. Otherwise, enter any other character or a carriage return alone. If you enter YES, SAVE writes the free fnodes map to the volume and displays the following message:

```
    free fnode map saved
```

In any case, SAVE next displays the following message:

```
    save space map?
```

If you want to write the reconstructed free space map to the volume, enter Y or YES. Otherwise, enter any other character or a carriage return alone. If you enter YES, SAVE writes the volume free space map to the volume and displays the following message:

```
    free space map saved
```

SAVE displays the following message if the bad blocks map is constructed:

```
    save bad blocks map?
```

If you want to write the reconstructed bad blocks map to the volume, enter Y or YES. Otherwise, enter any other character or a carriage return alone. If you enter YES, SAVE writes the volume bad blocks map to the volume and displays the following message:

```
    bad block map saved
```

DESCRIPTION

Whenever you perform a VERIFY function with the NAMED2 option (refer to the description of the VERIFY command for more information), VERIFY creates its own free fnodes map and volume free space map. It does this by examining all directories and fnodes on the volume, not by copying the maps that exist on the volume. To create the free fnodes map, it examines every directory on the volume to determine which fnodes represent actual files. To create the volume free space map, it examines the POINTER(n) fields of the fnodes to determine which volume blocks the files use.

If the volume has a bad blocks file, and you perform a VERIFY function with the PHYSICAL option (refer to the description of the VERIFY command for more information), VERIFY creates its own bad blocks map. It does this by examining every block on the volume, not by copying the maps that exist on the volume.

VERIFY then compares the newly created maps with the maps that exist on the volume. If a discrepancy exists, VERIFY displays a message to indicate the discrepancy.

The SAVE command takes the free fnodes map, the volume free space map, and the bad block map created during the VERIFY operation and writes them to the volume, replacing the maps that currently exist.

ERROR MESSAGE

Message	Description
nothing to save	You did not enter the VERIFY command with the NAMED2, NAMED, PHYSICAL, or ALL options prior to entering the SAVE command. Thus SAVE has no free fnode map, volume free space map, or bad block map with which to replace those that exist on the volume.

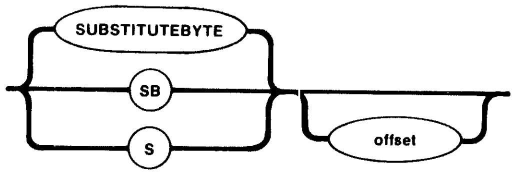
EXAMPLE

The following example illustrates the format of the SAVE command after you use VERIFY and the NAMED or NAMED2 option.

```
*SAVE
save fnode map? no
save space map? y
   free space map saved
*
```

SUBSTITUTEBYTE COMMAND

This command allows you to interactively change the contents of the working buffer (in byte format). You can abort this command by typing a CONTROL-C (press the CONTROL key, and while holding it down, press the C key). The format of the SUBSTITUTEBYTE command is:



x-242

INPUT PARAMETER

offset	Number of the byte, relative to the start of the working buffer, which the command can change in response to user input. This number can range from 0 to (block size - 1), where block size is the size of a volume block (and thus the size of the working buffer). If you omit this parameter, the command assumes a value of 0.

OUTPUT

In response to the command, SUBSTITUTEBYTE displays the specified byte and waits for you to enter a new value. This display appears as:

```
<offset>: val -
```

where <offset> is the number of the byte, relative to the start of the buffer, and val is the current value of the byte. At this point, you can enter one of the following:

- A value followed by a carriage return. This causes SUBSTITUTEBYTE to substitute the new value for the current byte. If the value you enter requires more than one byte of storage, SUBSTITUTEBYTE uses only the low-order byte of the value. SUBSTITUTEBYTE then displays the next byte in the buffer and waits for your further response.

- A carriage return alone. This causes SUBSTITUTEBYTE to leave the current value as is and display the next byte in the buffer. It then waits for your response.

- A value followed by a period (.) and a carriage return. This causes SUBSTITUTEBYTE to substitute the new value for the current byte. It then exits from the SUBSTITUTEBYTE command and gives you the asterisk (*) prompt, permitting you to enter any DISKVERIFY command.

- A period (.) followed by a carriage return. This exits the SUBSTITUTEBYTE command and gives you the asterisk (*) prompt, permitting you to enter any DISKVERIFY command.

DESCRIPTION

The SUBSTITUTEBYTE command gives you the ability to interactively change bytes in the working buffer. Once you enter the command, SUBSTITUTEBYTE displays the offset and the value of the first byte. You can change the byte by entering a new byte value, or you can leave the byte as is by entering a carriage return only. The command then displays the next byte in the buffer. In this manner, you can consecutively step through the buffer, changing whatever bytes are appropriate. When you finish changing the buffer, you can enter a period followed by a carriage return to exit the command.

The SUBSTITUTEBYTE command considers the working buffer to be a circular buffer. That is, entering a carriage return when you are positioned at the last byte of the buffer causes SUBSTITUTEBYTE to display the first byte of the buffer.

The SUBSTITUTEBYTE command changes only the values in the working buffer. To make the changes in the volume, you must enter the WRITE command to write the working buffer back to the volume.

ERROR MESSAGES

Message	Description
argument error	You specified a nonnumeric character in the offset parameter.
<offsetnum>, invalid offset	You specified an offset value that was larger than the number of bytes in the block.

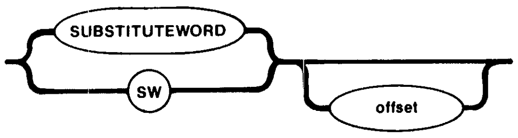
EXAMPLE

This example changes several bytes in two portions of the working buffer. Two SUBSTITUTEBYTE commands are used. Carriage returns are denoted by a <cr> to aid your understanding of this example.

```
    *SUBSTITUTEBYTE<cr>

    0000: A0 - 00<cr>
    0001: 80 - <cr>
    0002: E5 - <cr>
    0003: FF - 31<cr>
    0004: FF - .<cr>

    *SUBSTITUTEBYTE 40<cr>

    0040: 00 - E6<cr>
    0041: 00 - E6<cr>
    0042: 00 - .<cr>

    *
```

SUBSTITUTEWORD COMMAND

This command allows you to interactively change the contents of the working buffer (in word format). You can abort this command by typing a CONTROL-C (press the CONTROL key, and while holding it down, press the C key). The format of the SUBSTITUTEWORD command is:



x-243

INPUT PARAMETER

offset	Number of the byte, relative to the start of the working buffer, which the command can change in response to user input. This number can range from 0 to (block size - 1), where block size is the size of a volume block (and thus the size of the working buffer). If you omit this parameter, the command assumes a value of 0.

OUTPUT

In response to the command, SUBSTITUTEWORD displays the word beginning at the specified byte and waits for you to enter a new value. This display appears as:

```
    <offset>: val -
```

where <offset> is the number of the byte which begins the word, relative to the start of the buffer, and val is the current value of the word. At this point, you can enter one of the following:

- A value followed by a carriage return. This causes SUBSTITUTEWORD to substitute the new value for the current word. If the value you enter requires more than one word of storage, SUBSTITUTEWORD uses only the low-order word of the value. SUBSTITUTEWORD then displays the next word in the buffer and waits for your further response.

- A carriage return alone. This causes SUBSTITUTEWORD to leave the current value as is and display the next word in the buffer. It then waits for your response.

- A value followed by a period (.) and a carriage return. This causes SUBSTITUTEWORD to substitute the new value for the current byte. It then exits from the SUBSTITUTEWORD command and gives you the asterisk (*) prompt, permitting you to enter any DISKVERIFY command.

- A period (.) followed by a carriage return. This exits the SUBSTITUTEWORD command and gives you the asterisk (*) prompt, permitting you to enter any DISKVERIFY command.

DESCRIPTION

The SUBSTITUTEWORD command is exactly like the SUBSTITUTEBYTE command except that it allows you to interactively modify words instead of bytes. Once you enter the command, SUBSTITUTEWORD displays the offset and the value of the first word. You can change the word by entering a new word value, or you can leave the word as is by entering a carriage return only. The command then displays the next word in the buffer. In this manner, you can consecutively step through the buffer, changing whatever words are appropriate. When you finish changing the buffer, you can enter a period followed by a carriage return to exit the command.

