SOFTWARE MANUAL

INTRODUCTION TO AMOS

DWM-00100-65
REV. A00

alpha micro

SOFTWARE MANUAL

INTRODUCTION TO AMOS

DWM-00100-65
REV. A00

alpha micro

FIRST EDITION

May 1,1980

REVISIONS INCORPORATED	
REVISION	DATE

This book was originally created using the Alpha Micro text editor AlphaVUE and text formatter TXTFMT, and was then typeset using an Alpha Micro computer.

ALPHA MICROSYSTEMS
3501 Sunflower
P.O. Box 25059
Santa Ana, Ca 92799

IMPORTANT NOTE:

This manual is not a guide to system operation. For information on using the system, refer to the *AMOS User's Guide,* (DWM-00100-35), and the documents in the AM-100 documentation packet.

For a complete list of all Alpha Micro software documentation, refer to *A Guide to the Alpha Micro Software Documentation Library,* (DWM-00100-37).

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

This book is the first step in your Alpha Micro education.

As the Alpha Micro software documentation library grows larger and widens in scope, we find that we are talking to many different kinds of people. Our readers all have different needs and interests. And all of our readers come to the Alpha Micro system with various levels of knowledge and experience.

Much of our documentation provides little background information. That is, many of our documents assume that you are both familiar with computers in general and the Alpha Micro system in particular. Although such assumptions are often necessary for the sake of brevity, if you are new to computers or to the AMOS system, you may find that some of our documents just do not give you the information you need to really use your system to its fullest. We realize that dealing with unfamiliar jargon can be frustrating as well as irritating.

We first realized that a book of this type might be useful to some of our readers when we began to get questions about the concepts behind the AMOS system as well as questions about system operation. That is, besides questions like "How does this command work?", we began to get questions like "What is hexadecimal?". The major purpose of this book, then, is to fill in some of the *conceptual* gaps left by our other documentation.

This book does not pretend to be a complete textbook on computers. If you have no experience with computers, reading this book will not tell you everything you will need to know about them. It may, however, answer some of the specific questions that arise when you read our other documentation. We know that different readers will use this book in different ways- you may either want to read it straight through or, more likely, read just those chapters that define the terms you want information about. (For a discussion of which questions each chapter tries to answer, see Section 1.1, "How to Use This Book," below.)

When we began to organize this book, we asked ourselves, "What kinds of questions do people ask about this system?" We found that the kinds of terms we wanted to define and questions we wanted to answer fell rather neatly into three groups: questions that users with little computer experience might ask; questions about the programs that are available on this system; and questions about operating system terms.

Therefore, you will want to read *Introduction to AMOS* if you fall into one of these categories:

1. You are ready to begin using your Alpha Micro system, but you have little computer experience. You may have already looked at the system operations manual, *AMOS User's Guide,* (DWM-00100-35), but you found it tough going because many of the terms were unfamiliar to you. (For example, you may not be sure what "octal" is, or what a "file" or an "operating system" are.)

2. You have had some computer experience before, but are new to the AMOS system and would like some idea of what kinds of programs run under AMOS. In brief, you'd like an introduction to the various language processors, text processors, and utility programs on the system, without having to read through detailed discussions on how to use them.

3. You're either an experienced AMOS user or have had experience with other computer systems, but you are not a systems programmer and are not familiar with operating system concepts or terms.

> You may be interested in the general concepts behind the idea of an operating system, and you may feel the need for a very general overview of the entire Alpha Micro operating system.

Briefly, then, you will want to read this book if you want some background information about the software on your Alpha Micro computer system. We hope that these chapters will get you off to a good start in using the rest of the Alpha Micro software documentation.

Besides giving general information on computers and specific information on the AMOS system, this book also contains an extensive glossary that includes definitions for many of the terms you will run into in other Alpha Micro documentation. Also, note Appendix B, "Where Do I Go From Here?", which directs you to the Alpha Micro software documentation you will want to read next after you've finished this book.

1.1 HOW TO USE THIS BOOK

Depending upon your prior experience or knowledge, you may want to read this book carefully, or just skim it for specific details. The next few paragraphs discuss the contents of this book, and give you a better idea of which chapters you may want to read.

Throughout this book, we try to let you know when a section is coming up that may be of more interest to a specific group of readers than to our general audience. We also expect you to make your own judgments on what topics you want to read about. For example, if you are not interested just now in exploring the differences between "interpreters" and "compilers," we assume that you'll just skip the section in Chapter 10 titled "Compilers and Interpreters," and read something of more immediate interest to you.

We have tried to minimize the interconnections between the chapters so that you can read only those chapters you need, and leave the others until another time. Nonetheless, the chapters do build somewhat on the material presented in earlier chapters. For example, if you aren't sure what "memory" is, you probably won't want to read Chapter 14, "Memory Control and Management," until you have read Chapter 2, "What is a Computer?". The paragraphs below list each chapter with the kinds of questions we tried to answer in that chapter so that you can get some idea of where to find the information you need.

To suit the interests of our different readers, we've divided this book into three major sections:

PART I — GETTING STARTED

Part I is specifically for those of you who are not very familiar with computers, and who would like a little general information on computer concepts. These chapters talk about some common concepts you will often run into when reading other Alpha Micro software documentation. We also introduce you to the Alpha Micro Operating System (AMOS). The major questions these chapters answer are:

Chapter 2, "What is a Computer?":
What are some of the physical components computers are made up of? (For example, what is a "device," "CPU," "bus," "RAM," "memory," and "port"?) What areas of life are computers used in? What are some of the advantages of the Alpha Micro computer system?

Chapter 3, "Who is AMOS?":
What are "hardware" and "software"? How do we communicate with a computer? What is an operating system? What is the basic structure of the software on the Alpha Micro computer, and where does AMOS stand in that structure? What are some of the features of the Alpha Micro operating system?

Chapter 4, "Data and the Computer":
What is "data"? Why does the computer represent data in 1s and 0s? What are the "binary," "octal," and "hexadecimal" numbering systems, and why do we use them on computers? What are "ASCII" and "machine language"?

Chapter 5, "Introduction to Files":
What is the conceptual link between the way we organize information and the idea of a computer file? How does the physical representation of data that we talked about in Chapter 4 become structured into meaningful groups? What happens when the computer processes a file? How are files organized on the disk? What is an "account," "account directory," "project-programmer number," "password," "disk block," "sequential file," and "random file"? What is "logging in"?

Chapter 6, "Permanent Data Storage":
What are the major permanent storage devices we use on the Alpha Micro computer system? What are "System Disks," "System Devices," "hard disks," "floppy disks," "sectors," "tracks," "fixed disks," "platters," "disk packs," "disk cartridges," and "magnetic tape transports"?

Chapter 7, "Programs":
What is the concept of a "program"? What are "flowcharts" and "variables"?

PART II — PROGRAMS AVAILABLE ON THE AMOS SYSTEM

The chapters in this section discuss the major system programs available on the AMOS computer system. You may find these chapters of special interest if you have some prior computer experience, but are new to the AMOS system. These chapters acquaint you with some of the major language and text processor programs that run under AMOS, as well as some of the system utility programs, but do not go into any details on program operation. Some of the questions these chapters try to answer are:

Chapter 8, "AMOS Utility Programs":
What are some of the utility programs available on the AMOS system? What are "HELP files," "hash totals," and "the ISAM system"?

Chapter 9, "AMOS Text Processors":
What are "text processors"; what do you use them for? What are "text editors," "screen-oriented text editors," "character-oriented text editors," and "text formatters"? What text processors are available on the AMOS system?

Chapter 10, "AMOS Language Processors":
What is a "language processor," "computer language," "interpreter," "compiler," "assembler," "assembly language," and "macro"? What language processors are available on the AMOS system?

PART III — AMOS OVERVIEW

Part III is aimed at the programmer or general user of the system who wants more background information on how AMOS works to provide a multi-user, multitasking computer system. That is, Part III gives a general overview of the Alpha Micro Operating System, with information on the major components of AMOS, along with discussions of command execution, user partitions, and system initialization.

These chapters also talk about what happens at the time of system startup, how different terminals and devices are interfaced to the system, and how file accounts are structured on the disk.

Part III is not a guide for the System Operator (the person who manages the AMOS computer system). That is, these chapters do not tell you how to set up a system, or how to allocate user partitions, run disk diagnostic tests, etc. They do, however, give you the very general background information you will need before you go on to the documentation aimed at the System Operator. Some of the questions Part III attempts to answer are:

Chapter 11, "General Structure":
What is an operating system? (We explore this question in greater depth in this section than in Chapter 3, "Who is AMOS?".) What does an operating system do for you? What are the components of an operating system? What is the general structure of the Alpha Micro operating system? What are "terminals," "buffers," and "monitor calls"?

Chapter 12, "Introduction to Jobs":
What is a "job"? What portion of AMOS handles jobs? What is "job scheduling"? What are "Job Control Blocks," "queues," "quantum," and "job priorities"? What does "attaching terminals to jobs" mean?

Chapter 13, "Command Processing":
What is a "command"? How does AMOS handle user commands? What is a "command file," "DO file," "re-entrant program," "relocatable program," "transient program," and "Resident Program Area"?

Chapter 14, "Memory Control and Management":
What is "memory" and why is it important on a computer system? What is a "memory partition" or "user partition"? What are "memory management," "bank switching," "memory allocation" and "memory re-allocation"?

Chapter 15, "Terminal Handling":
What is a "terminal service system," "device driver program," "terminal driver," and "interface driver"? What are "TRMSER" and "character echoing"? What are "half duplex" and "full duplex"? How are characters transferred between AMOS and terminals?

Chapter 16, "How AMOS Handles Devices":
What is a "file service system"? What is a "logical I/O routine," "serial or parallel printer," "Dataset Driver Block," "special device driver," "disk service system," "DSKSER," "disk format," "bitmap," "Master File Directory," and "User File Directory"?

Chapter 17, "System Initialization and Startup":
What is "system initialization," and why is it important? What happens during system startup? What is the "system initialization command file (SYSTEM.INI)"? What does the SYSTEM.INI file do for you?

If you have little prior computer experience, we suggest that you read through the entire book, perhaps merely skimming Part III if you are not interested at this time in how AMOS works.

If you are interested only in an introduction to the major programs that run on the AMOS system, read Part II.

If you are already familiar with using the AMOS system, but would like a general understanding of how the components of AMOS work together, read Part III.

1.2 GRAPHIC CONVENTIONS USED IN THIS BOOK

To make our examples concise and easy to understand, we've adopted a number of graphic conventions throughout our manuals. Below is a list of the conventions we follow in this book:

___	Underlined characters indicate those characters that the computer prints on your terminal display. For example, you will often see examples that begin with an underlined dot. The underlined dot is the AMOS ***prompt symbol***- the symbol that the operating system displays when it is ready for you to enter a command.

The characters in the examples that you are supposed to type are not underlined.

RET — Carriage return symbol. This symbol indicates the place in an example where you would type a carriage return if you were entering the example to the computer. (The carriage return key on your terminal keyboard is usually labeled RETURN or RET.) When we say "Type a RETURN," we mean to say "press the RETURN key."

(DWM-00100-xx) — Part number symbol. Most of the references that appear in this book which are made to other documents are followed by a part number. For example:

> Refer to the *AMOS User's Guide*, (DWM-00100-35).

(If you order a document from Alpha Micro, be sure to refer to that document's part number as well as its title.)

PART I
GETTING STARTED

The purpose of Part I is to familiarize you with the Alpha Micro computer. If you are new to the AMOS system, but are already familiar with terms such as "files," "data," "program," "operating system," "disks," and "octal," you will probably want to merely skim Chapters 2-7, and then turn your attention to Parts II and III for information that is more specific to your AMOS system.

If you are new to computers, you will probably find the next few chapters to be of some help in filling you in on some of the concepts you will need to know before you begin to use the AMOS system.

If at any time you are confused by a particular word or phrase, turn to the Glossary for a fuller explanation of the word. You might also want to consult the Index to find other places in the manual where the word is used.

CHAPTER 2
WHAT IS A COMPUTER?

Imagine a society, if you will, which is dependent for many of its conveniences and services upon a vast number of highly motivated, quick thinking slaves. Picture these slaves as basically educable, having total recall and many inherent abilities, yet as being rather stupid when it comes to communicating with that society. They do as they are told– exactly as they are told– but they make no allowance for what is really meant, if it is left unsaid. To commands not precisely put in their own limited languages, they respond in highly unpredictable and certainly useless ways. Yet they toil, without the slightest effort or distress, through any task assigned to them as long as they are physically capable of doing the job in the first place.

The world-wide society in which we live is just such a society, of course. We increasingly benefit in our arts, humanities, science, business and industry as we expand our use of these so-called slaves. Our slaves are versatile machines known as computers.

2.1 DEFINITION OF A COMPUTER

A ***computer*** is most simply defined as a procedure follower: not too bright but very fast, and dependent upon procedures written in a special kind of language. The procedure, or program, which the computer must follow may vary in sophistication from one which is extremely simple or repetitive to one of almost incomprehensible complexity. But the real strength and value of any computer is that it can be guided through an entire procedure, however simple or complex, by a human programmer to solve real-world problems. And, the solution is fully as accurate as the program itself and the information provided allow it to be.

Most people never get to the point of programming a computer, because the computer is already an almost invaluable tool to them. It becomes to them a thing which appears to think for itself, to work for them, and yet it remains relatively undemanding of their time and resources. Other people learn the limited languages computers use and how to communicate their ideas for procedures to the computer. The power of a computer is greatly amplified here, since it may do a specific task for a certain person or group. And some people actually create computers, aiming the very design of the machine toward a specific field of tasks.

You do not need to understand the physical components of a computer in order to use it well. But a few concepts can be helpful to you for gaining overall comprehension of what a computer is, especially if we talk about those concepts at the outset. Physically, then, the computer is a machine comprised of electronic circuits and their support mechanisms. Electricity moves through myriads of these circuits in a way which is determined by several outside influences. First is the physical design of the circuits: electrons cannot travel paths which were not first placed there deliberately by human designers. Second is the operating system, which animates the computer and enables it to react to the outside world. The third determining influence, and certainly the most important, is you as the user! You initiate every physical action of the computer, making it work to obtain results you desire.

Only the physical construction of the computer becomes the limit to what the computer can do. Depending on how it is set up, a computer can do calculations, control external devices, or store information (known as ***data***) in its electronic memory; or, it may do a combination of any of these, as most complicated applications require.

Keep in mind that there are very few mechanical applications that do not fall, or cannot be made to fall, in one of these categories. This is why computers have become so useful, and why versatility on more powerful computer systems is so necessary.

2.2 COMPONENT STRUCTURE OF A COMPUTER

Most computers are designed to be expandable; that is, they are designed electronically to support other components which increase their overall versatility. We mention here some of the more commonly used components which add power to a computer system:

Central Processing Unit

The *central processing unit (CPU)* is the heart of any computer system. Internal control of the system, processing all instructions and information, and overseeing the interplay of other support equipment attached to the computer are a few of the tasks handled by the CPU.

The architecture of the CPU is designed with particular applications in mind. Some CPUs are designed for maximum versatility at a low cost, others are highly dedicated for specific jobs, and still others are very powerful with a large number of input and output ports and the ability to handle many tasks at once.

The CPU controls all the tasks occurring in the computer system, and it is the CPU which follows the procedure called the ***program.***

Busses

The ***busses*** are the actual paths which electronic signals travel upon among the components of the system. They are important because they are the physical means whereby the various components communicate together as the busses pass working data between them.

Temporary Storage Devices

The CPU must bring information into itself a small portion at a time for processing. But it must have the whole group of information it must proceed through instantly available to it. So, a copy of some information is taken from the place it is permanently kept and inserted into ***temporary storage devices*** where the CPU can access the information immediately. Information in temporary storage can be manipulated, changed, or eliminated altogether, but is lost if power is cut off from the system. (The most common temporary storage device is random-access memory; see below.)

NOTE: Any reference to "memory" in this book means temporary storage. The concept of memory is of primary importance, since the CPU is unable to consider any information not in memory. You will see the term often in various contexts.

Permanent Storage Devices

Before and after processing by the CPU, groups of information must be stored in facilities that are not affected by power removal, and which are accessible to the CPU (even if relatively slowly). These facilities are known as ***permanent storage devices.*** Updated information can be written back to permanent storage, perhaps to supersede and eliminate previous information. Information stays in permanent storage until deliberately altered or removed.

The most common types of permanent storage devices are disk drives and magnetic tape transports. We will talk about them more in Chapter 6, "Permanent Data Storage."

Random-access Memory Devices

Random-access memory (RAM) is quickly accessible memory into which data is copied temporarily for some process by the CPU. Locations in RAM can be drawn upon in any order. RAM is erased when power is cut off, and therefore is used as a temporary storage device only.

NOTE: As used in the computer industry, the word "random" does not necessarily imply a haphazard or unsystematic occurrence. In this case, it describes any group of similar identities (i.e., numbers, memory devices, etc.) in which any single identity may be directly (i.e., "randomly") accessed without reference to other elements in the group.

Read-only Memory Devices

Read-only memory (ROM) is a special kind of memory device known as unchangeable memory. Specific procedures are entered once only and remain regardless of whether or not power is applied. The procedures manufactured into the ROM device can be read by the CPU as instructions or information, but cannot be rewritten or altered in any way by the CPU or system.

Most often used as a device for getting initial instructions into the CPU when the system is starting up, ROM also may be used any time a non-changing, repetitive instruction is useful in a system. Elaborations on ROM include: ***Programmable ROM (PROM),*** a user-defined, fixed memory system which, after being manufactured, can be programmed on special electronic machines; and ***Erasable Programmable ROM (EPROM),*** which the user may actually change to contain other permanent instructions by using another kind of special equipment.

Input and Output Ports

Input/Output (I/O) ports are the physical method the various components of the computer system use to interconnect. They are actually extensions of the busses through hardware connection devices, with one important feature. They are provided with temporary storage devices so that information traveling on the particular bus may be stored until the receiving system component is ready to accept the information, and meanwhile the transmitting component can direct its attention somewhere else.

Peripherals

This term applies to all the devices actually controlled by the CPU which in turn feed responses back to the CPU, if necessary. Some very common peripherals are: interface boards, whereby the CPU can interact with components of a different design architecture; CRT or hard copy terminals, by which the users enter data into the computer and upon which data is returned by the computer; hard (metal) storage disks and "floppy" soft (mylar plastic) storage disks, which increase the system's permanent data storage; disk controllers, which control hard and floppy disks to present data quickly to the CPU; printers, which type data out on paper; magnetic tape recorders, which are also data storage devices; telephone equipment (especially in the reception or transmission of data from one computer to another); card sorters and readers, when data is recorded on punch cards; and paper tape readers, when data is recorded on rolls of paper via punched holes.

2.3 WHERE COMPUTERS ARE USED

Today's infinitely varied world-wide society makes great use of computers and their peripheral devices. From the small single-user systems where individuals find challenge and entertainment, to the gigantic corporate or military systems supporting thousands of users simultaneously, these cooperative machines have become indispensable to many millions. Here are some examples of the ways computers are being used:

As personal computers — Small computers have become microcosms of the very powerful systems. They are used in homes or very small businesses, in self or formal education, or as sophisticated games.

In business — Computers are in use in every aspect of business, including accounting, word processing (such as the text you are reading), inventory, marketing and sales, and in service support. Computers free many persons from chores that require repetition and strenuous precision, allowing them to apply themselves to more interesting endeavors.

In manufacturing and construction — Computers are employed as major tools in both the design and actual construction of products. Computers are being used in new ways by engineers to control machines in order to relieve workers from tasks requiring monotonous precision. Manufacturers are also finding ways to make computers both control and protect the environment for the comfort and safety of society.

World-wide interdependence — Virtually every modern military establishment depends heavily on the performance of computers, large or small. Telecommunications would be impossible at the current levels of usage around the world if it were not for the expandable complexity of computers. International businesses would simply be unable to keep track of their assets and liabilities without the central storehouse of data processed by computers.

Vast government programs — Our military network is beyond the comprehension of any individual, yet it is well cared for by a civilian government due to the extensive coordination possible using computer data. The military also depends upon the speed and accuracy of numerous kinds of computers aboard ships, aircraft and missiles. The National Aeronautics and Space Administration, in its space programs, calculates in moments what a man would take centuries to calculate by hand. As a single example, N.A.S.A uses incredibly complex formulae of rate, time, stress, and so on as it plans and executes a rocket launch. And in the civilian world, every bureau of the government keeps countless details available using the vast storehouses of memory accessed by extremely powerful computers.

2.4 THE ALPHA MICRO SYSTEM

Alpha Micro provides the low cost and highly expandable versatility that the small or medium-sized business or profession is most likely to require. At their inception, these endeavors are least likely to spend huge dollar amounts to buy the computer power they anticipate needing later on, yet they also do not want to be limited later by an inexpensive purchase they make now.

Below we list a few of the ways Alpha Micro solves this dilemma for the professional user. All of these concepts will be discussed elsewhere in this book in detail; this is an overview of the advantages of the Alpha Micro system.

1. The commands the CPU responds to are not an integral part of the operating system itself. Therefore, they can be replaced with new versions of those commands when you want to update your system software.

2. The number of commands is also expandable; that is, if you require a special command, you may write the command and add it to the system. Thus the system remains precisely what you require. And, when Alpha Micro develops a command for general use, you can conveniently add it to your system.

3. When the system starts up, all peripheral equipment that is incorporated into your system is defined to the operating system. This system initialization is set by you. Therefore, expanding the system to contain more terminals, printers or other devices is simply a matter of adding the devices themselves, then redefining the system initialization procedure and adding programs to control those devices.

4. Peripheral device independence allows a great deal of expansion (to the limits of control hardware) once the added device is included in the system initialization procedure. The system therefore can grow as the number of users increases, with no added expense but the cost of the device itself.

5. Timesharing among users is available, of course. In addition, as a user you may actually timeshare with yourself. (This is called ***multitasking.***) For example, you may print a listing and at the same time you can access a file to edit it. Also, several users, each programming the computer to do different tasks, may use either the same language or different languages.

Neither the personal accessibility of the small system nor the power of the relatively large system is outside the range of the Alpha Micro Operating System. The computer is an important tool in our modern society. Because the professions and business are so large a share of that society, Alpha Micro brings to the professional user a number of versatile, economic and expandable advantages. The chapters that follow define in much greater detail the specific means by which AMOS may be put to use for your precise needs.

CHAPTER 3
WHO IS AMOS?

AMOS (the Alpha Micro Operating System) is the heart of the Alpha Micro computer system; it is very obedient, has infinite patience, and is extremely quick and efficient.

Think of it as a cross between a master switchboard operator, language translator, dispatcher, and administrator. Because of AMOS, you can post today's inventory changes while someone else is updating payroll data. At the same time, another person might be composing and printing a book chapter while yet a fourth worker is creating his or her own program that works on calculus problems. AMOS makes possible the complicated interactions that arise as a result of multiple activities on the same computer at the same time.

We will talk in more detail about AMOS in Part III, "AMOS Overview," but we would like to give you a brief introduction to AMOS now. This introduction discusses the concept of an operating system, the basic structure of your Alpha Micro software, and AMOS's special abilities.

3.1 COMMUNICATING WITH THE COMPUTER

In the last chapter, we introduced you to some of the elements that make up a typical computer system. We usually call these physical components ***hardware.*** As we've already pointed out, computers are very fast, but rather straightforward. A computer will do nothing unless you give it instructions. The instructions that make the computer function are commonly called ***software.*** The purpose of software is to give you a way to communicate with hardware.

At its most primitive, this form of communication can consist of using switches or keys to enter numbers that the CPU can directly interpret as instructions.

These fundamental instructions make up what is called the ***instruction set*** of the computer, and usually are of very limited scope. For example, a typical machine instruction might tell the CPU to move a number from one memory location to another.

To get the computer to perform one simple function (for instance, to copy data from one disk to another) you might have to enter hundreds of these numbers. (Such a group of instructions is called a ***machine language program.***) But, the purpose of a computer is to save you work, not to cause it! Therefore, most computer systems allow you to communicate with your computer in a form that is much more convenient for humans. To make life easier for you, the more advanced computer systems provide a set of programs called the ***operating system*** or ***monitor.***

Since the CPU *only* understands numbers that represent its instruction set, it has no idea of what you are trying to say if you type in the word:

```
HELP (RET)
```

With AMOS (the Alpha Micro Operating System) acting as your translator, you can enter instructions in a form that is natural for you (that is, as letters and words); the operating system translates your orders into a form that the CPU can understand. Of course, the operating system has a finite vocabulary. If you type in:

```
HELP (RET)
```

AMOS gives you a list of topics it can supply information about. This is only because that ability has been programmed into AMOS. If you type in something outside of its vocabulary, for example:

```
    WHAT WAS GEORGE WASHINGTON'S HORSE'S NAME? RET
```

AMOS replies:

```
    ?WHAT ?
```

indicating that it is not able to understand you. We call the full range of AMOS's vocabulary the ***AMOS command language.***

The operating system allows you to give much more powerful and comprehensive orders to the computer than if you were forced to communicate directly in machine instructions.

Besides giving you a convenient way of communicating with the CPU, an operating system also provides a myriad of other services. It connects and supervises all of the various programs that take care of such things as communicating with terminals, writing data to the disks, running several users on the system at the same time, and finding disk files. (In Section 3.3, "The Alpha Micro Operating System," we discuss some of AMOS's special abilities.)

You could, of course, write your own programs to do all of the things you want to do on a computer system. But, then you would have to worry about all of the complicated procedures involved in transferring data between terminals and the computer, disks and the computer, etc. Getting all of those programs to work together without conflicting is a problem of the highest magnitude.

The purpose of an operating system is to make a computer look simple. Few people really know all of the complex and convoluted actions that take place behind the scenes when we ask AMOS to do something like copy a file from one disk to another. Fortunately, the operating system takes care of performing the trick for us, and we don't have to worry (or even think about) the intricate machinery behind the magic.

3.2 SYSTEM STRUCTURE

We'll go into much more detail on this subject in later chapters, but it is important at this point to get a feeling for the structure of your system. At the root of the structure is the hardware– the computer itself and the peripheral devices that allow you to enter data into the computer and to store, display, or print data.

Enabling you to communicate with the computer (by translating your orders into machine language) is the operating system, AMOS. (For a discussion of the various components within AMOS itself, see Chapter 11 "General Structure." For now, just remember that AMOS is simply another program, even though it oversees all other programs that run on the system.)

Supported by the AMOS program in this hierarchy are other system programs, such as: the screen-oriented text editor, VUE; the BASIC compiler, COMPIL; and the BASIC run-time package, RUN (which executes BASIC programs). AMOS oversees the running of every program on the system.

Sometimes other programs are at even a higher level in this structure. For example, when you execute a BASIC program, it runs under control of RUN, the BASIC run-time package. RUN, in turn, is controlled by AMOS. The BASIC program is thus higher in the structure than AMOS or RUN.

The diagram below illustrates this system structure:

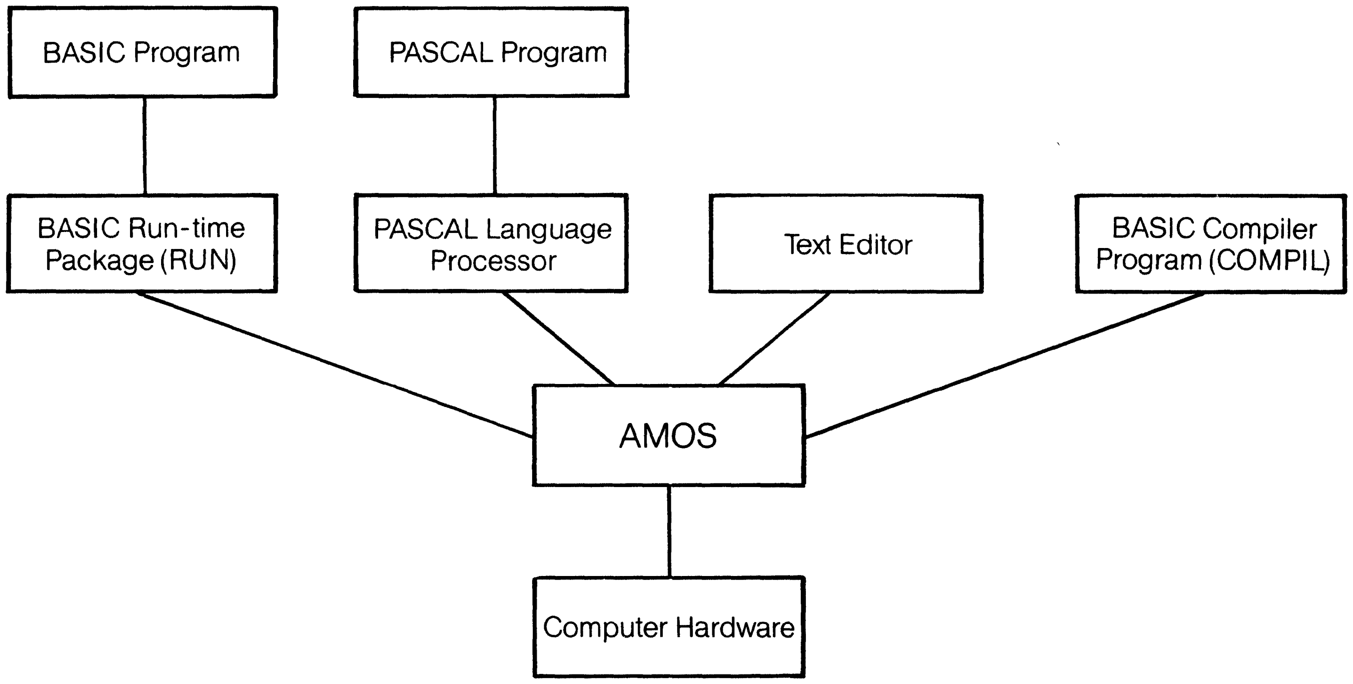

BASIC Program
PASCAL Program
BASIC Run-time Package (RUN)
PASCAL Language Processor
Text Editor
BASIC Compiler Program (COMPIL)
AMOS
Computer Hardware

Figure 3-1
Sample System Structure

Things are really a little more complicated than indicated by the diagram above (which lists just a few of the components of your system), but you now have the idea that many levels can exist within your system software. When you are at the level where you can communicate with AMOS, we say that you are "at AMOS command level."

The reason we emphasize the hierarchial nature of this structure is that it helps explain why entering a command phrased in the vocabulary that the BASIC language processor understands, for instance, doesn't work when talking to the operating system. For example:

```
LET AVERAGES=(B*C*D)/365
```

means nothing to AMOS, but is a perfectly valid statement when talking to BASIC. At AMOS command level, you can communicate directly with AMOS. When you are communicating with BASIC (that is, when you are "inside BASIC"), you can talk with BASIC, but not directly with AMOS. When you are using the text editor, VUE, you can enter VUE editing commands, but cannot enter BASIC commands or AMOS commands.

Often the particular program you are talking to displays a distinctive symbol called a ***prompt*** which serves to remind you which program you are communicating with. For example, when you are at AMOS command level you see a dot. This is the AMOS prompt. When you are inside EDIT (the Alpha Micro character-oriented text editor), you see an asterisk. A prompt tells you that the program you are communicating with is ready for a command.

3.3 THE ALPHA MICRO OPERATING SYSTEM

The complexity and sophistication of the Alpha Micro system requires a sophisticated and powerful operating system. One mark of the flexibility of the AMOS system is that most of the components of the operating system are not "canned into" the operating system as they are in some other computer systems. For example, the programs that allow you to access terminals and devices are not physically part of the operating system, but are separate programs called by AMOS. The same is true of the commands you can enter to the operating system. This means that you can continue to expand the capabilities of your system, simply by writing your own assembly language programs or by adding new device driver programs and commands as Alpha Micro makes them available.

If you have had some experience with other computer operating systems, you may be interested in taking a look at some of the features of AMOS:

Command Language AMOS translates your orders and performs the appropriate action. For example, when you type DIR followed by a RETURN, you are telling AMOS that you want to see a directory of the files in your account. (Files and accounts are structures AMOS uses to organize data on the disk. We'll talk more about them in Chapter 5, "Introduction to Files.") AMOS allows you to communicate with programs other than itself. For example, by typing "BASIC" followed by a RETURN, you tell AMOS that you want to communicate with the BASIC language processor. (BASIC is an easily learned language that you can use for writing computer programs.) You can also instruct AMOS to run programs you have created yourself. The set of valid orders you can give AMOS make up its ***command language.***

Device Independence AMOS talks with the various hardware components that control and interface with the ***peripheral devices*** on your system (disks, magnetic tape units, printers, terminals, etc.).

One of the strengths of the Alpha Micro computer system is that it allows you to change the configuration of your system (that is, add new disk devices, new terminals, etc.) by adding new software programs that handle those devices. When your programs communicate with hardware (for example, when a program sends data to a printer), the programs go through AMOS who is able to do the actual device access. The fact that you can add new devices to your system at any time, and that the system or a command is not restricted to any specific type of disk or terminal, is called ***device independence.***

Timesharing Besides overseeing the tasks you want to perform, at the same time AMOS also handles the needs of other users on the system. AMOS allocates a certain amount of CPU time for each user on the system. When you have used your allotted share of time (usually 1/60 of a second), the CPU turns its attention to another user. However, the computer can perform so many actions in such an incredibly short time, it is not usually evident that you are sharing the CPU with other users. In other words, the CPU returns its attention to you so quickly that you are often not aware of any delay between your command and the computer's response. This ability to handle several users on the system at the same time is called ***timesharing.***

In addition to allocating CPU time, AMOS also allocates other system resources such as disk space, memory use, printer use, etc.). You can instruct AMOS to let you perform more than one task at a time (for example, run a BASIC program at the same time that you are printing something). This process is called ***multitasking.***

Multi – programming

Unlike some timesharing systems, AMOS allows users running at the same time to run *different* programs. This ability is called ***multiprogramming.*** For example, you might be talking to the BASIC language processor at the same time that another user is executing a program written in the PASCAL language. Meanwhile, yet another user might be creating an office report using one of the system text editing programs.

Memory Management

AMOS also has another ability that is rare in microcomputers: it can access more than 64K of memory by using an option called ***memory management.*** A CPU that deals with 16-bit numbers can only reference 65,536 memory locations (that is, 64K of memory), because 65535 is the largest number we can represent in 16 bits. (Each location in memory is referenced by a unique number from 0 to 65535.)

This is a severe restriction, because it limits the number of users that can run on one system at a time. (All users on the system require a certain amount of memory to perform their tasks. Also, the operating system itself needs a certain amount of memory in which to work.)

On the Alpha Micro system, however, you can set up more than one set (or ***bank***) of memory so that AMOS can select among different banks. For example, you may be using memory locations 16384-32768 in Bank One while another user accesses memory locations with the *same* addresses, but in another bank. The system still restricts each individual user to a maximum of 64K of memory, but you can have several different sets of 64K on the system. For more information on memory management, see Chapter 14, "Memory Control and Management."

We will be talking more about most of these concepts in Part III, "AMOS Overview." In the next four chapters we discuss other general computer concepts such as "data," "files," "storage" and "programs." If you are already familiar with these terms, you may want to simply skim these chapters before you begin to read Part II, "Programs Available on the AMOS System."

CHAPTER 4 DATA AND THE COMPUTER

Before we go on to describe how AMOS is able to handle the many tasks you saw in the previous chapter, and many more which you will see later in this book, we will reduce such concepts as data, information, programs, numbering systems, ASCII and machine language to their simplest terms. We will also consider how a computer can communicate with the real world, and especially with you as the user.

Remember that a computer is an electronic device which merely conducts electricity through a vast maze of switch-like circuits in a way determined by three influences. One is the physical architecture of the computer's components and circuits; another is the operating system which vitalizes the computer and makes it responsive to the outside world. But, most importantly, *you* determine the computer's activities because it must handle data given by you and return results you desire.

4.1 WHAT IS DATA?

The word ***data*** is a general term for the symbols used to describe ideas, objects, situations, values or abstractions when they are collected for logical processing. We use the term as it applies to symbols presented to a computer, which is a logical processor. A computer cannot recognize the nearly unlimited varieties of content and structure symbols you can absorb from any information you are exposed to. It is an electrically oriented machine, limited to processing only a subset of information in small steps when expressed in a form (still comprehensible to you) that can be broken down into combinations of two numeric symbols (1 and 0). Therefore, data is defined as a precisely structured, specific kind of information retaining meaning for you, but which can be broken down by the computer into a series of physical units of electricity which are present or absent at a given point, in a given instant, within the computer.

