pdp 11

software
handbook

pdp11

software
handbook

CONTENTS

ix

CHAPTER 24 FOCAL

APPENDIX A GLOSSARY

PREFACE

This handbook describes the major operating systems, communications software, data management services, and programming languages available for the PDP-11 family of computers. It is intended for the system manager or programmer who needs a brief introduction to the range of PDP-11 software products and who is interested in determining which products best suit a particular processing environment.
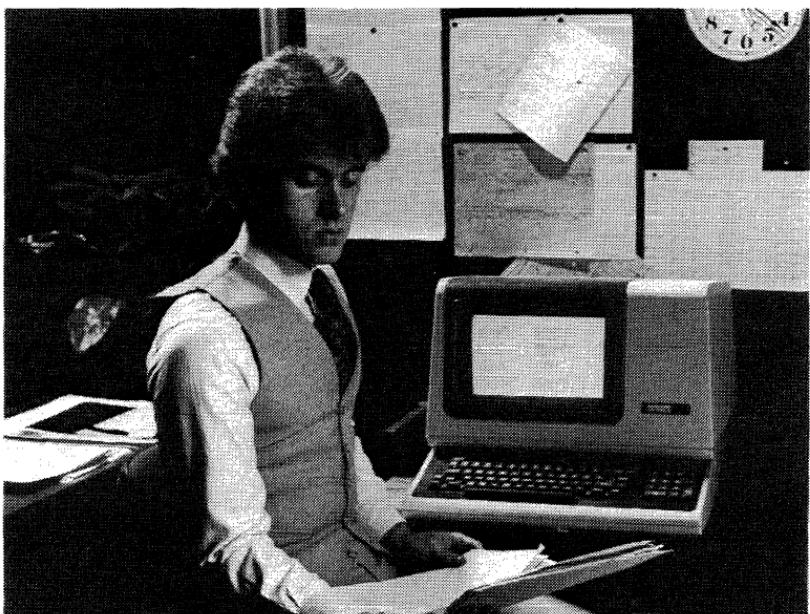
The technical descriptions provided in this handbook are not intended to be functional descriptions or operating procedures. This handbook is intended to be used in conjunction with the **PDP-11 Processor Handbooks** and the **PDP-11 Peripherals Handbook** to introduce the PDP-11 family's products. Complete technical information can be found in the set of software manuals that accompany each product.

Because DIGITAL is constantly developing new products and improving current ones, the information in this document is subject to change. In this connection, version numbers have been provided for each software product that specify what release of the software is being discussed. Users should consult their sales and software support representatives to obtain the latest information about a product's features and characteristics.

basic concepts

INTRODUCTION TO PDP-11 SOFTWARE

OVERVIEW

Upward compatibility is the star feature of PDP-11 software. This family of interactive software products has been designed to be compatible with DIGITAL's line of PDP-11 processors—ranging from board microcomputers to full multi-purpose computer systems. All of these processors are built upon a common architecture that uses a similar instruction set and input/output system; programs developed on one PDP-11 processor may therefore run on any other PDP-11 processor without major conversion.

This is the first of three chapters dealing with basic PDP-11 concepts.

FEATURE TOPICS

- Hardware/Software Systems
- Operating Systems
- Communications Software
- Data Management Services
- Languages and Language Processors
- PDP-11 Central Processors
- CPU/Operating System Comparative Chart

HARDWARE/SOFTWARE SYSTEMS

The PDP-11 computer family is a wide range of compatible processors complemented by a variety of peripheral devices, software, and services.

This handbook discusses the software that is available for the PDP-11 family of computers. Operating systems and programming languages may be available on either large or small hardware/software systems, but not both. For example, COBOL is available only on the larger systems. Other languages may be available on a wide range of systems, but may vary in characteristics signficant for a particular application. For example, FORTRAN IV is available on both large and small systems, but compilation speed may vary from system to system, depending on the hardware configuration.

The flexibility of PDP-11 hardware/software systems allows the user to select both the most appropriate hardware for a particular application's needs, and the operating system and languages that can serve immediate needs and still allow for possible growth.

DIGITAL offers a variety of operating systems, languages, data management services and communications software for the PDP-11 computer family. This handbook is structured around these major aspects of PDP-11 software:

BASIC CONCEPTS | Discusses the essential terms connected wuth PDP-11 operating systems and language processors.

OPERATING SYSTEMS | Discusses individual PDP-11 operating systems in detail.

DECnet | Discusses the family of PDP-11 software products used by the major operating systems to form communications networks.

DATA MANAGERS | Discusses the major PDP-11 data management services individually and in depth.

LANGUAGES | Discusses individual PDP-11 language options in detail.

It is assumed that the reader is familiar with operating system software and programming languages in general.

OPERATING SYSTEMS

An operating system not only provides access to the features of a processor in its size range, it also organizes a processor and peripher-

als into a useful tool for a certain range of applications. For example, the operating systems that run on the small processors are generally intended for dedicated applications. The operating systems that run on large processors are multi-purpose, and can provide a variety of services. The major operating systems to be discussed are:

RT-11   Real-Time Operating System for PDP-11 Proces-
                sors.