The SUBSTITUTEWORD command considers the working buffer to be a circular buffer. That is, entering a carriage return when you are positioned at the last byte of the buffer causes SUBSTITUTEWORD to display the first byte of the buffer.

The SUBSTITUTEWORD command changes only the values in the working buffer. To make the changes in the volume, you must enter the WRITE command to write the working buffer back to the volume.

ERROR MESSAGES

Message	Description
argument error	You specified a nonnumeric character in the offset parameter.
<offsetnum>, invalid offset	You specified an offset value that was larger than the number of bytes in the block.

EXAMPLE

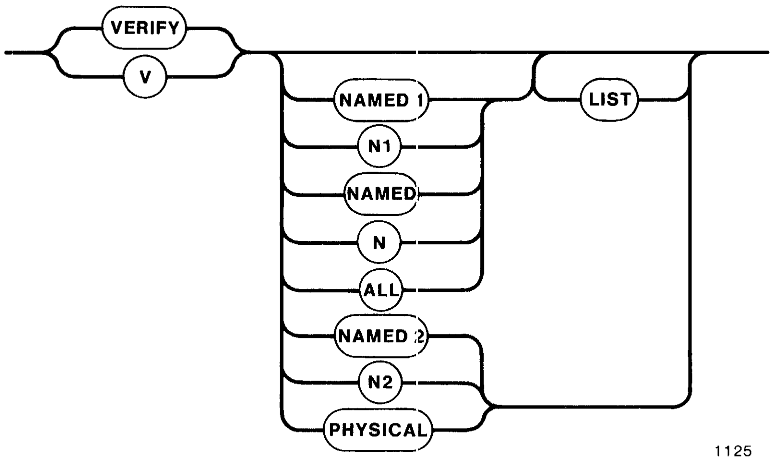
This example changes several bytes in two areas of the working buffer. Two SUBSTITUTEWORD commands are used. Carriage returns are denoted by a <cr> to aid your understanding of this example.

EXAMPLE (continued)

```
    *SUBSTITUTEWORD<cr>

    0000: A0B0 - 0000<cr>
    0002: 8070 - <cr>
    0004: E511 - <cr>
    0006: FFFF - 3111<cr>
    0008: FFFF - .<cr>

    *SUBSTITUTEWORD 35<cr>

    0035: 0000 - E6FF<cr>
    0037: 0000 - E6AB<cr>
    0039: 0000 - .<cr>

    *
```

VERIFY COMMAND

This command checks the structures on the volume to determine whether the volume is properly formatted. You can abort this command by typing a CONTROL-C (press the CONTROL key, and while holding it down, press the C key). The format of the VERIFY command is:



INPUT PARAMETERS

NAMED1 or N1	Checks named volumes to ensure that the information recorded in the fnodes is consistent and matches the information obtained from the directories themselves. VERIFY performs the following operations during a NAMED1 verification:

- Checks fnode numbers in the directories to see if they correspond to allocated fnodes.
- Checks the parent fnode numbers recorded in the fnodes to see if they match with the information recorded in the directories.
- Checks the fnodes against the files to determine if the fnodes specify the proper file type.
- Checks the POINTER(n) structures of long files to see if the indirect blocks accurately reflect the number of blocks used by the file.
- Checks each fnode to see if the TOTAL SIZE, TOTAL BLKS, and THIS SIZE fields are consistent.
- Checks the bad blocks file to see if the blocks in the file correspond to the blocks marked as "bad" on the volume.

NAMED or N	Performs both the NAMED1 and NAMED2 operations on a named volume. If you omit the parameter from the VERIFY command, NAMED is the default parameter.
ALL	Performs all operations appropriate to the volume. For named volumes, this option performs both the NAMED and PHYSICAL operations. For physical volumes, this option performs the PHYSICAL operations.
NAMED2 or N2	Checks named volumes to ensure that the information recorded in the free fnodes map and the volume free space map matches the actual files and fnodes. VERIFY performs the following operations during a NAMED2 verification: • Creates a free fnodes map by examining every directory in the volume. It then compares that free fnodes map with the one already on the volume. • Creates a free space map by examining the information in the fnodes. It then compares that free space map with the one already on the volume. • Checks to see if the block numbers recorded in the fnodes and the indirect blocks actually exist. • Checks to see if two or more files use the same volume block. • Checks to see if two or more files use the same fnode. • Checks the volume free space map for any bad blocks that are marked as "free."
PHYSICAL	Reads all blocks on the volume and checks for I/O errors. This parameter applies to both named and physical volumes. VERIFY also creates a bad blocks map by examining every block on the volume.
LIST	When you specify this option, the file information in Figure 2-4 is displayed for every file on the volume, even if the file contains no errors. You can use this option with all parameters that, either explicitly or implicitly, specify the NAMED1 parameter.

OUTPUT

VERIFY produces a different kind of output for each of the NAMED1, NAMED2, and PHYSICAL options. The NAMED and ALL options produce combinations of the first three kinds of output.

Figure 2-4 illustrates the format of the NAMED1 output (without the LIST option).

```
DEVICE NAME = <devname>      :  DEVICE SIZE = <devsize>  :  BLK = <blksize>

'NAMED1' VERIFICATION

FILE=(<filename>, <fnodenum>):  LEVEL=<lev>:  PARENT=<parnt>:  TYPE=<typ>
     <error messages>

FILE=(<filename>, <fnodenum>):  LEVEL=<lev>:  PARENT=<parnt>:  TYPE=<typ>
     <error messages>
    .           .           .           .           .           .
    .           .           .           .           .           .
    .           .           .           .           .           .
FILE=(<filename>, <fnodenum>):  LEVEL=<lev>:  PARENT=<parnt>:  TYPE=<typ>
     <error messages>
```

Figure 2-4. NAMED1 Verification Output

The following paragraphs identify the fields listed in Figure 2-4.

<devname>	Physical name of the device, as specified in the ATTACHDEVICE Human Interface command.
<devsize>	Hexadecimal size of the volume, in bytes.
<blksize>	Hexadecimal volume granularity. This number is the size of a volume block.
<filename>	Name of the file (1 to 14 characters).
<fnodenum>	Hexadecimal number of the file's fnode.
<lev>	Hexadecimal level of the file in the file hierarchy. The root directory of the volume is the only level 0 file. Files contained in the root directory are level 1 files. Files contained in level 1 directories are level 2 files. This numbering continues for all levels of files in the volume.
<parnt>	Fnode number of the directory which contains this file, in hexadecimal.
<typ>	File type, either DATA (for data files) or DIR (for directory files). If VERIFY cannot ascertain that the file is a directory or data file, it displays the characters "****" in this field.

<error messages>	Messages, which indicate the errors associated with the previously-listed file. The error messages which can occur are listed later in this section.

As Figure 2-4 shows, the NAMED1 option (without the LIST option) displays information about each file that is in error. If you used the LIST option with the NAMED1 option, the file information in Figure 2-4 is displayed for every file, even if the file contains no errors. The NAMED1 display also contains error messages which immediately follow the listing of the affected files.

Figure 2-5 illustrates the format of the NAMED2 output.

```
DEVICE NAME = <devname>     : DEVICE SIZE = <devsize> : BLK SIZE = <blksize>

'NAMED2' VERIFICATION

       BIT MAPS O.K.
```

Figure 2-5. NAMED2 Verification Output

The fields in Figure 2-5 are exactly the same as the corresponding fields in Figure 2-4.

If VERIFY detects an error during NAMED2 verification, it displays one or more error message in place of the "BIT MAPS O.K." message.

Figure 2-6 illustrates the format of the PHYSICAL output.

```
DEVICE NAME = <devname>     : DEVICE SIZE = <devsize> : BLK SIZE = <blksize>

'PHYSICAL' VERIFICATION

       NO ERRORS
```

Figure 2-6. PHYSICAL Verification Output

The fields in Figure 2-6 are exactly the same as the corresponding fields in Figure 2-4.

If VERIFY detects an error during PHYSICAL verification, it displays one or more error message in place of the "NO ERRORS" message.

If you specify NAMED verification, VERIFY displays both the NAMED1 and NAMED2 output. If you specify the ALL verification for a named volume, VERIFY displays the NAMED1, NAMED2, and PHYSICAL output. If you specify the ALL verification for a physical volume, VERIFY displays the PHYSICAL output.