4.2 THE BREAKDOWN OF DATA

Let's imagine for a moment a single switch, controlling, we'll say, a light bulb. The switch only has two states: it is either on and the light is lit, or it is off and the light is out. Since electricity is at the bulb and doing work when the switch is on, we'll label the ON position with a 1. When the switch is off, electricity is not present, so we'll label the OFF position with a 0.

Remember, the final result at the light bulb results from whether or not it has electricity. Setting the switch to 1 lights the bulb, or could be said to return a 1. Setting the switch to 0 turns off the bulb, or returns a 0.

Let's say you are stationed at the switch, and the light bulb is positioned to illuminate a room. Let's also assume a director, who (for reasons you neither know nor care about) speaks commands which are directed to you regarding the illumination of that room. When the director says "Light, go on," you recall that the words "go on," though they do not appear on either of the two labels on your switch, do mean "= 1." You quickly throw the switch to the position labeled 1. When the director says, "Light, I'm done looking now; you may shut off," you again search your memory to see what that instruction equals. When you do not find it, you either ignore the director or do what you've remembered to do when the director gives you an unexecutable command. That is, you can't handle his information because he didn't present it to you as an instruction, so you're not obligated to position your switch in the way that will return to him the desired "= 0" he was hoping for.

If the director says "Change," and that data recalls to you an instruction that resides in your memory telling you to push the switch the opposite direction, he gets his results as the light bulb goes out. You were able to execute several kinds of instructions because he gave you data in such a way that you could interpret it as instructions regarding the two positions on your switch.

In this example, our director was actually the user employing the computer as a tool to help accomplish some purpose of his own. You were the operating system, a model of AMOS as you saw it in chapter 3. You performed two different kinds of data handling in the example, though there are many more. First, you pulled interpretations out of your memory to convert data words into the numbers 1 and 0, which represented performable instructions. Second, you pulled out programs to perform the specific instructions and caused the switches, or physical hardware, to pass on various combinations of 1 and 0 as the presence or absence of electricity. Control of the light bulb, of course, was the result the user (director) wanted as he gave you specific data to convert to instructions, and then only as a tool to help him see into the room. With the proper data input, he did just that.

In a computer, there are literally millions of microscopic, electronic "switches" which control the electrical passage of data. Yet each switch that can pass data is set either at 1, providing an electrical ON (electricity present), or at 0, providing an electrical OFF (electricity absent), to the next switch down the line. The interaction of these switches, and the trillions of possible electrical paths they combine to form, provide the physical means for a computer to process data for the user.

4.3 BINARY NUMBERS AND DECIMAL NUMBERS

So that data can be represented using only 1s and 0s, the binary, or base 2, numbering system is the mathematical foundation upon which the computer is built. In the binary system, only 1 and 0 are used, and all quantities in the numbering system can be represented by a combination of those two symbols.

We commonly use the decimal, or base 10, numbering system, and you are familiar with these symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. When you see the combination of two symbols 1 and 0, or 10, you call them by a special name "ten," but they are really ONE-ZERO, base ten. When you count to "ten" you say, "(zero,) one, two, three, four, five, six, seven, eight, nine, ten." You identify by various sound-symbols the numerals representing quantities. In the binary system, you do the same thing, but to count quantities you say, "(zero,) one, one-zero." One-zero in base two describes the same quantity as represented by the symbol 2 in our familiar decimal system. The terms "binary" and "base 2" indicate that this numbering system contains only two symbols to represent all quantities in the system.

Each "place" in a decimal number represents a power of 10. For example, the number 204 (base 10) means:

2 hundreds, 0 tens, and 4 ones (or $2 \times 10^2 + 0 \times 10^1 + 4 \times 10^0$)

In the same way, each position in a binary number represents a power of 2. For example, the number 11001100 (base 2) means (in decimal notation):

1 one hundred twenty eight, 1 sixty four, 0 thirty two, 0 sixteen, 1 eight, 1 four, 0 two, and 0 one

(or $1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$)

which again equals 204 (base 10).

Conversion from decimal to binary and back again is a little ponderous for humans, but easy for computers. Since it helps to have the concept, we detail a few of the very basic methods of decimal-to-binary and binary-to-decimal conversion, along with conversion methods for other numbering systems, in Appendix A. Refer to any textbook on the subject for more detailed exploration. But keep in mind that the chart and explanations are dealing with alternate numeric symbols which *represent* the SAME quantities.

4.4 ASCII

There are no purely mathematical values equivalent to non-numeric symbols. Since you intend to symbolize not only quantities but ideas, objects, situations and the like as your data, you need a way to represent those non-numeric symbols by numbers of some sort, since the computer can only handle numbers. Therefore, computer designers and operators have agreed to use a system that simply assigns numeric values to non-numeric symbols. It is called ***ASCII,*** the American Standard Code for Information Interchange. (See Appendix A, "The ASCII Character Set," in the *AMOS System Commands Reference Manual,* (DWM-00100-49), for a complete listing of the conventions of symbols and their assigned numbers in the popular (among computer users!) octal and hexadecimal numbering systems. See Section 4.6 of this chapter for further explanations of those numbering systems.) Using ASCII, data presented to the computer can consist of symbols for letters, numbers, punctuation or control characters. The computer first recognizes the ASCII numeric equivalent of the symbol in the binary numbering system, which is converted to electricity right at the input terminal via hardware. Then the operating system checks to see what that binary data means. As we saw in our example above, valid data is operated upon, while invalid data causes some unintended return to the user from the computer.

As an actual example of an operating system recognizing and acting upon valid data, use one of the AMOS commands called DING (which rings the terminal bell). Type DING 3 followed by a RETURN after the AMOS prompt symbol which appears on your terminal display:

```
.DING 3 RET
```

You will immediately hear three rings or beeps. But the computer in the meanwhile took every key as you pressed it, converted it to its ASCII value in binary, and placed it in a buffer, or temporary storage area. Then, when you pressed the RETURN key, a string of data in the form of electricity was accepted by AMOS that we symbolize as:

0000 0000 0100 0100	Which equals upper case D
0000 0000 0100 1001	Which equals upper case I
0000 0000 0100 1110	Which equals upper case N
0000 0000 0100 0111	Which equals upper case G
0000 0000 0010 0000	Which equals a space
0000 0000 0011 0011	Which equals 3

AMOS recognized the DING command and executed it. The DING program determined that three electrical signals were required at the bell circuit, properly spaced and so on. When the command was processed, you heard the tones.

Incidentally, as written above, we have included spaces after every four digits of the binary number. This is for clarity, and not inherent to binary numbers or commonly accepted representations of them. Elsewhere you will see 16-digit binary numbers expressed with different spacing or with no spaces at all.

4.5 MACHINE LANGUAGE

Once data is represented in ASCII binary equivalents, it must be stored temporarily or permanently, reproduced, or otherwise computed. The operating system controls this computation. The computation is done physically (that is, in hardware) by the CPU and some of its peripherals, which are actually responding to the presence or absence of electricity as described in Section 4.2 of this chapter. ***Machine language*** is the set of electrical patterns (represented as 1s and 0s to indicate the presence or absence of electricity) which, by taking various paths through the architectural structure of the CPU, make the CPU act as if it is responding to instruction or data inputs. In other words, machine language is the one actual set of symbols directly converted to electricity that a given computer can physically handle. All symbols not found in the set of machine language instructions must first be transformed by some method into machine language to be acceptable to the CPU.

A particular machine language is unique to one type of CPU, because every CPU is deliberately designed by electronics engineers and scientists for different specific or general applications.

Once converted into machine language by the computer, data is both handled and stored in its binary form. Storage is mandatory for the computer to do any significant work with data. But data is almost never stored nor handled at a rate of a single 0 or 1 (electrical absence or presence) at a time. Those electrical states and their representative symbols 0 and 1, which are known as ***bits,*** are usually handled many at a time. The Alpha Micro Operating System is built around a CPU, for instance, that handles up to 16 bits of data at the same instant.

Those 16 bits arrive simultaneously on a 16-lane freeway known as the ***data bus.*** That is, electricity may be present on some of the data "lanes," while on the others electricity may be absent. (There is no difference in significance between the presence or absence of electricity at a given point, since both represent data of some kind. Bits are therefore not usually considered for their electrical value, but for the data they represent.) There are 65,536 (base 10) possible combinations of bits arriving or departing on the data or other system busses per machine cycle (which is measured in parts of millionths of seconds). This includes:

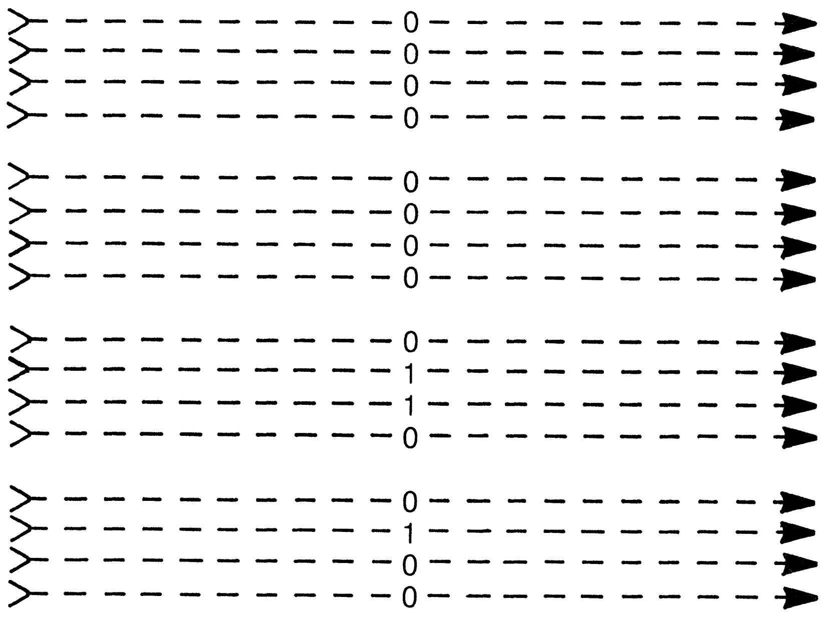
    0000 0000 0000 0000 which equals 0

and:

    1111 1111 1111 1111 which equals 65,535 (base 10), also known as 64K

There are some conventional terms regarding the grouping of bits. In most systems, 8 bits together are known as a ***byte.*** In the Alpha Micro system, two 8-bit bytes are known as a ***word.*** In some systems where 8 bits are the maximum unit, a byte and a word are synonymous.

The example below shows a "typical" word of data which might appear in a given cycle traveling one way on one of the bi-directional busses.



If you type the uppercase letter X followed by a RETURN on your terminal keyboard, the CPU sees the above word come in on its data bus. AMOS might not recognize that character as meaning anything significant, so it queries you:

?X ?

In order for the computer to display the message above, it must return to your terminal in 8 separate cycles these binary numbers:

```
0000 0000 0011 1111
0000 0000 0110 0100
0000 0000 0010 0000
0000 0000 0010 0000
0000 0000 0010 0000
0000 0000 0010 0000
0000 0000 0010 0000
0000 0000 0011 1111
```

which are interpreted by the terminal via ASCII to read ?X ? when displayed.

Bytes are usually further combined into large groups and quantified with the term "K," for "kilobytes" (e.g., 1K or 64K). The term was defined by engineers and scientists working with binary numbers who wanted a short and handy symbol to represent 2^{10} (two to the tenth power), or 1024, and multiples of it. Therefore, 32 Kbytes is, for example, 32,768 bytes and 64K is 65,536 bytes. The real-life handling of data, of course, finds large groups of bytes or words very commonplace.

Still another term for very large groups of bytes is "M," for "megabytes." 2^{20} (two to the twentieth power), or 2^{10} times 2^{10}, equals 1,048,580. 1 megabyte (or 1 Mbyte) is 1,048,580 bytes. Sometimes a group of data is several megabytes in length.

4.6 MORE NUMBERING SYSTEMS

As we said earlier, binary code is ponderous. If you need to program the computer in machine language (in the interest of program speed, for instance), you would find it extremely difficult to do lengthy or tedious programs the necessary byte or word at a time.

Bytes and words are oriented around multiples of 2 (i.e., 8 and 16), and are expressed in their normal form using binary numerals. But by using the numbering systems based upon those multiples of 2, it becomes possible to write binary values in what can be described as binary shorthand.

From the outset, please understand that the computer interprets any shorthand methods, expanding them into binary. There is never anything but binary in the machine. It is for the convenience of the programmer that shorthand methods are used. They are nothing more than a way to form shorter expressions of binary data; they are used because they require less symbols per value, and are generally easier to comprehend than their binary equivalents.

4.6.1 Octal

The octal numbering system, or base 8, is so called because it uses eight symbols to represent all quantities in the system. As a method of binary shorthand, it reduces the number of digits expressing a certain value as written in binary. A value expressed in sixteen digits (base 2) can be expressed in six digits (base 8). For example:

1 101 010 111 001 110 (base 2) = 1 5 2 7 1 6 (base 8)

And an eight-digit (base 2) numeral can be expressed in three digits (base 8):

10 010 111 (base 2) = 2 2 7 (base 8)

A quick method of mental conversion from binary to octal (once familiar with base 2, of course) is to make sure there is a multiple of three digits in the binary expression; if not, add or imagine placeholders (zeros) to the left until there is. Then, from left to right, convert each group of three digits into its octal equivalent just as you would its decimal equivalent. You will notice that no digit exceeds a 7:

```
000 = 0
001 = 1 (base 8)
010 = 2 (base 8)
011 = 3 (base 8)
100 = 4 (base 8)
101 = 5 (base 8)
110 = 6 (base 8)
111 = 7 (base 8)
```

Therefore, the sixteen-digit binary number 1101010111001110 seen above is converted:

```
[(00)1]   [101]   [010]   [111]   [001]   [110]
   1        5       2       7       1       6
```

Once again, remember that the octal system is merely a way to express binary numbers in shorthand, and is not something the computer recognizes except in its ability to convert it to the required binary.

4.6.2 Hexadecimal

The hexadecimal, or base 16 numbering system (often known as ***hex***), is so called because sixteen unique symbols are used to represent all values in the system. A binary value of sixteen digits is expressed in four digits (base 16). Here is an example with the same number we have been using:

1101 0101 1100 1110 (base 2) = D 5 C E (base 16)

An eight-digit binary numeral can be expressed in two digits (base 16):

1001 0111 (base 2) = 9 7 (base 16)

Quick mental conversion to hex is similar to octal except that binary digits are placed in groups of four rather than three. Again, placeholders are implied. Alphabetic symbols A, B, C, D, E and F represent numeric values also, and are not to be considered as letters. Convert the groups of four from left to right into their equivalent base 16 value:

0000 = 0		
0001 =	1 (base 16) =	1 (base 10)
0010 =	2 (base 16) =	2 (base 10)
0011 =	3 (base 16) =	3 (base 10)
0100 =	4 (base 16) =	4 (base 10)
0101 =	5 (base 16) =	5 (base 10)
0110 =	6 (base 16) =	6 (base 10)
0111 =	7 (base 16) =	7 (base 10)
1000 =	8 (base 16) =	8 (base 10)
1001 =	9 (base 16) =	9 (base 10)
1010 =	A (base 16) =	10 (base 10)
1011 =	B (base 16) =	11 (base 10)
1100 =	C (base 16) =	12 (base 10)
1101 =	D (base 16) =	13 (base 10)
1110 =	E (base 16) =	14 (base 10)
1111 =	F (base 16) =	15 (base 10)

Therefore, our sixteen-digit binary number is converted to hex:

[1101]	[0101]	[1100]	[1110]
D	5	C	E

Remember that this also is simply a method of shorthand to make binary numerals a little more handy to the programmer. Remember also that the characters A, B, C, D, E and F are simply value representations for the binary numbers 1010 through 1111.

4.7 THINGS TO COME

In the next chapter we will deal with larger groups of data made up by a multitude of bits contained within bytes or words. We will introduce and expand upon the idea of files, which are useful groups of data related by subject matter. Files are actually processed and saved as 1s and 0s, but may be returned to you in ASCII characters as octal, hexadecimal or binary numeric symbols. Files exist to allow you to comprehend quantities of related information, even though the computer still reduces that information to a form it can handle.

CHAPTER 5
WHAT IS A FILE?

We've already brought to your attention the fact that the computer does nothing but handle data. Whether that data is your company's payroll, a set of scientific calculations, or a book chapter, the computer can transfer, manipulate, erase, and transform that data with incredible quickness compared to the speed with which a human could perform those tasks.

Everything you do on the computer somehow involves handling data. Now that you know how the computer represents data internally, it's time to talk a little about how *you* can access that data. This chapter discusses the basic unit of data organization– the file– and how the computer handles files. (NOTE: IBM system users may know files as "data sets.")

5.1 HANDLING DATA

To begin our discussion of how the computer manages data, let's assume that you run an office which processes large amounts of information. How might you manage the problem of efficient data processing?

> Suppose you have a very large number of filing cabinets. These filing cabinets hold a great deal of data– in fact, each drawer may hold hundreds of manila file folders that contain memos, reports, numbers, etc.
>
> Accessing the information in the files takes a little while. (You have to find which drawer holds the information you need, and then you must go through the physical process of pulling out the proper manila folder.) So, every time you want to work on a particular file, you don't want to have to go get it from the filing cabinets. Besides taking time, this process keeps anyone else from working on the file at the same time. Therefore, you have ingeniously set up a photocopy machine next to the cabinets. When you are going to be working on a particular file, your administrative assistant quickly copies the file you want to work on, places the copy on your desk, and puts the original back in the filing cabinet. At your desk, you can work quickly and easily on the file.
>
> When you are finished with a file, your administrative assistant often places the edited file back into the filing cabinet, replacing the original file. Sometimes, of course, you don't change the file, but simply want to use it or look at it. In this case, the administrative assistant just throws away your copy of it when you are done, without updating the original copy in the filing cabinet.
>
> The only possible problem with this scheme is that you have an overly zealous janitorial staff that throws away any files left on your desk at the end of the day. Before you quit for the night, therefore, you must remember to place the files on your desk in the filing cabinet if you have changed them from the originals, because your desk is only a temporary storage area.
>
> Imagine now that you have ten to fifteen people in your office, all doing the same kind of data processing you are doing. Obviously, all of these people are going to have to go through your administrative assistant; otherwise, chaos is going to result as one worker goes to the filing cabinet to get a file that another worker is already photocopying. The administrative assistant assigns each worker his or her own locking filing cabinet drawer, and makes sure that workers don't put their files back into the wrong drawers.

This limited analogy gives us a starting point for our discussion. The computer, too, has very large permanent storage facilities and much smaller temporary storage areas. This distinction between permanent and temporary storage is a very important one.

The devices most frequently used for permanent storage are disk drives and magnetic tape units. These types of devices store data by magnetically writing the data onto a recording surface (somewhat as a tape recorder "writes" music to a tape). We call this permanent storage because the data remains on the recording media even if you turn off the power to the storage device. You may, of course, erase any of the data you no longer need.

Since the most commonly used permanent storage device is the disk drive, the rest of our discussion will assume that you are storing data on disks. Each location on the disk is associated with a unique number; AMOS can access that location by number when you want to read or write data. (For more information on disk drives, see Chapter 6, "Permanent Data Storage.")

Each one of these filing cabinet equivalents may contain a great deal of data. For example, one unit of a Phoenix disk drive may contain 15 million characters. At 2000 characters a page, one unit can store approximately 7,500 pages of text. You do not usually have a 7,500 page book to store. The groups of data you want to store are generally much smaller (for example, a 25-line BASIC program or a five-page report). Therefore, the disk must be divisible into groups of different sizes.

AMOS structures the data on the disk by grouping it into separate sets (such as memos, reports, programs, payroll data, etc.) called ***files.*** Files are much the same as the manila file folders in our example above. These files in turn are grouped within other structures called ***accounts.*** As in the cabinet drawers of our example above, accounts serve to identify the owners of files, and to help prevent users from accessing the same files at the same time. (Remember: even though files always contain binary data, that data can represent many different kinds of things, such as machine language programs, letters and reports, BASIC programs, etc.)

The temporary storage in your computer is called ***memory.*** Unlike the disk, when you turn off power to memory, its contents disappear (as in the case of the very efficient janitors in our example above who clear off your desk every night).

The advantage of temporary storage is that the computer can access the data in memory extremely quickly– much faster than it can data on the disk. Each location in memory is designated by a number (called an ***address***) that the computer uses to access the data in that location. The computer makes a file available to you by copying that file into memory (leaving the original file on the disk untouched). (The computer always copies into memory from the disk any data that it is going to be working on.) Think of memory as a temporary scratch pad that you use and then erase. Like the desk in our example above, memory gives you a place to quickly and easily work on files.

As you may already have guessed, the administrative assistant in our example above is AMOS itself. AMOS loads a copy of a file from the disk into your memory area, where you can edit it (in the case of text files) or execute it (in the case of program files). If you have changed the file, you might want to have AMOS write the new file back out to the disk. In any case, AMOS takes care of all of the details involved in handling the file. You don't need to know where the file is on the disk or what memory locations belong to you– AMOS handles it all for you.

5.2 EXAMPLE OF FILE HANDLING: EDITING A TEXT FILE

Let's look at an example of how AMOS helps you to handle files. Suppose you want to update an office report. This report exists as a text file on the disk (perhaps named REPORT.TXT).

The name of the file identifies it to AMOS. The characters that follow the dot at the end of the file name are called the file's ***extension.*** The extension identifies the file type, and also helps to identify the file if two or more file names are the same. For example, the .TXT extension tells AMOS that the file is a text file. (Any file containing ASCII binary data that can be converted into readable characters is called a ***text file.*** The characters that make up text include letters, numbers, spaces, tabs, carriage return symbols, and special control characters. We've already mentioned that the data you enter is stored internally by the computer as the ASCII values of the separate characters in their binary form. To present that data to you in a readable form, the computer translates it back into characters.)

It's important to keep the file extension in mind. You will find using the system much easier if you realize that the file REPORT.TXT is *not* the same file as REPORT.LST. While both are text files, REPORT.TXT might be a rough, working file, while REPORT.LST might be the polished, formatted, ready-to-print version of that file.

We want to use the screen-oriented text editor, VUE, to modify our report. (A screen-oriented text editor allows you to display the file you are editing on the CRT terminal screen. You then make changes in the file by using commands that change the text on the screen.)

We tell AMOS that we want to use VUE to edit the file REPORT.TXT by entering:

```
.VUE REPORT.TXT RET
```

AMOS first writes a copy of the VUE program into memory (that is, it "loads" VUE into memory). (VUE is a machine language program that exists on the disk as the file VUE.PRG.) Now VUE takes over. It looks for the file REPORT.TXT in the disk account you are logged into. If it finds the file, it copies it into your memory area (without touching the original on the disk). (If it doesn't find the REPORT.TXT, it asks you if you want to create a new file of that name.)

Using the VUE commands, you now make the changes you want. It is important to realize, however, that you are editing the copy of your file in memory– not the file itself. When you are finished, if you wish it to do so, VUE writes the modified file in memory back out to the disk, replacing the original file. (In the case of VUE, the original file now becomes a backup file– designated by a .BAK extension– and your newly edited file becomes the new source file. A backup file gives you a version of the file as it existed before you made your last set of changes.)

When you exit VUE, AMOS deletes it from memory. When you load something else into memory, AMOS writes it over the remains of your text file in memory. (But that's OK, because your edited file has already been written out to the disk.)

5.3 ORGANIZING FILES ON THE DISK

It is possible to have thousands of files on a single disk. This could present a difficult organization problem. AMOS uses the concept of "accounts" to organize files on the disk, and to establish you as a legal user of the system.

5.3.1 User Accounts

When you begin to do business with a bank, you establish an account with that bank. Your account identifies you to the bank and allows you to gain access to the funds you deposit there. In the same way, accounts on the AMOS system identify you to AMOS and give you a way to establish a business relationship with AMOS.

The System Operator is the person who sets up the accounts on a disk and assigns each user of the system one or more accounts. AMOS maintains the account structure on the actual disk. The System Operator must assign new accounts on each new disk on the system. (The System Operator is also known as the System Manager or System Administrator. The System Operator is the person who sets up user accounts, allocates memory, runs diagnostic tests, and performs other tasks necessary to the smooth functioning of the system.)

Each file on the disk "belongs" to one account. Each account maintains a list (called the ***account directory***) that tells AMOS what files belong to that account. Each account on the disk has a unique number associated with it– the ***project-programmer number*** (also known as the ***PPN*** or ***account number***)– that serves to identify the account. As you create new files, AMOS associates them with your account, and adds their names to your account directory. (We will talk more about the account structure on the disk in Chapter 16, "How AMOS Handles Devices.")

Before you can use the system, you must establish yourself as a user of an account by "logging into" the system. That is, you must tell AMOS which account you wish to work with by entering the LOG command and the project-programmer number associated with the particular account you want to access. For example:

```
.LOG DSK3:[110,4] RET
```

The command above tells AMOS to log you into account [110,4] which resides on the disk in device DSK3:. (For information on accounts, project-programmer numbers, and directories, see Chapter 6, "Identifying Yourself to AMOS," in the *AMOS User's Guide,* (DWM-00100-35).)

Once you have logged into an account, you can begin to use the system commands and access files. Usually, you will work within one account, working on files that belong to that account. Meanwhile, other users may be working on the files in their accounts. Only one person should work in the same account at the same time. Although you can always copy any file from any other account into the account you are logged into, only under special circumstances can you copy a file into someone else's account. The account structure on the disk helps both you and AMOS to keep straight which users may access files on that disk, and what files those users can access. The System Operator may log into a special account, called the ***System Operator's account*** (DSK0:[1,2]), which gives the System Operator special system privileges.

5.3.2 Passwords

Often the System Operator will assign a password to an account. This keeps unauthorized users from accessing that account. If a password is required before you can log into an account, AMOS asks you for it. For example:

```
.LOG HWK1:[34,1] RET
Password:
```

In this case, you must enter the correct password before AMOS logs you into the account. A password is only of value if it is secret; therefore, AMOS does not display your password as you enter it.

5.4 HOW AMOS ALLOCATES FILES ON THE DISK

We'll talk in much more detail about the structure of the disk in Chapter 16, "How AMOS Handles Devices." However, there is one concept that we ought to discuss here before you actually begin to use files.

Throughout the Alpha Micro documentation, you frequently see the terms ***sequential files*** and ***random files.*** (You may also see these mentioned as ***linked files*** and ***contiguous files.***) Because the kinds of actions you can perform on files depend on whether those files are sequential or random, you'll want to understand the difference between the two.

AMOS transfers files to and from the disk in units of 512 bytes. This 512-byte chunk is called a ***disk block.*** Even the smallest file consists of at least one disk block (though the block might be partially empty). The upper limit on the size of a file depends on the size of your storage devices. (That is, a file may not overlap onto two different disks, but must fit all on one unit.) Each disk block has a unique number by which AMOS can reference it.

AMOS has two different ways of writing files to the disk. The way that a file is allocated on the disk determines whether it is a sequential or a random file.

5.4.1 Sequential Files

Most files on the AMOS system are sequential files. We call these files ***sequential files*** because AMOS accesses the data in the file sequentially. That is, as AMOS reads each disk block of the file, that block tells it the disk address of the next disk block. AMOS proceeds through the file one block at a time. To find out where block #3 of the file is, AMOS looks at block #1, which points to block #2. Then AMOS looks at block #2, which points to block #3. The important thing to remember about a sequential file is that to acess one block of data in it, you have to access all preceding blocks.

When AMOS writes a sequential file to the disk, it looks for the first free disk block. It writes a copy of the first file block into that disk location. Next, it looks for another free disk block. This next disk block may or may not be anywhere near the first disk block used. This process goes on until the entire file is transferred to the disk. The disk blocks that make up the file may be scattered across the disk. How does AMOS keep track of the file? Each disk block in the file contains a portion of the file; it also contains the address of the next disk block used by the file.

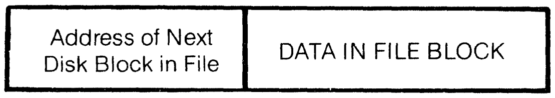


Figure 5-1
Sequential File Disk Block

Sequential files are also called ***linked files*** because the disk blocks are linked together by the information in each block that points to the address of the next disk block. (The last block in the file is designated as such by a link of zero.) For example:

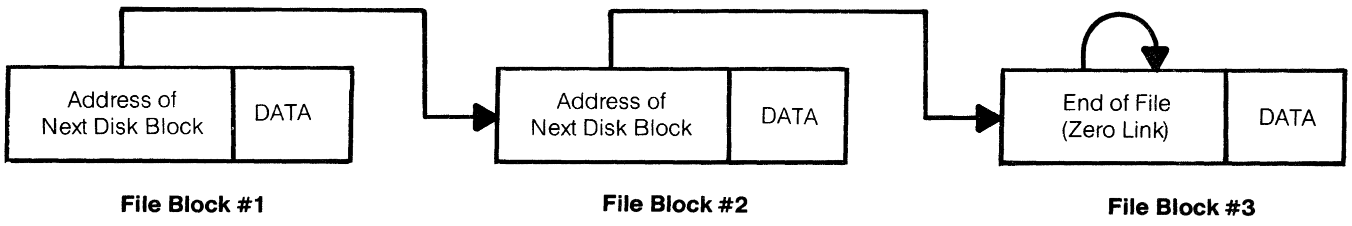


Figure 5-2
Disk Blocks in a Sequential File

The major advantage of a sequential file is that you can expand it. For example, suppose you are editing a text file that is four disk blocks long. AMOS is easily able to make the file larger by simply allocating a new disk block for the new material you want to add. It is for this reason that processes which expand files (such as text editing) can only be performed on a sequential file. If you try to use VUE on a random file, you see the error message:

```
    ?File type mismatch
```

Almost any time you create a file by using a command from AMOS command level (for example, using the MAKE or VUE commands), that file is a sequential file.

5.4.2 Random Files

Some files are called ***random files*** because AMOS can access the data in them "randomly." AMOS knows how long the files are, and also knows exactly where the files begin on the disk. AMOS can therefore access any block in a file by computing an offset value from the front of the file, and then reading the proper disk location. We say that the data access is random rather than sequential, because AMOS can access the disk blocks in any order, and does not have to step through the file to find the disk location of a specific block. AMOS can therefore find data in such a file quickly and efficiently.

When AMOS writes a random file to the disk, it looks for the first free group of *contiguous* disk blocks that is large enough to hold the entire file. That is, if your random file is 20 blocks long, AMOS looks for 20 disk blocks that physically adjoin on the disk. When it finds such a group of blocks, AMOS writes the file to the disk. If it cannot find a group of blocks large enough, you see a *Disk full* error message. This illustrates a major disadvantage of a random file. Even if you have 100 free disk blocks, you will not have room for a 20-block random file if 20 of those blocks are not in a contiguous group on the disk.

Because this kind of file is written into contiguous disk blocks, we also call it a ***contiguous file.***

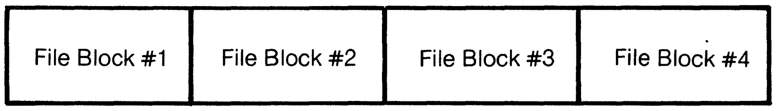


Figure 5-3
Disk Blocks in a Random File

Once a random file is allocated on the disk, it is not possible to expand it. Therefore, random files are used for applications where the file length remains constant (e.g., BASIC data files).

CHAPTER 6
PERMANENT DATA STORAGE

The two major devices for permanently storing data are the disk drive and the magnetic tape unit. Magnetic tape units are not as widely used as disk drives because they are relatively expensive and cannot be used as the primary permanent storage device. Therefore, we will center our discussions on disk drives. For more information on how data is structured on the disk, see Chapter 16, "How AMOS Handles Devices."

6.1 DISK DRIVES

All of the software that runs on the system exists as files on a disk. The special disk that contains the files that comprise AMOS and the various system commands is called the ***System Disk.*** Because the process of system startup involves reading certain files off the System Disk, we say that the system "boots off of" (that is, starts up from) the System Disk.

The System Disk is always known by the device specification "DSK0:". (For example, if the disk file CREATE.PRG appears in account [1,4] on the System Disk, its full file specification looks like this: DSK0:CREATE.PRG[1,4].)

The disk drive is the physical device that reads and writes data on the recording media (the disks). The disk drive that contains the System Disk is called the ***System Device.***

Disk drives come in two basic types: those that use "floppy" disks and those that handle hard disks.

A ***floppy disk*** is a very thin, flexible, circular piece of mylar plastic on which data can be recorded. A permanent cardboard envelope encloses the disk, which rotates within the envelope. The floppy disk drive reads the data on the disk through the cutouts in the cardboard envelope. (The cardboard envelope is an integral part of the floppy disk; do not remove it.) Although a floppy disk does not hold as much data as a hard disk, it is easily stored and very portable.

A hard disk drive contains at least one rigid circle of metal coated with iron oxide. (Each one of these units is called a ***platter*** or a ***surface.***) Platters are either "fixed" or "removable." A fixed platter cannot be removed from the disk drive; a removable platter can be removed and replaced with another removable platter. Depending on the disk drive, all of the platters may be fixed, or some may be fixed and some may be removable. Removable surfaces are enclosed in a protective plastic case; this disk together with its case is called a ***disk pack*** or a ***disk cartridge,*** and you may easily change one disk pack for another.

A typical example of a hard disk configuration is the Hawk hard disk drive which is available from Alpha Micro. When we talk about the Hawk drive, you may hear the phrase "one fixed and one removable." What this means is that one platter (containing five mega-bytes of data storage) is fixed in the bottom portion of the disk drive. Another platter (also containing five mega-bytes of data) is contained in a removable disk pack that loads into the top of the disk drive. Other hard disk drives may contain several fixed platters and one disk pack.

NOTE: Let's digress for a moment to talk about an issue that often confuses beginning AMOS users: that is, the fact that the cartridge is sometimes known by different disk specifications. (For example, on a Hawk disk, the cartridge is sometimes DSK0:, and other times it is DSK1:.)

Because the system must boot off DSK0:, the System Disk must always be known as DSK0:. On hard disk systems, the System Disk usually resides on the first fixed platter of the System Device. During normal operation, then, the system boots off the fixed platter System Disk; the first fixed platter is thus DSK0: and the cartridge is identified by some other specification. However, when your system is new, there is no System Disk on the fixed platter; in such cases, your system must be able to boot off of the cartridge so that you can transfer the system software from the cartridge to the fixed platter. At system startup, AMOS therefore checks to see if the cartridge is a System Disk; if it is, the system boots off the cartridge System Disk instead of the fixed platter (which may or may not contain a System Disk) and the cartridge is then known as DSK0: (since it is the disk the system started up from) and the fixed platter is known by some other device specification. (You can tell whether the system booted off the cartridge or the fixed platter by using the SYSTEM command; see the SYSTEM reference sheet in the *AMOS System Commands Reference Manual,* (DWM-00100-49).)

Hard disk drives are much faster at accessing data than are floppy drives, and can generally store a great deal more data. However, hard disks are much more expensive than floppy drives, and require that you handle them more gently.

Alpha Micro is constantly expanding the number of different types of disk drives you can use on your system. At the present time, your choice of disk drives ranges from disk drives using floppy disks that store about 250 thousand bytes up to hard disk drives that store 300 million bytes!

6.1.1 Disk Structure

Although we won't be going into much detail here, you might be interested in how disks are structured to hold data. Let's look at a single hard disk platter or a single floppy disk. The disk surface is organized into concentric rings called ***tracks.*** The disk is further divided into pie-shaped wedges. The area on a track between the beginning and end of a "wedge" is called a ***sector.*** Imagine a pie decorated with a "bulls-eye" target, which is then cut into wedges; the area within a wedge on a specific circle is a sector. The amount of data that is written into each sector varies, depending on the particular kind of disk. Except in very special circumstances, AMOS always writes data out in chunks of 512 bytes; usually the sector size on a hard disk is also 512 bytes.

Only under unusual circumstances will you ever have to worry about disk structure or sector size. AMOS takes care of communicating with the disk for you.