                A small, single-user foreground/background system
                that can support a real-time application job's
                execution in the foreground and an interactive or
                batch program development job in the background.

DSM-11   DIGITAL Standard Mumps Operating System for
                PDP-11 Processors.

                A small to large sized timesharing system that offers
                a unique fast access data storage and retrieval sys-
                tem for large data base processing.

RSTS/E   Resource-sharing Timesharing System/Extended
                Operating System for PDP-11 Processors.

                A moderate to large sized timesharing system that
                can support up to 63 concurrent jobs, which in-
                cludes interactive terminal user jobs, detached jobs,
                and batch processing.

RSX-11M   Real-Time Multiprogramming Executive Operating
                System for PDP-11 Processors.

                A small to moderate sized real-time multiprogram-
                ming system compatible with RSX-11D that can be
                generated for a wide range of application environ-
                ments — from small, dedicated systems to large,
                multi-purpose real-time application and program
                development systems.

RSX-11S   Real-Time Multiprogramming Executive Operating
                System for PDP-11 Processors.

                A small, execute-only member of the RSX-11 family
                for dedicated real-time multiprogramming applica-
                tions (requires a host RSX-11M or VMS system).

IAS   Interactive Application System for PDP-11 Proces-
                sors.

                A large, multi-user timesharing system, allowing
                real-time applications execution concurrent with
                timeshared interactive and batch processing.

3

TRAX   A dedicated high-volume transaction processing system offering real time and batch in a multi-user commercial environment.

Included in each chapter describing the operating systems are: a general description of the requirements for the system, the monitor/executive characteristics, the file structures and data handling facilities, the user interfaces, the programmed monitor services, the system utilities, and the language processors supported.

COMMUNICATIONS SOFTWARE
DIGITAL has provided the PDP-11 family of computers with a particularly useful range of communication software products. DECnet is a set of software tools that allows all DIGITAL systems to communicate programs and data among themselves. DECnet/11 software is designed specifically to connect he major PDP-11 operating systems together in a communications network.

DATA MANAGEMENT SERVICES
The PDP-11 family provides a full range of data management tools. The choice extends from the input/output support; to sequential and relative logical file/record support with a multi-key index sequential option; to a complete CODASYL-standard data base management system. The four main PDP-11 data managers discussed in this handbook are:

- SORT-11
- RMS-11
- DBMS
- DATATRIEVE-11

LANGUAGES AND LANGUAGE PROCESSORS
All PDP-11 operating systems offer a variety of programming language processors. A programming language is a tool that enables the user to state a problem that a computer can solve. A programming language is designed to be easily understood and manipulated by humans, while a language processor translates the problem into the machine's language.

In general, the language processors available to run under an operating system are commensurate with the kind of applications for which the operating system is designed. For example, a real-time application environment could be a laboratory in which a scientific programming language is useful for problem solving.

The programming languages discussed in this handbook are: APL, BASIC, BASIC-PLUS, BASIC-PLUS-2, COBOL, DIBOL, FORTRAN IV, FORTRAN IV-PLUS, MACRO, RPG II, FOCAL, and DSM.

Table 1-1 Language Table

MACRO	RT-11, RSX-11, RSX-11D, IAS, TRAX, VAX/VMS
FORTRAN IV	RT-11, RSX-11, RSX-11D, IAS, RSTS/E, VAX/VMS
FORTRAN IV-PLUS	RSX-11, RSX-11D, VAX/VMS
BASIC-11	RT-11, RSX-11, IAS
BASIC-PLUS-2	RSX-11, RSX-11D, IAS, TRAX, VAX/VMS
BASIC-PLUS	RSTS/E
RPG II	RSX-11M, RSTS/E
DIBOL	RT-11, RSTS/E
COBOL	RSX-11, RSX-11D, RSTS/E, TRAX, VAX/VMS

PDP-11 CENTRAL PROCESSORS

The PDP-11 family of processors is ordered in incremental steps of speed and size, and organized into four groups by their typical applications:

- PDP-11 microcomputers (LSI-based) for stable, programmable dedicated applications.
- PDP-11 minicomputers (11/04) for dedicated applications which may be planned for upward growth.
- PDP-11 system computers (11/34, 11/45) for multiple application tasks.
- PDP-11 high-throughput computers (11/55, 11/60, 11/70) for multi-purpose simultaneous application tasks.

This handbook uses these processor groups as the basis for discussing the range of hardware and software systems available in the PDP-11 family. An operating system that is designed to make maximum use of a particular processor is normally available on any processor in the same group. In addition, an operating system that runs on a particular

5

group of processors can often also run on processors in the group above or below it. As a general guide to size range, each processor group supports certain amounts of memory which will enable a system to possess specific operating capabilities:

In the following table, for example, the relationship between specific operating systems and the central processing units they may run on is charted. The features and capabilities listed there are intended as general guidelines and do not constitute strict rules for configuration.