DESCRIPTION

The VERIFY command checks physical and named volumes to ensure that the volumes contain valid file structures and data areas. VERIFY can perform three kinds of verification: NAMED1, NAMED2, and PHYSICAL. NAMED1 and NAMED2 verifications check the file structures of named volumes. They do not apply to physical volumes. A PHYSICAL verification checks each data block of the volume for I/O errors. PHYSICAL verification applies to both named and physical volumes.

As part of the NAMED2 verification, VERIFY creates a free fnodes map and a volume free space map which it compares with the corresponding maps on the volume. You can use the SAVE command to write the maps produced during NAMED2 verification to the volume, overwriting the maps on the volume.

When you perform a PHYSICAL verification on a named volume, VERIFY also creates a bad blocks map. You can use the SAVE command to write the bad blocks map produced during PHYSICAL verification to the volume; this destroys the bad blocks map already on the volume.

ERROR MESSAGES

Four kinds of error messages can occur as a result of entering the VERIFY command: VERIFY command errors, NAMED1 errors, NAMED2 errors, and PHYSICAL errors.

VERIFY command error

Message	Description
argument error	The parameter you specified is not a valid VERIFY parameter.

NAMED1 errors

The following messages can appear in a NAMED1 display, immediately after the file to which they refer.

Message	Description
<blocknum$_1$-blocknum$_n$>, block bad	The block numbers displayed in this message are marked as "bad."
<blocknum$_1$-blocknum$_n$>, invalid block# recorded in the fnode/indirect block	One of the POINTER(n) fields in the fnode specifies block numbers that are larger than the largest block number in the volume.
directory stack overflow	This message can indicate an internal error in the disk verification utility. However, it can also indicate that a directory on the volume lists, as one of its entries, itself or one of the parent directories in its pathname. If this happens, the utility, when it searches through the directory tree, continually loops through a portion of the tree, overflowing an internal buffer area.
file size inconsistent total$size = <totsize> : this$size = <thsize> : data blocks = <numblks>	The TOTAL SIZE, THIS SIZE, and TOTAL BLKS fields of the fnode are inconsistent.
<filetype>, illegal file type	The file type of a user file, as recorded in TYPE field of the fnode, is not valid. The valid file types and their descriptions are as follows:

Filetype	Description
DIR	directory
DATA	data
SMAP	volume free space map
FMAP	free fnodes map
BMAP	bad blocks map
VLAB	volume label file

<fnodenum>, allocation status bit in this fnode not set	The file is listed in a directory but the flags field of its fnode indicates that fnode is free. The free fnodes map may or may not list the fnode as allocated.
<fnodenum>, fnode out of range	The fnode number is larger than the largest fnode number in the fnode file.
<fnodenum>, parent fnode number does not match	The parent fnode number in the file does not match the parent fnode number in the directory. VERIFY displays the fnode number of the directory that contains the file, not the fnode number recorded in the PARENT field of the file's fnode.
invalid block# recorded in the fnode/indirect block	The file is a long file and one of the fnodes or indirect blocks specifies a block number that is larger than the largest block number in the volume.
insufficient memory to create directory stack	There is not enough dynamic memory in the system for the utility to perform the verification.
sum of the blks in the indirect block does not match block in the fnode	The file is a long file, and the number of blocks listed in a POINTER(n) field of the fnode does not agree with the number of blocks listed in the indirect block.
total-blocks does not reflect the data-blocks correctly	The TOTAL BLKS field of the fnode and the number of blocks recorded in the POINTER(n) fields are inconsistent.

NAMED2 errors

The following messages can appear in a NAMED2 display.

Message	Description
<blocknum_1-blocknum_2>, bad block not allocated	The volume free space map indicates that the blocks are free, but they are marked as "bad" in the bad blocks file.

<blocknum>, block allocated but not referenced	The volume free space map lists the specified volume block as allocated, but no fnode specifies the block as part of a file.
<blocknum>, block referenced but not allocated	An fnode indicates that the specified volume block is part of a file, but the volume free space map lists the block as free.
directory stack overflow	This message can indicate an internal error in the disk verification utility. However, it can also indicate that a directory on the volume lists, as one of its entries, itself or one of the parent directories in its pathname. If this happens, the utility, when it searches through the directory tree, continually loops through a portion of the tree, overflowing an internal buffer area.
Fnodes map indicates fnodes > max$fnode	The free fnodes map indicates that there are a greater number of unallocated fnodes than the maximum number of fnodes in the volume.
<fnodenum>, fnode map bit marked allocated but not referenced	The free fnodes map lists the specified fnode as allocated, but no directory contains a file with the fnode number.
<fnodenum>, fnode referenced but fnode map bit marked free	The specified fnode number is listed in a directory, but the free fnodes map lists the fnode as free.
Free space map indicates Volume block > max$Volume$block	The free space map indicates that there are a greater number of unallocated blocks than the maximum number of blocks in the volume.
insufficient memory to create directory stack	There is not enough dynamic memory in the system for the utility to perform the verification.
insufficient memory to create fnode and space maps	During a NAMED2 verification, the utility tried to create a free fnodes map and a volume free space map. However, there is not enough dynamic memory available in the system to create these maps.
insufficient memory to create bad blocks map	During a PHYSICAL verification, the utility tried to create a bad blocks map. However, there is not enough dynamic memory available in the system to create the map.

<blocknum>, multiple reference to this block	More than one fnode specifies this block as part of a file.
<fnodenum>, multiple reference to this fnode	The directories on the volume list more than one file associated with this fnode number.

PHYSICAL error

Message	Description
<blocknum>, error	An I/O error occurred when VERIFY tried to access the specified volume block. The volume is probably flawed.

other errors

The following error messages indicate internal errors in the disk verification utility. Under normal conditions these messages should never appear. However, if these messages (or other undocumented messages that also appear to indicate internal problems) do appear during a NAMED1 or NAMED2 verification, you should exit the disk verification utility and re-enter the DISKVERIFY command.

```
directory stack empty
directory stack error
directory stack underflow
```

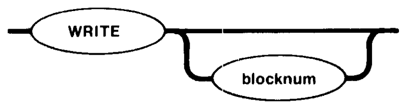
EXAMPLE

The following command performs both named and physical verification on a named volume.

```
*VERIFY ALL

DEVICE NAME = F1           : DEVICE SIZE = 0003E900 : BLK SIZE = 0080

'NAMED1' VERIFICATION


'NAMED2' VERIFICATION

   BIT MAPS O.K.

'PHYSICAL' VERIFICATION

   NO ERRORS

*
```

WRITE COMMAND

This command writes the contents of the working buffer to the volume. The format of this command is:



x-244

INPUT PARAMETER

blocknum	Number of the volume block to which the command writes the working buffer. If you omit this parameter, WRITE writes the buffer back to the block most recently accessed.

OUTPUT

In response to the command, WRITE displays the following message:

```
write to block <blocknum>?
```

where <blocknum> is the number of the volume block to which WRITE intends to write the working buffer. If you respond by entering Y or any character string beginning with Y, WRITE copies the working buffer to the specified block on the volume and displays the following message:

```
written to block number block <blocknum>
```

Any other response aborts the write process.

DESCRIPTION

The WRITE command is used in conjunction with the READ, DISPLAYBYTE, DISPLAYWORD, SUBSTITUTEBYTE, and SUBSTITUTEWORD commands to modify information on the volume. Initially you use READ to copy a volume block from the volume to a working buffer. Then you can use DISPLAYBYTE and DISPLAYWORD to view the buffer and SUBSTITUTEBYTE and SUBSTITUTEWORD to change the buffer. Finally, you can use WRITE to write the modified buffer back to the volume. By default, WRITE copies the buffer to the block most recently accessed by a READ or WRITE command.

A WRITE command does not destroy the data in the working buffer. The data remains the same until the next SUBSTITUTEBYTE, SUBSTITUTEWORD, or READ command modifies the buffer.

ERROR MESSAGES

Message	Description
argument error	You made a syntax error or specified nonnumeric characters in the blocknum parameter.
<blocknum>, block out of range	The block number you specified was larger than the largest block number in the volume.