6.2 MAGNETIC TAPE TRANSPORTS

The unit that reads and writes magnetic tapes is called a ***tape transport.*** This device works much like the home open-reel tape recorder you are familiar with. You can transfer disk files to tape or vice versa. Unlike disk drives, magnetic tape transports are never used as primary permanent storage devices. (You'll remember that AMOS itself is a set of files that exist on a special disk called the System Disk. Therefore, the primary permanent storage device on the system– the System Device– must always be a disk drive.) Magnetic tape may be used to back up data or (the most likely use) as a means of transferring files between different computer systems, but not as the System Device.

The main disadvantage of the magnetic tape unit (and another reason why it cannot be used as the System Device) is that it is not a random-access device. That is, you cannot access blocks of data on the tape except in the order in which they were written. Also, you can't rewrite data in the middle of the tape without losing all information past it on the tape. Nevertheless, it is a valuable tool for transferring data between computer systems.

CHAPTER 7
PROGRAMS

In the previous chapters of this section we mentioned that the special instructions which tell the computer exactly what to do are called the *program.* An audience is guided through a complex opera by a small, printed program that describes the chain of events which give meaning and continuity to the songs they hear. Similarly, a computer is guided through an event of data handling by the specially written program you enter or invoke from storage. The program describes to the computer the chain of steps it must perform to do a task. The brief summary of the action as written in the opera program helps the audience to follow and understand the opera's storyline. As a result the audience responds to the dramatic acting and singing with emotion at the right moments. The audience then signifies its appreciation to the creators of the opera by its applause as the event is completed. Likewise, the computer steps through your program, responding to the data presented it according to the steps you have put in the program, and at the end the computer returns to you a completed task.

A program is a set of steps that tell the computer exactly how to handle a complete task. Most programs include alternate steps to take care of variations. The result of processing a program is the automatic solution of a problem or the completion of a task based on the data you provide.

The concept of a program is not unfamiliar to you. You program all the time. Whenever you plan a set of steps, test conditions, and think of alternate steps, you are programming. For example:

Program to Enter the Swimming Pool

1. Change your clothes to your swimsuit.
2. Walk to the pool carefully so as not to slip.
3. Does the pool have clean water? If not, stop the program. If so, go on.
4. Test the water with a toe.
5. Is the water too cold? If so, wait awhile and then test it again. If not, go on.
6. Take a deep breath and hold it.
7. Leap into the pool.

7.1 THE DIFFERENCE BETWEEN DATA AND THE PROGRAM

Essential to understanding the concept of a program is the awareness that the program is different than the data that is manipulated by the program. They are both represented the same way within the computer– in machine language. They are both originally presented to the computer in the same ways (for instance, via the CRT terminal and ASCII code). However, the program is the method whereby you cause the computer to step through a process, going in the direction which is useful to you. The data is the raw material you present to the computer, which the computer draws upon as it proceeds to do its task. Data can be changed while it is being processed, so in order to use the same program on different or changing data, the program must be able to handle variables. A ***variable*** is a symbol that can represent data. However, the same variable can stand for a variety of data. For instance, in the above example of a program, you might let *trunks* replace the variable we have named *swimsuit.* Or you might substitute in *bikini.* (Or *frogman suit, lead overcoat* or *birthday suit.*) The variable allows you (and the program) to change the data the program works on. If a program does not contain variables, it can only work on one set of unchanging data.

At some point, the program is broken down into machine language for direct processing. You can program the computer using a number of different languages, depending on what your computer can handle. These languages are designed to allow you to put a program into the computer in a way that is convenient and logical to you. Most computers also allow you to enter a program in the actual machine language, using 1s and 0s or binary shorthand methods. The advantage to machine language programming is a vast increase in processing speed, since the computer does not have to interpret some other language first. The major disadvantage to machine language programming is the difficulty and tedium of entering each and every minor detail of instruction. The higher-level languages take care of most of these details for you. Some of the higher-level languages are discussed in Chapter 10 of this book.

7.2 STEPPING THROUGH A PROGRAM

The fact that most CPUs are so rapid makes it appear sometimes as if a great deal of data handling is occurring all at once. Yet, in a given cycle only one minute step of data processing is done. Many steps are required to do any significant computing, and in order for the computer to do what you want, you have to give it the steps you want it to go through. A machine language program details every single step. Programs in a higher-level language are translated into machine language instructions for you. Nevertheless, any program you use has to be set up to carry the computer step by step through a task.

When a program is created, the separate steps are determined by the programmer in as much detail as the language being used requires. A model of the program steps is called a ***flowchart,*** or ***flow diagram.*** Even when a programmer can create a whole program mentally, he or she must nevertheless decide how the logical flow of the program should go. A programmer must consider how data is entered, how it may vary, how the variables can all be contained in the program, what the program must do with or to the data, and where the result of the data handling must go when the program is complete.

A programmer is limited by the machine as well. Recall that when the CPU is going through a program, the program must be in very fast temporary storage, which is completely accessible to the CPU. This temporary storage is in every case limited somehow in size. This is a limit in the hardware, and one that the programmer must work around if possible by eliminating unnecessary steps and making the necessary steps efficient. Occasionally, time is a limit. If the programmer makes a step depend on a previous step, but has not allowed for the machine to take the required cycles it must have (each cycle taking a little time), the program doesn't work properly.

Flowcharting a program reveals how the program should work. Whether mentally conceiving or actually writing down a model of your program, you as a programmer must arrange your program in logical steps the computer can follow and perform. Below we give a lengthy example of a kind of flow diagram, more expansive than the swimming pool example above, and considerably more detailed. We bring in data and consider variables to show how a task is logically stepped through. It is not a program. It is a list of steps designed to illustrate how variables are anticipated and how the computer follows instructions precisely. For the sake of clarity, there is a standard set of symbols normally used when flowcharting. If you are curious or find it necessary to learn the standard symbols, please refer to a textbook on programming or flowcharting. There are literally hundreds of such books available today.

In our example, as we set about to list some logical steps from which a program could eventually be generated, we know that certain real objects exist and are available to us. We construct the flowchart to help us define how to handle all the variables which are possible regarding those steps, and to get us successfully through

13.3 CHARACTERISTICS OF PROGRAMS ON THE AMOS SYSTEM

Remember that all commands invoke command files or machine language programs. Let's talk a minute about the programs that make up the command routines. All command files and machine language programs originally exist on the disk as files. Most command routines are ***transient***; that is, they exist on the disk, are loaded into main memory only when needed, and then are automatically deleted from memory when execution is finished. Such command routines can be made non-transient by loading them into memory with the LOAD command. In that case, they remain in memory until explicitly deleted by a user.

All programs on the AMOS system are ***relocatable,*** That is, they will operate properly anywhere in memory, without being modified or reassembled. This is necessary because there is no way of knowing beforehand which memory locations a program will have to be loaded into, since all users on the system use a different area of memory. AMOS automatically takes care of making higher-level language programs and command files relocatable for you.

Some machine language programs are also re-entrant. A ***re-entrant program*** is one that can be used by more than one user at one time. For example, BASIC can be invoked by one user, interrupted by another user who also makes full use of the program, and then re-entered at the point of interruption by the first user. Both users get correct results. Re-entrant programs are also known as ***sharable*** programs.

So that a re-entrant program can be used by more than one person, it must be loaded into sharable memory (the area of memory used by the operating system and resident system programs). The System Operator can add programs to the Resident Program Area by modifying the system initialization command file. The obvious advantage to sharing programs is that each individual user does not have to load the program into his or her own area of memory, but can access the single copy of the program in sharable memory. The disadvantage to loading re-entrant programs into sharable memory is that this expands the size of this area of memory, and reduces the amount of memory available for individual users. If the System Operator loads a program into the Resident Program Area, that program *MUST* be re-entrant; the computer will exhibit strange and distressing behavior if several users are sharing, at the same time, a program that is not re-entrant. If you want to write re-entrant programs, consult the *AMOS Assembly Language Programmer's Reference Manual,* (DWM-00100-43), for hints on doing so.

CHAPTER 14
MEMORY CONTROL AND MANAGEMENT

When you see the term "memory" in this book, we are talking about the random-access memory that makes up the temporary data storage on your computer system. (Remember that we discussed temporary storage devices, permanent storage devices, and random-access memory in Chapter 2, "What is a Computer?".) Although previous chapters have mentioned the importance of memory as a component of your computer system, this chapter will go into some detail on how AMOS manages, controls, and allocates memory.

Before any program can be executed or data manipulated, the computer must transfer a copy of that program or data from a permanent data storage device (that is, the disk) into memory, where the CPU can work on it. (When we transfer a copy of a file from the disk into memory, we say that we have ***loaded*** that file into memory.)

Memory is the only form of storage that the CPU can work on directly. It differs from disk storage in that the computer system can access it *extremely* quickly (in billionths of a second), and because memory offers only temporary storage; when the power goes off, the contents of random-access memory disappear. Because of these unique attributes, the computer uses memory as a work area– a scratch pad, in other words.

Each location in memory is consecutively numbered; that number forms a unique address by which a job or the operating system can access a specific memory location. Memory addresses can run from 0 to 65535 on the Alpha Micro computer system. This is because the CPU handles 16-bit numbers; the maximum number you can represent in 16 bits is 65535. (**NOTE:** Although this would seem to limit us to a maximum of 64K memory on a system (locations 0-65535), the AMOS system uses a memory management technique that allows us to have multiple sets of 64K memory. See Section 14.3, "Memory Management," for a discussion of this technique.) Memory locations near location 0 are known as ***low memory;*** locations near the other end of memory are known as ***high memory.***

When the operating system loads a copy of a program into memory, it does so by consecutively writing one byte of data per memory location. The Memory Controller keeps track of which locations are available for use. It also keeps track of the areas of memory used by specific jobs, and allocates memory for different uses. Although these functions are important under any circumstances, they become even more significant on a timesharing system, where different users are operating in different areas of memory at the same time. If some entity were not managing memory resources, it would be impossible to keep the operating system and jobs from bumping into each other throughout memory, writing over each other's data and programs. Because the AMOS computer does not have memory mapping or memory protection built into the hardware, software must keep track of what areas are in use, and what areas are available.

14.1 MEMORY MAP

The pattern in which memory is distributed to the various jobs on the system and to the operating system itself, is often known as the ***memory map*** of that computer system. The memory map of your system changes every time you change memory allocations.

When the system starts up, AMOS writes itself into memory, beginning with location 0. The amount of memory taken up by AMOS depends on your particular system and the particular devices connected to that system.

The remaining memory is available for user jobs except for the top 256 bytes of memory, which are used as the I/O ports. (You can see, then, that our earlier assertion that memory addresses can run from 0 to 65535 is not strictly true. Because the top 256 bytes are the I/O ports, memory addresses really run from 0 to 65279.)

14.1.1 Memory Partitions

Each job has its own area of memory, called a ***memory partition*** or ***user partition.*** The memory partition allocated to a specific job may be anywhere in memory, depending on what memory was available when that partition was assigned. A typical memory map for a 64K system might look something like this:

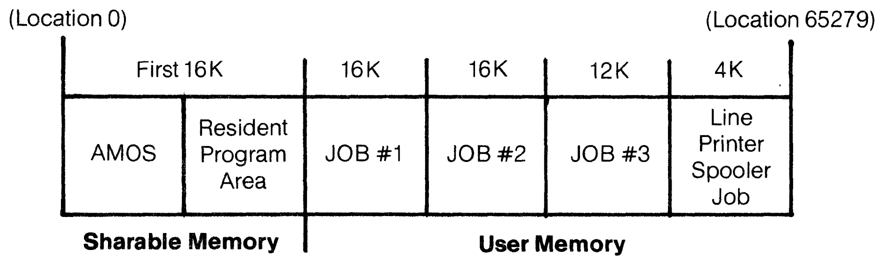


Figure 14-1
Typical Memory Map for a 64K system

The diagram above shows a system that uses the first 16K for the operating system and for resident programs. This 16K is called ***sharable*** or ***system memory,*** because all users on the system can access it. All users can access programs that are in the Resident Program Area without loading those programs into their own memory partitions. In addition, besides saving room in individual user partitions, putting programs in the Resident Program Area allows users to access those programs faster. This is because the programs do not have to be loaded into memory before they are used. Placing programs into the Resident Program Area is therefore a good idea if you have the room in sharable memory to do so, and if those programs are used frequently by most of the users on the system. (Placing a program into the Resident Program Area is done by the System Operator, who does so by modifying the system initialization command file.) **NOTE:** Any program loaded into the Resident Program Area *MUST* be re-entrant. (See our discussion of re-entrant programs in Section 13.4, "Characteristics of Programs on the AMOS System.")

The rest of the memory, called ***user memory,*** is nonsharable memory, and is devoted to user jobs. Our sample system has divided up the remaining memory into user partitions of 16K, 16K, 12K, and 4K.

The total amount of memory used in our sample system adds up to 64K (minus 256 bytes at the top of memory, which are used as the I/O ports and cannot be allocated to user partitions). Note that memory locations range from 0 to 65279. Each partition must contain contiguous memory locations. For example, all the memory locations that appear in JOB #1's memory partition must be consecutively numbered, with no gaps in those numbers.

Notice that the last 4K partition is set aside for a special job that is used by the line printer spooler program. The line printer spooler is a special program that allows your job to perform two tasks at once: printing a file while you are running a program in your partition. (The use of the spooler program is an example of ***multi-tasking;*** or, one user performing two or more tasks at the same time). What actually happens is that your job communicates with the line printer spooler job. Then the line printer spooler program allows you to place the

the task. We represent the real objects with symbols. In the flowchart, those symbols are words. In the program, those symbols (which are data) may be numbers, letters or other ASCII symbols. Were we to create a program using the flowchart as a model of the logical flow, and run that program, the computer would manipulate the symbols to simulate the manipulation of the actual objects. Even the final product of the manipulation would be simulated.

The rules of following a flowchart are that each step is an instruction, a decision or an action. As you consider the flow, you must imagine yourself as the computer, having the single purpose of exactly following and performing the steps. Some of the steps give you no choice but to follow them to another step. Other steps make you look in your memory and decide the current status of a certain variable, then take the appropriate path directed. (Sometimes the status of the variable is determined by the sensory inputs from the peripheral equipment in your system.) Some steps make you perform actions using the outputs of your peripheral equipment. If you are not forced, directly or following a decision, to a step out of numerical sequence, or when no direction is provided in the step at all, you must go to the next step in sequence automatically. The final rule to observe, when following a flowchart, is that if you come across an undefined variable, and do not meet the condition of the step, you must fall through the step to the next step in sequence.

In our example, the real objects we know to exist and be available to us are represented by the word-symbols "peanut butter," "jelly," "bread," "knife," and so forth. On our system, we have hands, ears and a mouth, can travel, make measurements and observe. The final result, after we have proceeded through the entire flowchart, should be a peanut butter sandwich:

1. START THE PROGRAM

2. GO TO A STORE

3. IS THE STORE A GROCERY STORE?

NO	YES
GO TO ANOTHER AND DIFFERENT STORE : GO TO STEP 3	GO TO STEP 4

4. GO IN THE STORE

5. GO TO A DEPARTMENT

6. LET THE TERM "GROCERY-NAME" TEMPORARILY MEAN "PEANUT BUTTER" : LET THE TERM "CONTAINER-NAME" TEMPORARILY MEAN "JAR" : LET THE TERM "VARIETY-NAME" TEMPORARILY MEAN "CRUNCHY" : GO TO STEP 9

7. LET THE TERM "GROCERY-NAME" TEMPORARILY MEAN "JELLY" : LET THE TERM "VARIETY-NAME" TEMPORARILY MEAN "GRAPE" : GO TO STEP 9

8. LET THE TERM "GROCERY-NAME" TEMPORARILY MEAN "BREAD" : LET THE TERM "CONTAINER-NAME" TEMPORARILY MEAN "LOAF" : LET THE TERM VARIETY-NAME" TEMPORARILY MEAN "WHITE"

9. DOES THE DEPARTMENT EQUAL THE GROCERY-NAME DEPARTMENT?

NO	YES
GO TO ANOTHER AND DIFFERENT DEPARTMENT : GO TO STEP 9	GO TO STEP 10

10. TOUCH A CONTAINER-NAME OF GROCERY-NAME

```
11.   IS IT VARIETY-NAME GROCERY-NAME?
                     NO                              |              YES
       RELEASE CONTAINER-NAME :                      |         GO TO STEP 12
       TOUCH ANOTHER AND DIFFERENT                   |
       CONTAINER-NAME OF GROCERY-NAME :              |
       GO TO STEP 11                                 |

12.   GRASP THE CONTAINER-NAME OF GROCERY-NAME

13.   IF 1 JAR IN GRASP GO TO STEP 7 :  IF 2 JARS IN GRASP GO TO STEP 8 : IF 2 JARS AND 1
      LOAF IN GRASP GO TO STEP 14

14.   LEAVE THE STORE WITH  CRUNCHY PEANUT BUTTER JAR AND GRAPE JELLY JAR AND
      WHITE BREAD LOAF IN GRASP

15.   RETURN TO HOME

16.   OPEN 1 JAR : WHEN 2 JARS OPEN GO TO STEP 18

17.   GO TO STEP 16

18.   OPEN A DRAWER

19.   IF DRAWER IS EMPTY GO TO STEP 22 : GRASP A UTENSIL

20.   IS IT A KNIFE?
                     NO                              |              YES
                GO TO STEP 21                        |         GO TO STEP 23

21.   DROP  A  UTENSIL  :  GO TO STEP 19

22.   OPEN ANOTHER AND DIFFERENT DRAWER : GO TO STEP 19

23.   OPEN LOAF : GRASP BREAD : STORE BREAD ON COUNTER : LET BREAD BE DIVIDED
      INTO HALF-BREAD AND OTHER-HALF-BREAD

24.   LOAD KNIFE WITH PEANUT BUTTER

25.   STORE PEANUT BUTTER ON HALF-BREAD

26.   LOAD KNIFE WITH JELLY

27.   STORE JELLY ON OTHER-HALF-BREAD

28.   LET SANDWICH BE HALF-BREAD + OTHER-HALF-BREAD

29.   DROP KNIFE : GRASP SANDWICH

30.   STORE SANDWICH IN MOUTH

31.   STOP THE PROGRAM
```

When you followed the steps, you mentally or symbolically constructed a final product and stored it. You handled all the procedures a step at a time, even though the steps often made you jump back and forth in the flowchart or go around in loops a number of times. The way you kept straight the manipulation of different kinds of symbols while going through the same procedure was by using variables, such as "GROCERY-NAME" and "UTENSIL." You exchanged variables, performed mathematical functions with them and, comparing them, acted on the result of the comparison.

You followed steps which described actions. Load, drop, and grasp in the example might be equivalent to the functions of Load (data), Erase, or Store, as performed by a computer in an actual program. Your external actions like "go," "run," and observations may be equivalent to a computer drawing on its peripheral devices to process, print or read data.

Above all, and the point of any flowchart from which a real program is to be developed, the steps were so organized that they could be manipulated by your mind in a logical way. You proceeded through the steps of the flowchart and automatically imagined or simulated the completion of your task. If a program were to be written based on the sequence of steps listed in the flowchart, and entered into the computer in the proper language, the computer would do the same thing.

7.3 ALPHA MICRO PROGRAMS

Please remember that AMOS, the Alpha Micro Operating System, is made up of a large number of individual programs, each designed efficiently and logically to handle a specific kind of task for you. It is important to reiterate here that as either a programmer or a user, you have many high-powered programs already available, and can always add command programs you write or which come from Alpha Micro.

7.4 THINGS TO COME

Now that you know a little bit about AMOS and about computer systems in general, it's time to talk about your system in particular.

The next few chapters of the manual are going to introduce you to some of the major programs that run on the AMOS system. We'll talk about the Alpha Micro language processor programs, text editing programs, and system utilities. For detailed information on these subjects, you will want to turn to the appropriate manuals. (See Appendix B, "Where Do I Go From Here?" for a short guide to the Alpha Micro software documentation library.)

PART II
PROGRAMS AVAILABLE ON THE AMOS SYSTEM

Now that you are familiar with some general computer concepts, you are probably interested in learning more about the AMOS system. The next three chapters introduce you to some of the major programs that run under the Alpha Micro Operating System.

We won't even try to introduce you to *all* of the programs that run on the AMOS system, but you will become acquainted with a few programs that represent the major types of system software available. (By "systems software," we mean those machine language programs supplied by Alpha Micro on the System Disk. There are over 120 system programs available. See the *AMOS System Commands Reference Manual,* (DWM-00100-49), for information on these programs.)

The programs we are going to talk about in Part II can be classified into three categories: utility programs, text processors, and language processors. In no case do we go into details on program operation. Rather, our purpose is to let you know that these *kinds* of programs exist, and to tell you where you can find more information about the programs.

CHAPTER 8
AMOS UTILITY PROGRAMS

This chapter introduces you to four major utility programs: HELP, SORT, DIR, and ISAM. We've chosen to discuss these very different kinds of programs because they are characteristic of the programs available to you. These programs do not have much in common with one another except for the fact that you can use them all at AMOS command level. Their applications range from simple "housekeeping" functions to sophisticated file processing. These "utility programs" provide general file or system services, and add to the flexibility and efficiency of your system. If you would like a complete list of all of the programs available for use at the AMOS command level, refer to Section 6.3, "Functional Summary of Commands," in the *AMOS System Commands Reference Manual,* (DWM-00100-49).

8.1 HELP

It is sometimes difficult for a new user on a complex, sophisticated system to become familiar with the multiplicity of commands and options available. The purpose of the HELP program is to aid the new user of the system by providing information on using the system and its commands. HELP displays text files that contain information on specific topics. To use HELP, enter the word "HELP" at AMOS command level. HELP then displays a list of the topics it can give you information about. For example:

```
.HELP RET
```

Now the screen clears and you see something like this:

```
Help is available for the following:

APPEND  BASIC   COPY    DATE     DIR    ERASE   LOG     ME
MOUNT   PASS    PRINT   RENAME   SET    TYPE    VUE
```

This list of topics tells you which text files the HELP program can show you. (The list of topics may vary depending on your system.) To see a particular HELP file, enter HELP followed by the topic you want information about; then type a RETURN.

Although the HELP program was primarily conceived as a way of giving you information about the system, you can also use the HELP program to provide your own information to other users. For example, if you use an accounts receivable program on your system, you might want to provide a HELP file that gives instructions for data entry.

The HELP program displays any text file in the proper account that has the extension .HLP. You can create your own HELP file using one of the system text editors. Depending on which account you place that HELP file into, you can make it accessible to: a) all users of the system; b) only the users that use the accounts in your own project; or c) only the users who log into a specific account.

For more information on HELP, see the HELP reference sheet in the *AMOS System Commands Reference Manual,* (DWM-00100-49).

8.2 DIR

You will often use the DIR program to find out what files are in your account. (That is, you will use DIR to look at your account "directory.")

DIR is capable of performing a great many other functions as well. Among other things, DIR can:

- Display a list of all of the files in any account or any group of accounts.
- Display a list of only specific files in one or more accounts. (As one example, DIR could list all files with the .TXT extension, all files whose names begin with the characters AR, and all files whose names end with the characters PRNT.)
- Search all accounts on a particular disk for a particular file or set of files, and tell you what accounts contain the files you are looking for.
- Search all mounted devices on the system for a particular file or set of files, and tell you which devices and accounts contain the files you are looking for.
- Display a list of the memory modules in memory.
- Send one or more directory displays to a printer or to a disk file.
- Format the directory display into a specified number of columns.

DIR usually gives you the following information on each account directory it displays: the account number, the names and extensions of the files, and the number of disk blocks taken up by each file. At the end of the directory display, DIR tells you how many files it listed and the total number of disk blocks used by those files.

You can instruct DIR to give you even more information about each file:

- The beginning disk address of the file.
- Whether the file is a random or a sequential file.
- A complete file specification for each file (i.e., disk and account specification as well as filename and extension).
- A hash total for each file. (A ***hash total*** or ***hash mark*** is a computed value based on characteristics of the file.) A file's hash total uniquely identifies that file. Two files will only have the same hash total if their contents are identical.

For more information on DIR, see Chapter 9, "The Wildcard File Commands," of the *AMOS User's Guide,* (DWM-00100-35), and the DIR reference sheet in the *AMOS System Commands Reference Manual,* (DWM-00100-49).

8.3 SORT

SORT alphabetically and numerically sorts data in a sequential file. There are many occasions when you would like to see the data in your file in a different order than the one in which you originally entered the data.

For example, suppose you have a data file that contains entries for customers. Each entry contains the following information: customer name, phone number, company name, address, and I.D. number. You may want a list of all of these customers with the company names in alphabetical order. Another day you may want the customer list ordered by customer name or I.D. number. SORT sorts the file for you and allows you to choose which item (called a ***key***) in the entry you want to sort by. You may sort in ascending or descending order.

Each entry (called a ***logical record***) must contain the same number of characters. (Remember to count spaces and punctuation, as well as letters and numbers!) Each record must be arranged so that each particular key is the same length as the same key in the rest of the records. (For example, if the customer name in the first record is fifteen characters long, the customer name key in all of the records must be fifteen characters long.) All keys must appear in the same order in each record. An example might help clarify matters a bit. Assume that each record has the following information:

NAME:COMPANY NAME:PHONE NUMBER:ADDRESS:I.D. NUMBER

Let's assume that we have decided how large each key must be:

NAME (15 characters):COMPANY NAME(15):PHONENUMBER(11):ADDRESS(20):ID#(8)

Each logical record is 15+15+11+20+8 characters, or 69. When we create the record, we must make sure that each key is equal to the size we have set aside for it. (For example, if the name we are entering is only 10 characters long, we must add 5 spaces– or other characters– to make it 15.)

Several records might look something like this:

```
Richards, J.R.]Racket Club]]]7145551212]123 Wicker,SanLeo,CA00002367
Greely, Horace]General Photon]5052315678]45 1st,Cedar City,UT00003456
Smyth, L.D.]]]]Acme SheepDip]]6721875645]12 Main,Ely NE]]]]]]00001200
```

Each logical record ends with a carriage return/line-feed character pair. When you use SORT, you tell it how long each logical record is (excluding the carriage return and line-feed characters at the end of each record). Then you tell it where each key begins in the record, and how long the key is.

Besides being able to sort by any one key in the logical record, SORT is also able to perform a ***hierarchial sort*** (a sort that also performs sub-sorts). One example of a hierarchial sort is the telephone book, which contains thousands of entries sorted by last name. Each group of entries with the same last name is then ordered by first name.

Let's look at a situation where you might want to perform a hierarchial sort. Suppose that you have a data file that contains information on forty of your salespeople. Each record contains: salesperson's name; region salesperson handles; number of sales to date; and, amount of commission earned to date.

You can, of course, sort your data file by any one of these items. But, you might want to arrange your data in a more sophisticated way. For example, you might want a list of your salespeople grouped by the regions they handle. Then, for each region, you might want to have the salespeople grouped by number of sales made this year. You might want each group of salespeople who made the same number of sales ordered by amounts of commission earned this year.

To perform this kind of sort, you specify three keys to SORT: region, number of sales, and amounts of commission. SORT performs only one sort, but your final listing might look something like this:

Smith, Edna	North East	023 sales	$2400
Arriza, J.R.	North East	023 sales	$1245
Calliano, John	North East	015 sales	$0932
Triatte, Susan	North East	007 sales	$0120
Williams, Loren	North East	007 sales	$0085
Wong, Henry	North West	051 sales	$3400
O'Brien, Jean	North West	051 sales	$1300
Ellison, Frank	South East	102 sales	$5076
Soto, Robin	South East	102 sales	$4600
Perlman, Andy	South East	034 sales	$1063

You may perform each sub-sort in ascending or descending order. For information on using SORT, refer to Chapter 10, "More File Commands," in the *AMOS User's Guide,*(DWM-00100-35), and the SORT reference sheet in the *AMOS System Commands Reference Manual,* (DWM-00100-49).

NOTE: SORT sorts data based on the ASCII values of the characters in the keys. That means that upper case letters come before lower case letters, and numbers come before letters.

8.4 THE ISAM SYSTEM

Many business applications programs create and maintain huge data files. Of major concern to the programmer is the task of accessing specific data in these files quickly and efficiently.

The ISAM system gives you a way to organize and use large data files. It also provides a standardized way to quickly retrieve and replace data in such files.

This section simply introduces you to the ISAM programming system. For information on using the ISAM system and the programs you can use to create and maintain ISAM files (ISMBLD, ISMCOM, ISMDMP, and ISMFIX), refer to the *ISAM System User's Guide,* (DWM-00100-06), and the *AlphaBASIC User's Manual,* (DWM-00100-01).

8.4.1 What is ISAM?

ISAM is simply a method for organizing and maintaining large data bases. The name stands for *I*ndexed *S*equential *A*ccess *M*ethod, and refers to the manner in which data is organized for later retrieval.

The information in an ISAM data file is accessed by searching a separate index file that contains a group of keys. The index file also contains pointers to the logical records in the data file that those keys are associated with. (Remember that we defined "keys" and "logical records" in the discussion above on SORT.)

To clarify this concept, let's look at an example where you automatically organize data much as ISAM would:

> When you need to find the meaning of a word in the dictionary, you don't start with the first page of the dictionary and scan every entry until you find the word you want. That would be a formidable task. Instead, you look at the tops of the pages to see which words are the first and last on that page. These words give you an "index" into the page. If the word you want falls somewhere between those two words, you decide which column of the page the word may fall into, and search that column. In other words, you search the index words instead of the individual entries on the pages until you reach the proper page. This procedure greatly increases the speed and efficiency with which you search the dictionary.

That was a very simple example. If we were to use the ISAM system to really organize a data base, the procedure would go something like this:

> Suppose you have a data file that contains a mailing list for a thousand customers. Each entry contains: customer name, address, state, zip code, and I.D. number.
>
> First, we build a file that contains one logical record for each entry in your mailing list. Each logical record has a number (called the ***relative record number*** or the ***relative key***) that marks its relative position in the file. (For example, the sixteenth record in the file is given the relative record number of 16.) (This file is called the ***ISAM data file.***)
>
> Next, we construct the ISAM index file. The index file contains the keys on which we want to base our search. For example, if you are going to be searching your mailing list data base by customer name, the index file would contain all of the customer names that appear in your data file.
>
> The index file is actually organized into three levels of indices. For example, suppose we are looking for customer "PERKOWSKI, JOHN." The first level index might contain entries such as "PACKER, G.B." - "PANDORA, MIKE" or "PEACH, GERTRUDE" - "PERLMAN, FRED." ISAM would know that our key falls into the second group. Each of these entries contains a pointer to another index level. The second index level sub-divides the keys even further. The third level gives us the actual relative record number of the data file logical record in which the key "PERKOWSKI, JOHN" occurs.
>
> In other words, ISAM swiftly searches all three levels of the index file to find the number of the data file record that we want to see. At no time does ISAM actually search the data file for the information we want; all data retrieval is done via the index file.
>
> ISAM also allows you to construct more than one index file for a given data file. This allows you to choose which key you are going to search by. (For example, one index file might contain customer names; another, zip codes.)

Although we talked in our example above as if *we* were building and maintaining the index file, remember that ISAM programs automatically build data files and index files for you in response to information that you supply. You can tell ISAM to search the index file/data file combination by giving commands from within your own BASIC programs or assembly language programs. You can then read or write data in the data file using the relative record number returned by ISAM.

CHAPTER 9
THE AMOS TEXT PROCESSORS

At the advent of the electronics age, it was predicted confidently that the written word would soon be made obsolete as television, audio and visual recorders, and those mysterious machines called computers became widespread. That turned out to be a pessimistic opinion. Rather than obsoleting the art of written communication, those and other electronic communications media began to meld with the written word to upgrade all of communications world-wide. The written word is more popular than ever before.

When writing, it is useful to be able to record thoughts rapidly, conveniently and accurately, before they escape. The computer has enhanced these abilities more than even the electric typewriter before it. Using even the most sophisticated electric typewriter, where text can be placed on paper via the fingertips and typewriter keyboard as fast as the writer can type, paper still has to be rolled into the machine, mistakes have to be erased laboriously or painted out messily, and a new or improved thought requires a newly retyped document. A finished, formatted document involves counting, spacing, tabulating and other mental calculations to make the result pleasing to the eye. There are always improvements, electrically and mechanically, but limitations still remain using the electric typewriter or any predecessor of it.

Using the computer, however, you may pass beyond those limitations. Though you still enter your thoughts through a keyboard, you can get the computer, programmed accordingly, to work with you to greatly decrease the work you must do and increase the speed in which you do it. The programs which are run on the computer to assist you in creating, modifying and formatting text are, as a group, called ***text processors.***

There are two broad categories of text processors. One category is known as the ***text editor.*** Text editors are used to generate text, such as that which you are reading or any other kind of document. The other category is known as the ***text formatter.*** You use text formatters to arrange your text into a format you desire, which could be as complex as the format of technical documentation, complete with chapter heading, spacing, topic hierarchy and right and left margins.

Naturally, there are many sub-categories of text editors. However, the two main sub-categories of text editors are the screen-oriented and the character-oriented types. The features and differences between these two kinds of text editors will be discussed briefly in this chapter.

Text formatters are also multi-categoried. Some work on general texts, some work specifically to create unique kinds of text, and so on. These also will be discussed and compared at more length in this chapter.

Please keep in mind that text editors and text formatters are separate entities. And whereas certain kinds of text processors have, for user convenience, some of the features of editors incorporated into formatters, and vice versa, they are best considered separately. In this chapter we will define more specifically what a text editor and a text formatter are, and introduce you to the Alpha Micro text editors VUE and EDIT. Also, you will meet the Alpha Micro text formatters called TXTFMT and PDLFMT.

9.1 THE TEXT EDITOR

A text editor is a computer program which assists you in the creation, alteration and eventual presentation of a document of written text. Remember that, in relation to computers, ***text*** is data that is entered as characters into the system and stored as equivalent ASCII binary numbers. Later these binary numbers are translated back into characters for display. Text characters include letters, numbers, spaces, tabs, carriage returns, and special control symbols. The characters you type in, except for editing commands to the text editor itself, are treated literally by the text editor program. That is, they are merely repeated in their relative positions within the file and have no meaning to the text editor.

Using a text editor to write a document of any sort, a file is created and named. You then type text into that file (which resides in temporary storage) a character at a time, using a keyboard. You may edit what you create immediately or at a future time, just as you would any other existent file. When you are ready to save what you have done, you exit the text editor and your file is saved in permanent storage upon your command.

In the modifying process, a copy of the existing file is placed again into temporary storage. Then you may insert, delete, move or change characters or words, entire lines or whole blocks of text. In order to help you locate the text you wish to modify, you may search for a certain character or string of characters throughout the text. Some text editors allow you to change each appearance of a certain string of characters to some other string, or delete them altogether, with a single command. When you have modified the text to your satisfaction, you exit the text editor with a command and the program puts the updated text back into permanent storage.

On the AMOS system, at this point the original (or most recent previous version) of the file is saved as a backup file in your account. The filename of the version prior to updating remains the same but the extension changes to .BAK. This gives you a final chance to change your mind or recover from an error if there are problems with your most recent editing session.

9.1.1 Character-oriented and Screen-oriented Text Editors

The difference between a character-oriented and a screen-oriented text editor is largely in the way the text is presented to you for editing.

Before the CRT terminal was invented, character-oriented text editors were designed that could be used with hard copy terminals. Hard copy terminals are machines with typewriter keyboards for input to the computer and printers (usually built in) for output from the computer. Hard copy terminals cannot permit the fast display allowed by CRT terminals, since they only provide output in permanent print, otherwise known as ***hard copy.*** A character-oriented text editor uses a special pointer which moves through the text from ASCII character to character under your control. Once the pointer is placed where you want it, the text editor (by your command) alters or displays a certain group of text before, after or otherwise oriented around the character you have pointed to.

The power of character-oriented text editors, aside from their ability to create or alter text without a video terminal, is the high-speed capability of handling complicated, tedious or repetitious operations upon the text. In executing a series of steps again and again as you have delineated in a single command, first the pointer of a character-oriented text editor locates a particular character or group of characters (meaning any combination of characters ordered in words, lines or some other pattern). Then the editor program alters the character or group as you have instructed and goes to the next character selected to repeat the operation.