LSI-11 BASED 11/04 11/34 11/45 11/55 11/60 11/70	**RT-11 Foreground/Background or Single-Job Operating System** 16K to 256K bytes of memory. In 16K bytes: Single-Job (SJ) operation; subset MACRO included; BASIC, FORTRAN IV, FOCAL as options. In 32K bytes: Foreground/Background (FB) or SJ operation; languages can support string operations, laboratory and graphics peripherals; full MACRO assembler included; multi-user BASIC available as option supporting as many as 4 users (under SJ monitor). MU BASIC supports as many as 8 users in 48K bytes under SJ monitor and as many as 4 in 56K bytes under FB monitor. Languages: MACRO included; FORTRAN IV; BASIC, MU BASIC, FOCAL, and APL are options
11/04 11/34 11/45 11/55 11/60 11/70	**DSM-11 DIGITAL Standard MUMPS-11 (Multi-User)** 64K to 1Mb of memory. 64K bytes will allow approximately 2 to 4 users to operate simultaneously. A maximum of 63 jobs may be supported depending on processor and partition size, supports many users accessing a common data base for easy applications development. Languages: DSM-11 included.

6

11/34 11/45 11/55 11/60 11/70	**RSTS/E General Purpose Timesharing System** 96K to 248K bytes of memory, or 96K to 3840K bytes on 11/70. Depending on disk and memory configuration, RSTS/E can support a maximum of 63 jobs. Languages: BASIC-PLUS and MACRO included; COBOL, BASIC-PLUS-2, FORTRAN IV, DIBOL, RPG II, DATATRIEVE-11, and APL are options.
LSI-11 BASED 11/04 11/34 11/45 11/55 11/60 11/70 VAX-11/780	**RSX-11S Execute-Only Real-Time Multi-Programming System** 16K to 3840K bytes of memory. 8K-byte system allows 4K for user tasks. 16K bytes required for on-line task loading or support for tasks written in FORTRAN. Languages: Program development on host RSX-11D/M, IAS, or VAX/VMS.
11/04 11/34 11/55 11/60 11/70	**RSX-11M Small-to-Moderate-Sized Real-Time Multi-Programming System** 32K to 248K bytes of memory or 32K to 3840K bytes on 11/70. At least 48K bytes are required for full MACRO support, concurrent program development and application tasks execution or memory management support. Error logging supported. Languages: MACRO included; FORTRAN IV and FORTRAN IV-PLUS, BASIC, BASIC-PLUS-2, COBOL, RPG, DATATRIEVE, and CORAL 66 are options.

11/45
11/55
11/60
11/70

IAS Multi-Purpose Multi-Programming System

128K to 248K bytes of memory or 128K to 3840K bytes on 11/70. Timeshared interactive and batch job processing with concurrent real-time applications execution. Depending on disk and memory configuration, as many as 10 interactive users can be supported on an 11/60; as many as 20 interactive users on an 11/70. Error logging supported.

Languages: MACRO included; FORTRAN IV, FORTRAN IV-PLUS, COBOL, BASIC, BASIC-PLUS-2, RPG, and CORAL 66 are options.

11/04
11/34
11/45
11/60
11/70

TRAX Dedicated Transaction Processing System

192K to 3840K bytes on 11/70. Interactive transaction processing characterized by sets of predefined procedures with multi-user protection built in.

Languages: COBOL, BASIC-PLUS-2, FORTRAN IV, APL, and DATATRIEVE are options.

OPERATING SYSTEMS

OVERVIEW
The success of the PDP-11 family of operating systems is largely attributable to its ability to handle many diverse data processing applications. For example, RT-11 provides a single-user environment with foreground/background processing; RSTS/E provides a multi-user environment with economical timesharing; RSX-11M provides a multi-user on-line environment with data collection and process control; DSM-11 (MUMPS) provides the same with data base information systems; and IAS provides a multi-user environment with simultaneous timesharing, real-time, and batch processing.

Basic concepts pertaining to the structure of these systems are presented in this chapter.

FEATURE TOPICS
- Components and Functions
- Processing Methods
- Data Management
- Data Storage and Transfer Mode
- I/O Devices and Physical Data Access Characteristics
- Physical Device Characteristics
- File Structures and Access Methods
- Directories and Directory Access Techniques
- File Protection/File Naming
- User Interfaces
  - Special Terminal Commands
  - I/O Commands
  - Monitor and Command Language Commands
- System Utilities
- Operating System Comparative Chart

INTRODUCTION

Operating systems have two basic functions: they provide services for application program development and act as an environment in which application programs run. The character that an operating system has, that is, the services and environment it supplies, is appropriate only for a certain range of program development and application requirements, in order to serve selected needs efficiently. Operating systems for the PDP-11 family of computers, however, share many similar program development techniques and processing environments.