EXAMPLE

The following command copies the working buffer to the block from which it was read.

```
*WRITE
   write to block 4B? y
*written to block number : 4B
```

***

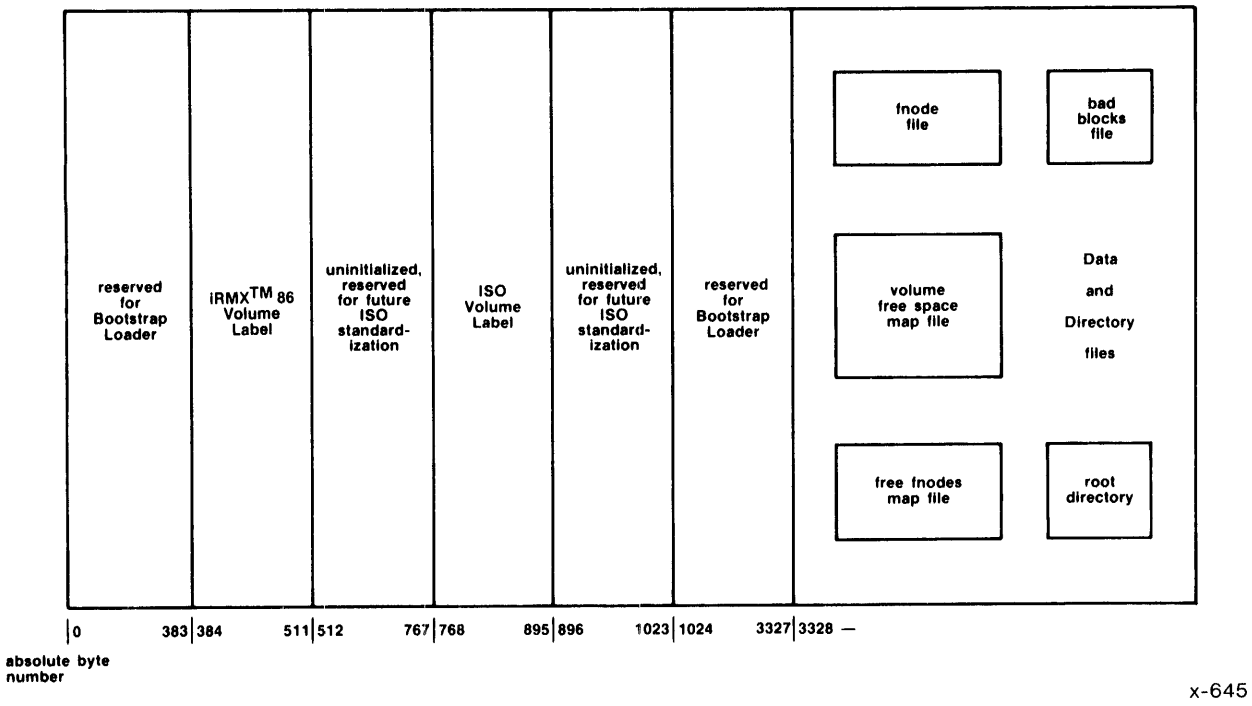
APPENDIX A STRUCTURE OF iRMX™ 86 NAMED VOLUMES

This appendix describes the structure of an iRMX 86 volume that contains named files. Those users who wish to examine named file volumes or create their own formatting utility programs can use this information.

This appendix is intended for system programmers who have had experience in reading and writing actual volume information. It does not attempt to teach the reader these functions.

INTRODUCTION

Each iRMX 86 named volume contains ISO (International Organization for Standardization) label information as well as iRMX 86 label information and files. Figure A-1 illustrates the general structure of a named file volume.



reserved for Bootstrap Loader
iRMX™ 86 Volume Label
uninitialized, reserved for future ISO standardization
ISO Volume Label
uninitialized, reserved for future ISO standardization
reserved for Bootstrap Loader
inode file
bad blocks file
volume free space map file
Data and Directory files
free inodes map file
root directory
0
383
384
511
512
767
768
895
896
1023
1024
3327
3328 —
absolute byte number
x-645

Figure A-1. General Structure Of Named Volumes

This appendix discusses the structure in more detail. It includes information concerning the following:

- ISO Volume Label
- iRMX 86 Volume Label
- fnode file
- volume free space map file
- free fnodes map file
- bad blocks map file
- root directory

It also discusses the structure of directory files and the concepts of long and short files.

The blocks in Figure A-1 that are reserved for the Bootstrap Loader are not discussed. To include these blocks on a new volume that you are formatting, you should copy them from an already formatted volume.

NOTE

The following sections of this appendix refer to a data type called DWORD. DWORD must be declared literally as POINTER. This results in a 32-bit variable for the PLM/86 models COMPACT, MEDIUM, and LARGE.

VOLUME LABELS

This section describes the structure of the volume labels that must be present on a named volume. These labels are the ISO volume label and the iRMX 86 volume label.

ISO VOLUME LABEL

The ISO (International Organization for Standardization) volume label is recorded in absolute byte positions 768 through 895 of the volume (for example, sector 07 of a single density flexible diskette). The structure of this volume label is as follows:

```
DECLARE
    ISO$VOL$LABEL  STRUCTURE(
        LABEL$ID(3)         BYTE,
        RESERVED$A          BYTE,
        VOL$NAME(6)         BYTE,
        VOL$STRUC           BYTE,
        RESERVED$B(60)      BYTE,
        REC$SIDE            BYTE,
        RESERVED$C(4)       BYTE,
        ILEAVE(2)           BYTE,
        RESERVED$D          BYTE,
        ISO$VERSION         BYTE,
        RESERVED$E(48)      BYTE);
```

where:

LABEL$ID(3)	Label identifier. For named file volumes, this field contains the ASCII characters "VOL".
RESERVED$A	Reserved field containing the ASCII character "1".
VOL$NAME(6)	Volume name. This field can contain up to six printable ASCII characters, left justified and space filled. A value of all spaces implies that the volume name is recorded in the iRMX 86 Volume Label (absolute byte positions 384-393).
VOL$STRUC	For named file volumes, this field contains the ASCII character "N", indicating that this volume has a non-ISO file structure.
RESERVED$B(60)	This is a reserved field containing 60 bytes of ASCII spaces.
REC$SIDE	For named file volumes, this field contains the ASCII character "1" to indicate that only one side of the volume is to be recorded.
RESERVED$C(4)	This is a reserved field containing four bytes of ASCII spaces.
ILEAVE(2)	Two ASCII digits indicating the interleave factor for the volume, in decimal. ASCII digits consist of the numbers 0 through 9. When formatting named volumes, you should set this field to the same interleave factor that you use when physically formatting the volume.
RESERVED$D	This is a reserved field containing an ASCII space.
ISO$VERSION	For named file volumes, this field contains the ASCII character "1", which indicates ISO version number one.
RESERVED$D(48)	This is a reserved field containing 48 ASCII spaces.

iRMX™ 86 VOLUME LABEL

The iRMX 86 Volume Label is recorded in absolute byte positions 384 through 511 of the volume (sector 04 of a single density flexible diskette). The structure of this volume label is as follows:

```
    DECLARE
        RMX$VOLUME$INFORMATION  STRUCTURE(
                VOL$NAME(10)        BYTE,
                FLAGS               BYTE,
                FILE$DRIVER         BYTE,
                VOL$GRAN            WORD,
                VOL$SIZE            DWORD,
                MAX$FNODE           WORD,
                FNODE$START         DWORD,
                FNODE$SIZE          WORD,
                ROOT$FNODE          WORD,
                DEV$GRAN            WORD,
                INTERLEAVE          WORD,
                TRACK$SKEW          WORD,
                SYSTEM$ID           WORD,
                SYSTEM$NAME(12)     BYTE,
                DEVICE$SPECIAL(8)   BYTE);
```

where:

VOL$NAME(10) — Volume name in printable ASCII characters, left justified and zero filled.

FLAGS — BYTE which lists the device characteristics for automatic device recognition. The individual bits in this byte indicate the following characteristics (bit 0 is right-most bit):

Bit	Meaning
0	VF$AUTO flag. When set to one, this bit indicates that the FLAGS byte contains valid data for automatic device recognition. When set to zero, it indicates that the remaining flags contain meaningless data.
1	VF$DENSITY flag. This bit indicates the recording density of the volume. When set to one, it indicates modified frequency modulation (MFM) or double-density recording. When set to zero,

Bit	Meaning
	it indicates frequency modulation (FM) or single-density recording.
2	VF$SIDES flag. This bit indicates the number of recording sides on the volume. When set to one, it indicates a double-sided volume. When set to zero, it indicates a single-sided volume.
3	VF$MINI flag. This bit indicates the size of the recording media. When set to one, it indicates a 5 1/4-inch volume. When set to zero, it indicates an 8-inch volume.
4	VF$NOT$FLOPPY. This bit indicates the type of disk you are using. When this bit is set to one and when Bit 0 is set to one, it indicates a Winchester disk.