With the increasing popularity of the CRT terminal, the screen-oriented text editors also became popular for their convenience and power. A screen-oriented text editor visually produces a screenful of text which you can observe. Also, a moving indicator of light known as a ***cursor,*** which you can control in four directions with keys on the keyboard, appears on the screen. By moving the cursor up and down and through the file, you can get positioned to alter the text. You can add, delete or move characters, lines or whole blocks of text. Whereas the screen-oriented text editors do not permit the automatic accomplishment of repetitive tasks with quite the facility of character-oriented text editors, they have become the preferred text editors overall because it is much easier to keep track of one's whereabouts while moving through the text. You can see more of the text without specifically asking for it, and can make alterations more quickly.

Incidentally, it is important to note that character-oriented text editors will work on CRT terminals to display text. However, screen-oriented text editors cannot work on hard copy terminals.

9.2 ALPHA MICRO TEXT EDITORS

As specific examples of both screen- and character-oriented text editors, the Alpha Micro text editors VUE and EDIT are particularly powerful and multi-featured, representing the general attributes of the two types of text editors. We will introduce VUE and EDIT to you separately in this section, touching on some of their primary features. Remember that you can use either VUE or EDIT to create any kind of text file. Such a file might be a book chapter, a business report, a BASIC program, a LISP program, or any other collection of ASCII characters.

9.2.1 VUE

VUE is a high speed text editor designed for AMOS users. It is screen-oriented; therefore it only works with CRT terminals. And the text it can work on must be in a sequential file, so that the file can expand as text is added. Although it has over 70 commands, it is easy to learn and use. VUE can be used to build programs and write documentation. For instance, the text included in this book was originally written and edited using VUE, as were the various examples of programs throughout the book.

When you tell VUE to edit a non-existent text file with the intention of creating one by that name, VUE immediately tells you that the file does not exist, and asks if you would like to create it. If you type Y (meaning yes) and a RETURN, your CRT screen fills up with asterisks (*) and the cursor waits at the top line. Then you can begin entering text of any sort. This is how you create a new file.

Since the text itself is meaningless to VUE, you can enter almost any combination of characters and spaces. As you create text, you are working in temporary storage. When you exit VUE, your text is written out to permanent (disk) storage by VUE and assigned the filename you called it when you were creating it.

When you again use VUE to read, continue, update or otherwise alter the file, VUE copies it from the disk into temporary storage. (If a file is too large to fit entirely in your available temporary storage, VUE writes in as much as will fit). Then VUE displays the beginning several lines of your file (i.e., a screenful) on your CRT and awaits your commands. This is called the Screen-editing mode. You can see the entire file a screenful at a time by moving the cursor anywhere within the text. Therefore, you can edit what you see on your screen by moving the cursor to the characters, words, whole lines or blocks you want to add or delete, and then entering the appropriate commands.

Sometimes you may wish to move coherent blocks of text from one area to another in the file. Or, you might want to search out all the occurrences of a certain string of characters. Or just automatically change all occurrences of a string of characters to something else. These and many more abilities are available in VUE by first striking the ESCAPE key (labeled ESC or ALT MODE on your keyboard). Your screen clears and then displays the title of the file you are editing and several other items of information regarding the progress of VUE. The cursor rests beside the VUE prompt character, >, as VUE waits for a command from you. This is known as the Command mode of VUE.

In Command mode you can issue many different VUE commands. You can move or copy blocks of text, search out character strings, or replace all occurrences of a certain string throughout a text. You also have commands to review and replace a single string at a time, swap strings, copy a string or block to a different place without removing the original, mark a location and return to it later, and other commands which adjust the text within the file.

You have the ability, operating from the Command mode, to access other text files in your account without leaving VUE. For instance, with one command, you can temporarily display the names of the files in your account. With another command, you can bring in a separate file from your account and add it to your text. Or you can take an excerpt from your text and create a new file with it.

Several other commands available in Command mode allow you to set parameters used by VUE in handling your text. That is, you can set or disable certain features, such as instructing VUE to automatically enter a carriage return when you reach the end of the screen line.

You can exit from VUE several ways, too. You can give the Finished command, telling VUE to write the copy of text in temporary storage out to permanent storage and create a backup file. You can merely quit (the Q command), and nothing you have done since you last updated the file in permanent storage will be saved. (In other words, the copy you have in temporary storage will not be written out to the disk, but will be erased from temporary storage.) Finally, you can finish-and-go with the G command, which is a user-defined command you set up so that VUE does a particular sequence of tasks upon exiting. A typical user-defined finish-and-go command might tell VUE to write the file out to the disk and then (for a BASIC program file), compile the BASIC program and run the compiled program with a small set of test data.

Each of the many Screen-editing and Command mode VUE commands makes moving around in any written text quick, versatile and convenient for you. For a complete list of the various commands of VUE, refer to the *AlphaVUE User's Manual,* (DWM-00100-15).

9.2.2 EDIT

EDIT is a sophisticated character-oriented text editor capable of processing complex strings of commands, enabling you to perform large, repetitious editing tasks quickly and efficiently. You can use EDIT either on a CRT or hard copy terminal.

You cannot begin to enter text into an empty file using EDIT until you first create the file in your account. This is done with a program supervised by AMOS called MAKE. After using MAKE, the file is created but empty. Then, using the insert mode of EDIT, you can enter text into the file.

You may use EDIT on any existing sequential text file. Invoking EDIT brings a copy of the file specified into temporary storage. EDIT gives you a prompt symbol, an asterisk, as it awaits your commands. A pointer (called DOT) indicates your current position in the copy of the text in temporary storage. Upon invoking EDIT, your current position is the first character in the file. Almost all of the more than 60 commands you can give to EDIT either move DOT to a character position in the file, or somehow alter the text oriented around that character. Various EDIT commands can search for characters or strings, add or delete characters, lines or sections of text, do replacements of words throughout the text automatically, and exhibit portions of the text for your scrutiny. EDIT commands are one or two characters long and some require arguments. That is, you follow them with strings of text (as in a search procedure) or by a numeric symbol (as in a character-advance command, specifying how many characters or lines you want DOT to advance).

You may enter the commands one at a time or in groups of several. You can group the commands by signaling to EDIT that you are finished with the insertion of one command, but that it must not execute the command until others are inserted. Finally, when your group of commands is complete, you enter a signal to EDIT to execute the entire group of commands.

The greatest power of EDIT today, with the inherent power of screen-oriented text editors making them more popular, is its ability to do tedious or repetitious tasks more easily. For instance, say you are designing a complex chart of numbers for printing with lines of division between all the rows and columns. Creating that chart can be a time consuming and tedious job using a screen editor. However, using EDIT, the chart can be done quickly and efficiently. With a single group of commands, EDIT can make a column wall character and a row character, enter the text (the number), move a few characters to the next column and repeat the process until the line is complete, then drop to the next line and repeat the whole series. And so on, until the chart is complete. There are many similar real-life examples of uses for EDIT, where by defining what you want via the many commands of EDIT, you can save a considerable amount of effort on your own part.

For a list of all the commands available to you using EDIT, refer to *EDIT: Character-oriented Text Editor,* (DWM-00100-39).

9.3 THE TEXT FORMATTER

A text formatter is a computer program which arranges characters and spaces to place text in a pattern which you command. Just as a person might convert a series of typewritten notes into a cohesive typewritten document with a centered header, a body subdivided into paragraphs, and a summary, a text formatter takes rough text and rearranges it into a finished format according to commands you embed in the rough text as you create it.

The finished format can be similar to a business, personal or legal letter, a report, a book with chapters, a pamphlet, a computer program, or any other document that has a form which is standard, traditional, convenient or efficient. Instead of painstakingly adjusting your rough text to meet the required form, you may insert a command to do the same thing. For instance, when typing, to center a title in the middle of a page you must first find the middle, then backspace once for each two characters in the title, and finally type the title itself. Using a text formatter with this ability, you simply insert the command for centering, then write the title. When the formatter analyzes the text, ignoring all but the text formatting commands, it automatically centers the title on the screen or printout for you.

Naturally, the text formatter and the specific purpose for which it is designed determines what you can or cannot do with it. A text formatter designed for general document formatting cannot be expected to format a highly specialized computer program. Conversely, a text formatter originated to help design programs will not do well in arranging a book into chapters and sections. Therefore, a variety of formatters are required. In the following two sections, you will meet Alpha Micro's main formatter, TXTFMT, which arranges many varieties of text; and you will briefly meet PDLFMT, the Program Design Language Formatting System, a tool that helps you to produce a program-design document.

9.3.1 TXTFMT

Most of the Alpha Micro documentation and publications, and many thousands of pieces of written material in existence, have been put into their final, readable form using the Alpha Micro program called TXTFMT. TXTFMT is a program that enables you to easily format documents in conjunction with one of the text editors VUE or EDIT. Rather than having to laboriously type your document in a finished form, you embed the short TXTFMT commands in your original text (called the ***source file***) to design the eventual appearance of your text. Filling lines, numbering pages, titling and other formatting considerations are taken care of according to your commands when you run your source file through the TXTFMT program. And because the formatting commands are embedded in the source file itself, you may simply run the source file through TXTFMT following any changes you later make to the text. Your text will again be formatted properly, without causing you extensive retyping. Also, if it becomes necessary to change the format itself, the formatting commands are as easily accessible as the text by using the text editor.

Usually, TXTFMT commands are entered as the text is being written. For instance, if you are writing a document and wish to set off a quote in double indentations, you can insert the TXTFMT command /DOUBLE INDENT as you come to the quote and /END DOUBLE INDENT when you finish it. The commands are made recognizable to TXTFMT by the slash (/) in column 1 (the first character position on the line). All other material in your file is regarded as text by TXTFMT. TXTFMT moves the text itself as needed. For instance, TXTFMT "fills" each line as it processes the file, meaning that as many words as possible are put on each line until the addition of another word would exceed the righthand margin. Thus, when you enter text, you need not worry about how many words you place on each line. You can place as few as one character or as many as 300 characters on a line. (A blank line is retained by TXTFMT, as in a break between paragraphs.) As another instance, if you insert the command /JUSTIFY at the beginning of your source file, TXTFMT adds spaces between words in a subtle pattern until all the lines of the text are precisely the same length. This makes a straight, or justified, righthand margin. The page you are reading is an example of righthand justification.

The margin widths themselves can be controlled by your command via TXTFMT, as can the linesize, spacing between lines, page numbering, pagesize and so on. These are among certain modes and formatting instructions automatically assumed by TXTFMT before you insert any TXTFMT commands. At the beginning of your source file you must insert TXTFMT commands to alter any or all of these automatic default commands if they are not suitable to you.

Some of the TXTFMT commands, including several of the default commands, require decimal number arguments. As an example, you have to supply a number to the command that specifies the number of lines that will appear on a page, if the assumed default number of lines is inappropriate for you. Other TXTFMT commands require a text argument. For instance, to tell TXTFMT that a certain line of text should appear at the top of every page as a page header, use the /TITLE command:

```
/TITLE Year-to-Date Summary
```

The phrase "Year-to-Date Summary" is the text argument you supply to the /TITLE command, which causes that title to automatically appear at the top of every page in the document.

TXTFMT allows you to organize your documents into numbered sections. For example, notice that you are reading Section 9.3.1, "TXTFMT." At no time did the authors have to worry about just what numbers ought to be assigned to this section– TXTFMT did the numbering for us based on the numbers of the preceding sections.

When the source file is complete with the textual matter and the text formatting commands that suit your purpose, you are ready to use the TXTFMT program itself. At the AMOS command level, you type "TXTFMT" and the name of the source file you have completed, along with the names of any other files you want to format into the same document. You must give the filenames in the order in which you want the files to appear in the finished, formatted document. From the separate source files, one longer, concatenated list file is created by TXTFMT which is assigned the name of the first source file you gave, and the extension of .LST.

As it proceeds through the source files, TXTFMT reports verbatim any incorrect text formatting commands embedded in the files. To correct these errors, you may again use VUE or EDIT on those source files, search out the errors using the features of the text editor, and correct them. Then exit the text editor normally and once again use TXTFMT on the files. Your final result will be a structured .LST file containing text arranged according to the commands you have given throughout the source file or files, with only a minimum amount of work on your part.

There are over 60 separate commands available to you with TXTFMT. For a complete, explanatory list of these commands, possible error messages and other instructions regarding TXTFMT, please refer to the *TXTFMT User's Manual,* (DWM-00100-07).

9.3.2 PDLFMT

The other Alpha Micro text formatter is PDLFMT, standing for the Program Design Language Formatting System. Since it is a very specifically applied tool, only the briefest introduction of its concepts is necessary here.

PDLFMT helps you to produce a program-design document. A ***program-design document*** is a type of outline that helps you in the arrangement of a computer program, as a step beyond merely flowcharting the program. A program-design document looks like a printout of a program with the major parameters of your requirements filled in. It enables you to add the more specific details of each section as you reason them out.

To use PDLFMT, you must first use one of the text editors EDIT or VUE to write your document of parameters in a very rigid structure. That structure contains four commands, each preceded by a slash in column 1. Those commands are /T (Design Title), /S (Section Name), /P (Procedure Name), and /R (Reference Tree). You begin by assigning the design title to the document. Then you name the first section and describe the section and your intentions for it. You detail the procedures of the section, giving each procedure a name and a description of design. There are several keywords you may insert within the procedures which PDLFMT uses to format the design document. Each following section of the design document is structured the same way, until the entire program design is outlined.

As the last command of the document, you may optionally make a single request for one or more reference trees. When you include a /R and a list of the procedure names from the various sections, PDLFMT generates a structured tree for each procedure, showing which other procedures it calls.

Once your source file is finished, exit from the text editor normally and at AMOS command level type "PDLFMT" and the source file name. (PDLFMT assumes a .PDL extension if you have not assigned another to the source file.) When formatting the source file, PDLFMT reports errors to you. The formatted version of the file is assigned the extension .LST, and has a table of contents, a program-design outline and (optionally) one or more reference trees.

For a complete list of instructions regarding the use of PDLFMT, refer to *Program Design Language Formatting System,* (DWM-00100-26).

CHAPTER 10
AMOS LANGUAGE PROCESSORS

This chapter will introduce you to the major language processor programs on the AMOS system. You'll meet BASIC, PASCAL, LISP, and the AMOS macro-assembler program, MACRO. The first few sections talk about some basic concepts behind the AMOS language processors. If you are already familiar with these ideas ("computer languages," "compilers," and "interpreters") you may want to skip on to later sections that talk specifically about the AMOS language processors (e.g., Section 10.4, "AlphaBASIC").

10.1 COMPUTER LANGUAGES

In earlier chapters, we mentioned that one of the major benefits of an operating system is that it allows you to communicate with your computer system in a way that is more comfortable for you than entering the machine language understood by that computer. (You'll remember that we defined machine language as the set of all of the instructions that you can give to the CPU which it can understand directly.) Because the operating system has very specific functions, the instructions that you can enter to it are also very specific and related only to system activities. For example, you cannot ask the operating system to perform a general function, such as to compute a mathematical value, but you can ask it to tell you what files are in your disk account. Therefore, your computer system gives you another tool that allows you to communicate in a more general way with the computer– computer languages.

A working definition of "language" might be that it is a large set of meaningful patterns that communicate ideas. A computer language allows you to write groups of instructions (a program) that the computer can carry out. A computer language can be machine language (which directly communicates instructions and data to the CPU) or it can be a higher-level language (such as BASIC) that is interpreted by a machine language program. (Those languages we call ***higher-level*** are more comfortable for us than machine language because they look much more like the languages we humans are used to, and because they let one instruction perform the same function as perhaps hundreds of machine language instructions.)

The type of computer language you use to write your programs depends on the applications performed by that program. For example, one computer language might have commands that make numeric computation very simple, while another language might be very good for writing programs that manipulate text. Still another language might generate programs that execute very quickly.

Just as English has a set of rules concerning the ways in which you can combine words into sentences and paragraphs, so computer languages also have strict rules that govern the ways you can form their elements into program lines and programs. The specific commands and ways in which you can use them vary widely among computer languages, but all computer languages have one thing in common– they help you to communicate with the computer. The computer languages you can use on the AMOS system are: BASIC, PASCAL, LISP, and assembly language.

10.2 WHAT IS A LANGUAGE PROCESSOR?

Anytime we talk to the computer in a language that is not machine language, we must use a translator of some sort that can transform our communication into a pattern that the computer can handle. Such a translator is a *language processor.*

Usually when we use the term "language processor," we refer to a program that can understand and act upon one of the familiar computer languages such as BASIC or PASCAL. (Of course, in a limited sense, AMOS itself is a language processor. We mentioned in Chapter 3, "Who is AMOS?", that the set of instructions that you can give AMOS make up the ***AMOS command language.***)

At this point we have already made the distinction between a computer language and the program that processes that language. Remember that when we talk about the BASIC language, we are talking about an abstract computer language named "BASIC." Programs written in this language are not directly executable by the operating system unless they are first translated and controlled by the BASIC language processor. The language processor program almost always bears the same name as the language it processes. For example, when you type:

```
.BASIC RET
```

you are telling AMOS to execute the language processor that understands the BASIC language. So, remember that when we talk about being "in BASIC," we mean that you are communicating with the program that processes programs written in the BASIC language. On the other hand, when we talk about a "BASIC statement," we are talking about an element in the BASIC language.

10.3 INTERPRETERS AND COMPILERS

We class language processors into one of two groups, according to how they process programs. These two groups are interpreters and compilers.

10.3.1 Language Interpreters

An *interpreter* reads each line of a program, and performs the commands in that line as it reaches them. For example, when a BASIC interpreter sees this line in your program:

```
PRINT "HELLO"
```

the interpreter recognizes the PRINT command. It then transfers control to the routine within itself that handles the PRINT command. Then control passes back to the main portion of the interpreter which scans and interprets the next program statement.

When it finishes reading, interpreting, and acting upon a program line, the interpreter goes on to the next line until it reaches the end of the program. If you ask the interpreter to execute your program again, it goes through this entire process of recognizing and acting upon program commands again, as if it had never seen your program before.

A major advantage of an interpreter is that such language processors are usually ***interactive.*** That is, you can enter program statements to the language processor, and the interpreter will tell you as you enter it whether a program statement is in legal form. You can even enter language statements that are not part of a program, and the interpreter will execute them directly as you type them in, instead of waiting to execute an entire program.

Of course, interpreters also have disadvantages. Because it must go through the same process of recognizing and executing program statements each time it processes your program, an interpreter is usually slower than a compiler when executing the same program. Another drawback to the use of interpreters is that you must keep the interpreter in memory along with your program; this can use up quite a bit of memory, since most interpreters are relatively large programs.

10.3.2 Language Compilers

The second type of language processor is called a ***compiler*** because it actually "compiles" or translates your program into another version which is closer to machine language. (A traditional, "strict" compiler translates a source program directly into machine language. Other compilers compile source programs into a pseudo-machine language. Regardless of its type, the compiler reduces your program to a form that is closer to the computer's own level.) This process of translation is called ***compiling*** a program or ***program compilation.***

After compiling a program, you have two versions of it: the original (or ***source***) program and the translated version (or ***compiled***) program.

One of the benefits of compiling a program is that most of the scanning and recognition process that an interpreter must do every time it reads your program is done only once by the compiler– at the time that it compiles your program.

The compiler thus allows you to separate the processes of program analysis and execution. You only need to compile a source program once. Then you can execute the compiled program at any time without going through the preliminary processing done earlier. (Of course, if you change your source program, you will need to re-compile it so that the compiled program reflects those changes.) Since the source program has been translated and reduced, the compiled program takes up less memory, and since the compiled program is closer to the computer's own machine language than the source program, it runs faster than an interpreted version. In addition, when you execute the compiled program, you do not have to have the compiler in memory along with your program. Depending on the compiler, your compiled program may either be executed directly by the computer, or you may need to have a small machine language program in memory called a ***run-time package,*** which completes the translation between your compiled program and the computer's machine language.

10.3.3 Theory Versus Fact

Now that we've given a very general discussion of interpreters and compilers, we should mention that although in theory the differences between them are very clear, in reality, most interpreters and compilers share some features of each other. That is, you will rarely find a "strict" compiler or a "strict" interpreter.

For example, most interpreters perform a process called ***tokenization,*** wherein the processor substitutes special tokens for BASIC statements. That is, when it reads in a line from a BASIC program, an interpreter might (for example) substitute a special one-byte symbol for the PRINT command. This allows the interpreter to go much faster later when it processes that line, because it doesn't need to scan the entire word "PRINT," but only needs to recognize the special PRINT token. A tokenized program also takes up less room in memory. Tokenization is a very limited type of compilation.

As another example, very few compilers actually translate programs directly into machine language. Most compilers (e.g., AlphaBASIC) compile programs into a pseudo-machine language form, which is then executed by a small machine language program called a ***run-time package.*** (In the case of AlphaBASIC, the run-time package is called ***RUN.***)

10.4 ALPHABASIC

Because AlphaBASIC is by far the most widely used language processor on the Alpha Micro system, we will talk about it in some depth. For detailed information on using AlphaBASIC, refer to the *AlphaBASIC User's Manual,* (DWM-00100-01).

BASIC is the most popular higher-level language on microcomputers today. Much of this popularity stems from the fact that it is relatively easy to learn. It was designed as a computer language for beginning programmers; in fact, its name is an acronym for "Beginner's All-purpose Symbolic Instruction Code."

Because it has been implemented by so many manufacturers on numerous computers, BASIC is no longer a standardized language. Many different versions of it exist, each slightly different from the other. Alpha Micro's BASIC, ***AlphaBASIC,*** is an extremely powerful version of BASIC that contains a number of unusual features that make it uniquely suitable for business and scientific applications programming.

If you've never seen a BASIC program, you might be interested in taking a look at a very small and simple program written in AlphaBASIC:

```
10    REM Calculate number of years it takes to double your money.
20    REM Get information and initialize new principal and number of years.
30    START:
40       INPUT "Enter principal: $", MAINPRINCIPAL
50       INPUT "Enter rate of interest (in %): %", RATE
60       YEARS = 1
70       PRINCIPAL = MAINPRINCIPAL
80    REM Now calculate the new principal
90    CALCULATE:
100      INTEREST = PRINCIPAL * (RATE/100)
110      PRINCIPAL = PRINCIPAL + INTEREST
120      IF PRINCIPAL >= (2 * MAINPRINCIPAL) THEN GOTO SOLUTION
130      YEARS = YEARS + 1
140      GOTO CALCULATE
150   REM We're finished.  Print the answer.
160   SOLUTION:
170      PRINT "At";RATE;"%, in";YEARS;"years you will have $";PRINCIPAL
```

Figure 10-1
Sample BASIC Program

(**NOTE:** The characters "REM" in the example above designate ***comments,*** known in BASIC as ***remarks.*** A program comment explains the purpose of the surrounding program statements. Comments are not processed by the language processor, but are solely for the programmer's benefit in helping him or her to figure out what the program is supposed to be doing.)

Below we discuss some of the features that make possible AlphaBASIC's power and versatility:

1. Although AlphaBASIC is a compiler, it incorporates the best features of an interactive interpreter as well. It accomplishes this by allowing you to use BASIC in two different modes: you can either use BASIC in compiler mode (as a traditional compiler) or you can use BASIC in interactive mode (simulating the operation of an interactive interpreter). To use BASIC in compiler mode, create your program using one of the system text editors. You may then compile that program from AMOS command level by using the COMPIL command. To run the compiled program, use the system RUN command. (COMPIL is the compiler portion of AlphaBASIC; RUN is the AlphaBASIC run-time package.) Note that at no time in this process

do you "ente: BASIC; that is, you are operating at AMOS command level when you compile and execute the program, and you cannot enter direct commands to BASIC except through your program.

To use AlphaBASIC in interactive mode, enter AlphaBASIC by using the system BASIC command. You are now communicating with AlphaBASIC (that is, you are "in BASIC"), and can type in your program, load in an existing program, or enter individual statements to be executed directly. (The BASIC command loads into memory both the compiler and run-time package portions of AlphaBASIC.) In interactive mode, AlphaBASIC simulates an interactive interpreter by compiling each line of your program as you enter it, giving you immediate feedback if an error occurs.

2. Unlike conventional BASICs (which usually allow only two-character variable names), AlphaBASIC allows variable names of any length in either upper or lower case. (For example, instead of being restricted to cryptic variable names such as "A1", you can use a more descriptive variable name such as "InvoiceNumber".)

3. To help you to structure your programs so that they are easy to read and maintain, AlphaBASIC allows the use of program labels to identify specific program modules. (See our sample program above– the program lines that end with colons are labels.)

4. BASICs break down data into two types: floating point numeric data and string data. (A ***string*** is data made up of ASCII characters. ***Floating point*** is a method the computer uses to internally represent numbers by storing the significant digits of the number along with a number (the exponent) that tells the computer where to insert a decimal point when it displays that number. Floating point representation thus allows the computer to store a number that is physically too large or too small to be expressed directly by the CPU.) AlphaBASIC also supports variables that contain binary, integer, and unformatted data.

5. Much of the flexibility and sophistication of AlphaBASIC that makes it so useful for business applications programming result from some unique data-handling features. These features include:

   - The ability to form a template in memory (via MAP statements), which allows BASIC to transfer data between your program and the disk with optimum speed and flexibility. MAP statements are most often used to define groups of variables which will be transferred in and out of disk files, but MAP statements are also useful for sophisticated array allocation and for linking with assembly language subroutines.

   - Advanced string subscripting. Besides the more common LEFT$, MID$, and RIGHT$ functions which allow you to extract portions of a string, AlphaBASIC also supports a powerful form of string subscripting which allows you to excerpt a substring by specifying the beginning and ending positions of that substring within the master string, or by specifying the beginning position and the length of the substring. (These forms of string subscripting accept either positive or negative values, which specify either left-relative positions or right-relative positions within the master string.)

     AlphaBASIC also has functions that allow you to compare strings, compute the length of a string, search a string to see if another string is embedded in it, and convert strings to numeric data (and vice versa).

   - Mode independence. Most BASICs require the use of special functions to convert string data to numeric form, or vice versa. Although AlphaBASIC also supports these functions, it automatically converts strings and numeric data that appear in expressions so that the results of such expressions are in the proper format. (**NOTE:** An ***expression*** (e.g., PRINCIPALX100) is a combination of ***constants*** (i.e., unchanging values) and/or variables.) For example, the phrase:

"34"+"5"

concatenates two strings to form the string: "345". However, the expression:

34+"5"

adds a numeric constant (34) and a string ("5"). AlphaBASIC converts the string "5" to the number 5, so that the result of the expression is 39.

- Formatting data displays. The PRINT USING statement (an extension of the standard data display command– PRINT), allows you to format data into specific forms that are suitable for business reports, screen displays, etc. For example, you can use the PRINT USING statements to format a list of numbers so that dollar signs precede each number, and commas are inserted every three digits to the left of the decimal point.

Other features in AlphaBASIC include: calls to external assembly language routines (such as sort routines, multi-user file locking routines, etc.); chaining to system commands, other BASIC programs, or command files; use of the system ISAM package for efficient data retrieval; allocation and use of sequential and random files; error and Control-C trapping and processing; terminal display functions; and a complete set of mathematical functions.

AlphaBASIC also supports several unique debugging features. While using BASIC in interactive mode, hitting the line-feed key tells BASIC to ***single-step*** the program in memory (that is, to execute one line at a time). After each line is executed, you can enter direct statements which allow you to examine and change the value of the program variables, open and close files, and so on. The BREAK command allows you set and clear breakpoints. (A ***breakpoint*** designates the spot in your program where you want BASIC to interrupt execution so that you can examine the values of program variables or perform other debugging functions.)

NOTE: We've already mentioned the term "function" as an element of a program. A ***function*** is a special command that accepts one or more strings or numbers (called the ***arguments*** of that function), and then returns an answer to you by performing some computation on those arguments. We say that we "pass" arguments to functions, and that they "return" an answer. An example of a function is the BASIC command, SQR. SQR accepts a number as an argument and returns the square root of that number. For example, the function SQR(39) returns the number 6.245. Other functions perform more sophisticated procedures. Some computer languages allow you to write your own functions, called ***user-defined functions.***

10.5 ALPHAPASCAL

PASCAL was developed in the early 1970s by Jensen and Wirth as a response to the need for a programming language that encouraged good programming "style." It was designed specifically to teach the concepts of computer programming and to make easier the task of efficiently implementing large programs. PASCAL is a general-purpose language that is applicable to a wide range of numeric and non-numeric problems. It contains a relatively small number of statements, but these can be combined in a variety of ways to construct powerful programs.

PASCAL is being used with increasing frequency in academic and industrial installations. The number of students that are taught PASCAL as their first programming language increases every year.

A noteworthy feature of PASCAL is that it encourages the use of ***step-wise refinement.*** That is, the programmer breaks down his or her task into a number of subtasks. Each subtask is coded into a subprogram (called a ***procedure***) within the main program. This encourages the programmer to construct a well-structured, modular program, and creates a program that is easier to read and maintain. PASCAL also encourages the creation of an easy-to-read program by requiring that the front of the program contain definitions of the data types used by the variables in that program. (We call this process ***declaring the variables.***)

An important feature of the Alpha Micro PASCAL is that it allows you to execute programs that are larger than your memory area. (That is, we say that it is a *virtual* PASCAL.)

The major advantage of PASCAL over other high-level programming languages is that it is a standardized language: that means that programs written in standard PASCAL can run on most other computers that support PASCAL. Other features PASCAL supports which make it a powerful programming language are:

- User-defined functions. Unlike AlphaBASIC, PASCAL allows you to create your own functions.
- Advanced data structures. PASCAL supports advanced data structures (such as sets and linked lists) that are usually quite difficult to construct in other programming languages.

If you are curious about what a PASCAL program looks like, take a look at the very simple example below:

```
(* Read 2 integers and print their sum *)
PROGRAM ADDTWO (INPUT,OUTPUT);
VAR VALUE1, VALUE2, SUM: INTEGER;
BEGIN (* ADDTWO *)
        WRITE ('Please enter the 1st number, a space, and 2nd number: ');
        READ (VALUE1,VALUE2);
        SUM := VALUE1 + VALUE2;
        WRITELN ('The sum is: ',SUM)
END.    (* ADDTWO *)
```

Figure 10-2
Sample PASCAL Program

PASCAL is usually implemented as a compiler that compiles your PASCAL programs either to actual machine language or to a pseudo-machine language level. For information on Alpha Micro's specific implementation of PASCAL, see the *AlphaPASCAL User's Manual,* (DWM-00100-08).

10.6 ALPHALISP

The LISP (*List Processing*) programming language is based on John McCarthy's 1960 work on non-numeric computation. It has become increasingly popular in the last few years as more and more programmers begin to realize the power of LISP in certain programming applications.

LISP is an awkward language for applications that require a great deal of numeric analysis (i.e., programs that do massive "numbercrunching"), but LISP can perform powerful and sophisticated symbol manipulation. Therefore, it is widely used in academic installations for research in natural language and artificial intelligence and it is also used in the business world for relational data base applications. LISP is often used for programs that associate items of information (for example, mailing list programs) and other programs that do large amounts of information retrieval.

LISP is unique among programming languages, with a look and "feel" all its own. Although some programmers find a LISP program hard to read and maintain, others feel that LISP's power in the areas at which it excels more than make up for the fact that it is not one of the easier programming languages to learn and that LISP programs look much different than those written in other languages.

The major structure of the LISP program is the list. All data and programs are made up of lists, and have the same syntax. This means that LISP makes little distinction between programs and data. For example, it is very easy to write a LISP program that reads another LISP program as data.

Alpha Micro's implementation of LISP (the AlphaLISP interpreter) is based on a version of LISP called "UCI LISP," which is in turn based on Stanford Artifical Intelligence Project's LISP 1.6. For information on using AlphaLISP, refer to the *AlphaLISP User's Manual,* (DWM-00100-05).

If you are interested in what a small LISP program looks like, take a look at the sample below:

```
(DEFPROP DBASE
 (LAMBDA NIL
   (PROG (NAME KEY INFO ACTION)
   LOOP (TERPRI)
        (PRINC @"Enter 'action (update or query)' 'name' 'key' ")
        (SETQ ACTION (READ))
        (SETQ NAME (READ))
        (SETQ KEY (READ))
        (COND
          ((EQ ACTION @update)
           (PRINC @" Enter data ")
           (SETQ INFO (READ))
           (PUTPROP NAME INFO KEY))
          ((SETQ INFO (GET NAME KEY))(PRINC INFO))
          ((PRINC @"No such information.")))
        (GO LOOP)))
 EXPR)
```

Figure 10-3
Sample LISP Program

The example above is part of a LISP program that associates information (for example, names and phone numbers) for a data base management system.

10.7 ASSEMBLERS

The earlier sections in this chapter concerned themselves with higher-level language processors such as BASIC and PASCAL. Another kind of language processor (called an ***assembler***) transforms your program (called an ***assembly language*** program) directly into machine language. MACRO is the Alpha Micro assembler.

How does assembly language differ from a higher-level language? Remember that a higher-level language statement may translate into hundreds of machine language elements. Each assembly language statement translates directly into only one machine language statement. (That's why assembly language is not called a higher-level language– it is, in effect, just another way of writing machine language.) (If you are interested in the assembly language used by the Alpha Micro system, refer to the *WD16 Microcomputer Reference Manual,* (DWM-00100-04).)

Assembly language was developed to help machine language programmers. You can imagine how tedious it would be to enter into the computer a machine language program that consists of thousands of numbers. This is just how things were done until someone finally had the brilliant idea of associating each machine language instruction with a symbolic name that was easier to remember than the actual numeric value of the machine language instruction, and then creating a program that translated those symbolic instructions into machine language.

Instead of having to remember the actual numbers that represent machine language instructions, machine language programmers (also known as ***assembly language programmers***) only have to remember a set of instruction names (called assembly language ***mnemonics***). The programmer constructs his or her program out of these statements, and then has the assembler program "assemble" the program into machine language. Besides mnemonics that translate directly into machine instructions, assemblers also recognize special symbols. (Some of these special symbols and statements are called program labels, CPU register symbols, data storage definitions, assembly control statements, and macro definitions.)

Even though assembly language programming was made much easier with the advent of the assembler program, assembly language is still much harder for most people to learn and use than a higher-level language. For one thing, assembly language programs must deal much more directly with the CPU than programs written in higher-level languages. That is, assembly language programs are "machine-oriented." Therefore, the assembly language programmer must usually have a greater understanding of the hardware with which he or she is communicating than does the programmer of a higher-level language. The assembly language programmer must also have a greater understanding of the inner workings of the operating system.

Assembly language programs usually contain many more program statements than equivalent programs in a higher-level language, and assembly language programs are usually harder to test, debug, and modify. Why, then, is so much programming done in assembly language? (For example, all of the AMOS commands, drivers, and language and text processors were written in assembly language.)

The assembled, machine language versions of assembly language programs are usually much smaller than higher-level programs that perform the same functions. Because machine language programs communicate directly with the computer, they generally execute much faster than higher-level language programs– sometimes hundreds of times faster. Another reason why much systems programming is still done in assembly is that most higher-level languages, while undeniably powerful, were not designed to deal with hardware- and operating system-related problems. It is sometimes just too cumbersome to "get down to the computer's level" in a higher-level language. So, for applications that interface with the hardware and the operating system, and that demand high speed, assembly language is still widely used.

If you would like to see what part of an assembly language program looks like, take a look at the example below:

```
; Sample program to print powers of two.
;
;
        COPY    SYS
;
        MOVI    1,R1            ; Set R1 to 1 (2UP0XUP).
LOOP:
        DCVT    0,2             ; Output number in decimal.
        CRLF                    ; Move cursor to next line.
        ASL     R1              ; Shift R1 left one bit.
        BNE     LOOP            ; Keep looping unless R1=65536 (i.e., 0).
        EXIT
```

Figure 10-4
Sample Assembly Language Program

The sample above computes and displays powers of two. (**NOTE:** The semicolons indicate assembly language program comments.)