COMPONENTS AND FUNCTIONS

An operating system is a collection of programs that organizes a set of hardware devices into a working unit that people can use. Figure 2-1 illustrates the relationship between users, the operating system, and the hardware. PDP-11 operating systems basically consist of two sets of software: the monitor (or executive) software and the system utilities.
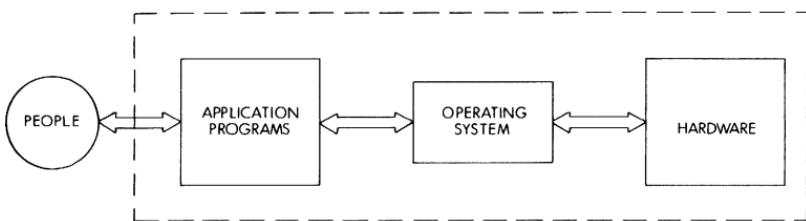
Figure 2-1 Computer System

An operating system monitor is an integrated set of routines that acts as the primary interface between the hardware and a program running on the system, and between the hardware and the people who use the system. The monitor's basic functions can be divided among the routines that provide the following services:

- device and data management
- user interface
- programmed processing services
- memory allocation
- processor time allocation

In general, a monitor can have two distinct operating components: a permanently resident portion and a transient portion. When a monitor

is loaded into memory and started, all of the monitor is resident in memory. Its first duty is to interface with the operator running the system. The monitor simply waits until an operator requests some service, and then performs that service. In general, these services include loading and starting programs, controlling program execution, modifying or retrieving system information, and setting system parameters. In most systems, these functions are serviced by transient portions of the monitor.

In some cases, when the monitor initiates another program's execution, the transient portion of the monitor can be over-written by the loaded program or swapped out. The permanently resident portion remains in memory to act on requests from the program. These generally include I/O services such as file management, device dependent operations, blocking and unblocking data, allocating storage space, and managing memory areas. In large systems, these services might also include inter-task communication and coordination, memory protection and parity checking, and task execution scheduling.

The dividing line between permanently resident and transient portions of the monitor, however, is not strictly based on user-interface functions and program-interface functions. In some systems, special monitor routines that service either the operator or programs might be stored on the system device, and are called into memory only as needed. The concern for space in small systems usually determines what portions of the monitor are resident at any time. The programmer or operator can control the size of the monitor, based on the needs for memory.

In some cases, the user can adjust the size of the monitor by eliminating features that are not needed in an application environment. RSTS/E, RSX-11M, and RSX-11S are examples of such systems. The RSX-11S system's monitor (called an executive) is always permanently resident when the system is operating. In this case, the user concerned with size can eliminate routines that perform unneeded operations. In general, however, all PDP-11 operating systems are designed to be flexible enough to operate in a relatively wide range of hardware environments.

System utilities are the individual programs that are run under control of the monitor to perform useful system-level operations such as source program assembly or compilation, object program linking, and file management.

System utility programs enhance the capabilities of an operating system by providing users with commonly performed general services. There are three classes of system utilities: those used solely or primar-

ily for program development, those used for file management, and those used to perform special system management functions.

Program development utilities include text editors, assemblers and compilers, linkers, program librarians, and debuggers. File management utilities include file copy, transfer, and deletion programs, file format translators, and media verification and clean-up programs. System management utilities vary from system to system, depending on the purpose and functions the system serves. Some examples are system information programs, user accounting programs, and error logging and on-line diagnostic programs.

PROCESSING METHODS

The basic distinctions among operating systems are in the processing methods they use to execute programs. The distinctions to be discussed here are:

- single-user vs. multi-user
- single-job vs. foreground/background
- foreground/background vs. multi-programmming
- timesharing vs. event-driven multi-programming

A single-user operating system views demands upon its resources as emanating from a single source. It has only to manage the resources based on these demands. As a result, these systems do not require account numbers to access the system or data files. RT-11 is a single-user operating system.

A multi-user operating system receives demands for its resources from many different individuals. The system must manage its resources based on these demands. For example, several users may want sole access to the same device at the same time. The system must control access to these devices. In addition, the individuals may be using the system for different purposes, implying that some privacy must be maintained. As an effect, a multi-user system normally has an account system to manage different user's files. The IAS, RSTS/E, and RSX-11M systems are all multi-user systems, and all provide device allocation control and file accounts. In the case of the IAS, RSTS/E and systems, the file account structure is also used to keep track of the amounts of system resources an individual uses. Furthermore, the RSTS/E system extends privacy by protecting individual users at a system level from the effects of any other users of the system.

An RT-11 system can operate in two modes: as a single-job system, or as a foreground/background system. In a foreground/background system, memory for user programs is divided into two separate re-

gions. The foreground region is occupied by a program requiring fast response to its demands and priority on all resources while it is processing; for example, a real-time application program. The background region is available for a low-priority, preemptable program; for example, a compiler.

Two independent programs, therefore, can reside in memory, one in the foreground region and one in the background region. The foreground program is given priority and executes until it relinquishes control to the background program. The background program is allowed to execute until the foreground program again requires control. The two programs effectively share the resources of the system. When the foreground program is idle, the system does not go unused. Yet, when the foreground program requires service, it is immediately ready to execute. I/O operations are processed independently of the requesting job to ensure that the processor is used efficiently as well as to enable fast response to all I/O interrupts.