FILE$DRIVER	Number of the file driver used with this volume. For named file volumes, this field is set to four.
VOL$GRAN	Volume granularity, specified in bytes. This value must be a multiple of the device granularity. It sets the size of a logical device block, also called a volume block.
VOL$SIZE	Size of the entire volume, in bytes.
MAX$FNODE	Number of fnodes in the fnode file. Refer to the next section for a description of fnodes.
FNODE$START	A 32-bit value which represents the number of the first byte in the fnode file (byte 0 is the first byte of the volume).
FNODE$SIZE	Size of an fnode, in bytes.
ROOT$FNODE	Number of the fnode describing the root directory. Refer to the next section for further information.

DEV$GRAN	Device granularity of all tracks except track zero (which contains the volume label). This field is important only when the system requires automatic device recognition.
INTERLEAVE	Block interleave factor for this volume. This value indicates the physical distance, in blocks, between consecutively-numbered blocks on the volume. A value of one indicates that consecutively-numbered blocks are adjacent. A value of zero indicates an unknown or undefined interleave factor.
TRACK$SKEW	Offset, in bytes, between the first block on one track and the first block on the next track. A value of zero indicates that all tracks are identical.
SYSTEM$ID	Numerical code identifying the operating system that formatted the volume. The following codes are reserved for Intel operating systems:

Operating System	Code
iRMX 86	0 - 0Fh
iRMX 88	10h - 1Fh
OS 88	20h - 2Fh

Currently, the iRMX 86 Operating System places a zero in this field.

SYSTEM$NAME(12) Name of the operating system which formatted the volume, in printable ASCII characters, left justified and space filled. Zeros (ASCII nulls) indicate that the operating system is unknown. The iRMX 86 Operating System currently places several pieces of information into this field, as follows:

- The left-most six bytes of this field contain the ASCII characters "iRMX86" to identify the operating system. Former iRMX 86 releases filled this field with zeros.

- The next byte is an ASCII character which identifies the program that formatted the volume. The following characters apply:

Character	Formatting Program
F	Human Interface FORMAT command
U	iRMX 86 Files Utility

If the formatting program is unable to provide this information, it places an ASCII space in this field.

- The next two bytes contain a two-digit ASCII sequence number which is incremented by the formatting program each time the formatting program changes in a way that affects the volume format. The Release 4 FORMAT Human Interface command places the characters "00" in this field.

- The right-most three bytes of the field contain a three-digit ASCII number specifying the version of the Basic I/O System that was used in formatting the volume (for example, the characters "030" would indicate version 3.0). If the formatting program is unable to obtain this information, it places ASCII spaces in this field.

DEVICE$SPECIAL(8) Reserved for special device-specific information. When no device-specific information exists, this field must contain zeros. If the device is a Winchester disk with an iSBC 215 controller or if the device is a disk with an iSBC 220 controller, the iRMX 86 Operating System imposes a structure on this field and supplies the following information:

```
SPECIAL              STRUCTURE(
CYLINDERS            WORD,
FIXED                BYTE,
REMOVABLE            BYTE,
SECTORS              BYTE,
SECTOR_SIZE          WORD,
ALTERNATES           BYTE);
```

where:

CYLINDERS	Total number of cylinders on the drive.
FIXED	Number of heads on the fixed disk or Winchester disk.
REMOVABLE	Number of heads on the removable disk cartridge.
SECTORS	Number of sectors in a track.
SECTOR_SIZE	Sector size, in bytes.
ALTERNATES	Number of alternate cylinders.

The remainder of the Volume Label (bytes 430 through 511) is reserved and must be set to zero.

INITIAL FILES

Any mechanism that formats iRMX 86 named volumes must place five files on the volume during the format process. These five files are the fnode file, the volume free space map file, the free fnodes map file, the bad blocks file, and the root directory. The first of these files, the fnode file, contains information about all of the files on the volume. The general structure of the fnode file is discussed first. Then all of the files are discussed in terms of their fnode entries and their functions.

FNODE FILE

A data structure called a file descriptor node (or fnode) describes each file in a named file volume. All the fnodes for the entire volume are grouped together in a file called the fnode file. When the I/O System accesses a file on a named volume, it examines the iRMX 86 Volume Label (described in the previous section) to determine the location of the fnode file, and then examines the appropriate fnode to determine the actual location of the file.

When a volume is formatted, the fnode file contains six allocated fnodes. In addition to the six allocated files, there also any number of unallocated fnodes. The original number of unallocated fnodes depends on the FILES parameter of the FORMAT command. These allocated fnodes represent the fnode file, the volume free space map file, the free fnodes map file, the bad blocks file, the root directory, and one other file. Later sections of this chapter describe these files. The size of the fnode file is determined by the number of fnodes that it contains. The number of fnodes in the fnode file also determines the number of files that can be created on the volume. The number of files is set when you format the storage medium.

The structure of an individual fnode in a named file volume is as follows:

```
       DECLARE
               FNODE             STRUCTURE(
                      FLAGS                WORD,
                      TYPE                 BYTE,
                      GRAN                 BYTE,
                      OWNER                WORD,
                      CR$TIME              DWORD,
                      ACCESS$TIME          DWORD,
                      MOD$TIME             DWORD,
                      TOTAL$SIZE           DWORD,
                      TOTAL$BLKS           DWORD,

                      POINTR(40)           BYTE,

                      THIS$SIZE            DWORD,
                      RESERVED$A           WORD,
                      RESERVED$B           WORD,
                      ID$COUNT             WORD,

                      ACC(9)               BYTE,
                      PARENT               WORD,
                      AUX(*)               BYTE);
```

where:

FLAGS — A WORD which defines a set of attributes for the file. The individual bits in this word indicate the following attributes (bit 0 is the right-most bit):

Bit	Meaning
0	Allocation status. If set to one, this fnode describes an actual file. If set to zero, this fnode is available for allocation. When formatting a volume, this bit is set to one in the six allocated fnodes. In other fnodes, it is set to zero.
1	Long or short file attribute. This bit describes how the PTR fields of the fnode are interpreted. If set to zero, indicating a short file, the PTR fields identify the actual data blocks of the file. If set to one, indicating a long file, the PTR fields identify indirect blocks. Indirect blocks are described later in this section. When formatting a volume, this bit is always set to zero, since the initial files on the volume are short files.

Bit	Meaning
2	Reserved bit which is always set to one.
3-4	Reserved bits which are always set to zero.
5	Modification attribute. Whenever a file is modified, this bit is set to one. Initially, when a volume is formatted, this bit is set to zero in each fnode.
6	Deletion attribute. This bit is set to one to indicate that the file is a temporary file or that the file is going to be deleted (the deletion may be postponed because additional connections exist to the file). Initially, when the volume is formatted, this bit is set to zero in each fnode.
7-15	Reserved bits which are always set to zero.

TYPE	Type of file. The following are acceptable types:

Mnemonic	Value	Type
FT$FNODE	0	fnode file
FT$VOLMAP	1	volume free space map
FT$FNODEMAP	2	free fnodes map
FT$ACCOUNT	3	space accounting file
FT$BADBLOCK	4	bad device blocks file
FT$DIR	6	directory file
FT$DATA	8	data file
FT$VLABEL	9	volume label file

During system operation, only the I/O System can access file types other than FT$DATA and FT$DIR. These file types are discussed later in this section.