10.7.1 The Alpha Micro Assembly Language Programming System

The Alpha Micro Assembly Language Programming System consists of: the assembler program ***MACRO***; the linkage editor ***LINK***, which connects different program segments into one program and resolves references segments make to each other; the symbol table generator ***SYMBOL***, which makes the user-defined symbols in your program accessible to the Alpha Micro debugging program, DDT; and, the Alpha Micro assembly language program debugger ***DDT***, which allows you to examine and change your program in memory. For information on using MACRO, LINK, SYMBOL, and DDT, refer to the *AMOS Assembly Language Programmer's Reference Manual*, (DWM-00100-43). Refer to the *AMOS Monitor Calls Manual*, (DWM-00100-42), for information on the calls your assembly language programs can make to routines embedded in the operating system.

When you write an assembly language program, the usual sequence of events is:

1. You create the source program using a system text editor.
2. Next you use MACRO to assemble your source program. If any errors occur during assembly, you probably will want to edit the program again to find the problems. When your program assembles without error, you can go on to the next step.
3. If the program contains references to other program segments, you now use the LINK program to link the segments together.
4. If you are going to be using the debugger program, DDT, you now use SYMBOL to create a symbol table file. This makes sure that DDT can access the user-defined symbols (such as program labels) that are in your source program.
5. Finally, you can now use DDT if you want to test and examine your program.

The Alpha Micro assembler, MACRO, is perhaps the most important part of this programming system. It does much more than just translate mnemonics directly into machine language. Some of the features that make MACRO a powerful macro-assembler are:

- MACRO is a ***multi-pass assembler***. That means that it scans your assembly language program more than once, unlike single-pass assemblers. This gives you flexibility in writing your programs, since MACRO is able to resolve program statements that refer to later portions of the program.
- MACRO is a full ***macro-assembler***. In other words, it is able to process and use macros. (A ***macro*** is a special instruction that you define which can assemble to a number of machine instructions. In effect, you tell the assembler, "When you see this macro, substitute in this series of instructions.") You may nest macro calls and pass arguments to macros.
- You may access external libraries of programs and routines. For example, the COPY SYS command tells MACRO that you want to use one or more of the 70 calls (which appear in macro form in the SYS library) that enable you to use many of the routines embedded within AMOS.
- MACRO allows you to generate ***relocatable code*** (i.e., programs that are independent of absolute memory addresses). The programs you assemble can thus be run by any user, regardless of the specific memory addresses he or she is using as a workspace.

- Assembly language programs assembled by MACRO are ***segmentable.*** That is, you can divide a large program into smaller, independent segments.
- MACRO understands ***conditional assembly directives.*** That means that you can control whether or not certain portions of your program are assembled, depending on a variable's value when you assemble the program. This allows you to generate several different machine language programs from one assembly language program. In other words, you can tailor a specific program for several different uses. You may nest conditional assembly directives.

10.8 THINGS TO COME

Now that you know something about the programs that run under AMOS, it is time to talk about AMOS itself.

The final section of this book introduces you to the concepts behind an operating system. You'll also learn about the actual components that combine to form the Alpha Micro Operating System.

PART III
AMOS OVERVIEW

Chapter 11 discusses the basic concept of an operating system. This chapter also gives a general overview of the Alpha Micro Operating System. The rest of the chapters in this book talk in more detail about the concepts introduced by Chapter 11. Therefore, we suggest that you read Chapter 11 first; then, read the particular chapter in Part III that is organized around the topic you are interested in. We remind you that this is not a section aimed at applications. We do not, for instance, discuss how to write terminal or device driver programs or how to use the AMOS monitor routines. Systems programmers will want to read the *AMOS Monitor Calls Manual,* (DWM-00100-42), and the *AMOS Assembly Language Programmer's Reference Manual,* (DWM-00100-43), for information on interfacing their programs with the Alpha Micro Operating System. System Programmers should also refer to the documents in the "System Programmer's Information" section of the AM-100 documentation packet.

CHAPTER 11
GENERAL STRUCTURE

Since you are reading Part III of this book, we assume that you are interested in the inner workings of AMOS. Because this book serves as an introduction to your system, and because the operating system can change with each new system software release, we won't be going into great detail on the various components of the Alpha Micro Operating System. Our main purpose is just to give you some idea of how the major portions of AMOS interact and work together to provide a flexible and sophisticated computer system.

> **NOTE:** This seems to be a good point to inject a note of caution. We have simplified our discussions of the operating system so that you can get a general feeling for how things work. But, this is not meant as a technical manual. Remember that some details have been simplified or omitted. For detailed information, see the *AMOS Monitors Call Manual,* (DWM-00100-42), and the "System Programmer's Information" section of the AM-100 documentation packet.

11.1 WHAT IS AN OPERATING SYSTEM?

The next few chapters discuss in some detail the various functions of AMOS, but for now let's just talk in general about what an operating system does. Of course, operating systems vary widely in power and complexity, but most of them perform the same kinds of functions (although with differing degrees of sophistication).

Imagine, for the moment, that you have the following units:

One or more terminals.

A hard or floppy disk drive.

Memory.

A printer.

You also have a computer that contains all of the additional hardware necessary to connect these devices. Hardware doesn't do you any good unless you can communicate with it, so now you need the software that can interface these components into a computer *system.*

Few people are really interested in entering machine language programs into a computer via toggle switches. Even fewer people would want to include in each machine language program the code that enables that program to communicate with disks, terminals, and other devices. If you are faced with providing a computer system with software, at this point you probably decide that it would be much more efficient if all of the support programs could be written only once, and then made available to other programs. This collection of programs, whose purpose is to assist in the running of other programs, is called an ***operating system*** or ***monitor.***

Your first task, then, is to design your own operating system. Even for a single-user system, this is a difficult task; when we consider a system that can handle multiple users and multitasking, we are talking about a formidable endeavor. But, let's start by breaking our operating system down into the major tasks we need to perform:

1. The first thing we have to worry about is being able to communicate with the CPU via our terminal keyboard, instead of with toggle switches. So, perhaps the first programs we write will be those that handle transferring data between the terminals and the computer system.

   (**NOTE:** Terminals are simply the hardware devices that enable us to communicate with the computer. ***CRT terminals*** or ***video display terminals*** are terminals with a video screen on which the computer displays data. A ***hard copy terminal*** is a terminal that prints your communications with the computer on paper instead of displaying the interaction on a video screen. Both types of terminals have keyboards on which you enter data and commands.)

   The Terminal Service System handles such things as echoing characters on our terminal display as we enter them and discarding control-characters from our input that the operating system does not use. The most important function that the Terminal Service System performs is to copy the characters we enter into a buffer so that other portions of the operating system can access and process our input. (**NOTE:** A ***buffer*** is a storage area in memory into which data is copied so that it can be worked upon.)

2. Perhaps the most important portion of our operating system is the Command Processor. This group of programs analyzes the characters that we enter and performs the proper responses to the commands made up of those characters.

3. The next thing we will want to do is to develop a set of routines that will help our programs to read from and write data to peripheral devices. These are the Logical I/O Routines. They allow our programs to communicate with devices without worrying about the physical attributes of those devices. For example, our program might know that it wants to read the first record of a file. It tells the Logical I/O Routines so, and this portion of the operating system decides what physical device address to access.

4. Special I/O problems exist when we talk to the disk because it is organized into files, so we need a set of programs to handle disk management. These programs (called a Disk Service System) work with the account and directory structure of the disk.

5. Another important function of our operating system is the handling of memory. The computer must use main memory for program execution and data manipulation. This group of programs tells us what areas of memory are free, and assigns memory areas needed by the programs we run on the system.

6. Many of our operating system programs will often need to perform the same kinds of functions (e.g., numeric conversions, file lookups, and display of account numbers). So, another type of operating system routine will be the utility routine. The utility routines are those programs used by other operating system programs (and, sometimes, your own programs as well) to perform frequently needed "housekeeping" functions. Perhaps the most important of the utility routines will be the ones that perform numeric conversion of data. All data is stored and manipulated in binary within the computer, yet we must be able to enter and display it in a format familiar to humans. The numeric conversion routines allow us to convert binary numbers to octal, hexadecimal, decimal, and character representations, and vice versa. We must also be able to handle any special representations used by our particular system.

In summary, our basic operating system has these components:

- Terminal Service System — Gives our terminal a way to communicate with the software that runs on the system.
- Command Processor — Processes and responds to the commands we enter.

- Logical I/O Routines and Disk Service System — Takes care of reading from and writing to devices.
- Memory Controller — Controls and allocates memory.
- Utility Routines — Perform frequently used "housekeeping" functions.

If we are able to create an operating system that performs these functions, we now have a fairly complete operating system for a one-person computer. However, a timesharing operating system is more complicated–it has to worry about all the added difficulties that arise when we have more than one user on the system at the same time.

Besides increasing the complexity of all of the functions above, a time sharing operating system requires several new sets of programs:

1. Job Scheduler and Controller — This component of the operating system schedules user jobs that run on the system and allocates CPU time and resources to those jobs.
2. Memory Management System — When several users are running on the system, it often becomes necessary to add memory beyond the direct addressing limit of the CPU. The memory management system allows you to set up multiple banks of memory, so that the total amount of memory on the system can exceed the normal limit.

11.2 BASIC STRUCTURE OF AMOS

Fortunately, you don't have to write your own operating system. AMOS performs all of the functions discussed above as well as a variety of others. We'll talk about some of these components of the operating system in more detail in later chapters, but first we'd like to give you a brief overview of how these routines work together to form the Alpha Micro Operating System.

To begin with, AMOS contains a program (we'll call it the ***Exec***) that coordinates all of the operating system functions. It also handles all of the job scheduling and control functions.

When you enter a command at AMOS command level, the Terminal Service System (TRMSER) places the various characters into a buffer assigned to your job. Exec (which is always waiting for new characters to be input), takes these characters and sends them to the Command Processor.

The Command Processor then looks for the program or command file requested by your command, and loads and executes it. (If the Command Processor can't find a program or command file that corresponds to your input or for some reason cannot load and execute the program, AMOS displays the appropriate error message via the Terminal Service System.)

Meanwhile, the Exec communicates with the Memory Controller and the File Service System (FILSER) in response to the demands made by your commands and your programs. (FILSER performs the functions of the Logical I/O Routines we discussed earlier.) The Exec may also call on various support routines (such as numeric conversion routines) in the process of carrying out its duties.

The diagram below may give you some idea of how this entire interaction takes place:

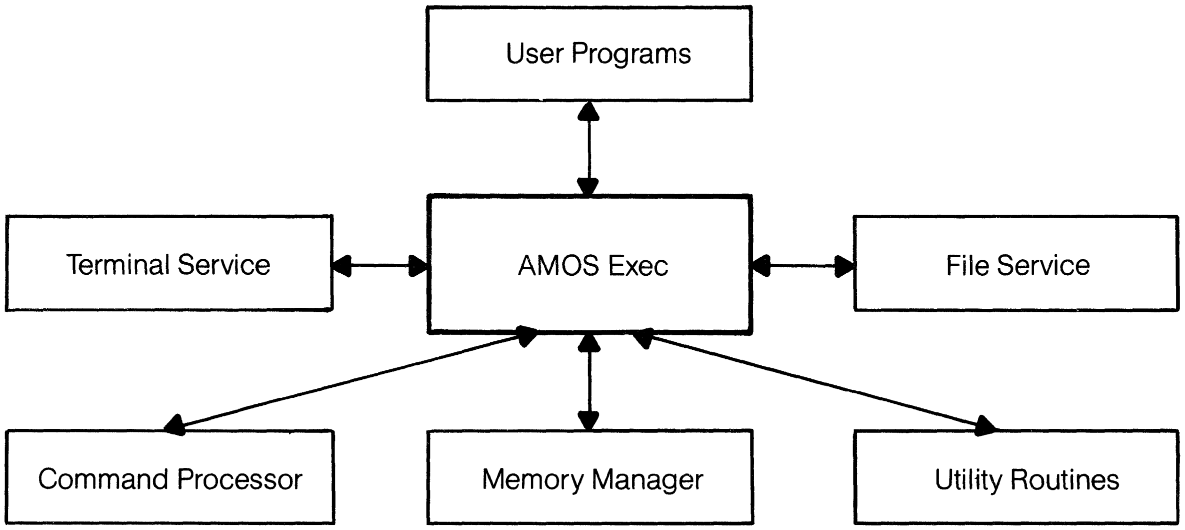


Figure 11-1
General Structure of AMOS

NOTE: Notice that the communication between the various parts of AMOS is not one-way. Different components communicate indirectly with one another by going through Exec.

11.3 AMOS MONITOR CALLS

Many of the functions that AMOS performs are functions that your assembly language programs will have to perform too. For example, your assembly language programs will often have to convert binary data into ASCII or RAD50 form, and vice versa. (***RAD50*** is a special, tightly-packed data format used by some programs in the operating system that formats three ASCII characters into two bytes, instead of the three bytes normally required.)

As another example, if your programs are going to be reading or writing data on the disk, you will have to include code that performs those operations, as well as file lookups and error detection.

Since the operating system already has routines that perform these kinds of functions, it would clearly be of great help to programmers if they could take advantage of these routines in their own assembly language programs. Many of these routines are, in fact, accessible to your programs.

AMOS contains over 70 operating system routines that your programs can use. They perform such widely ranging functions as: displaying a user account specification in ASCII form; checking to see if a particular disk file exists; fetching a line of data from a user terminal; and, performing a physical write operation to a disk.

You access these routines by including a monitor call in your assembly language program. (A ***monitor call*** is the coding used by your program to access a particular routine that is embedded in the operating system. These are also known as ***supervisor calls***) For information on the operating system routines you can use, refer to the *AMOS Monitor Calls Manual,* (DWM-00100-42).

CHAPTER 12
INTRODUCTION TO JOBS

In some of our earlier discussions of AMOS, we talked as if you were the only person on the system, and as if AMOS devoted its whole attention to your commands. Of course, you already know that a major advantage of the AMOS system is that it is a "timesharing" operating system. That is, it handles more than one user on the system at one time by sharing CPU time among users. Each user of the system has the illusion that he or she is the only one talking to AMOS, even though AMOS is almost simultaneously communicating with several other people. (In addition to being a timesharing system, we also say that AMOS is ***multi-user*** because it runs more than one user, and ***multitasking*** because it allows one user to run more than one task at the same time.)

Although we talk about AMOS communicating with you, the user, of course the operating system has no way of knowing who "you" are. Instead, the system initialization command file sets up the system to run a specific number of *jobs.*

12.1 WHAT IS A JOB?

The idea of a "job" is a very important one on the AMOS system, but it's a little hard to define. A job is *not* the same thing as a "user." Let's be fanciful for a moment, and compare a timesharing system to a carousel. The operating system is the carousel operator that supervises the running of the carousel. The painted horses are the system jobs, and the people riding the carousel are users. You might think of the horses as the way the carousel operator has of keeping track of the riders and of connecting the riders to the carousel. When you get on the carousel, you pick a horse and climb on. As far as the carousel operator is concerned, you then become that horse. You can easily change horses, and then you become a different horse. Whether or not the carousel has any riders, the same number of horses exist, and the carousel can only have as many riders at a time as it has horses. If more potential riders exist than horses, some riders must wait until current riders get off the carousel before they can get on. At any given time, any rider can be on any horse.

The system initialization command file defines a list of jobs that can be run on the system. Before a job can be used, it must be associated with a terminal, so that the job can communicate with AMOS. The system initialization command file usually attaches jobs to terminals. When you log into the system, you become the job that is attached to the terminal you are using. If you log in on another terminal, you become the job attached to that terminal. (Once logged in, you can re-attach your terminal to another job.) Once you log off the system, any other user can use the terminal you were using and log in as the job attached to that terminal.

Jobs are therefore the mechanism that AMOS uses to connect you, the user, to the operating system. Although jobs are generally used by human users, jobs can be set up to run the line printer spooler or other special tasks, instead.

The operating system safeguards itself from unauthorized users by the use of accounts. (An account is a structure on the disk that organizes the files on that disk.) To log into the system, you must enter a valid account number. As an additional security measure, accounts may have passwords assigned to them which you must enter to log into those accounts.

12.2 JOB SCHEDULING

The Exec contains the component of the operating system that handles jobs– the Job Scheduler. The ***Job Scheduler*** allocates, controls, and schedules jobs on the system.

Each job that runs on the system has two components that are unique to that job: the Job Control Block (JCB) and a memory partition. (A ***memory partition*** is the specific area of memory used by a job. We'll talk about memory allocation and control in Chapter 14, "Memory Control and Management.") The ***Job Control Block*** is an area allocated for each job within the operating system. Each JCB maintains specific information about its job. When it runs a job, the Job Scheduler consults that job's JCB for information on the status of the job.

The Job Scheduler maintains a list of active jobs on the system, called the ***run queue.*** Each job in the queue gets a specific amount of CPU time so that AMOS can process the job's requests for system resources. This amount of time (called a ***quantum***) is usually one-sixtieth of a second, or one tick of the real-time clock. You can change the length of a job's quantum by using the JOBPRI command. We say that if you increase a job's quantum, you increase its ***priority.***

When the quantum of a particular job is through, the Job Scheduler directs its attention to the next job in the queue. When the Job Scheduler reaches the end of the run queue, it begins again at the front. With the single-minded dedication that is one of the most important features of a computer, as the Job Scheduler moves to the next job in the queue, it quickly "remembers" where it left off during the last quantum of that job, and then temporarily "forgets" about the rest of the jobs in the queue. It devotes itself entirely to the specific job it is working on. The computer works so quickly that you are usually unaware that the operating system is handling jobs other than your own.

If a job is in a wait state, that job is removed from the run queue until the job can run again. (The term ***wait state*** describes the status of a job that is not ready to run because it is waiting for input or a system resource.) When it is no longer in wait state, the job is placed back into the run queue. If no jobs are ready to run, the Job Scheduler patiently waits until the run queue again contains an active job.

CHAPTER 13
COMMAND PROCESSING

To begin with, what is a "command"? A ***command*** is simply an order that you give to the computer that tells it to act in a specific way. Different computer systems handle commands in different ways. The most common way of processing operating system commands is to use a command table. A ***command table*** is a special list embedded in the operating system itself. This list contains the names of all of the legal system commands along with the addresses of the routines within the operating system that perform the functions requested by the commands. When such an operating system receives a command from you, it compares it against its command table. When it makes a match, it transfers control to the appropriate routine. The major disadvantage of this scheme is that the commands you can use are built into the operating system. That is, you cannot add new commands to the system without rewriting the operating system. Also, even if you do not use some of the commands, the routines that carry out those functions are still part of the operating system, taking up valuable space in memory.

The AMOS system offers a unique approach to command handling. Instead of canning command routines into the operating system (and causing you to reassemble the operating system whenever you want to change or delete a command), AMOS treats each command as the name of a program or command file. This gives your system optimum flexibility, since adding a new command consists merely of adding a new machine language program or command file to the disk. (**NOTE:** We talk about command files and DO files below.)

This method of command processing allows you to write your own commands, rename old commands, and delete commands you don't want to use. Changing your system software to reflect the newest enhancements introduced by Alpha Micro becomes a simple matter of adding new Alpha Micro command routine files to your System Disk.

13.1 COMMAND AND DO FILES

You've seen the terms "command file" and "DO file" frequently in earlier chapters. Command file processing significantly expands the range and power of the AMOS command language by allowing you to give AMOS a stream of commands and data by entering the specification of a single command file. (Those of you who are familiar with large, mainframe systems might think of command files as a way of doing "batch processing.") To perform all of the commands in a command file, just enter the name of the command file at AMOS command level.

A ***command file*** contains ASCII characters. Each line of the file is a valid AMOS command line or a line of data. ***DO files*** are a type of command file that can contain special symbols for which you can substitute arguments when you invoke the DO file. Command files can contain AMOS commands, specifications of other command files, and data for the specified programs to work on.

You may create your own command files by creating a text file with one of the system text editors. You might consider creating a command file to perform any series of commands that you enter frequently. For instance, if you often back up files onto a backup disk, you might want to create a command file to do this function for you. For example, the command file BACKUP.CMD looks like this:

```
:T
:< Command file to back up our working accounts onto DSK5:.
>

:< Make sure backup cartridge labeled BACKUPA is in disk drive.
Hit RETURN when ready.>
:K
;
; Make sure cartridge is mounted.
MOUNT DSK5:

; Log into System Operator's account [1,2] so we don't
; run into protection violations when we copy into accounts
; that are in projects we're not logged into.
LOG DSK5:[1,2]

; Copy files from one account into same account on DSK5:
; (the cartridge).
COPY =DSK1:[20,1],DSK2:[40,1],DSK3:[50,1],DSK4:[ ]

:< Remember to put today's date on cartridge label.

All done...
>
```

Figure 13-1
Sample Command File

The :T symbol above is a special symbol that must appear at the top of a command file if you want to see that file as it is processed by AMOS. The semicolons denote comments within the command file. The Command Processor ignores comments except to display them as it processes the command file if you have included the :T at the front of the file. The [ ] symbol is a "wildcard" symbol. (A ***wildcard symbol*** is a single symbol that can represent a range of elements. For instance, [ ] represents *all* accounts on a particular disk.) In the example above, the [ ] symbol tells the COPY command to copy all accounts on DSK4: over to the cartridge, DSK5:. The :K and :<> symbols are special command file elements that allow you to display messages to the user of your command file and to ask for input.

We discuss command files in this chapter because it is the Command Processor that handles command files. In fact, after the Command Processor loads a command file into memory, it processes the lines in that file almost as if you were entering the lines from the terminal keyboard (except for the fact that your command files and DO files can contain special symbols that you cannot enter from the keyboard). For detailed information on using command files and DO files, refer to Chapter 8, "Command Files and DO Files," in the *AMOS User's Guide,* (DWM-00100-35).

A special command file called the ***system initialization command file*** plays a very important part in the process of system configuration and initialization. This file is on your System Disk in account [1,4] as SYSTEM.INI.

We'll talk about SYSTEM.INI in more detail in Chapter 17, "System Initialization and Startup." For now, just remember that SYSTEM.INI is a very important command file that the Command Processor handles in a special way at the time of system startup. The file defines the configuration of your system, and tells AMOS what terminals, disks, special devices, and jobs you will be using on your system. The operating system actually builds itself in sharable memory in response to the information in the system initialization command file.

13.2 PROCESSING COMMANDS

When you enter a command at AMOS command level, the Exec accepts the command from the Terminal Service System. Then the Exec transfers control to the Command Processor. The Command Processor sets about looking for the program or command file that bears the same name as your command. (The search that it makes follows a very well-defined pattern. See the next section of this chapter for a detailed discussion of this search.) If the Command Processor cannot find the proper program or command file, the Exec lets you know that no command routine was found: it echoes your command back to you, bracketed by question marks. For example:

```
.PRIMPT RET
?PRIMPT?
```

If the Command Processor does find the correct program or command file (and if the module is not already in sharable memory or in the area of memory you are using), the Command Processor loads a copy of it from the disk into your area of memory. Then AMOS executes the program or processes the command file. When the program or command file finishes, it exits, returning control to the Exec. The Exec then performs some general cleanup functions which may include automatically deleting the program from memory and returning to AMOS command level. (The Command Processor always deletes the module unless that module has been loaded into memory via the LOAD command, or unless the module is a machine language program that itself orders the Command Processor not to delete it.) Those commands that you are not using do not take up any room in the memory used by the operating system or your area of memory.

13.2.1 The Command Processor Search List

The Command Processor follows a specific search procedure as it tries to identify your command. (Of course, if you enter the full file specification of the command, telling AMOS the device, account, and name and extension associated with the file, AMOS does not have to go through this search procedure.) Let's say you enter a command called REMOVE:

```
.REMOVE RET
```

Let's follow the imaginary thoughts of the Command Processor as it searches for the proper program or command file (remember that you are the "user"). Each numbered paragraph below details one step that the Command Processor makes in its search. If a step is unsuccessful, the Command Processor goes on to the next.

1. Is the memory module REMOVE.PRG in sharable memory or the user's area of memory? (The .PRG extension identifies a machine language program; ***sharable memory*** is that area of memory used by the operating system.)

2. Is the disk file REMOVE.PRG in the System Program Library Account, DSK0:[1,4]?

3. Perhaps the module is not a machine language program– for now, we'll assume that it is a command file. Is the memory module REMOVE.CMD in sharable memory or the user's area of memory? (The .CMD extension identifies a command file.)

4. Is the disk file REMOVE.CMD in the System Command File Library Account, DSK0:[2,2]?

5. Let's start looking for a .PRG file again. Is the disk file REMOVE.PRG in the account the user is currently logged into?

6. Is the disk file REMOVE.CMD in the account the user is logged into?

7. Is the file a .PRG file in the user's library account? (The ***library account,*** also known as the ***project library account,*** has the same project number as the account the user is logged into, but has a programmer number of zero (e.g., [110,0]). The library account contains programs and files that all users can share if their accounts are within that project.)

8. Is the file a .CMD file in the user's project library?

9. Well, the file does not appear to be a .PRG or a .CMD file. Now we load into memory the file DSK0:MDO.PRG[1,4] which helps us search for DO files.

   a. Is the disk file REMOVE.DO in the account the user is logged into?

   b. Is the disk file REMOVE.DO in the user's project library account?

   c. Is the disk file REMOVE.DO in the System Command File Library Account, DSK0:[2,2]?

10. If we still haven't found the disk file or module that corresponds to the command we are processing, the user's input must have been in error. Echo the command back to the terminal, bracketed by question marks to let the user know that we couldn't find the correct file or module.

    ```
    ?REMOVE?
    ```

We can summarize the search list in this way:

1. Look for .PRG module in sharable memory.
2. Look for .PRG module in user memory.
3. Look for .PRG file in DSK0:[1,4].
4. Look for .CMD file in user memory.
5. Look for .CMD file in DSK0:[2,2].
6. Look for .PRG file in user account.
7. Look for .CMD file in user account.
8. Look for .PRG file in user's project library account.
9. Look for .CMD file in user's project library account.
10. Load in file DSK0:MDO.PRG[1,4]:

    a. Look for .DO file in user account.

    b. Look for .DO file in user library account.

    c. Look for .DO file in DSK0:[2,2].

Now that you know how the Command Processor "thinks," you will know why, if you have several command files or programs of the same name, the particular account you are logged into can affect which program or command file AMOS selects to execute.

13.3 CHARACTERISTICS OF PROGRAMS ON THE AMOS SYSTEM

Remember that all commands invoke command files or machine language programs. Let's talk a minute about the programs that make up the command routines. All command files and machine language programs originally exist on the disk as files. Most command routines are ***transient***; that is, they exist on the disk, are loaded into main memory only when needed, and then are automatically deleted from memory when execution is finished. Such command routines can be made non-transient by loading them into memory with the LOAD command. In that case, they remain in memory until explicitly deleted by a user.

All programs on the AMOS system are ***relocatable,*** That is, they will operate properly anywhere in memory, without being modified or reassembled. This is necessary because there is no way of knowing beforehand which memory locations a program will have to be loaded into, since all users on the system use a different area of memory. AMOS automatically takes care of making higher-level language programs and command files relocatable for you.

Some machine language programs are also re-entrant. A ***re-entrant program*** is one that can be used by more than one user at one time. For example, BASIC can be invoked by one user, interrupted by another user who also makes full use of the program, and then re-entered at the point of interruption by the first user. Both users get correct results. Re-entrant programs are also known as ***sharable*** programs.

So that a re-entrant program can be used by more than one person, it must be loaded into sharable memory (the area of memory used by the operating system and resident system programs). The System Operator can add programs to the Resident Program Area by modifying the system initialization command file. The obvious advantage to sharing programs is that each individual user does not have to load the program into his or her own area of memory, but can access the single copy of the program in sharable memory. The disadvantage to loading re-entrant programs into sharable memory is that this expands the size of this area of memory, and reduces the amount of memory available for individual users. If the System Operator loads a program into the Resident Program Area, that program *MUST* be re-entrant; the computer will exhibit strange and distressing behavior if several users are sharing, at the same time, a program that is not re-entrant. If you want to write re-entrant programs, consult the *AMOS Assembly Language Programmer's Reference Manual,* (DWM-00100-43), for hints on doing so.

CHAPTER 14
MEMORY CONTROL AND MANAGEMENT

When you see the term "memory" in this book, we are talking about the random-access memory that makes up the temporary data storage on your computer system. (Remember that we discussed temporary storage devices, permanent storage devices, and random-access memory in Chapter 2, "What is a Computer?".) Although previous chapters have mentioned the importance of memory as a component of your computer system, this chapter will go into some detail on how AMOS manages, controls, and allocates memory.

Before any program can be executed or data manipulated, the computer must transfer a copy of that program or data from a permanent data storage device (that is, the disk) into memory, where the CPU can work on it. (When we transfer a copy of a file from the disk into memory, we say that we have ***loaded*** that file into memory.)

Memory is the only form of storage that the CPU can work on directly. It differs from disk storage in that the computer system can access it *extremely* quickly (in billionths of a second), and because memory offers only temporary storage; when the power goes off, the contents of random-access memory disappear. Because of these unique attributes, the computer uses memory as a work area– a scratch pad, in other words.

Each location in memory is consecutively numbered; that number forms a unique address by which a job or the operating system can access a specific memory location. Memory addresses can run from 0 to 65535 on the Alpha Micro computer system. This is because the CPU handles 16-bit numbers; the maximum number you can represent in 16 bits is 65535. (**NOTE:** Although this would seem to limit us to a maximum of 64K memory on a system (locations 0-65535), the AMOS system uses a memory management technique that allows us to have multiple sets of 64K memory. See Section 14.3, "Memory Management," for a discussion of this technique.) Memory locations near location 0 are known as ***low memory;*** locations near the other end of memory are known as ***high memory.***

When the operating system loads a copy of a program into memory, it does so by consecutively writing one byte of data per memory location. The Memory Controller keeps track of which locations are available for use. It also keeps track of the areas of memory used by specific jobs, and allocates memory for different uses. Although these functions are important under any circumstances, they become even more significant on a timesharing system, where different users are operating in different areas of memory at the same time. If some entity were not managing memory resources, it would be impossible to keep the operating system and jobs from bumping into each other throughout memory, writing over each other's data and programs. Because the AMOS computer does not have memory mapping or memory protection built into the hardware, software must keep track of what areas are in use, and what areas are available.

14.1 MEMORY MAP

The pattern in which memory is distributed to the various jobs on the system and to the operating system itself, is often known as the ***memory map*** of that computer system. The memory map of your system changes every time you change memory allocations.

When the system starts up, AMOS writes itself into memory, beginning with location 0. The amount of memory taken up by AMOS depends on your particular system and the particular devices connected to that system.

The remaining memory is available for user jobs except for the top 256 bytes of memory, which are used as the I/O ports. (You can see, then, that our earlier assertion that memory addresses can run from 0 to 65535 is not strictly true. Because the top 256 bytes are the I/O ports, memory addresses really run from 0 to 65279.)

14.1.1 Memory Partitions

Each job has its own area of memory, called a ***memory partition*** or ***user partition.*** The memory partition allocated to a specific job may be anywhere in memory, depending on what memory was available when that partition was assigned. A typical memory map for a 64K system might look something like this:

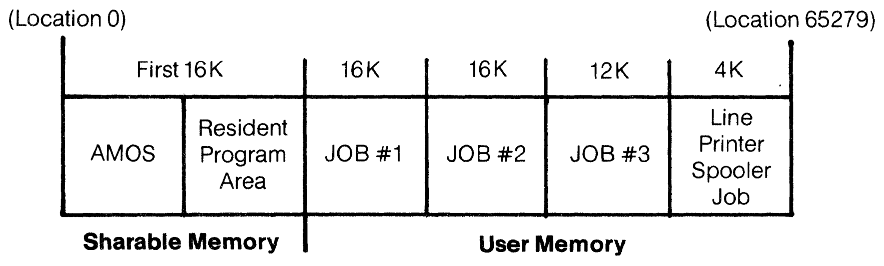


Figure 14-1
Typical Memory Map for a 64K system

The diagram above shows a system that uses the first 16K for the operating system and for resident programs. This 16K is called ***sharable*** or ***system memory,*** because all users on the system can access it. All users can access programs that are in the Resident Program Area without loading those programs into their own memory partitions. In addition, besides saving room in individual user partitions, putting programs in the Resident Program Area allows users to access those programs faster. This is because the programs do not have to be loaded into memory before they are used. Placing programs into the Resident Program Area is therefore a good idea if you have the room in sharable memory to do so, and if those programs are used frequently by most of the users on the system. (Placing a program into the Resident Program Area is done by the System Operator, who does so by modifying the system initialization command file.) **NOTE:** Any program loaded into the Resident Program Area *MUST* be re-entrant. (See our discussion of re-entrant programs in Section 13.4, "Characteristics of Programs on the AMOS System.")

The rest of the memory, called ***user memory,*** is nonsharable memory, and is devoted to user jobs. Our sample system has divided up the remaining memory into user partitions of 16K, 16K, 12K, and 4K.

The total amount of memory used in our sample system adds up to 64K (minus 256 bytes at the top of memory, which are used as the I/O ports and cannot be allocated to user partitions). Note that memory locations range from 0 to 65279. Each partition must contain contiguous memory locations. For example, all the memory locations that appear in JOB #1's memory partition must be consecutively numbered, with no gaps in those numbers.

Notice that the last 4K partition is set aside for a special job that is used by the line printer spooler program. The line printer spooler is a special program that allows your job to perform two tasks at once: printing a file while you are running a program in your partition. (The use of the spooler program is an example of ***multitasking;*** or, one user performing two or more tasks at the same time). What actually happens is that your job communicates with the line printer spooler job. Then the line printer spooler program allows you to place the name of the file you want to print in a printer request queue until the printer can get to it. (A ***queue*** is a waiting line.) A ***spooler,*** in a more general sense, is any program that enters (or "spools") items into a queue. (**NOTE:** Systems programmers may be interested in more information on the monitor queue system. See Chapter 5, "Monitor Queue System Calls," in the *AMOS Monitor Calls Manual,* (DWM-00100-42).)

14.1.2 Memory Modules

The contents of a memory partition may be divided up into groups called ***memory modules.*** Once you load a disk file into memory, that copy of the file becomes a memory module. The kinds of disk files you can load into memory are: programs, command files, and data. You can reference a memory module by giving the specification of its original disk file. For example, if you load the disk file HWK1:STDMOD.PRG[100,2] into memory, the copy of the file in memory becomes memory module STDMOD.PRG. Modules are built within a partition upward from the beginning of the partition as long as there is available space. As modules are deleted from memory, all modules above them in the partition are automatically shifted downward to fill up space left by the deleted module. When any module changes size, modules above it are shifted accordingly. This ensures that all available memory is always at the top of your partition. AMOS processes command files by loading them into the upper part of your partition and processing them there. For more information on the format of a memory module, see Chapter 3, "Memory Control System Calls," in the *AMOS Monitor Calls Manual,* (DWM-00100-42).

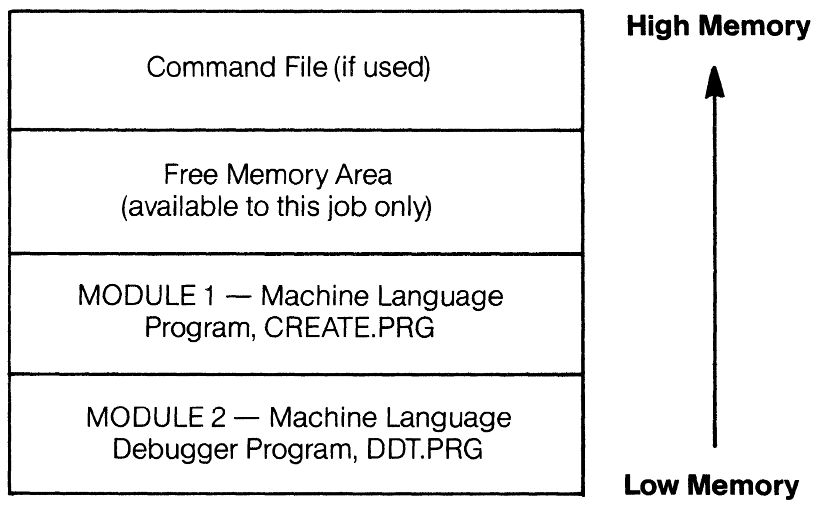

Command File (if used)
Free Memory Area (available to this job only)
MODULE 1 — Machine Language Program, CREATE.PRG
MODULE 2 — Machine Language Debugger Program, DDT.PRG
High Memory
Low Memory

Figure 14-2
Memory Map of a Typical User Partition

The diagram above shows the arrangement of modules in a typical user partition. Note that command files always are processed and executed at the top of the partition. Other modules are loaded in from the base of the partition up through the available memory area. When AMOS tells you:

```
?Memory map destroyed
```

it means that AMOS has lost track of the positions of the modules in your partition. If you continue to receive this message, you must reboot the system when convenient, or switch to another terminal and job, since AMOS's map of your partition has been destroyed beyond recovery.