The basis of foreground/background processing is the sharing of a system's resources between two tasks. An extension of foreground/background processing is multiprogramming. In multiprogrammed processing, many jobs, instead of only two, compete for the system's resources. While it is still true that only one program can have control of the CPU at a time, concurrent execution of several tasks is achieved because other system resources, particularly I/O device operations, can execute in parallel. While one task is waiting for an I/O operation to complete, for example, another task can have control of the CPU.

The RSX-11 family of operating systems employs multiprogrammed processing based on a priority-ordered queue of programs demanding system resources. In this case, memory is divided into several regions called partitions, and all tasks loaded in the partitions can execute in parallel. Program execution, as in the RT-11 foreground/background system, is event-driven. That is, a program retains control of the CPU until it declares a significant event—normally meaning that it can no longer run, either because it has finished processing, or because it is waiting for another operation to occur. When a significant event is declared, the RSX-11 executive gives control of the CPU to the highest priority task ready to execute. Furthermore, a high-priority task can interrupt a lower-priority task if it requires immediate service.

The RSTS/E and MUMPS-11 systems also perform concurrent execution of many independent jobs. RSTS/E and MUMPS-11, however, process jobs on a timesharing rather than an event-driven basis, since

13

this is best suited for an interactive processing environment. Each job is guaranteed a certain amount of CPU time (a time slice), and jobs receive time one after another, in a round-robin fashion based on job priority levels set by the system. The system itself manages timesharing processing to obtain the best overall response depending generally on whether jobs are compute-bound or I/O-bound. The system manager or privileged users can also specify the minimum guaranteed time for a particular job when it gets service, as well as modifying its priority.

The IAS system effectively combines event-driven and timeshared processing in order to handle both real-time processing needs and interactive timesharing needs. In IAS, I/O tasks and any user-designated real-time tasks are assigned high priorities and receive service on an event-driven basis. All other tasks run at lower priorities on a timeshared basis, using any CPU time remaining after real-time, high-priority tasks have been serviced.

DATA MANAGEMENT
Digital computers deal with binary information only. The way in which people interpret and manipulate the binary information is called data management.

This section describes PDP-11 software data management structures and techniques, from the physical storage and transfer level to the logical organization and processing level. This includes:

- ASCII and binary storage formats — how binary data can be interpreted
- physical and logical data structures — the difference between how data storage devices operate and how people use them
- file structures — how physical units of data are logically organized for easy reference
- file directories — how files are located and retrieved
- file protection — how files are protected from unauthorized users
- file naming conventions — how files are identified

Physical and Logical Units of Data
Physical units of data are the elements which digital computer devices use to store, transfer and retrieve binary information. A bit (binary digit) is the smallest unit of data that computer systems handle. An example of a bit is the magnetic core used in some processor memories that is polarized in one direction to represent the binary number 0 and in the opposite direction to represent the binary number 1.

14

In PDP-11 computers, a byte is the smallest memory-addressable unit of data. A byte consists of eight binary bits. An ASCII character code can be stored in one byte. Two bytes constitute a 16-bit word. A word is the largest memory-addressable unit of data. Some machine instructions are stored in one word.

The smallest unit of data that an I/O peripheral device can transfer is called its physical record. The size of a physical record is usually fixed and depends on the type of device being referenced. For example, a card reader can read and transfer 80 bytes of information, stored on an 80-column punched card. The card reader's physical record length is 80 bytes.

A block is the name for the physical record of a mass storage device such as disk, DECtape or magnetic tape. An RK05 disk block consists of 512 contiguous bytes. Its physical record length is 512 bytes.

Physical blocks can be grouped into a collection called a device or a physical volume. This collection generally has a size equal to the capacity of the device medium. The term physical volume is generally used with removable media, such as disk packs or magnetic tape.

Logical units of data are the elements manipulated by people and user programs to store, transfer and retrieve information. The information has logical characteristics, for example, data type (alphabetic, decimal, etc.) and size. The logical characteristics are not device dependent; they are determined by the people using the system.

A field is the smallest logical unit of data. For example, the field on a punched card used to contain a person's name is a logical unit of data. It can have any length determined arbitrarily by the programmer who defines the field.

A logical record is a collection of fields treated as a unit. It can contain any logically related information, in any one of several data types, and it can be any user-determined length. Its characteristics are not device dependent, but can be physically defined. For example, a logical record can occupy several blocks, or it can reside in a single block, or several logical records can reside in a single block. Its characteristics are determined by the programmer.

A file is a logical collection of data that occupies one or more blocks on a mass storage device such as a disk, DECtape or magnetic tape. A file is a system-recognized logical unit of data. Its characteristics can be determined by the system or the programmer.

A file can be a collection of logical records treated as a unit. An example is a employee file which contains one logical record in the file for each employee. Each record contains an employee's name and ad-

dress and other pertinent information. If the logical record length is 50 bytes and there are 200 employees, the complete employee file could be stored in 20 512-byte blocks. Depending on the file structure used in the system, the blocks could be scattered over the disk, or could be located one after the other.