GRAN	File granularity, specified in multiples of the volume granularity. The default value is 1. For the files initially present on the volume (fnode file, volume free space map file, free fnodes map file, bad blocks file, root directory), this value can be set to any multiple of the volume granularity.
OWNER	User ID of the owner of the file. For the files initially present on the volume, this parameter is important only for the root directory. For the root directory, this parameter should specify the user WORLD (FFFFH). The I/O System does not examine this parameter for the other files (fnode file, volume free space map file, free fnodes map file, bad blocks file) and so a value of zero can be specified.
CR$TIME	Time and date that the file was created, expressed as a 32-bit value. This value indicates the number of seconds since a fixed, user-determined point in time. By convention, this point in time is 12:00 A.M., January 1, 1978. For the files initially present on the volume, this parameter is important only for the root directory. A zero can be specified for the other files (fnode file, volume free space map file, free fnodes map file, bad blocks file).
ACCESS$TIME	Time and date of the last file access (read or write), expressed as a 32-bit value. For the files initially present on the volume, this parameter is important only for the root directory.
MOD$TIME	Time and date of the last file modification, expressed as a 32-bit value. For the files initially present on the volume, this parameter is important only for the root directory.

TOTAL$SIZE	Total size, in bytes, of the actual data in the file.
TOTAL$BLKS	Total number of volume blocks used by this file, including indirect block overhead. A <u>volume block</u> is a block of data whose size is the same as the volume granularity. All memory in the volume is divided into volume blocks, which are numbered sequentially, starting with the block containing the smallest addresses (block 0). Indirect blocks are discussed later in this section.
POINTR(40)	A group of bytes on which the following structure is imposed:

```
PTR(8)         STRUCTURE(
     NUM$BLOCKS     WORD,
     BLK$PTR(3)     BYTE);
```

This structure identifies the data blocks of the file. These data blocks may be scattered throughout the volume, but together they make up a complete file. If the file is a short file (bit 1 of the FLAGS field is set to zero), each PTR structure identifies an actual data block. In this case, the fields of the PTR structure contain the following:

NUM$BLOCKS	Number of volume blocks in the data block.
BLK$PTR(3)	A 24-bit value specifying the number of the first volume block in the data block. Volume blocks are numbered sequentially, starting with the block with the smallest address (block 0). The bytes in the BLK$PTR array range from least significant (BLK$PTR(0)) to most significant (BLK$PTR(2)).

If the file is a long file (bit 1 of the FLAGS field is set to one), each PTR structure identifies an indirect block (possibly consisting of more than one contiguous volume block), which in turn identifies the data blocks of the file. In this case, the fields of the PTR structure contain the following:

NUM$BLOCKS	Number of volume blocks pointed to by the indirect block.

BLK$PTR(3)	A 24-bit volume block number of the indirect block.

Indirect blocks are discussed later in this section.

THIS$SIZE	Size, in bytes, of the total data space allocated to the file. This figure does not include space used for indirect blocks, but it does include any data space allocated to the file, regardless of whether the file fills that allocated space.
RESERVED$A	Reserved field which is set to zero.
RESERVED$B	Reserved field which is set to zero.
ID$COUNT	Number of access-ID pairs declared in the ACC(9) field.
ACC(9)	A group of bytes on which the following structure is imposed:

```
ACCESSOR(3)    STRUCTURE(
     ACCESS          BYTE,
     ID              WORD);
```

This structure contains the access-ID pairs which define the access rights for the users of the file. By convention, when a file is created, the owning user's ID is inserted in ACCESSOR(0), along with the code for the access rights. The fields of the ACCESSOR structure contain the following:

ACCESS Encoded access rights for the file. The settings of the individual bits in this field grant (if set to one) or deny (if set to zero) permission for the corresponding operation. Bit 0 is the right-most bit.

Bit	Data File Operation	Directory Operation
0	delete	delete
1	read	display
2	append	add entry
3	update	change entry
4-7	reserved (must be 0)	

ID ID of the user who gains the corresponding access permission.

PARENT	Fnode number of directory file which lists this file. For files initially present on the volume, this parameter is important only for the root directory. For the root directory, this parameter should specify the number of the root directory's own fnode. For other files (fnode file, volume free space map file, free fnodes map file, bad blocks file) the I/O System does not examine this field.

AUX(*)	Auxiliary bytes associated with the file. The named file driver does not interpret this field, but the user can access it by making GET$EXTENSION$DATA and SET$EXTENSION$DATA system calls. The size of this field is determined by the size of the fnode, which is specified in the iRMX 86 Volume Label. The Files Utility allocates three bytes for this field by default. If you use the Human Interface FORMAT command or create your own utility to format a volume, you can make this field as large as you wish; however, a larger AUX field implies slower file access.

Certain fnodes designate special files that appear on the volume. The following sections discuss these fnodes and the associated files.

FNODE 0 (FNODE FILE)

The first fnode structure in the fnode file describes the fnode file itself. This file contains all the fnode structures for the entire volume. It must reside in contiguous locations in the volume. Fields of fnode 0 must be set as follows:

- The bits in the FLAGS field are set to the following (bit 0 is the right-most bit):

Bit	Value	Description
0	1	Allocated file
1	0	Short file
2	1	Primary fnode
3-4	0	Reserved bits
5	0	Initial status is unmodified
6	0	File will not be deleted
7-15	0	Reserved bits

- The TYPE field is set to FT$FNODE.

- The GRAN field is set to 1.

- The OWNER field is set to 0.

- The CR$TIME, ACCESS$TIME, and MOD$TIME fields are set to 0.

- Since the iRMX 86 Volume Label specifies the size of an individual fnode structure and the number of fnodes in the fnode file, the value specified in the TOTAL$SIZE field of fnode 0 must equal the product of the values in the FNODE$SIZE and MAX$FNODE fields of the iRMX 86 Volume Label.

- The TOTAL$BLOCKS field specifies enough volume blocks to account for the memory listed in the TOTAL$SIZE field. The product of the value in the TOTAL$BLOCKS field and the volume granularity equals the value of the THIS$SIZE field, since the fnode file is a short file.

- Since the fnode file must reside in contiguous locations in the volume, only one PTR structure describes the location of the file. The value in the NUM$BLOCKS field of that PTR structure equals the value in the TOTAL$BLOCKS field. The BLK$PTR field indicates the number of the first block of the fnode file.

- The ID$COUNT field is set to zero, indicating that no users can access the file.

FNODE 1 (VOLUME FREE SPACE MAP FILE)

The second fnode, fnode 1, describes the volume free space map file. The TYPE field for fnode 1 is set to FT$VOLMAP to designate the file as such.

The volume free space map file keeps track of all the space on the volume. It is a bit map of the volume, in which each bit represents one volume block (a block of space whose size is the same as the volume granularity). If a bit in the map is set to one, the corresponding volume block is free to be allocated to any file. If a bit in the map is set to zero, the corresponding volume block is already allocated to a file. The bits of the map correspond to volume blocks such that bit n of byte m represents volume block (8 * m) + n. The bits in the remaining space allocated to the map file (those that do not correspond to actual blocks of memory) must be set to zero.

When the volume is formatted, the volume free space map file indicates that the first 3328 bytes of the volume (the label and bootstrap information) plus any files initially placed on the volume (fnode file, volume free space map file, free fnodes map file, bad blocks file) are allocated.

FNODE 2 (FREE FNODES MAP FILE)

The third fnode, fnode 2, describes the free fnodes map file. The TYPE field of fnode 2 is set to FT$FNODEMAP to designate the file as such.

The free fnodes map file keeps track of all the fnodes in the fnodes file. It is a bit map in which each bit represents an fnode. If a bit in the map is set to one, the corresponding fnode is not in use and does not represent an actual file. If a bit in the map is set to zero, the corresponding fnode already describes an existing file. The bits in the map correspond to fnodes such that bit n of byte m represents fnode number (8 * m) + n. The bits in the remaining space allocated to the map file (those that do not correspond to actual fnode structures) must be set to zero.

When the volume is formatted, the free fnodes map file indicates that fnodes 0, 1, 2, 3, and 4 are in use. If other files are initially placed on the volume, the free fnodes map file must be set to indicate this as well.

FNODE 4 (BAD BLOCKS FILE)

The fifth fnode, fnode 4, contains all the bad blocks on the volume. The TYPE field of fnode 4 is set to FT$BADBLOCK to indicate this.

If there are any unusable blocks on a volume, this fnode must be initialized to describe a file which consists of all such bad blocks. If there are no bad blocks on the volume, the fnode must still be set up as allocated, and of the indicated type, but it should not assign any actual space for the file.

FNODE 5 (VOLUME LABEL FILE)

This fnode contains the first 3328 bytes of any volume. The information in this file defines the volume as a whole. The TYPE field of this node is set to FT$VLABEL. You cannot write to this fnode.