14.2 MEMORY MANAGEMENT

Figure 14-1, "Typical Memory Map for a 64K System," gives a sample memory allocation scheme for a system with a total of 64K of memory. Notice that the largest user partition on this sample system is 16K. Although that is sufficient for some applications, jobs that do a great deal of program development or text preparation, usually require a larger memory partition. To make the partitions larger on our sample system, the System Operator must either reduce the size of the Resident Program Area or reduce the number of jobs on the system. Either solution presents problems of its own.

Because it is difficult to run many users on the system at the same time in 64K of memory, the Alpha Micro system uses an optional memory control technique called ***memory management*** to allow your computer to access more than 64K of memory.

Remember that the largest memory address we can form with a 16-bit CPU is 65535. The only way, then, we can have more than 64K of memory is to allow separate memory locations to have the same addresses. We differentiate between memory locations that have the same addresses by setting memory up into ***banks.*** Each bank may contain up to 64K of memory. The banks on the system are numbered consecutively beginning with number zero. You can have as many banks as you want; the limit depends only upon the amount of memory you can physically place in your computer chassis. One system can thus have over one million bytes of memory. Each bank cannot exceed 64K, however, and each job cannot use more than one bank.

In essence, we "fool" the CPU into thinking that there is only 64K of memory on the system by turning "on" the bank of memory used by the job currently being processed, and turning "off" those banks used by other jobs. This process is called ***bank switching.*** At any one time, therefore, the CPU thinks that it is accessing a maximum of 64K. AMOS, however, knows which bank of memory should be active at any one time and which banks should be inactive. These banks of memory (because they are "swapped" or "switched") are called ***switchable memory.***

Non-switchable memory is the area of memory we have been calling sharable memory. Sharable memory is used by all jobs on the system, since it contains the operating system and any resident programs. Since sharable memory is accessed by whatever job is currently being handled by the Job Scheduler, sharable memory is *never* turned "off"; it is always active. In fact, sharable memory is considered to be part of the maximum 64K of memory that can be active at any one time. Because of this, the actual size of each bank can never be more than 64K minus the size of sharable memory. Therefore, if sharable memory must take up 16K, each bank must never be larger than 48K. (In other words, sharable memory plus the currently active bank equals the total amount of memory active at any one time.)

To make these concepts clearer, let's assume that we have a system that has three 64K memory boards, for a total of 192K of memory. We will dedicate 16K to sharable memory and divide the rest among four banks. Our memory configuration might look something like this:

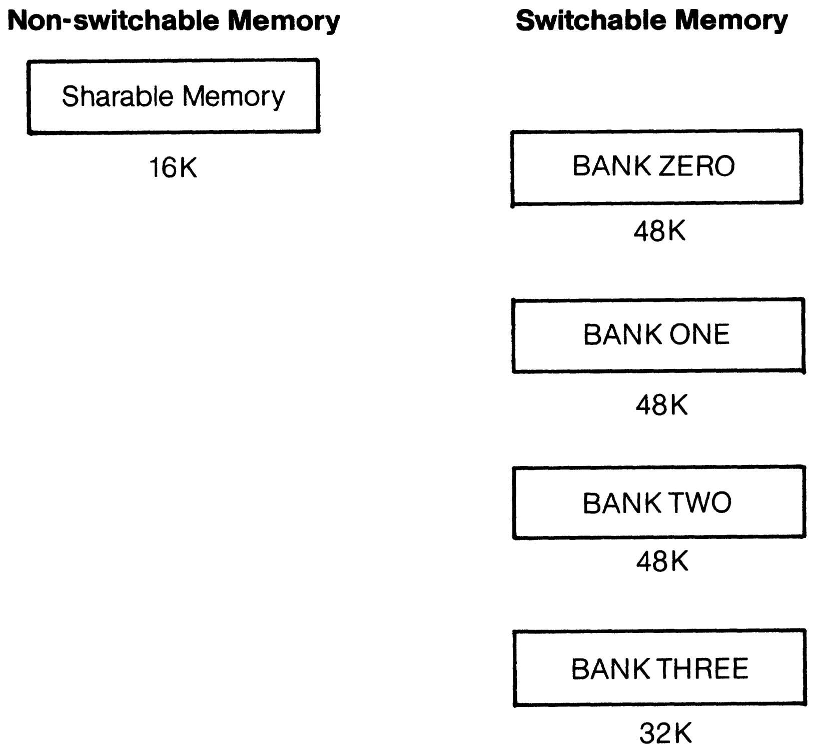


Figure 14-3
Bank Switching Memory

If you are interested in how the three physical 64K memory boards make up four banks, this is how we do it:

> The first memory board makes up the 16K sharable memory and the first switchable 48K bank. The second memory board makes up the second 48K bank and the first 16K of the third bank. The third memory board makes up the last 32K of the third bank, and this board also makes up the last bank, which contains only 32K.
>
> You can see, then, that memory banks are a *logical* grouping of memory, not a physical one. That is, one bank may consist of memory from more than one physical memory board. In our example, three 64K memory boards 192K of memory) make up 16K of sharable memory and 176K of switchable memory.

Of course, unless you are actually setting up the memory used on your system, the memory management AMOS performs is not something you have to be aware of. When your job is active, the memory your job uses is active too, and it appears to you that you are the only user on the system. Memory management offers a certain amount of memory protection, since it is very difficult for one job to get to a bank of memory outside of its own.

To tell AMOS that you want the system to bank switch memory, the System Operator must modify the system initialization command file to include the proper instructions. For information on setting up memory management, refer to the "System Operator's Information" section of the AM-100 documentation packet.

14.3 MEMORY ALLOCATION

Whether a system bank switches memory or uses only 64K of memory, the System Operator must assign memory partitions. Until user partitions are allocated, all memory past the operating system remains available but unused. Even once memory banks are set up (see the discussion of memory management above), the memory in those banks does not become useful until it is assigned to jobs. Several jobs may share one memory bank but no single job may use more than one bank. All memory locations within a single user partition must be contiguous (that is, the addresses of the locations must be consecutive, with no gaps).

On a system that uses memory management, you use the JOBMEM command to allocate memory to job partitions. This command requires that you give both the number of the bank that job is going to use, and the addresses (in octal) of the first and last memory locations used by the partition. The system initialization command file usually contains JOBMEMs that perform the initial memory allocations for jobs, but once the system is up and running you can also change those allocations by using the JOBMEM command.

If your system is not a bank-switched system, use the MEMORY command to allocate memory to jobs. You do not assign actual memory locations, but simply amounts of memory. AMOS allocates the amount you desire beginning with the first available memory area. The initial memory allocations are usually done from within the system initialization command file. Once the system is up and running, you can change memory allocations (again, by using the MEMORY command). Re-allocating can become a bit tricky if several users are on the system.

14.3.1 Re-Allocating Memory

On occasion, it becomes necessary to change the size or the location of user partitions. For example, one job may want to run a program that requires more memory than is allocated to that job's current partition. At the same time, another job may not need all of its allocated memory. AMOS allows you to change the allocations of memory while the system is up and running. Naturally, jobs must not be executing programs or command files while someone is changing the memory locations in which those jobs operate.

The methods for doing re-allocation of memory differ, depending on whether or not your system uses memory management. Systems that use memory management use the JOBMEM command to re-allocate user partitions. One job may perform all of the re-allocations necessary by using a series of JOBMEM commands.

Systems that do not bank switch memory use the MEMORY command to re-allocate memory. Usually all jobs on the system cooperate together to shrink and expand their own partitions until the desired configuration is achieved. (However, by using FORCE commands, one job may enter the necessary series of MEMORY commands– e.g., FORCE JOB2 MEMORY 0.)

When you re-allocate user partitions, you actually change the size and locations of those partitions in memory. Often, changing one partition requires that you change the locations of other partitions as well.

To make clearer the idea of changing the locations of user partitions in memory, we'll step through a simple example. The sample below applies to systems that do not bank switch memory. Let's say that your memory is currently being used in this way:

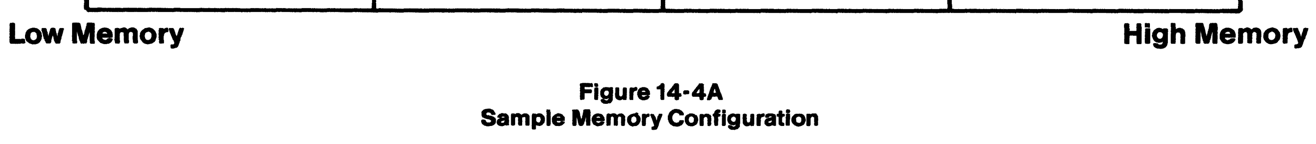


Figure 14-4A
Sample Memory Configuration

We have 16K of memory available. If we want to expand JOB2's partition, we can simply use the MEMORY command to tell AMOS to include the top 16K of memory in JOB2's current partition. This gives JOB2 32K of contiguous memory locations.

Expanding JOB1's partition is a little more complicated. Remember that all memory in a partition must be contiguous. Which means that, as matters stand, JOB1 cannot access the unused 16K because JOB2's partition is in the way. The only way that JOB1 can make its partition larger is to expand into JOB2's area. To do this requires a little cooperation from JOB2. First, JOB2 must use the MEMORY command to free up *all* of the memory in its partition for the use of JOB1. To do this, JOB2 enters:

```
.MEMORY 0 RET
```

To free up all of the memory in its partition, JOB2 must enter a zero to the MEMORY command; a non-zero number (no matter how small) will leave some amount of memory assigned to the partition, and will leave JOB2's partition standing in the way of JOB1's access of the unused 16K of memory that is at the high end of memory.

JOB1 now has all memory from its own partition up to the end of memory (excluding, of course, the top 256 bytes for the I/O ports) available for its use. It uses the MEMORY command to expand its partition. To get a partition that contains a total of 32K of memory, JOB1 enters:

```
.MEMORY 32K RET
```

Now JOB1 has the first 32K of memory past the operating system. We've now expanded JOB1's partition in memory, and must now re-allocate memory to JOB2. JOB2 now enters:

```
.MEMORY 16K RET
```

AMOS allocates the last 16K of memory to JOB2's partition. We've now sucessfully moved the partitions belonging to JOB1 and JOB2. Our final memory configuration looks like this:

16K (Operating System)	32K (JOB1)	16K (JOB2)

Low Memory **High Memory**

Figure 14-4B
Memory Configuration After Re-allocation

IMPORTANT NOTE: One reason why JOB2 has to use the MEMORY 0 command is that its partition would otherwise stand in JOB1's way. Another important reason is that when a job attempts to perform a task and it has *no* partition at all (that is, no memory is assigned to it), AMOS attempts to allocate to it any memory available to that job. Therefore, if you allocate a zero amount of memory to a job, the next time that job tries to do something, AMOS will give it all the memory it can. (You can see that in the case above of re-allocating memory to JOB2, we didn't really have to use the MEMORY 16K command; AMOS would automatically have allocated the top 16K of memory to JOB2 when JOB2 next tried to run a command program.)

CHAPTER 15
TERMINAL HANDLING

The Terminal Service System (TRMSER) is the portion of AMOS that makes possible the communication between terminals and software on the system. (We've already talked about terminals in Chapter 11, "General Structure." Briefly stated, a ***terminal*** is a character-oriented device that allows both input and output. The terminal is the device that allows you to communicate with the computer system.)

TRMSER is a general terminal processing routine that allows your programs (as well as the programs that make up AMOS itself) to communicate with terminals connected to the system. Those programs are thus not required to know what kinds of terminals or interface boards must actually be physically accessed. It acts as a kind of clearing house for data, making input from the terminals accessible to AMOS and your programs, and transferring output from AMOS to the terminals.

Think of TRMSER as a switchboard operator. You can tell the switchboard operator that you want to talk to Ms. Jones; the operator then worries about which office Ms. Jones occupies and what line to use to make the call. Your call may be routed through a complicated maze of equipment, but you need do nothing more than contact the switchboard operator. In the same way, programs talk to terminals by going through TRMSER.

TRMSER takes care of such general functions as: retrieving your terminal input and placing it in a memory buffer so that other portions of the operating system can access that input; funneling operating system output to your terminal; and echoing characters on your terminal display.

15.1 DRIVER PROGRAMS

Different kinds of terminals handle input and output differently. For example, the same code sequence may cause two different terminals to perform radically different screen display options. We need a mechanism for translating TRMSER's commands into a form recognized by a specific terminal. We also need a way to transfer data from the terminal to the particular interface board to which it is connected. (***Interface boards*** are the hardware that physically connect the terminal to the computer system, and provide a physical pathway for data to flow between the terminal and the system. Several terminals may be connected to the same interface board, and you may have more than one type of interface board in your system. Different interface boards handle input and output differently.)

So, we need software that allows the Terminal Service System to communiate with a specific terminal and a specific interface board. The programs that allow it to do so are called ***terminal drivers*** and ***interface drivers.*** These driver programs take data from TRMSER and translate it into the form required by the hardware; then they route the data to the terminal through the interface board.

Terminal and interface drivers are two types of drivers; a ***driver*** is a program that allows the system to communicate with an ***I/O device*** connected to the system, such as a disk, magnetic tape unit, terminal, printer, or paper tape reader.

15.1.1 Terminal Drivers

The terminal driver processes all input and output characters, and determines whether or not these characters need special handling because of the special characteristics of the terminal. For example, most CRT terminals have the ability to back up and erase a character on the screen when we type a RUBOUT; a hard copy terminal, on the other hand, can only print forward, and therefore responds to a RUBOUT by echoing the deleted characters on the printed display.

Because different kinds of terminals have different characteristics, most kinds of terminals require their own terminal driver programs. For example, a terminal driver for a SOROC 120 terminal may not work for a BEEHIVE 100 terminal. When you add a new kind of terminal to your system, you will probably also have to add a new terminal driver for that device. (**NOTE:** It is sometimes possible to use one terminal driver for several different kinds of terminals; check with your System Operator for advice on this subject.)

Most terminal drivers are written to handle actual hardware devices. Some special cases, however, may not require that a job be physically connected to a terminal (for example, when you want to discard output characters or effect inter-job communication without use of a terminal). For these situations, where you do not use an actual terminal, AMOS allows you to define ***pseudo terminals.*** Two pseudo terminal drivers are included within AMOS and are called PSEUDO and NULL. They are used in conjunction with the PSEUDO interface driver. A good example of a job using a pseudo terminal is the job that runs the line printer spooler. The line printer spooler does not use an actual hardware terminal for job input or output.

Terminal drivers are files with .TDV extensions. They appear on your System Disk in account [1,6]. For example, the file DSK0:SOROC.DVR[1,6] is the terminal driver for the SOROC terminal. (PSEDUO and NULL are not .TDV disk files, but actually exist as part of AMOS.)

15.1.2 Interface Drivers

The interface driver is the program that transfers characters back and forth between the terminal and the hardware interface board that the terminal is connected to. Interface drivers are usually quite small programs. They perform any necessary initialization of the interface boards and set up interrupt processes if used by the boards.

Interface drivers are files on the System Disk in account [1,6] that have the extension .IDV. A special interface driver called PSEUDO is for use with the pseudo terminal drivers NULL and PSEUDO, and defines a pseudo interface for inter-job communication and control. (The PSEUDO interface driver is not a .IDV disk file, but is actually part of AMOS.)

15.2 HOW TRMSER WORKS

The Terminal Service System works with the terminal and interface drivers to provide a generalized way of getting information to and from terminals.

At the time of system initialization, AMOS builds a terminal definition block within sharable memory for each terminal. (AMOS consults the TRMDEF commands in the system initialization command file when building these units. The TRMDEF commands contain information on each terminal connected to the system.)

The ***terminal definition block*** maintains information about the terminal (e.g., what interface that terminal is connected to, what terminal driver that terminal uses, status of that terminal, and so on). AMOS then loads into sharable memory the correct terminal driver program and interface driver program for each terminal (unless these programs are already in sharable memory), and then executes the interface driver, which performs any necessary interface initialization.

TRMSER consults a terminal's definition block when transferring data between AMOS and that terminal. (Whn a terminal becomes attached to a job, the terminal definition block becomes linked to that job's Job Control Block. A terminal is said to be "detached" if it is not linked to a JCB.)

The most important function TRMSER performs is the input and output of characters. Terminal service systms are notoriously convoluted and involved. Just to give you an idea of what goes on when TRMSER is at work, we give some very simplified diagrams below. Remember, however, that the reality is somewhat more complicated than our simplifications. For more information on the Terminal Service System, see Chapter 7, "Terminal Service System," in the *AMOS Monitor Calls Manual,* (DWM-00100-42).

15.2.1 Inputting Characters

When you enter a character on your terminal keyboard, the character is routed through a complex network of processors. The process looks something like this:

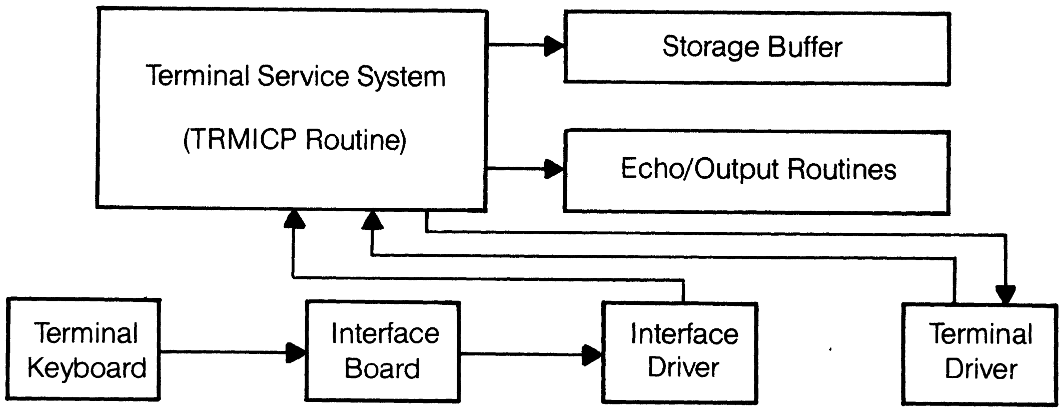


Figure 15-1
Inputting a Character through TRMSER

What actually happens is this:

1. The character is physically transferred from the terminal to the interface board by the interface hardware.
2. The character is then accepted by the interface driver program which passes it to TRMSER.
3. TRMSER sends the character to the terminal driver which does any character conversion needed.
4. The terminal driver sends the character back to TRMSER.

5. TRMSER puts the character into a buffer (unless it is to be acted upon immediately- e.g., a Control-C) so that some other portion of the operating system (or your program) can read the character. At this point, TRMSER echoes the character unless the terminal driver has already done so. (See Section 15.2.2., "Outputting Characters," for information on echoing characters and on the output routine.)

15.2.2 Outputting Characters

Terminal output characters come from TRMSER either as input characters to be echoed or as output from AMOS or your programs. Each character goes from TRMSER to the terminal driver so that the driver can perform any necessary conversions. We've simplified the process somewhat, but outputting goes something like this:

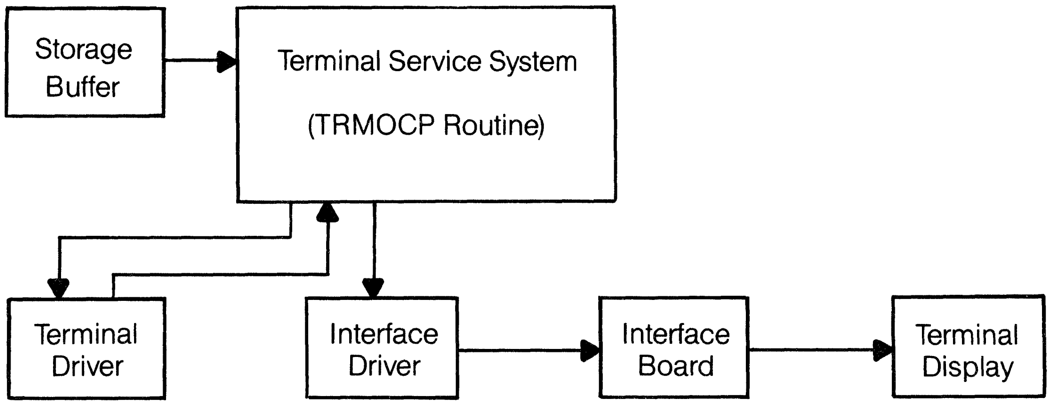


Figure 15-2
Outputting or Echoing a Character Through TRMSER

What happens is this:

1. When there is room and time to output or echo another character, the interface driver asks TRMSER for an output character.

2. TRMSER sends the character from the storage buffer to the terminal driver, which performs any necessary conversions. (For example, for timing purposes, some printing terminals require that the operating system send a special sequence of nulls after every line-feed.)

3. The terminal driver processes the character and sends it back to TRMSER for position processing. (For example, if we are trying to output a TAB, TRMSER has to calculate how many characters were output since the last tab stop on the screen so that TRMSER will know how many spaces to output for the TAB character.)

4. TRMSER then passes the output character (or converted character) directly to the interface driver.

5. The interface driver sends the character to the interface board, and thus to the terminal display.

Let's digress for a moment and talk about how your terminal echoes characters. (The next two paragraphs are especially for those of you who have always wondered why the FULL DUPLEX/HALF DUPLEX switch exists on your terminal.)

When a terminal is being used to communicate with the computer system, the characters that you enter on the keyboard are not automatically displayed by that terminal; instead, they are first processed by AMOS and then "echoed" (or repeated back) to the terminal display by TRMSER. This process is so fast that you are unaware of any delay between typing a character and seeing it on your terminal display. This mode (in which the computer echoes characters that it receives from a device back to that device) is called ***full duplex mode.***

Full duplex mode is the usual way data is transferred between the terminal and the computer. In ***half duplex mode,*** the terminal itself echoes characters back before the computer can receive and translate those characters. In this case, both the computer AND the terminal are echoing characters back to your terminal display, and you see two of each character that you type. This problem arises because your terminal is operating in half duplex mode, but the computer is operating in full duplex mode. Make sure that the switch on your terminal is turned to the FULL DUPLEX or FULL setting so that the computer and your terminal operate in harmony. (**NOTE:** Terminal drivers can be written to control terminals that only operate in half duplex.)

CHAPTER 16
HOW AMOS HANDLES DEVICES

This chapter introduces you to the portions of AMOS that handle devices. A ***device*** is any hardware unit that inputs or outputs data to the computer system. Devices are also known as ***peripherals,*** and include such units as disk drives, paper tape punches, printers, terminals, and magnetic tape transports.

One of the most valuable services that an operating system performs for you is to provide logical I/O routines that allow your programs to deal with devices on a "logical" rather than a "physical" basis. That is, your programs may want to access information on a disk without knowing anything about the physical attributes of that device. Logical I/O routines allow you to designate the data that you want to access by specifying the logical grouping in which that data appears (e.g., a file). AMOS then figures out the physical address of the device that the system must access to retrieve the data you want.

The two major portions of AMOS that handle data input and output are the Terminal Service System (see Chapter 15, "Terminal Handling") and the File Service System. These two sets of programs give your computer system a powerful flexibility by making it "device independent." That is, upon adding a new device such as a new type of terminal or disk drive, you need only add a new machine language program (called a ***driver***) that interfaces that device to the Terminal or File Service Systems. The operating system itself does not change to reflect the changes in your hardware, nor do you need to change your programs that access the devices on your system.

The Terminal Service System handles devices such as terminals and some printers. (To handle a printer through TRMSER, that printer must be defined as a terminal rather than as a general device and must be connected to an interface board. Also, a terminal driver and an interface driver must exist that can control that printer.)

The File Service System handles communication with non-terminal devices such as disks, magnetic tape units, and some printers. (To handle a printer through FILSER, that printer must be defined as a general system device and be connected to its own interface board. We must also have a device driver program that controls the printer.)

Let's talk a moment about why we define some printers as general devices, and some printers as terminals:

> We usually define serial printers as terminals because we can connect them to a terminal interface board (e.g., the AM-310), which makes them easier to interface to the system. We usually define parallel printers as general devices (thus requiring a device driver program rather than a terminal driver program for that printer) because such a printer requires its own hardware interface board.
>
> (A ***serial printer*** is one that outputs data serially- one bit at a time; a ***parallel printer*** outputs data in parallel- eight bits at a time. Parallel printers are usually faster than serial printers.)

16.1 THE FILE SERVICE SYSTEM

Some devices input and output data in a stream of sequential characters (for example, a paper tape reader); these devices are called ***non-file structured devices.*** Other devices contain data that is organized into complex structures called ***files.*** (See Chapter 5, "Introduction to Files," for information on files.)

The File Service System (FILSER) allows access of devices that maintain files (e.g., disk drives) and those that do not (e.g., magnetic tape unit). When FILSER has to communicate with a file-structured device, it does so by going through the Disk Service System (DSKSER). (We talk about DSKSER in Section 16.2, "The Disk Service System," below.)

FILSER and DSKSER work through special programs called ***device drivers.*** A device driver is a machine language program that allows the system to communicate with a specific I/O device connected to the system. They differ from terminal and interface drivers in that they perform very general, all-purpose I/O functions. (Terminal and interface drivers are highly optimized forms of device drivers that perform very specialized tasks.) Device drivers are files with one- to three-character names and the extension .DVR. They appear on your System Disk in account [1,6].

At the time of system initialization, the system initialization command file defines the devices that your system will be using. In response to the DEVTBL commands in the system initialization command file, AMOS builds within sharable memory a special structure called the ***device table.*** The device table tells AMOS what device driver programs will have to be used when accessing the devices on the system, and how many units (e.g., drives) each device contains. The device table also points to the bitmaps associated with disk devices.

When your job requests that AMOS perform I/O to a device, AMOS calls on FILSER, which accesses the proper device driver for the device you want to use. FILSER controls the device access by referring to the device table to make device assignments, and to a special unit in memory called the ***Dataset Driver Block (DDB).***

The program you are running sets up DDBs so that your program can pass information to FILSER; the DDB contains defining parameters such as device name, drive number, filename and extension, project-programmer number, buffer address, etc. One separate DDB exists for each file or device that is currently in use.

You might picture FILSER's interaction with devices and device drivers in this way:

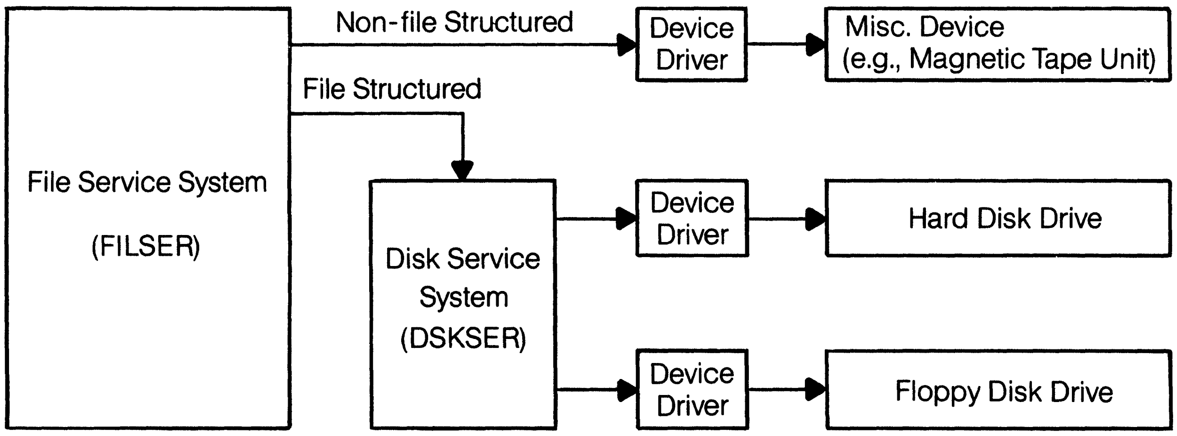


Figure 16-1
File Service System and Disk Service System

FILSER makes its decision whether or not to go through DSKSER based on characteristics of the device driver referenced in the device table. (That is, if the device is file structured, FILSER goes through DSKSER; otherwise, FILSER goes directly through a device driver.)

16.1.1 Special Device Drivers

The device table that you define at the time of system initialization contains the names of the devices you will be using on your system. The names of the devices are the same as the names of the device driver programs that will be used to access those devices. AMOS thus knows which device driver to use for which device. Most of these device names identify actual hardware devices such as disk drives or magnetic tape transports. Three special device names that do not identify actual devices should also appear in your device table; they are MEM, RES and TRM.

The drivers MEM.DVR, RES.DVR, and TRM.DVR are generalized device drivers that allow you to use as devices several units that normally would be handled by portions of AMOS other than FILSER.

- MEM.DVR allows you to access your memory partition as just another device. For example, the DIR command allows you to see what files exist in a disk account. Normally when you use DIR, you specify the disk on which the account occurs (e.g., DIR DSK0:*.*[1,6]). MEM.DVR allows you to reference memory as a device. For example:

  ```
  .DIR MEM:*.* RET
  ```

  The command above tells DIR to give you information about all files (that is, all memory modules) in your own memory partition.

- RES.DVR allows you to access the Resident Program Area as a device. For example, you can reference RES: to the DIR command:

  ```
  .DIR RES:*.* RET
  ```

  to find out what memory modules exist in the Resident Program Area.

- TRM.DVR allows you to access terminals as general devices. FILSER can access any terminal as though it were a device by using the TRM driver.

  For example, the COPY command allows you to copy data between devices (e.g., COPY DSK1:=DSK2:VUE.MAC). You can use TRM.DVR to send the output to a terminal rather than to another disk device. For example:

  ```
  .COPY TRM:TERM6=DSK2:VUE.MAC RET
  ```

  (**NOTE:** The name following the TRM specification, TERM6, is the name of the terminal we are trying to copy data to. The terminal name is assigned at the time of system initialization by the TRMDEF commands in the system initialization command file.)

Please note that there are some limitations placed on the use of special devices. For example, although you can use RES.DVR to find out what modules are in the Resident Program Area, you cannot use it with the COPY command to copy modules into the Resident Program Area. For more information on these special devices and device drivers, see *The System Initialization Command File,* (DWM-00100-09), in the "System Operator's Information" section of the AM-100 documentation packet.

16.2 DISK SERVICE SYSTEM

Without the benefit of a Disk Service System, any time your machine language program needed to access a disk, it would be necessary for you to create your own routines to perform file operations. DSKSER keeps track of files, block links, and block counts, so that your program doesn't have to. When your program wants to read or write a particular logical record in a file, DSKSER translates that request so that AMOS gives you the proper physical location on the disk. DSKSER also takes care of renaming files, deleting files, and performing file lookups for other portions of the operating system (or your programs).

As we mentioned in Chapter 6, "Permanent Data Storage," disks are physically structured into concentric rings called ***tracks.*** Those tracks are further divided up into ***sectors.***

Each sector contains a specific number of bytes of data. We call a sector a ***disk physical record,*** since it is an actual, physical grouping of data on the disk (as opposed to a ***logical record,*** in which data is grouped without regard to the physical attributes of the disk).

The particular pattern in which the data on the disk is written is called the ***format*** of that disk. Disk formats vary, depending on the type of disk drive. (For information on disk formats, see *Disk Drivers and Formats,* (DWM-00100-32), in the "System Operator's Information" section of the AM-100 documentation packet.) No matter what format is used by a particular disk drive, AMOS almost always reads and writes data in a group of 512 bytes. We call this 512-byte group a ***disk block.*** (The exception to this 512-byte block size occurs in the special case of the IMG: device driver. You use IMG: to access data on special devices in blocks of 128 or 256 bytes.)

Most hard disks available for use on your AMOS system also have sector sizes of 512 bytes. So, for hard disk systems, disk blocks are the same size as disk physical records. Your programs only concern themselves with logical records. (Although AMOS itself uses a disk block of 512 bytes, your BASIC programs, for example, might define data files that use logical records that are smaller than 512 bytes.) The Disk Service System translates your program requests for logical records into requests for the actual disk blocks used by those logical records. Your programs therefore do not have to keep track of how sequential file blocks are linked together, or how disk blocks map into the physical records on the disk.

16.2.1 Disk Structure

AMOS organizes every disk into the same basic structure. This section discusses briefly how AMOS organizes data on the disk, but for detailed information on disk structure, see Appendix A, "Disk Structure Format," in the *AMOS Monitor Calls Manual,* (DWM-00100-42).

The first block on the disk (block 0) is the disk I.D. block. Alpha Micro uses this disk block to maintain permanent identification information about the disk.

The next block (block 1) is the Master File Directory (MFD). We will talk about the MFD in the paragraphs below.

Beginning at block 2 is the disk bitmap. A ***bitmap*** is the structure AMOS uses to keep track of which blocks on the disk are in use, and which are available. The bitmap contains one bit for each block on the disk. If a block is in use, the bit in the bitmap that represents that disk block is a 1; if the block is available for use, its bit in the bitmap is a 0. The bitmap is permanently stored on the disk beginning with block 2 and extending as far as necessary. The last two words in the bitmap form a hash total. (A ***hash total*** is a special value that is computed based on the characteristics of a group of data. The hash total is used to check the integrity of a group of data or to uniquely identify that data.) Every time AMOS accesses the bitmap, it re-computes the hash total (by adding up all of the bits in the bitmap); if the new hash total and the old hash total are not the same, AMOS knows that some data in the bitmap has inadvertently been destroyed. It then gives you the error message:

```
    ?Bitmap kaput
```

You must then reconstruct the disk bitmap by running the disk analysis program, DSKANA.

Whenever AMOS attempts to write data to the disk, it:

1. Finds in memory (or loads into memory, if the proper bitmap is not already there) a copy of the bitmap of the disk it wants to access.
2. Computes the hash total of the bitmap and checks it against the old hash total.
3. Consults the bitmap to see what disk block is free for it to write into.
4. Changes the bitmap to show that the block is now in use.
5. Re-computes the bitmap hash total to reflect the modified bitmap.
6. Writes the modified bitmap back out to the disk.
7. Writes the data into the chosen disk block.

Whenever you change the disk in a drive (that is, when you change a floppy disk or change a hard disk cartridge), you must mount the new disk by using the MOUNT command. If you do not, AMOS has no way of knowing that the bitmap it may have in memory for that disk is no longer valid. Forgetting to mount a disk can cause AMOS to use the wrong bitmap when allocating blocks on that disk, which can destroy the data on your disk.

16.2.1.1 Account Structure— You already know that all files are organized into accounts. To make use of the AMOS system, you must log into a specific disk account. (Each account has a number called a ***project-programmer number (PPN)*** associated with it. To identify a specific account when you log onto the system, you supply the LOG command with the PPN of the account you want to use.) To allocate accounts on the disk (or to optionally assign passwords to those accounts), the System Operator uses the SYSACT command.

Of course, AMOS does not *physically* organize the disk area into accounts. That is, each account does not have a certain disk area for its files. Instead, AMOS keeps track of what files belong to which account by maintaining an account structure on the disk. This structure consists of a Master File Directory, various User File Directories, and the files themselves.

Block 1 (the block before the bitmap) is always the ***Master File Directory (MFD).*** Each disk contains one MFD. The MFD is one block long, and contains one entry of four words for each user account allocated on that disk. (Since the MFD is only one block long, each disk can thus have a maximum of 63 user accounts.)

Each entry in the MFD identifies a specific account directory. (Individual account directories are known as ***User File Directories***, or ***UFDS.***) The entry contains: the account project-programmer number; the number of the first disk block used by the UFD; and, the password (if assigned) of that account. The MFD, therefore, contains one entry for every UFD on the disk. The SYSACT command allows the System Operator to add, delete, and change MFD entries.

One User File Directory exists for each user account; it contains one entry for each file in that account. The UFD entry contains: the name and extension of the file; the number of blocks in the file; the number of active data bytes in the last block of the file (since the last group of data in the file may not fill the entire block); and, the number of the first disk block in the file. A UFD may consist of more than one disk block; if it is larger than one block, the first word in the UFD is nonzero and gives the link to the next UFD block.