A logical volume is a collection of files that reside on a single disk or DECtape. It is the logical equivalent of a physical device unit (a physical volume) consisting of physical records, such as a disk pack. The files on a volume may have no specific relationship other than their residence on the same magnetic medium. In some cases, however, the files on a volume may all belong to the same user of the system.

Figure 2-2 illustrates some of the kinds of physical and logical units of data that PDP-11 computer systems handle.
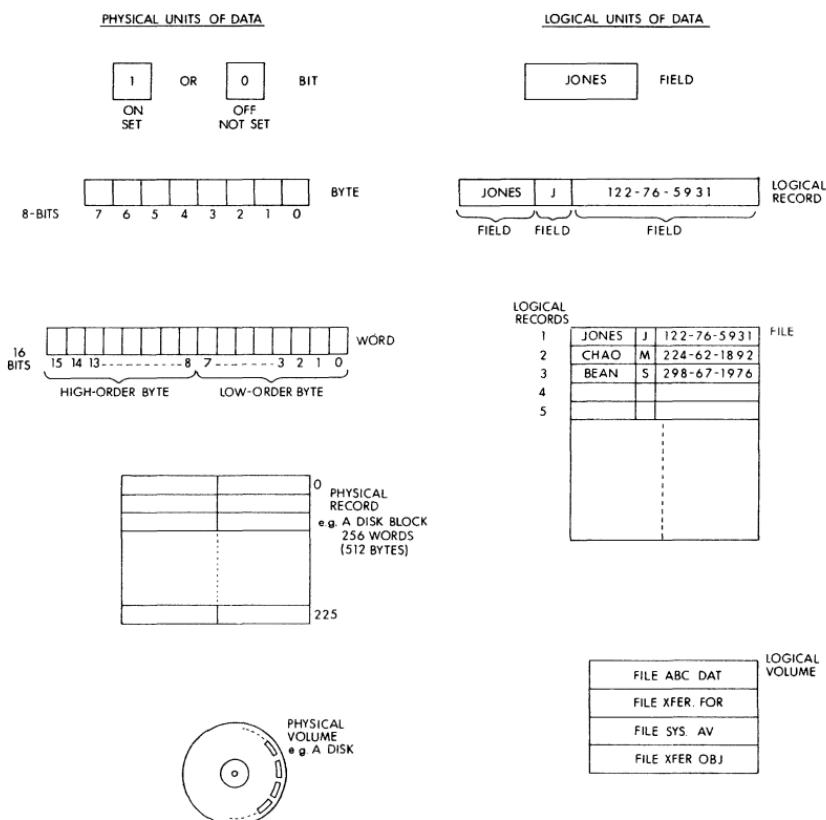
Figure 2-2 Physical and Logical Data Storage

16

Data Storage and Transfer Modes

All PDP-11 operating systems use two basic methods of data storage: ASCII and binary. Data stored in ASCII format conform to the American National Standard Code for Information Interchange, in which each character is represented by a 7-bit code. The 7-bit code occupies the low-order seven bits of an 8-bit byte. Depending on the operating system's storage techniques, the high-order bit may be used for parity checking and special formatting, or it may be ignored. Text files such as source programs are examples of data stored in ASCII format.

Binary storage always uses all eight bits of a byte to store information. The significance of any bit varies depending on the kind of information to be stored. Machine instructions, 2's complement integer data, and floating point numeric data are some examples of data stored in binary format.