FNODE 6 (ROOT DIRECTORY)

The root directory is a special directory file. It is the root of the named file hierarchy for the volume. The iRMX 86 Volume Label specifies the fnode number of the root directory. The root directory is its own parent. That is, the PARENT field of its fnode specifies its own fnode number.

The root directory (and all directory files) associates file names with fnode numbers. It consists of a number of entries that have the following structure:

```
DECLARE
    DIR$ENTRY           STRUCTURE(
        FNODE           WORD,
        COMPONENT(14)   BYTE);
```

where:

FNODE	Fnode number of a file listed in the directory.
COMPONENT(14)	A string of ASCII characters that is the final component of the path name identifying the file. This string is left justified and null padded to 14 characters.

When a file is deleted, its fnode number in the directory entry is set to zero.

OTHER FNODES

When a volume is formatted, one other fnode is set up, fnode 3, representing a file of type FT$ACCOUNT, The fnode is set up as allocated, and of the indicated type, but it does not assign any actual space for the file.

When formatting a volume, no other fnodes in the fnode file represent actual files. The remaining fnodes must have bit zero (allocation status) set to zero.

LONG AND SHORT FILES

A file on a volume is not necessarily one contiguous string of bytes. In many cases, it consists of several contiguous blocks of data scattered throughout the volume. The fnode for the file indicates the locations and sizes of these blocks in one of two ways, as short files or as long files.

SHORT FILES

If the file consists of eight or less distinct blocks of data, its fnode can specify it as a short file. The fnode for a short file has bit 1 of the FLAGS field set to zero. This indicates to the I/O System that the PTR structures of the fnode identify the actual data blocks that make up the file. Figure A-2 illustrates an fnode for a short file. Decimal numbers are used in the figure for clarity.

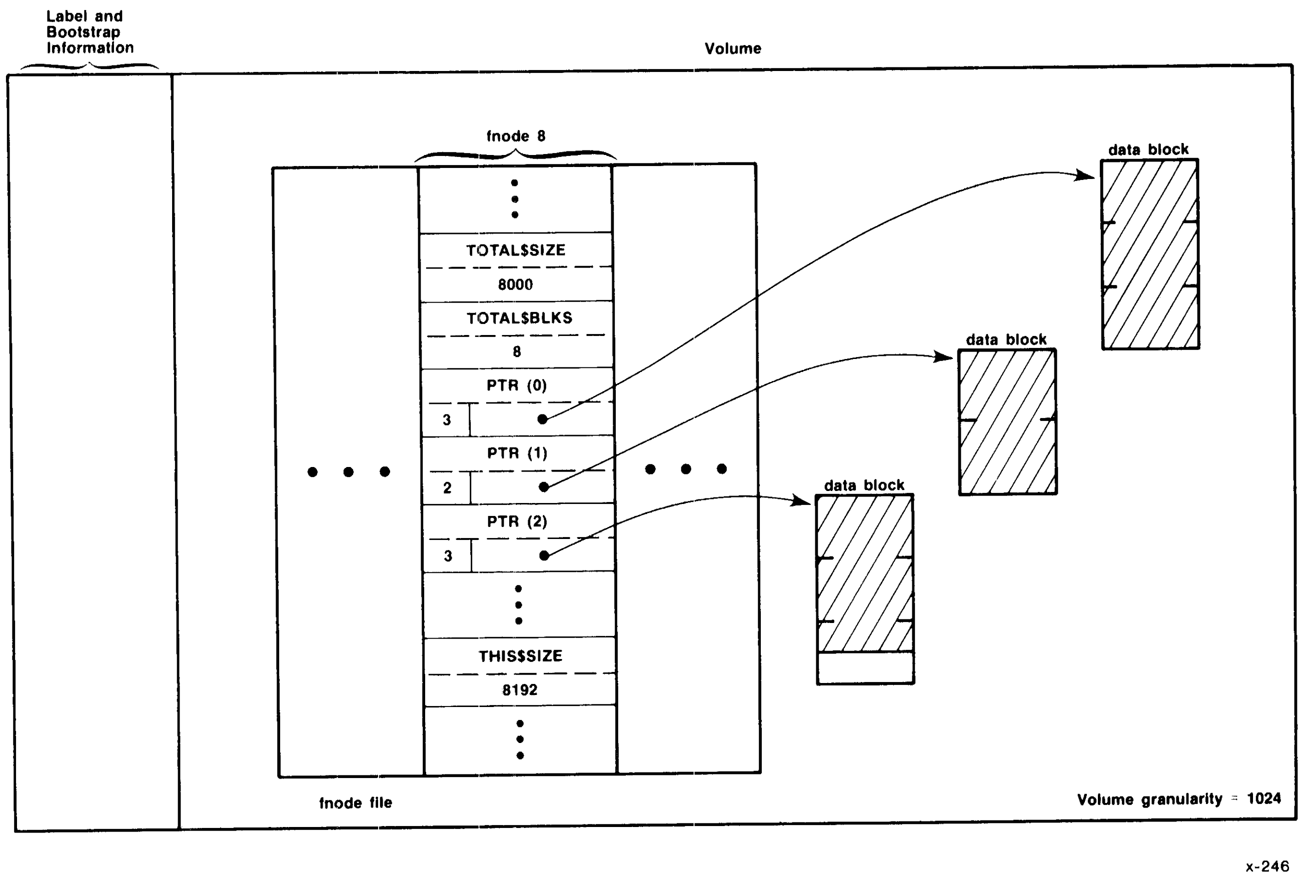


Figure A-2. Short File Fnode

As you can see from Figure A-2, fnode 8 identifies the short file. The file consists of three distinct data blocks. Three PTR structures give the locations of the data blocks. The NUM$BLOCKS field of each PTR structure gives the length of the data block (in volume blocks) and the BLK$PTR field points to the first volume block of the data block.

The other fields shown in Figure A-2 include TOTAL$BLKS, THIS$SIZE, and TOTAL$SIZE. The TOTAL$BLKS field specifies the number of volume blocks allocated to the file, which in this case is eight. This equals the sum of NUM$BLOCKS values (3 + 2 + 3), since short files use all allocated space as data space.

The THIS$SIZE field specifies the number of bytes of data space allocated to the file. This is the sum of the NUM$BLOCKS values (3 + 2 + 3) multiplied by the volume granularity (1024) and equals 8192.

The TOTAL$SIZE field specifies the number of bytes of data space that the file occupies. This is designated in Figure A-2 by the shaded area. As you can see, the file does not occupy all the space allocated for it, and so the TOTAL$SIZE value (8000) is not as large as the THIS$SIZE value.

LONG FILES

If the file consists of more than eight distinct blocks of data, its fnode must specify it as a long file. The fnode for a long file has bit 1 of the FLAGS field set to one. This tells the I/O System that the PTR structures of the fnode identify indirect blocks. The indirect blocks identify the actual data blocks that make up the file.

Each indirect block contains of a number of indirect pointers, which are structures similar to the PTR structures. However, an indirect block can contain more than eight structures and thus can point to more than eight data blocks. In fact, an indirect block can consist of more than one volume block; however, all volume blocks of of an indirect block must be contiguous. The structure of each indirect pointer is as follows:

```
    DECLARE
        IND$PTR        STRUCTURE(
            NBLOCKS    BYTE,
            BLK$PTR    BLOCK$NUM);
```

where:

NBLOCKS	Number of volume blocks in the data block.
BLK$PTR	A 24-bit volume block number of first volume block in the data block. Volume blocks are numbered sequentially throughout the volume, starting with the block with the smallest address (block 0).

The iRMX 86 Operating System determines how many indirect pointers there are in an indirect block by comparing the NBLOCKS fields of the indirect pointers with the NUM$BLOCKS field of the fnode. It assumes that the indirect block contains as many pointers as necessary for the sum of the NBLOCKS fields to equal the NUM$BLOCKS field.

Figure A-3 illustrates an fnode for a long file. Decimal numbers are used in the figure for clarity.