The last element of the disk account structure is the file itself. We have already discussed sequential files and random files in Chapter 5, "Introduction to Files." You remember that a sequential file consists of blocks that are not necessarily next to one another on the disk, but which are linked together by special address words at the front of each block. Random file blocks are allocated in contiguous blocks on the disk.

A typical account structure might look something like this:

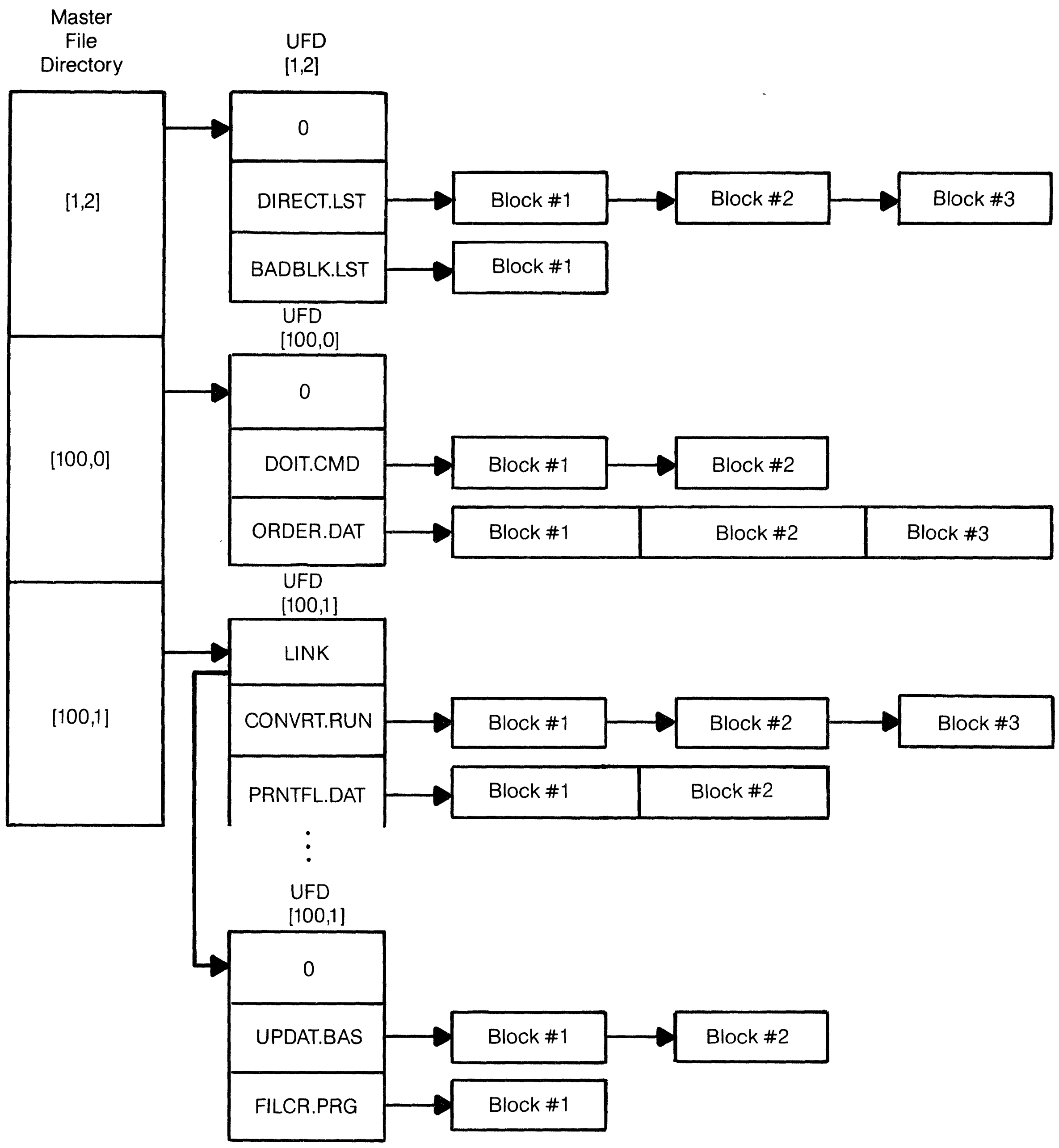


Figure 16-2
Sample Disk Account Structure

The diagram above shows a very simple account structure for a disk that has only three accounts. Notice that the UFDs above point to both sequential and random files. (For example, DOIT.CMD[100,0] is a sequential file; ORDER.DAT[100,0] is a random file.)

CHAPTER 17
SYSTEM INITIALIZATION AND STARTUP

Earlier chapters emphasized the flexibility of the AMOS system in adapting to new devices and system configurations. The key to this flexibility is the system initialization command file, which plays a major part in the processes of system initialization and startup. Other chapters have mentioned these processes in passing, but this chapter discusses in greater detail how your system starts up and the role that the system initialization command file plays in system initialization. (For more information on these topics, refer to the "System Operator's Information" section of the AM-100 documentation packet.)

The process of system initialization is a very important one on the AMOS system, because you can change the initialization procedure to reflect changes and additions to your system hardware. Those of you who are familiar with the operation of large-scale, mainframe computers may recognize system initialization as the equivalent of the ***sysgen*** (system generation) procedure used by such large computers.

Traditionally, small microcomputers have never faced the problem of having to perform a sysgen to integrate the various devices attached to the system, because such systems usually do not allow you to add or change the devices originally included with the system. Or, if you can add new devices, you must first get a reassembled version of the operating system from the computer manufacturer.

Large computer systems, on the other hand, often have to deal with the fact that users want to add new disk drives, printers, terminals, and other devices. Starting up such a system, which can handle a variety of devices, is called "performing a sysgen," and is considered a difficult, painstaking, and tedious task. Sysgens are performed only by a well-informed, experienced System Operator. Even though sysgens are not easily performed, they provide one answer to the problem of changing an operating system to reflect changes in hardware.

The philosophy behind the AMOS system initialization is to give users the ability to change the hardware configuration of their system without being forced to install a new version of the operating system or go through a lengthy and difficult sysgen procedure. This results in one of the most important features of the AMOS system– ***device independence.*** That is, the operating system is not restricted to working with a particular set of disks, terminals, or special devices. Adding a new type of terminal or disk drive to the system is as simple as adding a new device driver program to the System Disk and modifying the system initialization command file. In addition, because of the logical I/O routines embedded in the operating system, programs that access disks or terminals do not have to change when you add new devices, since AMOS takes care of translating program requests into the actual physical device read/write operations. For example, if you change your System Device from a Hawk disk to a Phoenix disk drive, your programs that access files on the System Device do not have to change even though the System Device is different.

17.1 SYSTEM STARTUP

Whenever you power up your system or hit the RESET button, AMOS sets about establishing itself in sharable memory. This process is called ***system startup*** or ***booting the system.*** This section gives a brief description of the system startup process, but for more details on exactly what happens when the system boots, see *The System Initialization Command File,* (DWM-00100-09), in the "System Operator's Information" section of the AM-100 documentation packet.

You already know that when you turn off your computer, all contents of random-access memory (RAM) disappear. Yet, AMOS must be in main memory if it is to supervise and control the jobs running on your system. This means that every time you power up your computer, AMOS must copy itself into RAM. Every time you reset the system, this process occurs again.

Your CPU contains permanent, wired-in instructions that tell it to start executing instructions contained in PROMs (Programmable, Read-Only Memory) that are located in your computer hardware.

The program in the PROMs transfers itself into RAM (between locations 31K-32K) where it begins to execute. This program (called a ***bootstrap loader***), reads in the file SYSTEM.MON from account [1,4] on your System Disk and loads it into RAM, starting at location zero. (SYSTEM.MON is the skeleton monitor that contains the kernel of AMOS.)

Incidentally, the term "bootstrap" is derived from the reference to the loop of leather found sewn to the top of some boots. It was once suggested that if a person could only lift himself up hard enough by his bootstraps, he might actually leave the ground. AMOS is successful in this feat by getting itself up and running.

17.2 SYSTEM INITIALIZATION

After it has been loaded into RAM, SYSTEM.MON takes over and begins to initialize your system. System initialization is the process by which AMOS completes building itself in memory. System initialization sets AMOS up so that it conforms to the specific configuration of the hardware on your system.

What actually occurs during system initialization is that the operating system brings into memory those driver programs necessary for communicating with the specific terminals connected to your system. The operating system also builds within itself areas in which to store information about the various jobs and terminals running on the system (e.g., terminal definition blocks, job control blocks, memory bank tables, system queues, and device tables). These processes all "customize" the operating system for the actual combination of devices and jobs that you want to use.

How does AMOS know what terminals and disk drives you are going to want to use? And how does it know how many jobs you want to run on the system? The mechanism that the Alpha Micro system uses for giving such information to AMOS is a simple one. When SYSTEM.MON begins to initialize the system, it sets up a temporary user partition in the top 8K of memory. It places this partition at the top of memory so that AMOS has room to expand as it builds itself in memory. (If your system bank switches memory, SYSTEM.MON sets up this user partition in the top 8K of Bank Zero.)

Then SYSTEM.MON loads into that partition a special command file, called the system initialization command file. (This file is found on your System Disk as SYSTEM.INI[1,4].) The first job and terminal defined in SYSTEM.INI is used by SYSTEM.MON for processing the SYSTEM.INI command file.

SYSTEM.MON finishes the system initialization process under the control of SYSTEM.INI, which contains information about the configuration of your system. The last instruction in SYSTEM.INI tells SYSTEM.MON to de-allocate the temporary user partition at the top of memory. The system is now up and running and ready for your commands.

17.2.1 The SYSTEM.INI File

The system initialization command file contains a series of command lines, each of which represents one system function or parameter that determines the characteristics of the running operating system.

Remember that until it begins to process SYSTEM.INI, AMOS is incomplete– a skeleton monitor. SYSTEM.INI is necessary if AMOS is to know what kinds of terminals are being used on the system, what kinds of disks are being used, whether or not memory management is being used, what jobs are being assigned, and so on. SYSTEM.INI actually defines your system to AMOS.

Your system comes with a SYSTEM.INI that has been set up to run with your hardware. Because SYSTEM.INI is simply an ASCII text file, you can easily change it with one of the system text editors to reflect changes in your system. Then, the next time you reboot the system, AMOS builds itself in memory in accord with the new instructions in your SYSTEM.INI.

We will not go into any detail here on the actual contents of the SYSTEM.INI file. Briefly, however, you can think of the commands at the front of SYSTEM.INI as describing and defining the hardware you want to use on your system (for example, defining terminals, devices, etc.). That is, the first part of SYSTEM.INI defines the resources you have available on your system. The second part of SYSTEM.INI generally takes care of allocating those resources (for example, allocating memory, attaching terminals, etc.) For detailed information on the modification and use of SYSTEM.INI, and for a sample SYSTEM.INI, refer to *The System Initialization Command File,* (DWM-00100-09) in the "System Operator's Information" section of the AM-100 documentation packet.

The SYSTEM.INI file gives some important information to AMOS about your system. Below we list some of the important functions performed by the SYSTEM.INI file. At the end of each paragraph in this list, we give the name of the command which performs the described task.

Job Definition: SYSTEM.INI tells AMOS how many jobs will be running on the system and the names of the jobs. AMOS builds a Job Control Block within sharable memory for each defined job. The Job Scheduler consults each job's JCB for information about the current status of that job. ***JOBS Command.***

Terminal Definition: AMOS must know how many terminals will be running on the system, their names, and the terminal and interface driver programs necessary to run each kind of terminal. This portion of SYSTEM.INI also defines sizes of the buffers used by each terminal, which are allocated in sharable memory. AMOS builds a table in sharable memory that contains information about each terminal. It also loads into memory the terminal drivers needed by the terminals. ***TRMDEF Command.***

Device Table Definition: SYSTEM.INI provides a list of what devices will be used on the system. You must tell AMOS about any devices such as disk drives or magnetic tape drives so that it can access the proper device driver programs. (You do not have to tell AMOS about your System Device, since SYSTEM.MON has already been set up to run with the device driver for that disk drive.) AMOS builds a device table in sharable memory that contains information about the devices in use on the system. The File Service System consults the device table when it accesses a device. ***DEVTBL Command.***

Memory Management: If your system is going to use memory management, your SYSTEM.INI must define the memory banks you are going to use so that AMOS can tell which portions of your memory boards belong to which bank. ***MEMDEF Command.***

Bitmap Definition: Each disk that you are going to be using must have a bitmap defined in memory. (If your system uses memory management, these bitmaps may be in either sharable or switchable memory; otherwise, they must all be in sharable memory.) ***BITMAP Command.***

Expanding the System Queue: If you want additional blocks allocated to the system queue, your SYSTEM.INI must tell AMOS how many more blocks you need. AMOS builds the system queue in sharable memory. ***QUEUE Command.***

Resident Program Area Definition: If you want to load any programs into the Resident Program Area of sharable memory, you can only do so at the time of system initialization from within your SYSTEM.INI. These programs must be re-entrant and relocatable. ***SYSTEM Command.***

Because SYSTEM.INI is the system initialization command file, it can contain commands that other command files cannot. Many of the SYSTEM.INI commands either cannot be used as user commands once the system is up and running or perform different functions if used at that time.

All of the system definition functions listed above must be done at the time of system initialization. For your convenience, you may also place within SYSTEM.INI any command that can be performed at AMOS command level.

Some of the AMOS command level functions that are often performed by the SYSTEM.INI file are:

1. Attaching terminals and jobs via the ATTACH command. (Except for the first job and terminal defined, all other jobs and terminals must be explicitly attached to one another if those jobs are to be able to use terminals for input and output.)

2. Allocating memory to jobs. If your system uses memory management, you may use JOBMEM commands to allocate user partitions in switchable memory. If your system does not use memory management, you can use the FORCE and MEMORY commands to allocate user partitions. (The FORCE command sends input to another job. In this case, the job that is processing SYSTEM.INI is sending (or "forcing") a MEMORY command to another job in order to allocate a memory partition for that job.)

3. Setting terminal characteristics. SYSTEM.INI can use the SET command to set certain display options for the terminal attached to the job that is processing SYSTEM.INI.

4. Setting up a line printer spooler. This is usually done at the time of system initialization. (Remember that the spooler is the program that allows all users on the system to print files at the same time that they perform other tasks.)

5. Mounting disks. Before you can write to a disk, the bitmap for that disk must be in memory. The MOUNT command ensures that the proper bitmap will be used when AMOS tries to access the mounted disk.

6. The final statement in SYSTEM.INI is MEMORY 0. The MEMORY 0 command de-allocates the temporary 8K user partition that was allocated by SYSTEM.MON for processing SYSTEM.INI.

Any time you change the hardware configuration of your system, you must change the SYSTEM.INI to reflect those changes.

WARNING: Changing the SYSTEM.INI file is usually the responsibility of the System Operator. Although modifying your SYSTEM.INI is not difficult, it requires a thorough understanding of the system, and should not be done lightly. If you do modify it, be sure to read *The System Initialization Command File,* (DWM-00100-09), in the "System Operator's Information" section of the AM-100 documentation packet very carefully before doing so. Also, never modify SYSTEM.INI directly; make a copy of it under a different name and modify the copy. Then you can use the MONTST command to test the copy. That way, if something should go wrong, you still have a good SYSTEM.INI from which to boot the system.

Now that we've talked about the myriad of functions performed by the system initialization command file, you have a better idea of the power and flexibility the initialization process gives to your computer system.

EPILOG

The intention of this book has been to carry you from a very basic introduction of the structure, limitations, uses and science of computers to an awareness of the vocabulary, concepts and power of the Alpha Micro Operating System. So that the potential usefulness of the system can be better realized by you, the emphasis of the book has been on understanding why the system works as it does, rather than just discussing how to work it.

Most importantly, we have tried to illustrate the "big system" philosophy behind the Alpha Micro computer system.

Where you go from here depends on you. You now have a background in the concepts used by the other Alpha Micro software documentation– you are ready to really begin your self-education on the Alpha Micro system by tackling the specific documentation aimed at your particular needs. Be sure to read Appendix B, "Where Do I Go From Here?", for suggestions on what documents to read next. If you are confused by any of the phrases used in this book, refer to Appendix C, "The Glossary", for clarification.

This book was meant as a beginning and an introduction. Now that you've made AMOS's acquaintance, you've just begun– now it is time for you to establish a satisfying and successful working relationship with the system. We hope that this book has gotten you off to a good start in that endeavor.

And finally, it would be of great help to us in our future documentation efforts if you would take a moment to fill out the "Software Documentation Reader's Comments" form at the back of this book. We appreciate your comments and suggestions.

APPENDIX A
CONVERSION CHARTS

When using computers, a basic understanding of various numbering systems may be desirable. AMOS account numbers, for instance, are octal numbers. If you were to select an account number outside of the octal set (that is, a number with an 8 or 9 in it), you would receive back an unwanted result. It would be easier to understand why the problem occurred and how to remedy it if you realized you were dealing in octal rather than decimal.

We review some basic features of the binary (base 2), octal (base 8), decimal (base 10) and hexadecimal (base 16) numbering systems in this appendix. Then we give you a conversion chart for the first 100 expressions of those four numbering systems and tell you some of the easier ways to convert among them.

The stars in the below charts represent the object or objects which the character set enumerates; in reality we can only express numbers (which are ideas or concepts) by numerals, which are symbols.

A.1 DECIMAL OR BASE 10

The decimal numbering system is the one we have been familiar with since childhood. The entire character set consists of:

0	
1	★
2	★ ★
3	★ ★ ★
4	★ ★ ★ ★
5	★ ★ ★ ★ ★
6	★ ★ ★ ★ ★ ★
7	★ ★ ★ ★ ★ ★ ★
8	★ ★ ★ ★ ★ ★ ★ ★
9	★ ★ ★ ★ ★ ★ ★ ★ ★

The base number of the system is TEN, written 10, and represents ★★★★★★★★★★ objects.

A.2 BINARY OR BASE 2

The binary numbering system is the one computers actually deal in. The characters of the binary system represent electrical signals absent or electrical signals present, either of which is significant. The entire character set consists of:

0	
1	★

The base number of the system is TWO, written 2 (base 10) or 10 (base 2) and represents ★ ★ objects.

A.3 OCTAL OR BASE 8

The octal numbering system is used in the Alpha Micro System and other popular systems as a form of binary shorthand. The entire character set consists of:

0	
1	★
2	★ ★
3	★ ★ ★
4	★ ★ ★ ★
5	★ ★ ★ ★ ★
6	★ ★ ★ ★ ★ ★
7	★ ★ ★ ★ ★ ★ ★

The base number of the system is EIGHT, written 8 (base 10) or 10 (base 8) and represents ★★★★★★★★ objects.

A.4 HEXADECIMAL OR BASE 16

The hexadecimal numbering system is used as a form of binary shorthand also, requiring even fewer digits than base 8 to represent the same number. The entire character set consists of:

0	
1	★
2	★ ★
3	★ ★ ★
4	★ ★ ★ ★
5	★ ★ ★ ★ ★
6	★ ★ ★ ★ ★ ★
7	★ ★ ★ ★ ★ ★ ★
8	★ ★ ★ ★ ★ ★ ★ ★
9	★ ★ ★ ★ ★ ★ ★ ★ ★
A	★ ★ ★ ★ ★ ★ ★ ★ ★ ★
B	★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★
C	★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★
D	★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★
E	★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★
F	★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★

The base number of the system is SIXTEEN, written 16 (base 10) or 10 (base 16) and represents ★★★★★★★★★★★★★★★★ objects.

A.5 CHART OF CONVERSION TO EQUIVALENTS FROM 1 (BASE 10) TO 100 (BASE 10)

BINARY	OCTAL	DEC	HEX
00000001	1	1	1
00000010	2	2	2
00000011	3	3	3
00000100	4	4	4
00000101	5	5	5
00000110	6	6	6
00000111	7	7	7
00001000	10	8	8
00001001	11	9	9
00001010	12	10	A
00001011	13	11	B
00001100	14	12	C
00001101	15	13	D
00001110	16	14	E
00001111	17	15	F
00010000	20	16	10
00010001	21	17	11
00010010	22	18	12
00010011	23	19	13
00010100	24	20	14
00010101	25	21	15
00010110	26	22	16
00010111	27	23	17
00011000	30	24	18
00011001	31	25	19
00011010	32	26	1A
00011011	33	27	1B
00011100	34	28	1C
00011101	35	29	1D
00011110	36	30	1E
00011111	37	31	1F
00100000	40	32	20
00100001	41	33	21
00100010	42	34	22
00100011	43	35	23
00100100	44	36	24
00100101	45	37	25
00100110	46	38	26
00100111	47	39	27
00101000	50	40	28
00101001	51	41	29
00101010	52	42	2A
00101011	53	43	2B
00101100	54	44	2C
00101101	55	45	2D
00101110	56	46	2E
00101111	57	47	2F
00110000	60	48	30
00110001	61	49	31
00110010	62	50	32
00110011	63	51	33
00110100	64	52	34
00110101	65	53	35
00110110	66	54	36
00110111	67	55	37
00111000	70	56	38
00111001	71	57	39
00111010	72	58	3A
00111011	73	59	3B
00111100	74	60	3C
00111101	75	61	3D
00111110	76	62	3E
00111111	77	63	3F
01000000	100	64	40
01000001	101	65	41
01000010	102	66	42
01000011	103	67	43
01000100	104	68	44
01000101	105	69	45
01000110	106	70	46
01000111	107	71	47
01001000	110	72	48
01001001	111	73	49
01001010	112	74	4A
01001011	113	75	4B
01001100	114	76	4C
01001101	115	77	4D
01001110	116	78	4E
01001111	117	79	4F
01010000	120	80	50
01010001	121	81	51
01010010	122	82	52
01010011	123	83	53
01010100	124	84	54
01010101	125	85	55
01010110	126	86	56
01010111	127	87	57
01011000	130	88	58
01011001	131	89	59
01011010	132	90	5A
01011011	133	91	5B
01011100	134	92	5C
01011101	135	93	5D
01011110	136	94	5E
01111111	137	95	5F
01100000	140	96	60
01100001	141	97	61
01100010	142	98	62
01100011	143	99	63
01100100	144	100	64

A.6 CONVERSION METHODS

We will discuss in this section a few convenient methods of converting numbers among the four popular numbering systems. If you need to do more than become comfortable with the concept of converting between numbering systems, it might be wise to check entire books on the subject for more than the elementary explanation provided here.

A.6.1 Binary to Decimal

Binary arithmetic is based around 2 (base 10) and powers of 2. Conversion from base 2 to base 10 is a matter of identifying which quantity in base 10 is being represented by a base 2 expression. Let's assume a number in base 2 that we will convert to base 10 a digit at a time:

10110111

Picture this number in a grid which separates the digits:

128	64	32	16	8	4	2	1
1	0	1	1	0	1	1	1

Notice that the headers on the grid begin, from right to left, with 1 and increase by powers of 2. That is, 2 raised to the 0 power (2^0) equals 1. 2^1=2. 2^2=4. 2^3=8... ... and so on. Since we are dealing here with an eight-digit number, we have only gone as high as 2^8=128 (in which we *could* represent up to 255 (11111111) quantities. Of course, we could go on indefinitely.

Converting the number in the 2^0 column (or 1's column) first, we mentally scratch it down on our memory pad. Since a 1 appears in the 1's column, we write down a 1. A 0 in the 1's column would mean we would write down a 0, or, in fact, simply ignore that column. Then we look at the 2^1 or 2's column. A 0 in the 2's column would again mean we would write a 0. But a 1 in the 2's column means we put a 2 on our scratch pad under the 1. (We are going to add them later.) A 1 in the 2^2 column (or 4's column) means we put a 4 under the 2 on our scratch pad. A 0 in the 8's (or 2^3) column means we ignore it. A 1 in the 2^4 (16's column) means we put a 16 under the 4 on the scratch pad. We also put a 32 under the 16 on the scratch pad from the 2^5 column (32's). We ignore the 2^6or 64's column and record a 128 on the scratch pad from the 2^7 column. Therefore, on our mental scratch pad we hold this:

$$\begin{array}{r} 1 \\ 2 \\ 4 \\ 16 \\ 32 \\ \underline{+128} \end{array}$$

which we add to equal 183

Therefore, 10110111 (base 2) = 183 (base 10), which we derived by adding the appropriate powers of 2.

A.6.2 Decimal to Binary

Starting with a decimal number and converting to binary is a matter of subtracting from the base 10 number the highest power of 2 which can be extracted, leaving a difference greater than or equal to 0. We can record the power of 2 in a mental grid similar to the one used in the section above. The power of 2 is represented in the grid by a 1 in the proper column. If the next lowest power, subtracted from the difference, is less than 0, that power is represented in the grid by a 0. If subtracting the next lowest power leaves a difference greater than or equal to 0, then record it also in the appropriate column of the grid. And so on until the difference equals 0, at which time all remaining columns to the right, if any, are filled with zeros.

Let's convert the decimal number 204:

$204 - 128 = 76$	$(76>0)$	Place a 1 in the 128's column
$76 - 64 = 12$	$(12>0)$	Place a 1 in the 64's column
$12 - 32 = -20$	$(-20<0)$	Place a 0 in the 32's column
$12 - 16 = -4$	$(-4<0)$	Place a 0 in the 16's column
$12 - 8 = 4$	$(4>0)$	Place a 1 in the 8's column
$4 - 4 = 0$	$(=0)$	Place a 0 in the 4's column
0	$(=0)$	Place a 0 in the 2's column
0	$(=0)$	Place a 0 in the 1's column

128	64	32	16	8	4	2	1
1	1	0	0	1	1	0	0

A.6.3 Binary to Octal

The octal numbering system, or base 8, is very simple to convert from binary, since 8 is a direct multiple of 2. Since 8 is 2^3, we will transform each group of 3 digits in a binary expression into its equivalent octal expression of 1 digit. The binary digits in groups of 3 and their octal equivalents are:

000 = 0
001 = 1 (base 8)
010 = 2 (base 8)
011 = 3 (base 8)
100 = 4 (base 8)
101 = 5 (base 8)
110 = 6 (base 8)
111 = 7 (base 8)

Let's convert the binary numeral 100101010 to its octal equivalent:

100 101 010

Notice that we have broken the digits into groups of three. This is only for convenience, and not by standard or convention. To begin, take the group to the left, 100, and find its octal equivalent, which is 4. Record 4 on your mental scratch pad for a moment. Convert the middle group to octal, which is 5. Record this to the right of the 4. Finally, convert the group to the right into its octal equivalent, which is 2, and record that to the right of the 5. The numeral on your mental scratch pad is the octal equivalent of the binary expression.

Therefore, 100101010 (base 2) = 452 (base 8)

A.6.4 Binary to Hexadecimal

The hexadecimal, or base 16 numbering system (often known as ***hex***) is also very simple to convert from binary, since 16 is a direct multiple of 2 as well. Since 16 is 2^4, we will transform each group of 4 digits in a binary expression into its equivalent hex expression of 1 digit. Again, placeholders are implied. Alphabetic symbols represent values also, and are not to be considered as letters. The binary digits in groups of 4 and their hex equivalents are:

0000 = 0
0001 = 1 (base 16) = 1 (base 10)
0010 = 2 (base 16) = 2 (base 10)
0011 = 3 (base 16) = 3 (base 10)
0100 = 4 (base 16) = 4 (base 10)
0101 = 5 (base 16) = 5 (base 10)
0110 = 6 (base 16) = 6 (base 10)
0111 = 7 (base 16) = 7 (base 10)
1000 = 8 (base 16) = 8 (base 10)
1001 = 9 (base 16) = 9 (base 10)
1010 = A (base 16) = 10 (base 10)
1011 = B (base 16) = 11 (base 10)
1100 = C (base 16) = 12 (base 10)
1101 = D (base 16) = 13 (base 10)
1110 = E (base 16) = 14 (base 10)
1111 = F (base 16) = 15 (base 10)

Let's convert the binary number 1101010111001110 to hex:

1101 0101 1100 1110

To begin, take the group to the left, 1101, and find its hex equivalent, which is D. Record the D on your mental scratch pad. Convert the next group to 5, and record that also, to the right of the D. The third group converts to C, so record that also. Finally, the last group to the right converts to E, so record that. Now you have on your scratch pad the hexadecimal equivalent of the binary number above.

Therefore, 1101010111001110 (base 2) = D5CE (base 16).

Remember that this also is simply a method of shorthand to make binary data a little more handy to the programmer.

A.6.5 Decimal to Octal

Octal arithmetic is based around 8 (base 10) and powers of 8. You have to keep in mind when converting to octal that you must consider the multiples of the powers of 8. That is, 64 is 8 to the second power (8^2). But 128 is 2*(8^2), or the second multiple of 8 to the second power. When converting from decimal to octal, it is the multiples of the powers of 8 TIMES the powers of 8 which become the octal equivalent.

To illustrate, we will assume the decimal numeral 705. (In this illustration, all numerals appear in their base 10 form.)

> 705 is between 8^3 (512) and 8^4 (4096). 8^3 is the closest power of 8 which is less than or equal to 705,so we now consider which multiple of 8^3, or 512, is closest to yet less than or equal to 705. 2 * 512 is 1024, greater than 705. So it is, of course, 1*512 or 512. We calculate that 705 - 512 = 193. We record the multiple 1 on our scratch pad (we'll see it become the leftmost octal digit.) Having begun at 8^3, we must step down through the powers consecutively until we reach 8^0, or 1. 193 is the difference between the decimal number we're converting (705) and the multiple (1) times the power (8^3, or 512). We check to see whether the difference is greater than or equal to the next lowest power (8^2, which is 64) or less than it. If the difference is greater than or equal to the next lowest power, as it is in the example, we again consider which multiple of 64 is closest to but less than or equal to 193. We find that it is 3 * 64 = 192. We record the 3 on our scratch pad to the right of the 1. We subtract 192 from 193 to equal 1. 1 is the new difference, and this time it is less than the next lowest power (8^1 or 8). The power must be lower than the difference, so we simply record a 0 as placeholder on our scratch pad to the right of the 3. Once again, we step down to the next lowest power, which is 8^0 or 1. The difference of 1 equals the multiple (1) of the power, which is 1 also. Subtracting the multiple of the power from the difference would give us a new difference of 0. We record the last digit on the scratch pad, which is the multiple 1, next to the 0. Were there any more digits to the right to be filled, they would be placeholders, or zeros. Reading the scratch pad, we see 1301. This is the octal equivalent to 705 in the decimal system. That is, 705 (base 10) = 1301 (base 8).

Let's do another example, pictorially. We will convert 93 (base 10). Again, the numerals used here are the decimal equivalents of the values unless otherwise specified.

$$\boxed{\begin{array}{lll} & 93 > 64\ (8^2) & \\ & 93 < 512\ (8^3) & \quad\text{Subtract:}\ \begin{array}{r} 93 \\ \underline{-64} \\ 29 \end{array} \\ \text{So:} & 93 >= 1\times 64 & \end{array}} \longrightarrow \text{Write down a 1}$$

$$\boxed{\begin{array}{lll} & 29 > 8\ (8^1) & \\ \text{So:} & & \quad\text{Subtract:}\ \begin{array}{r} 29 \\ \underline{-24} \\ 5 \end{array} \\ & 29 >= 3\times 8 & \end{array}} \longrightarrow \text{Write down a 3}$$

$$\boxed{\begin{array}{lll} & 5 > 1\ (8^0) & \\ \text{So:} & & \quad\text{Subtract:}\ \begin{array}{r} 5 \\ \underline{-5} \\ 0 \end{array} \\ & 5 >= 5\times 1 & \end{array}} \longrightarrow \text{Write down a 5}$$

Therefore, 93 (base 10) = 135 (base 8)

Perhaps a convenient check to you is to convert the decimal number first to binary, then the binary to octal:

705 (base 10) = 1 011 000 001 (base 2) = 1301 (base 8)

93 (base 10) = 1 011 101 (base 2) = 135 (base 8)

A.6.6 Octal to Decimal

Converting from the octal to the decimal numbering system can be visualized most readily in a grid, as converting binary to decimal was. In this case, however, first a multiplication and then an addition must take place to convert each digit. Let's convert 577 (base 8) to its decimal equivalent:

64	8	1
5	7	7

Notice that the headers of the column are the powers of 8; from left to right they are 8^2 or 64, 8^1 or 8 and 8^0 or 1. The theory of conversion by this method is to multiply the number in each column by the header of that column, then add all the results. 5*64 is 320. We record 320 on our mental scratch pad. 7*8 = 56, so we write 56 under the 320. 7*1 = 7, and we write 7 under the 56. Then we add:

$$\begin{array}{r} 320 \\ 56 \\ \underline{+7} \end{array}$$

And the sum is 383.

Therefore 577 (base 8) = 383 (base 10).

Let's do another quick one pictorially:

We will convert 100 (base 8) to base 10.

$$\begin{array}{rcr} 1\times 8^2 & = & 64 \\ 0\times 8^1 & = & 0 \\ 0\times 0^0 & = & \underline{+0} \\ & & 64 \end{array}$$

Therefore, 100 (base 8) = 64 (base 10).

Again, perhaps the quickest way to check is to convert to binary from octal, then from binary to decimal. Octal to decimal conversion is rather easy, however, and a check may not usually seem necessary.

577 (base 8) = 101111111 (base 2) = 383 (base 10)
100 (base 8) = 1000000 (base 2) = 64 (base 10)

APPENDIX B
WHERE DO I GO FROM HERE?

Now that you've been introduced to the system, you are ready to begin communicating with AMOS and the programs that run under its control. You may be wondering which of the Alpha Micro software manuals to read next. We realize that the profusion of Alpha Micro software documentation may appear bewildering at first glance. The purpose of this appendix is to direct you to the next step in your Alpha Micro education. The discussions below assume that you are new to the AMOS system.

For a complete list of the documents available from Alpha Micro, refer to *A Guide to the Alpha Micro Software Documentation Library,* (DWM-00100-37).

B.1 IF YOU ARE THE SYSTEM OPERATOR

IMPORTANT NOTE:

If your system software has not been set up to reflect the particular combination of terminals and disk devices on your system (that is, if you are installing the system software yourself), you *MUST* read the "System Operator's Information" section of the AM-100 documentation packet before you do anything else! After the system is up and running, you will want to read the general user documentation, the *AMOS User's Guide,* (DWM-00100-35).

The system software installation may already have been done for you before you bought your system. In that case, you probably will want to read the *AMOS User's Guide,* (DWM-00100-35), right away before you begin to use the system. This manual gives actual system operation instructions for the beginning user of the AMOS system.

With every new software release, you ought to read the Software Release Notes at the front of the AM-100 documentation packet. This document will inform you of any important changes in the new software release. You will also need to consult the "System Operator's Information" section of that packet to see if any important changes have been made to the system software.

Once you have become an experienced user of the system, you will often refer to the *AMOS System Commands Reference Manual,* (DWM-00100-49), which contains reference sheets on all AMOS commands, including those privileged commands that only the System Operator may use.

(**NOTE:** Some of the commands discussed in the *AMOS System Commands Reference Manual* are potentially dangerous to the system if used incorrectly, and should be used only by you, the System Operator. If you are in charge of distributing software documentation, you may want to remove the reference sheets for those commands before distributing the *AMOS System Commands Reference Manual.*)

B.2 IF YOU ARE A GENERAL USER OF THE SYSTEM

The first thing you will want to read is the *AMOS User's Guide,* (DWM-00100-35). This manual gives general information on using the system and discusses the most important AMOS commands.

Next, you should glance through the "User's Information" section of the AM-100 documentation packet to see if you are interested in reading any of the documents in that section. (For example, if you are not going to be using the ISAM system for organizing and retrieving data, you will probably not want to read *Important Notice for ISAM Users.* However, you may be interested in one of the other documents in that section.)

Because you will probably want to be doing text editing at some point (for example, creating reports, letters, or programs), you should read the VUE reference manual, *AlphaVUE User's Manual,* (DWM-00100-15). You will also want to read the *TXTFMT User's Manual,* (DWM-00100-07) to find out how to use the Alpha Micro text formatting program.

After you have become familiar with using the system, you will want to refer to the *AMOS System Commands Reference Manual,* (DWM-00100-49), for information on all of the AMOS commands. (The *AMOS User's Guide* will introduce you to the AMOS commands you will use the most; however, the *AMOS System Commands Reference Manual* discusses all of the commands available on the system.)