Figure 2-3 illustrates the way in which binary data can be interpreted as either ASCII data or machine instructions. The figure shows two examples of a word of storage containing the same sequence of bits, interpreted first as two ASCII characters and second as a machine instruction. When a word of storage is interpreted as two ASCII characters, the binary digits are grouped into octal digits in a bytewise manner. Each byte is grouped into three octal digits. The low-order two octal digits contain three binary digits. The high-order octal digit contains two binary digits. When a word of storage is interpreted as a machine instruction, the binary digits are grouped into six octal digits in a wordwise manner. Proceeding from the low-order binary digit, each group of three binary digits is interpreted as an octal digit. The single remaining high-order binary digit is interpreted as an octal digit.
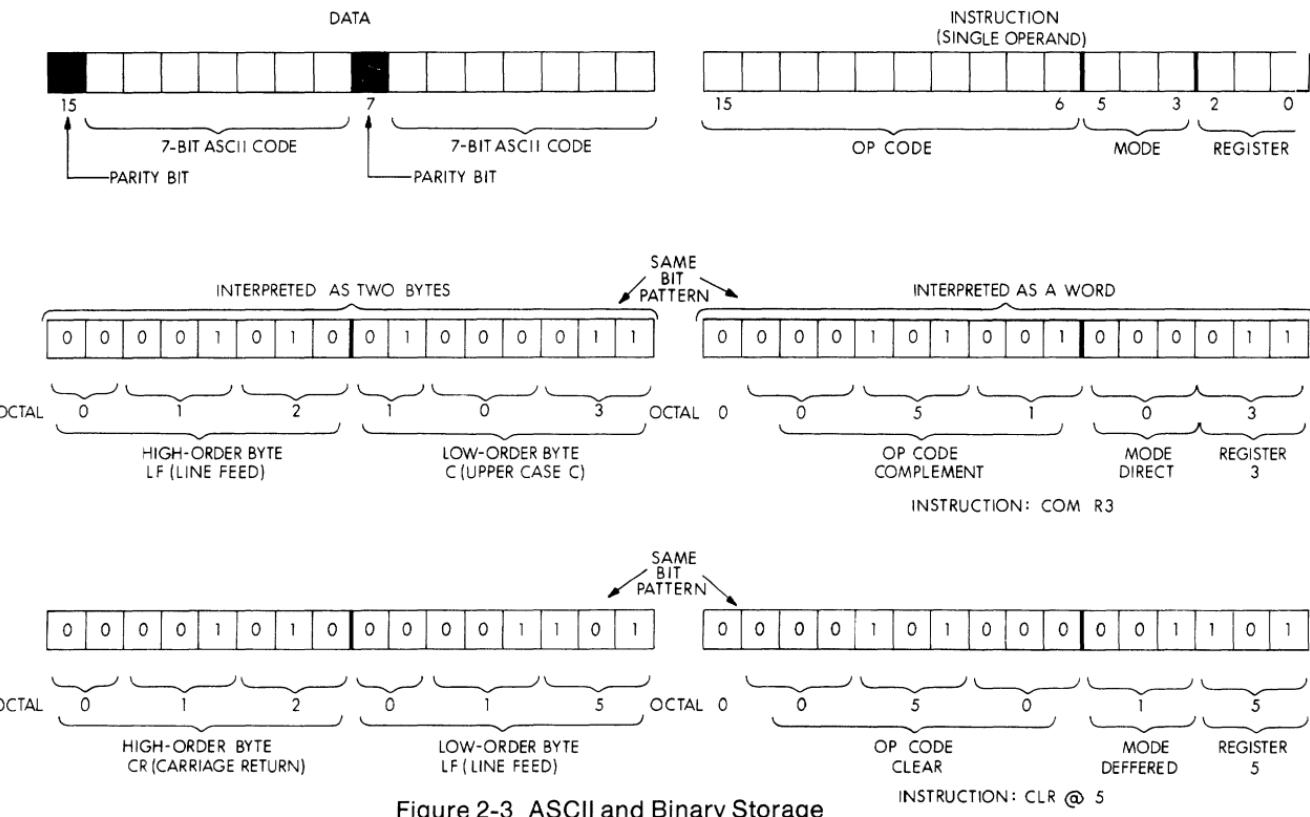
Figure 2-3 ASCII and Binary Storage

In large, sophisticated systems such as RSTS/E, RSX-11, and IAS, the way in which data are stored on the byte or bit level is rarely a concern of the application programmer. The operating system handles all data storage and transfer operations. In smaller systems such as RT-11, the programmer can become involved in data storage formats. A particular application may require the selection of a particular storage format.

The data storage format is related to the way in which data are transferred in an I/O operation.
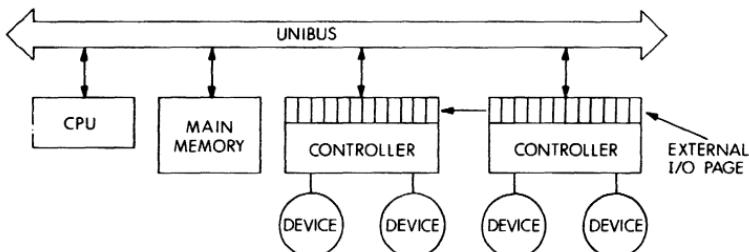
Formatting can also be applied at a higher level to define the type of data file being processed. In the RT-11 system, there are four types of binary files; each type signifies that a special interpretation applies to the kind of binary data stored. For example, a memory image file is an exact picture of what memory will look like when the file is loaded to be executed. A relocatable image file, however, is an executable program image whose instructions have been linked as if the base address were zero. When the file is loaded for execution, the system has to change all the instructions according to the offset from base address zero.

I/O Devices and Physical Data Access Characteristics

In a PDP-11 computer system, data moves from external storage devices into memory, from memory into the CPU registers, and out again. The window from external devices to memory and the CPU is called the I/O page. Each external I/O device in a computing system has an external page address assigned to it. Figure 2-4 illustrates the data movement path in a PDP-11 computing system.

PHYSICAL ORGANIZATION

FROM THE PROGRAM'S VIEWPOINT
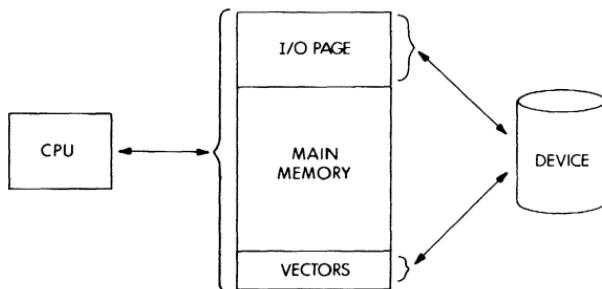
Figure 2-4 Memory and I/O Devices