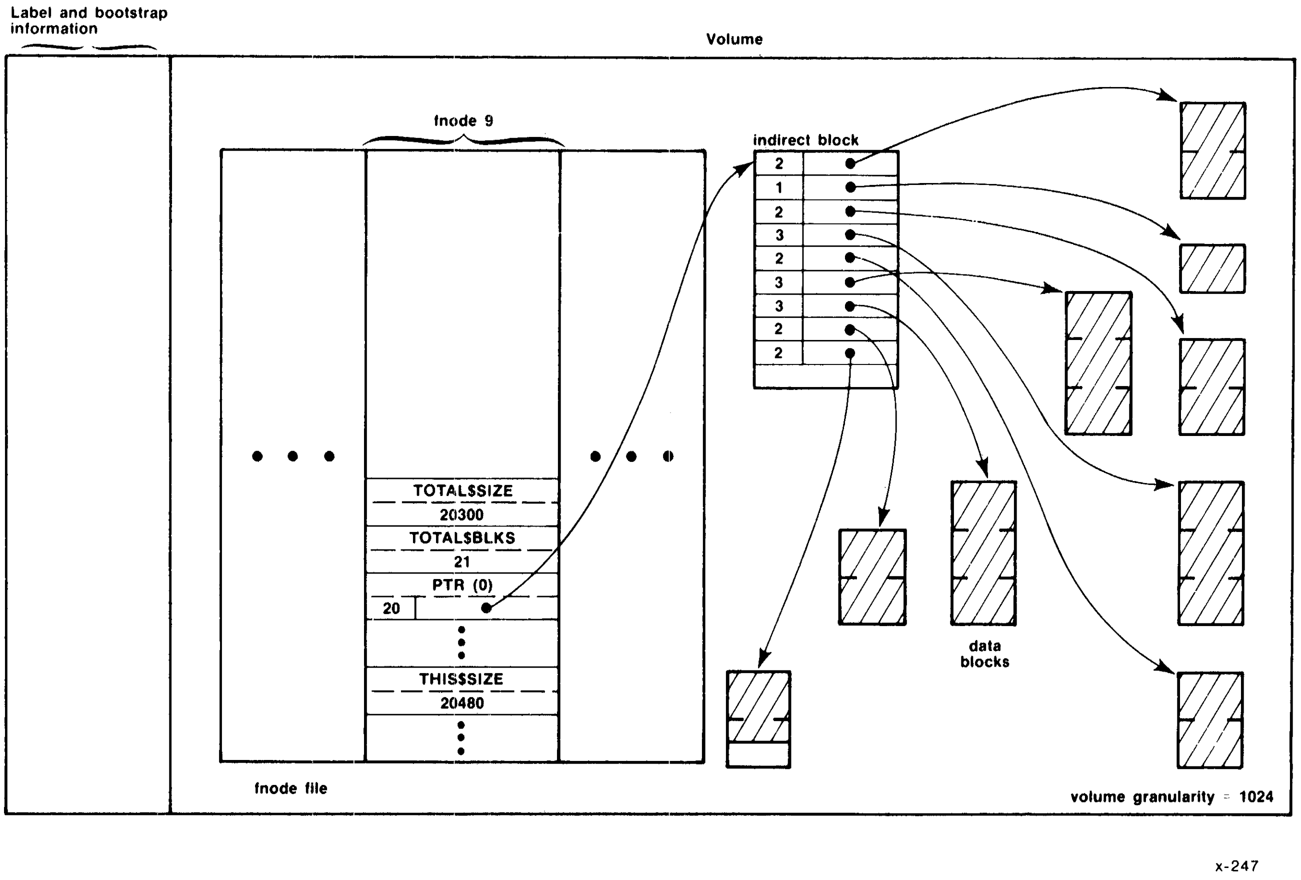


Figure A-3. Long File Fnode

As you can see from Figure A-3, fnode 9 identifies the long file. The actual file consists of nine distinct data blocks. One PTR structure and an indirect block give the locations of the data blocks. The NUM$BLOCKS field of the PTR structure contains the number of volume blocks pointed to by the indirect block. The BLK$PTR field points to the first volume block of the indirect block.

In the indirect block, each NBLOCKS field gives the length of an individual data block and each BLK$PTR field points to the first volume block of a data block.

Figure A-3 also lists the TOTAL$BLKS, THIS$SIZE, and TOTAL$SIZE values, which are more complex than for a short file. The TOTAL$BLKS field specifies the number of volume blocks allocated to the file, which in this case is 21. Twenty of the volume blocks are used for actual data storage and one of the blocks is used for the indirect block.

The THIS$SIZE field specifies the number of bytes of data space allocated to the file, and does not include the size of the indirect block. This size is equal to the NUM$BLOCKS value (20) or the sum of NBLOCKS values in the indirect block (2 + 1 + 2 + 3 + 2 + 3 + 3 + 2 + 2 = 20) multiplied by the volume granularity (1024) and equals 20480.

The TOTAL$SIZE field specifies the number of bytes of data space that the file currently occupies. This is designated in Figure A-3 by the shaded areas. As you can see, the file does not occupy all the space allocated for it, and so the TOTAL$SIZE value (20300) is not as large as the THIS$SIZE value.

FLEXIBLE DISKETTE FORMATS

The flexible diskette device drivers supplied with the iRMX 86 Basic I/O System can support several diskette characteristics. Tables A-1 and A-2 list these characteristics.

Table A-1. Eight-Inch Diskette Characteristics

Sector Size	Density	Sectors per Track	Device Size (in bytes)	
			One sided	Two Sided
128	Single	26	256256	512512
256	Single	15	295168	590848
512	Single	8	314880	630272
1024	Single	4	315392	630784
256	Double	26	509184	1021696
512	Double	15	587264	1177600
1024	Double	8	626688	1255424

Table A-2. 5 1/4-Inch Diskette Characteristics

Sector Size	Density	Sectors per Track	Device Size (in bytes)			
			One Sided		Two Sided	
			40 Tracks	80 Tracks	40 Tracks	80 Tracks
128	Single	16	81920	163840	163840	327680
256	Single	9	91904	184064	184064	368384
512	Single	4	81920	163840	163840	327680
1024	Single	2	81920	163840	163840	327680
256	Double	16	1617921	325632	325632	653312
512	Double	8	1617921	325632	325632	653312
1024	Double	4	1617921	325632	325632	653312

For compatibility with ECMA (European Computer Manufacturers Association) and ISO (International Organization for Standardization), the iRMX 86 device drivers, when called by the formatting tools (the FORMAT Human Interface command and the FORMAT Files Utility command), can format the beginning tracks of all flexible diskettes in the same manner. A configuration option for each driver allows you to specify the following:

- For all 5 1/4-inch and 8-inch flexible diskettes, the device drivers format track 0 of side 0 with single-density, 128-byte sectors, with an interleave factor of 1.
- In addition, for 8-inch, double-sided, double-density flexible diskettes, the device drivers format track 0 of side 1 with double-density, 256-byte sectors.

The iRMX 86 device drivers map the sectors on these beginning tracks into blocks of device granularity size so that the Basic I/O System and Bootstrap Loader can treat flexible diskettes as if they contain a contiguous string of blocks, all of the same size.

However, this mapping is not exact when you use 8-inch, double-sided, double-density diskettes and specify a device granularity of 512 or 1024. A problem arises because there are 26 128-byte sectors in a track, which is not an integral mapping for device granularities of 512 or 1024. Thus the device driver combines the left-over 128-byte sectors of track 0, side 0 with the first sectors of track 0, side 1 in order to make a block of device granularity size. This continues throughout track 0, side 1, but the same problem occurs with the last 256-byte sectors of track 0, side 1; there are not enough sectors to make a block of device granularity size. When the device driver tries to combine these left-over sectors of track 0, side 1 with the first sectors of track 1, side 0, it finds that the sectors of track 1, side 0 are already of device granularity size. Therefore, since the device driver cannot access partial sectors, it is left with one block (the left-over sectors of track 0, side 1) that is less than device granularity size. When the device granularity is 512, this small block is block 19; when the device granularity is 1024, the small block is block 9.

If nothing is done to exclude this smaller-than-normal block from use, the device driver will treat this block as a normal block, assuming it is of device granularity size. Thus if you try to write information to that block, the driver will attempt to write an entire device granularity block of information into a block that is much shorter, causing you to lose information.

To prevent this situation, the Human Interface FORMAT command automatically declares this smaller-than-normal block as allocated in the volume free space map when it formats the volume. This prevents the Basic I/O System from ever writing information into this block. If you write your own formatting utility, you should also declare this block as allocated.

***

INDEX

Primary references are underscored.

***