B.3 IF YOU ARE A BASIC PROGRAMMER

First, you will want to read the *AMOS User's Guide,* (DWM-00100-35), to learn how to use the system.

Then, read the *AlphaBASIC User's Manual,* (DWM-00100-01), for information on Alpha Micro BASIC.

The next thing you will want to read is the VUE reference manual- *AlphaVue User's Manual,* (DWM-00100-15), so that you can use VUE to create BASIC programs outside of BASIC.

Finally, you will want to read the "BASIC Programmer's Information" section of the AM-100 documentation packet for information on some of the BASIC subroutines available for your use on the AMOS system.

B.4 IF YOU ARE AN ASSEMBLY LANGUAGE PROGRAMMER

To gain a general knowledge of system operation, read the *AMOS User's Guide,* (DWM-00100-35).

Your next step will be to read the *AMOS Assembly Language Programmer's Reference Manual,* (DWM-00100-43) and the *WD16 Microcomputer Reference Manual,* (DWM-00100-04). These manuals introduce you to assembly language programming on the AMOS system.

Once you begin to write assembly language programs, you will want to refer to the documentation on DDT (a machine language debugger program) in the *AMOS Assembly Language Programmer's Reference Manual,* (DWM-00100-43) to find out how to test and modify your programs. Also, for information on the Alpha Micro screen-oriented debugger, see the *AlphaFIX User's Manual,* (DWM-00100-69).

For information on making use of assembly language routines embedded in the operating system itself, refer to the *AMOS Monitor Calls Manual,* (DWM-00100-42).

Finally, you ought to read the "System Programmer's Information" section in the AM-100 documentation packet.

You will probably want to refer to the *AMOS System Commands Reference Manual,* (DWM-00100-49), for information on all of the commands available on the AMOS system.

B.5 IF YOU HAVE SPECIAL USES FOR THE SYSTEM

Beside the documentation we mention above, other manuals and documents are available that explain the use of various system programs.

For example, if you are interested in using the ISAM system for organizing and accessing data, you will want to read the *ISAM System User's Guide,* (DWM-00100-06), and the *Important Notice for ISAM Users,* (DWM-00100-36). If you want to program in the PASCAL or LISP languages, you will want to refer to the *AlphaPASCAL User's Manual,* (DWM-00100-08), and the *AlphaLISP User's Manual,* (DWM-00100-05), for information on those language processors.

APPENDIX C

GLOSSARY

This glossary contains entries for words and phrases that appear in this book. These terms also appear in other Alpha Micro software documentation. If an explanation below still leaves you feeling at a loss, turn to the Index and see if the word appears in this book. It may be that the context in which that word is used will make its meaning clearer to you.

The words or phrases for which we have included definitions appear below in **boldface.** Words in *italics* are terms that appear elsewhere in this glossary with their own definitions.

C.1 THE GLOSSARY

Absolute Memory Address — Specifies a memory location by giving its actual address within memory, rather than by specifying its position relative to some known reference point, such as a program or buffer.

Absolute Memory Area — A group of memory locations in memory specified by their *absolute memory addresses.* See *Absolute Memory Address.*

Account — A method of organizing individual files on a disk, usually related by their significance to you. See *Project-Programmer Number (PPN).*

Account Directory — See *Directory.*

Account Number — An *Octal* number identifying an account on a disk. See *Account.* See *Project-Programmer Number (PPN).*

Account Specification — Same as *Account Number.* See *Project-Programmer Number (PPN).*

Addressing Limit — The highest-numbered address which can be referenced by a CPU. In the Alpha Micro system, the CPU can access 2^{16} addresses (0 through 65,535).

AlphaBASIC — Alpha Micro version of the BASIC programming language. It is a *compiler* rather than an *interpreter* and can operate in both *interactive* and *compiler* modes. See *BASIC Language.*

AlphaLISP — Alpha Micro version of the *LISP* programming language. Designed especially for performing non-numeric computation such as symbol manipulation. See *LISP.*

Alpha Micro Operating System — Alpha Micro's program that supports all other programs on the *system.* See *Operating System.*

AMOS — Acronym for *Alpha Micro Operating System.*

AMOS Command — Any of more than 100 words or letter combinations entered by a user which AMOS can interpret to perform a specific operation.

AMOS Computer System — Any of Alpha Micro's combinations of *hardware* and *software* as employed by a single group of users.

AMOS Monitor Call — Any of more than 70 various codings to be embedded in an *assembly language* program which cause the access of *monitor routines* when the program is run.

AMOS Prompt — The dot or period symbol (.), indicating that AMOS is ready for a command to be entered by the user.

AMOS System — See *AMOS Computer System.*

Architecture — 1. In the CPU, the physical and electrical construction of the several integrated circuit devices combined with many interactive switching devices. 2. In a computer system, the *CPU, interface* devices and other *peripheral* devices connected together to interact for certain kinds of operations. 3. In a program, the construction of several logical procedures built upon one another to interact for a desired program result.

Argument — One or more numeric or alphabetic characters accepted by a *function* which affect the operation of that function and supply the data on which the function will work. For example, when commanding your system to print a file, PRINT is the function and the file specification you give is the argument. The function is thus limited to printing that file.

Array — A set or sets of characters arranged in a meaningful way or pattern, such as a grid of line segments or a list of words.

ASCII — Acronym for American Standard Code for Information Interchange. It is a widely accepted standard for representing all common and some special keyboard symbols with numeric values in the *Binary* numbering system.

Assembler Program (MACRO) — See *MACRO.*

Assembly Language — A group of symbolic names called *mnemonics* which can be arranged to form instructions that translate one for one to the numeric value of *machine language* instructions. The mnemonics are easier to remember and work with than the series of numbers that are machine language instructions. They are converted to numbers by use of an *assembler program (MACRO).*

Assembly Language Subroutine — See *Subroutine.*

Attaching Terminals — Before a *job* can communicate with the computer system, it must be attached to a terminal. This is done with the AMOS command called ATTACH. See *Job.*

Backup File — After using one of the Alpha Micro text editors, when writing a modified file back out to the disk, the version of the file before the modification is safeguarded in a special transfer process and then renamed with the same filename, but given the extension .BAK.

Bank — See *Bank of Memory.*

Bank of Memory — A method of adding memory to the system beyond the 64K addressing capability of the CPU. Separate banks of memory have duplicate addresses to 65,535, but only one bank is turned on in a given instant as the system timeshares between *jobs.*

BASIC Language — BASIC is an acronym for Beginner's All-purpose Symbolic Instruction Code. There are many slighty different versions of BASIC now, so it is not considered a standardized language. BASIC is the most popular computer language because it permits the use of familiar English words and mathematical symbols to perform operations.

BASIC Language Processor — Translates *BASIC language* commands into *pseudo machine language* commands which the *run-time package* can accept and process. The BASIC language processor can be an *interpreter* or a *compiler.*

BASIC Statement — A command in the *BASIC* language which is the equivalent to many *machine language* commands. BASIC statements are often similar to English terms or mathematical operators, and simple to learn.

Binary — The binary numbering system has 2 as its base and expresses all quantities with the numerals 0 and 1. The value of binary digits is evaluated by position in ascending order from right to left.

Bit — *A binary digit 0 or 1. It is the smallest unit of data in a computer.*

Bitmap — *The structure AMOS uses to keep track of which disk blocks are in use, and which are available.*

Block — *See Disk Block.*

Block of Text — *A general term meaning two or more lines of text as they appear on a CRT screen or a printout.*

Booting the System — *Or, "bootstrapping," from the phrase, "Pulled himself up by his bootstraps." At system startup, permanent instructions wired into the computer cause AMOS to start loading itself into memory.*

Bootstrap Loader — *The program in Programmable ROM (PROM) which transfers itself into Random-Access Memory (RAM) at system startup and loads the skeleton monitor program from the system disk.*

Breakpoint — *Designates the spot in a program where you want to interrupt execution to perform debugging functions.*

Buffer — *A storage area in memory where data is copied so it can be processed.*

Bus — *(pl. busses) - A circuit or circuits which provide a path of communication between two or more computer system devices.*

Byte — *Eight bits. See Bit.*

Card Punches, Readers and Sorters — Electro-mechanical devices for writing, reading and sorting *punch cards,* a media of permanent storage often processed in large batches at a time.

Carriage Return — Also, a keyboard key named CARRIAGE RETURN, RETURN, or RET. On a terminal, returns the *cursor* to column 1. At the AMOS command level, it signals the operating system that the commands in the keyboard buffer are ready to be processed.

Central Processing Unit or CPU — The CPU controls all tasks which occur in the computer system and all interfacing with the other components.

Chaining — In BASIC, the ability of a program to connect or link another existing program to itself during execution rather than having to incorporate a similar sub-program into itself.

Character-oriented Text Editor — A *text processor* using a pointer moved through the text by commands you enter to perform operations on that text.

Character String — One or more ASCII characters grouped to form an element of data.

Circuit — One or more electric or electronic devices, plus connectors, which form a complete, closed path for electron flow.

Code — 1. A set of instructions. 2. To prepare a set of instructions.

Command — An order which you give to a computer to cause it to respond in a specific way.

Command File — An ASCII text file of commands and data you define, whose contents can be processed simply by entering the name of that file.

Command Language — The set of all commands. The command language of AMOS is well over 100 separate commands.

Compiled Program — A program which has been translated by a *language processor* into *machine language* or a form which is close to it.

Compiler — A language processor which translates a *source program* into a *compiled program.*

Computer Language — Two types of computer languages, *higher-level languages* or *assembly language,* are terms for sets of elements we can combine to form programs. Each command can represent at least one and possibly hundreds of *machine language* elements in a way that is more convenient for humans. Ultimately, all commands are processed by the computer as machine language.

Computer System — See *AMOS Computer System.*

Concatenate — To connect or link in a series or chain. For example, to concatenate two files is to append one onto the other.

Conditional Test — An element of a program. The computer tests one or more *variables* to see if they meet the specified condition. Depending on the result of the test, the computer can be made to do different operations.

Contiguous Disk Blocks — Blocks of storage which physically adjoin on a disk.

Contiguous File — See *Random File.*

Control Character — Any of a special class of characters used to represent certain control functions. For example, a Control-M represents a carriage return symbol, and indicates that the RETURN key has been pressed. AMOS uses some of the control-characters to represent special functions (e.g., a Control-U deletes the current input line). To type a control-character, hold down the Control key and press the appropriate letter-key.

Control-C processing or trapping — In BASIC, the ability to redefine Control-C to do other functions besides interrupting the BASIC program. For example, to go to a second routine embedded in the program.

Copying — The process of duplicating data from one area or kind of storage to another without touching the original set.

CPU — See *Central Processing Unit.*

CRT Terminal — A hardware device enabling communication with the CPU, that has a video (Cathode Ray Tube) screen which displays *ASCII* characters.

Cursor — A moving indicator of light on a CRT terminal display that indicates the last position where characters were displayed (either your input or AMOS's output).

Data — A subset of information especially formatted for logical processing.

Data File — A file which contains related data. See *File.*

Data Processing — The rearrangement and refinement of data to make that data suitable for our purposes. Such data handling may include word processing, numeric computation, re-ordering, etc.

Data Structure — The particular way in which data is represented in memory, such as a matrix, a doubly-linked list or an array.

DDT — An interactive program that allows you to display and modify *assembly language* programs to detect programming errors.

Debugging — The process of detecting and removing errors in a program.

Debugging Program — See *DDT.*

Declaring Variables — In structured programming, the process of defining what types of *variables* will appear in the program (e.g., string variables, integer data, floating point).

Default — Data assumed by a command from internal sources if you do not supply information. For example, if you do not supply an extension to the file specification you give to the PRINT command, it defaults to .LST (indicating one kind of a text file).

Device — The peripheral units outside your CPU that it can communicate with.

Device Driver — A program that allows the CPU to communicate with a specific I/O device connected to the system.

Device Table — Generated by AMOS, it contains information about the devices in use on the system.

Diagnostic Test — Any of various programs that detect errors in temporary or permanent storage. (e.g., A memory diagnostic program called DIAG3 diagnoses problems, if any, in system RAM.

Directory — A list of files associated with an account.

Disk — A permanent storage device consisting of one or more disk-shaped, magnetically sensitive recording surfaces rotating inside a mechanism called a disk drive. There are two major types of disks. A "hard" disk is made of metal, and a "floppy" disk is made from a thin film of mylar plastic.

Disk Account — See *Account.*

Disk Bitmap — See *Bitmap.*

Disk Block — Divisions of 512-byte units which AMOS imposes on a disk for structuring purposes, regardless of the built-in structuring attributes of the disk. Disk block 0 contains the first 512 bytes on the disk.

Disk Controller — An electronic device enabling the CPU to control, read from, and write to the disk device. Some Alpha Micro disk controllers are: AM-210, AM-410, and AM-500.

Disk Format — The physical pattern in which data is written on the disk.

DO File — A special *command file* you can create whose parameters are assigned by arguments you enter when you invoke the DO file. See *Command File.*

Driver — See *Device Driver.*

Echoing — The response of the computer to a character typed on the keyboard. The image appearing on the terminal display is the echo from the computer.

EDIT — 1. The AMOS character-oriented text editor. 2. (verb) The process of making written material suitable for presentation.

Error Message — A message appearing on your display explaining that a command you have just entered cannot be performed, and usually telling you why.

Error Trapping — Within a language processor, the ability to react within the program to recover from an error rather than crashing the system or returning to AMOS command level.

Escape — A special key on the terminal keyboard (labeled ESC, ESCAPE or ALT MODE) that is used by different programs on the AMOS system to initiate special functions.

Execute — (verb) Carrying out the instructions that make up a program.

Extension — The characters following the *filename* that tell you and AMOS the kind of file it is. The name and extension of the file are separated by a dot.

External Library of Programs — A set of existing programs you can call to supplement programs you write.

File — A structure that groups logically related data together.

File Extension — See *Extension.*

Filename — a label identifying the name of a *file.*

File Specification — Consists of the disk name, filename, extension and the account number (e.g., DSK2:MYFILE.TXT[50,1]).

Floating Point Number — A method used by the computer to express a number physically either too small or too large (because of the number of digits) for the CPU to handle conventionally. By using scientific notation and shifting the decimal point, the number can be expressed in two segments- the significant digits and the exponent (e.g., 1,234,567,890 is expressed as approximately 1.23457E9, or 1.23457 X 10 to the 9th power; 1 billion).

Floppy Disk — A small disk of mylar plastic coated in magnetically sensitive material and enclosed in a cardboard sheath. Used for permanent storage, it is easy to handle and store.

Flow Diagram — See *Flowchart.*

Flowchart — A written or drawn outline of the logical "flow" of a procedure, to be converted in essence to a program.

Format (Disk) — See *Disk Format.*

Formatting (Text) — The process of arranging text into a finished format according to commands you embed in the source file.

Function — A special command that accepts an *argument* and returns an answer which the computer derived based on the argument.

Hard Copy Terminal — A *terminal* that prints on paper all interaction between you and the computer, as opposed to displaying it on a CRT screen. See *CRT Terminal.*

Hard Disk — A permanent storage device which consists of hard metal disks coated with a magnetically sensitive compound upon which data can be recorded. Some disks (called disk "cartridges" or "packs") are removable from the disk drive.

Hardware — A general term for the mechanical, electric and electronic aspects of the computer system.

Hardware Configuration — The arrangement of the physical components of your computer system, including the *CPU, peripherals,* interconnecting devices and memory boards.

Hash Mark — See *Hash Total.*

Hash Total — A computed value based on characteristics of the file, such that the hash total can uniquely identify the file.

Hawk Disk Drive — A brand of *hard disk* drive having one fixed and one removable disk.

Hierarchy — A structure built on different levels of power, importance or supervision.

Higher-level Language — Any *computer language* where one statement can take the place of groups of machine language commands for programmer convenience.

I/O — Abbreviation for input/output. The interface of the computer with the real world is through its ability to input and output data.

Input/Output Port — The mechanism for inputting (bringing in) and outputting (sending out) data that occurs when the computer communicates with its devices.

Instruction Set — The set of machine language instructions the CPU recognizes.

Integer Data — A type of data made up of integers. An integer is a "whole number." That is, it contains no numbers to the right of the decimal point.

Interactive — An interactive program allows you to change or modify the behavior of that program while it is running.

Interface Board — A *hardware* device that does the actual data transfer from the computer to the *terminal.*

Interface Driver — A *machine language* program that transfers data back and forth between the *interface board* and the *terminal.*

Interpreter — Reads each line of a BASIC program and performs all the commands on that line as it reaches them. See *Compiler.*

ISAM — An acronym for Indexed Sequential Access Method. It is an AMOS utility program for organizing and maintaining a large group of data.

Job — The structure AMOS uses to connect you, the user, to itself and to accept the data you enter via one of the system terminals.

Job Control Block (JCB) — An area allocated for each *job* within the operating system, maintaining specific information about the job.

Job Priority — May be increased from the normal allotment (usually 1/60 of a second) of CPU time dedicated sequentially to each job for processing that job's current tasks. Increasing the job priority allots more time per sequence to that job and decreases the user's overall waiting time for the task to be finished.

Job Scheduling — A method of organizing jobs in a timesharing system, where the computer must keep track of what each job wants, provide a portion of that service, and go on to the next job many times per second.

Key — The item in a file record you want to sort the file by.

Language Processor — A program that can understand and act upon commands in one of the various *computer languages.*

Linefeed — A key used primarily to allow you to move the terminal cursor down one line. It is usually marked with a down-arrow or LINE-FEED. Some programs use it for specific functions, such as single-stepping. The linefeed character is also used as a delimiter for BASIC data records.

LINK — See *Linkage Editor (LINK).*

Linkage Editor (LINK) — A program that ties separate *assembly language* program modules together and resolves references modules make to each other.

Linked File — A file whose records are not necessarily near each other on the disk, but are linked together by references in each record. Also called a *sequential file.*

LISP — Acronym for LISt Processing. A computer language especially useful for processing non-numeric computations.

List File — A file comprised of ASCII code, having the extension .LST. It usually contains text that is formatted and ready for presentation.

Logging In — Performing the sequence of events that tell AMOS that you are a legitimate user of the system and have an account (and password if required).

Logical Record — Where data is grouped within a file regardless of the physical sector sizes of the disk.

Loop — A routine to execute one or more instructions repeatedly. The instructions are the same in each repetition, but the data on which they operate is not.

Machine Language — The set of *binary* symbols and the instructions for combining them in a way that can be directly processed by the CPU.

MACRO — The Alpha Micro macro-assembler program which translates assembly language *mnemonics* to machine language code.

Magnetic Tape Unit — A permanent storage device accessible by the computer which drives a magnetically sensitive tape on reels at high speeds.

MAP Statement — In BASIC, a data-handling capability most often used to define groups of *variables* which are transferred in and out of disk files.

Master File Directory (MFD) — Contains a list of all user accounts (PPNs and associated passwords) that have files on the disk. The MFD also stores the disk address of each account's User File Directory. *See User File Directory (UFD).*

Memory — 1. The ability to store one or more *bits* of data electrically over a period of time. 2. Temporary data storage on a computer system.

Memory Allocation — The process of assigning *memory* to *jobs* by setting up *memory partitions* or *user partitions.*

Memory Bank — The logical grouping of memory locations into separate, numbered sets. Different banks can contain the same addresses, but the system can reference each bank independently. Allows the *memory management* option to make possible the use of more than 64K of memory on one system. See *Addressing Limit.* See *Memory Management.*

Memory Location — The physical location of a given *byte* of memory, which is permanently assigned an address relative to the beginning of memory (e.g., address 0 is the first byte of memory).

Memory Management — The technique of organizing memory where you can meet the needs of the operating system, resident programs and several jobs, using several times the amount of memory the CPU can normally address in a way that is transparent to the CPU.

Memory Mapping — A method of keeping track of which areas of memory are in use, and which are available.

Memory Module — A file that is copied into a user partition is a memory module. The term is used to differentiate a *file* in memory from a file on the disk. **Memory Partition or User Partition** — That area of memory reserved for one particular *job*; can be anywhere in nonsharable memory.

Memory Re-allocation — Using AMOS commands, you as the *System Operator* can re-allocate memory to the various *jobs.*

MFD — See *Master File Directory (MFD).*

Microcomputer — A general term for relatively small systems using a CPU based around a microprocessor, usually addressing 8 bits per cycle, or as many as 32 bits. Recent technological advances empower the microcomputer with computing abilities competitive with the large, mainframe computers.

Mnemonics — Symbolic names that are arranged to form the equivalents of *machine language* instructions. Mnemonics are in the form of words, rather than numbers, which are easier to remember. See *Assembly Language.*

Mode — A way a program or system may operate; implies that there is at least one other way. For instance, using the shift key of your *terminal* takes you out of lowercase mode, putting you into uppercase mode.

Modular Program — A well-structured program broken down into several sub-programs.

Module — See *Memory Module.*

Monitor — A term interchangeable with the phrase *"Operating System."*

Monitor Call — Any of more than 70 various codings to be embedded in an *assembly language* program which allow your program to access *monitor routines.* Also known as Supervisor Call.

Monitor Routine — Any of more than 70 existing programs in AMOS you can call from within your *assembly language* program, rather than writing your own.

MONTST Command — Used after modifying a copy of the *System Initialization command file* to test its validity. *MONTST* is an abbreviation of Monitor Test.

Mounting Disks — Using the MOUNT command to mount a disk tells AMOS to read into memory the bitmap for that disk.

Multiprogramming — The ability to have two or more system users running different programs at the same time.

Multi-user System — The ability of having two or more users using the same computer system at one time. The computer switches its attention from one user to another at a very high rate of speed, giving the appearance to each user that all the computer's time is being dedicated to that user.

Multiple Users — See *Multi-User System.*

Multitasking — The ability of the computer to do two or more tasks simultaneously for a single user.

Non-switchable Memory — The area of memory used by the operating system and any resident programs.

Numbercrunching — A term to describe the computer's ability to, or the act of doing, progressive numeric computations.

Numbering System — A system by which abstract quantities can be represented numerically. Each numbering system uses a different "base number." Each digit-position in a number represents a power of the base number used by the specific numbering system. In computers, the numbering systems most often used to represent a given value are base 2 *(Binary),* base 8 *(octal),* base 10 *(decimal)* and base 16 *(hexadecimal).*

Numeric Constant — A number appearing in a program which does not change value during the processing of the program.

Octal — The Base 8 numbering system where all values are expressed in relation to 8 or powers of 8. 8 (base 10) is written 10 in base 8; that is, $1*8^1$ (or 8) + $0*8^0$ (or 0).

Offset Value — A value expressing the positioning of an object or marker in relation to a known position, as opposed to positioning according to a fixed address.

Operating System — Consists of many machine language programs that provide an interface between you and the computer.

Paper Tape Reader — An electro-mechanical device which reads data stored in binary form using holes punched into a roll of paper tape, and sends that data into the computer for processing.

Parameter — Used synonymously with *Argument* in our documentation.

PASCAL — A *higher-level language* which promotes structured programming, used for both numeric and non-numeric data handling.

Password — An account can be protected from unauthorized use by requiring that a user enter a password before being allowed to log into the account.

PDLFMT — Abbreviation for Program Design Language Formatting System, the Alpha Micro text formatter that helps you create a *Program-Design Document.* It is a very specialized *Text Formatter.*

Peripheral — Any device used in association with a computer which adds to the versatility or the power of the computer to interface with the outside world.

Permanent Storage Device — Any device that stores binary data permanently (regardless of whether or not power is applied to the device). Common permanent storage devices are disks, magnetic tape, and paper tape.

Personal Computer — A computer used by a single user but usually severely limited in its expansion capabilities.

Phoenix Disk Drive — A brand of *hard disk* drive using five fixed and one removable disk.

Physical Record — The physical sector on the disk.

Pointer — An abstract marker controlled by a program where text is affected relative to the pointer. The pointer is movable in the text by commands you enter.

Port — See *Input/Output Port.*

PPN — See *Project-Programmer Number (PPN).* See *Account.*

Printer — A *peripheral* device which types data characters on paper under computer control.

Printout — A listing or record typed on paper representing data processed by a computer.

Priority — See *Job Priority.*

Procedure — A set of sequential steps to conduct an operation upon data. See *Program.*

Program — A sequence of instructions executed from first to last unless explicitly commanded in well-defined ways to break the sequence (as in a LOOP). The program is static, but defines the process which the processor, or computer, must perform.

Program Design Language Formatting System — See *PDLFMT.*

Program Execution — The act of running, or performing, the program.

Program Module — A portion of a program nearly or completely self-contained as a sub-program.

Program-design Document — A program-design document (the output of the PDLFMT program) defines and outlines the structure of the program you want to construct. Gives you a well-defined design to follow when coding your program.

Programmable ROM (PROM) — An electronic integrated circuit device acting as memory which can only be written to once (this is done after its manufacture, via special equipment). Since it cannot be written to again, it contains permanent instructions (e.g., a *bootstrap loader*). PROMs retain their instructions whether or not power is applied to the system.

Project-Programmer Number (PPN) — Also known as an *Account Specification* or *Account Number,* it is an *octal* number which identifies a specific account on a disk.

Prompt — The symbol returned by AMOS or another program that indicates that the program is ready for a command to be entered by the user.

Pseudo Terminal — Whereas normally AMOS requires that a terminal be assigned to a *job,* a pseudo (or software controlled, rather than hardware controlled) terminal may be defined to AMOS when a job does not require an actual terminal for input or output.

Punch Card — A card with small holes punched in it as a form of permanent data storage.

Quantum — The portion of time in a timesharing system that the CPU dedicates to each job before directing its attention to the next job. In the Alpha Micro system, it is usually 1/60th of a second. The quantum is alterable. See *Job Priority.*

Queue — A line of objects waiting to be processed. For example, the line printer queue.

RAD50 Form — A special form of data representation used internally by the computer that condenses three *bytes* of ASCII data into one 16-bit *word.*

Random — Describes the ability to access any one of a similar group of elements without referencing any of the other elements (e.g., a random-access device).

Random-access Memory (RAM) — A temporary storage device which can hold data as long as it has electrical power. (Some types of RAM devices require that the computer refresh the data within the RAM periodically.) Used as memory by the CPU.

Random File — Also *contiguous file.* A file whose blocks are physically adjoined on the disk, which the computer can access randomly (i.e., at any block and in any order) by computing an *offset value* from the front of the file. A method of structuring files which is efficient for data retrieval, but files are not expandable in length.

Read-only Memory (ROM) — An electronic integrated circuit device which is encoded with instructions at the time of manufacture that cannot be changed afterward. ROM is not dependent upon electrical power to hold its instructions, and is most often used to contain initial instructions for system startup procedures since it reads as quickly as temporary storage.

Reboot — To reset the computer system by causing a new copy of the operating system program to build itself in sharable memory. See *Booting the System.*

Record — See *Physical Record.*

Re-entrant — Also *Sharable Program.* A program that can be used by more than one person, and which must be loaded into sharable memory to do so.

Register — A temporary storage unit which can be used to store data for reference or manipulation. One register contains one data word.

Relocatable Code — A program that is independent of an absolute memory address, and which may be moved within memory.

Relocatable Program — See *Relocatable Code.*

Reset Button — The button on the CPU chassis which resets the system. See *Reboot.*

Resident Program Area — That area of system memory AMOS uses to load *resident system programs* into.

Resident System Program — Any of several re-entrant, sharable programs which are in the *Resident Program Area* of sharable memory.

Righthand Justification — The process of aligning the righthand margin vertically by spacing each line of text in a way that brings it flush to the margin.

RUN — 1. (as a verb) To perform or to execute. 2. In the *BASIC language processor,* the command telling the run-time package to execute the specified program.

Run Queue — The list of active jobs on the system, maintained by AMOS.

Run-time Package — The small machine language program which executes a previously *compiled* program.

Screen-oriented Editor — Displays text you are editing on the CRT terminal screen. Such an editor allows you to modify your text by moving the cursor to the appropriate position on the screen and then entering the proper commands.

Sector — A section of a *track* on a *disk.*

Segment — *MACRO* allows you to divide large assembly language programs into modules, called segments. See *MACRO.* See *Assembly Language.*

Sequential File — A file whose records are not necessarily near each other on the disk, but are linked together by addresses (links) in each record. AMOS accesses these records in sequence according to the links. See *Linked File.*

Sharable Memory — An area of memory on the system, which contains the operating system. As each job becomes active, it can access sharable memory.

Sharable Program — See *Re-entrant.*

Single-user System — A small computer system that can only support one user.

Single-step — The process of stepping through a program one line of statements at a time.

Skeleton Monitor — A program (*SYSTEM.MON*) on the *System Disk* which runs at system startup to initialize the system and complete the loading of AMOS.

Software — Programs that control the operation of computer system hardware. Also, any program available to the computer that is created by a person.

SORT (data) — The arrangement of data according to a *key* you enter. For example, sorting a list of names alphabetically according to last names.

Source File — A file you create (using a *text editor*) which appears exactly as you type it in. Various programs are used to create different versions of the source file.

Source Program — A program you create (using a *language program* or a *text editor*) which appears exactly as you type it in. You can then use another program (see *compiler*) to process that program and create a new, executable version of it.

Spooler — As a line printer spooler, a program that allows several system users to enter requests to print files at the same time, or one user to print a file while doing another task. More generally, any program that allows you to insert requests for action into a queue.

Startup — The process of energizing the system, then loading in the *skeleton monitor* that performs the system initialization, which in turn loads in the complete version of the operating system to make the system functional.

Step-wise Refinement — The division of a task into sub-tasks by the programmer, which are coded as *sub-programs.* Such a program is more easily understood than a less well-structured program.

String Data — Alphanumeric data made up of *ASCII* characters.

String Subscripting — In BASIC, the ability to excerpt the contents of a portion of a string. The *substring* may be in various forms in the string. See *String Data.*

Structured Program — A program built with the technique of using *sub-programs* (enabling people to more easily understand that program).

Subroutine — A small program found within the confines of a larger program, which is usually complete within itself.

Sub-sort — A refinement of a *sort*. For instance, when sorting by last names, for several entries with the same last name, sub-sorting those by first names.

Substring — A portion of a string. See *String Data.*

Switchable Memory — On a system that uses memory management, memory which may be allotted to the various jobs according to their requirements. No jobs may share the same switchable memory locations.

Symbol — In a general sense, any item that represents information. For example, the carriage return character is a symbol that represents a press of the RETURN key. More specifically, in *assembly language* programs, a symbol is a program label or *variable* that stands for data or a memory address.

Symbol Table — When using the AMOS debugging program *DDT*, instructions in a program which point to relative addresses when processed can be re-associated to their English names (which are labels) for easier analysis by the programmer. The list of these symbols and their relative position within the program comprise the symbol table.

Syntax — The form that a command or other entry to the computer must take. This includes correct punctuation and the proper order of command words.

System Administrator — See *System Operator.*

System Command — See *Command Language.*

System (Computer) — A complete structure of interacting hardware and software that work together to provide an integrated whole.

System Disk — A disk reserved for various system programs of the Alpha Micro system, including AMOS, the *Text Processors* and the *Language Processors.*

System Initialization — The process of identifying to the *Monitor* the hardware configuration, *jobs* and other parameters unique to a particular system.

System Initialization Command File — See *SYSTEM.INI.*

System Manager — See *System Operator.*

System Operator — A person familiar with the computer system who is responsible for the maintenance of a particular system, including initialization, operation and upkeep.

System Queue — A special queue set up by AMOS for its own use which is built in sharable memory.

System Resource — Any of the available resources on a particular system that may be allocated to a job for its use (e.g., disk access and printer use).

System Software — The various machine language programs supplied by Alpha Micro that make up the operating system and its support. This includes drivers, language processors, text editors and formatters, and utility programs.

SYSTEM.INI — The *System Initialization Command File* which contains all the information appropriate to your system configuration as required by AMOS. This file is accessed at system startup and reset.

SYSTEM.MON — The program which completes the system initialization process according to the instructions found in the *SYSTEM.INI* file.

Systems Programming — *Assembly language* programming that expand the power of the system, typically making use of *monitor calls. Device drivers, terminal drivers* and disk access routines are examples of such programming.

Tape Transport — The *Magnetic Tape Unit* mechanism which moves the magnetically-sensitive tape between the reels at high speeds.

Temporary Storage — *Random-Access Memory (RAM)* available to the CPU which can be written to and read from within billionths of a second, and where data remains while power is applied and is erased when power is interrupted.

Temporary Storage device — Any device supporting RAM of any type, and attached to a CPU in a way which is accessible. System memory.

Terminal — A device allowing communication between the computer and yourself, given the proper interface devices and programs. Especially a CRT terminal with a keyboard and a television-like screen, or a hard copy terminal with a keyboard and an associated printer.

Terminal Driver — A *machine language* program which translates data into a form that the *terminal* can understand.

Text Editor — A program which, during execution, enables you to create or modify text you enter via your terminal.

Text File — A file of ASCII data.

Text Formatter — A program which, during execution, reads commands embedded in the text of a file and, by following those commands, formats the text a special way and creates a finished file.

Text Processor Program — In the Alpha Micro system, any program that enables you to create, modify or format text. See *Text Editor.* See *Text Formatter.*

Timesharing — Several jobs using sequential portions of CPU time at a speed so quick each job usually appears to the user as if it were alone on the system.

Timesharing Operating System — An operating system, like AMOS, which is able to perform *timesharing.*

Toggle Switch — A device that usually has two positions, and that can be manually placed in one or the other of the positions.

Tokenization — The substitution of an abbreviated (usually 1-byte) symbol for a more memory-consuming *BASIC statement.*

Track — One of many concentric rings around a disk which are part of the data storage format of the disk.

UFD (User File Directory) — See *User File Directory.*

Unauthorized User — Any user either not permitted on the system by the *System Operator* or who cannot give the password to access a protected account.

User — A general term referring to a programmer, operator, or any other person who may apply the computer to a task or purpose of his or her own.

User File Directory (UFD) — A method of organizing files into accounts. Each UFD contains a list of all files in a specific account.

User Job — See *Job.*

User Partition — See *Memory Partition.*

User-defined Symbol — Any symbol in a program created by a programmer, such as a label or *sub-program* name.

Utility Routine — One of several programs that perform housekeeping and general usage functions such as file lookup, numeric conversion or display functions.

Variable — A symbol whose numeric or string value changes from one repetition of a program to the next, or changes within each repetition of a program.

Video Display Terminal — See *CRT Terminal.*

VUE — The Alpha Micro screen-oriented text editor that helps you create or edit any text file.

Wildcard Symbol — Any of several special symbols (e.g., * or [ ]) which allow you to select a range of elements using only one specification. For example, the DIR command recognizes the file specification *.TXT to mean all .TXT files of any name.

Word Processing — Applying the computer to creating, modifying and formatting text documents.

INDEX

TECHNICAL PUBLICATIONS READERS COMMENTS

We appreciate your help in evaluating our documentation efforts. Please feel free to attach additional comments.
If you require a written response, check here: ☐

NOTE: This form is for comments on documentation only. To submit reports on software problems, use Software Performance Reports (SPRs), available from Alpha Micro.

Please comment on the usefulness, organization, and clarity of this manual:

Did you find errors in this manual? If so, please specify the error and the number of the page on which it occurred.

What kinds of manuals would you like to see in the future?

Please indicate the type of reader that you represent (check all that apply):

- ☐ Alpha Micro Dealer or OEM
- ☐ Non-programmer, using Alpha Micro computer for:
  - ☐ Business applications
  - ☐ Education applications
  - ☐ Scientific applications
  - ☐ Other (please specify):
- ☐ Programmer:
  - ☐ Assembly language
  - ☐ Higher-level language
  - ☐ Experienced programmer
  - ☐ Little programming experience
  - ☐ Student
  - ☐ Other (please specify):

NAME: ________ DATE: ________

TITLE: ________ PHONE NUMBER: ________

ORGANIZATION: ________

ADDRESS: ________

CITY: ________ STATE: ________ ZIP OR COUNTRY: ________

STAPLE

FOLD

PLACE
STAMP
HERE

CUT ALONG LINE

alpha micro

3501 Sunflower
P.O. Box 25059
Santa Ana, CA 92799

ATTN: TECHNICAL PUBLICATIONS

FOLD