Although all external devices transmit and receive data through the UNIBUS, devices differ in their ability to store, retrieve or transfer data. Almost all PDP-11 operating systems provide device independence between devices that have similar characteristics and, where possible, between differing devices in situations where the data manipulation operations are functionally identical. Primarily, PDP-11 operating systems differentiate between:

- file-structured and non-file-structured devices
- block-replaceable and non-block-replaceable devices

Terminals, card readers, paper tape readers, paper tape punches and line printers are examples of devices that do not provide any means to store or retrieve physical records selectively. They can transfer data only in the sequence in which they occur physically.
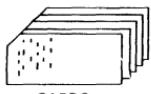
In contrast, mass storage devices such as disk, DECtape, floppy disk, magnetic tape and cassette have the ability to store and retrieve physical records selectively. For example, an operating system can select a file from among many logical collections of data stored on the medium.

Mass storage devices are called file-structured devices since a file, consisting of a group of physical records, can be stored on and retrieved from the device. Terminals, card readers, paper tape readers/punches and line printers are called non-file structured devices because they do not have the ability selectively to read or write the physical records constituting a file.

Finally, mass storage devices differ in their ability to read and write physical records. Disk and DECtape devices are block-replaceable devices because a given block can be read or written without accessing or disturbing all the other blocks on the medium. Magnetic tape and cassette are not block-replaceable devices.

A device's physical data access characteristics determine which data transfer methods are possible for that device. Non-file structured devices allow sequential read or write operations only. Non-block replaceable devices allow sequential or random read operations, but allow sequential write operations only. Block-replaceable devices allow both sequential and random read or write operations. Figure 2-5 summarizes the read/write capabilities of each category of I/O device.
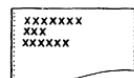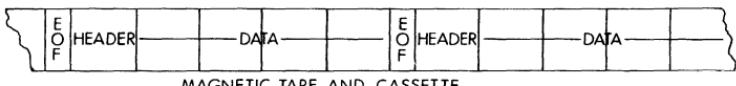
CARDS   PAPER TAPE   LINE PRINTER

SEQUENTIAL READ OR WRITE ONLY
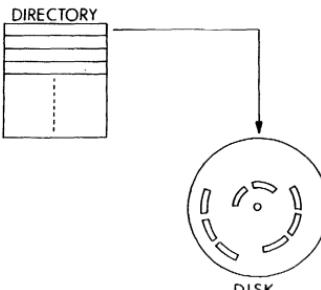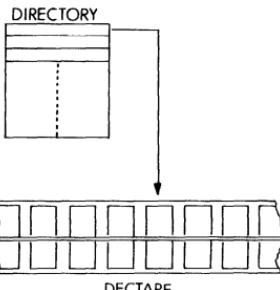
NON-FILE-STRUCTURED

FILE-STRUCTURED

MAGNETIC TAPE AND CASSETTE

READ AND WRITE SEQUENTIAL

NON-BLOCK REPLACEABLE

BLOCK REPLACEABLE

DIRECTORY   DIRECTORY

DECTAPE   DISK

READ & WRITE SEQUENTIAL OR RANDOM

File Protection

Master File and User File Directories form the basis for file access protection in multi-user systems. Unauthorized users cannot access a file unless they know the account under which it is stored and can obtain access to that account. Account systems and file access protection techniques are related.

Multi-user systems identify the individuals who use the system by account numbers called User Identification Codes (UIC). The system manager normally gives a user an account number under which the user can log in to the system and obtain access to its services. In general, a UIC consists of two numbers: the first number is used to identify a group of users, the second number is used to uniquely identify an individual user in the group.

In RSTS/E systems, an individual file can be protected against read access or write access where distinctions are made on the basis of the UIC account number under which a file is stored. For example, a file can be read protected against all users who are not in the same account group and write protected against all users except the owner.

The RSX-11/IAS file system provides a protection scheme for both volumes and files. It is possible to specify protection attributes for an entire volume as well as for the files in the volume. A file or an entire volume can be read-, write-, extend- or delete-protected. Distinctions are made on the basis of account number, where the system recognizes four groups of users: privileged system users, owner, owner's group, and all others.

File Naming

The most common way users communicate their desire to process data is through file specifications. A file specification uniquely identifies and locates any logical collection of data which is on-line to a computer system.

A compiler, for example, needs to know the name and location of the source program file that it is to compile; it also needs to know the name that the user wants to use for the output object program and listing files it produces. Most PDP-11 operating systems share the same basic format for input and output file specifications.

In the RT-11 system, a file specification consists of the name of the device on which the file resides, a file name, and a file name extension in the following format:

dev:filnam.ext

The colon is part of the device name, separating it from the file name

on the right. The period is part of the file name extension, separating it from the file name on the left.

PDP-11 operating systems use the same device names for the devices they can access. A device name consists of a two-letter mnemonic and, for multiple devices of the same kind, a one-digit number indicating the device unit number. For example, the name "DK1:" is used to identify the RK11 disk drive unit number 1. The name "DP0:" identifies the RP11 disk drive unit number 0.

In the RT-11 system, a file name is a 1- to 6-character alphanumeric name designated by the user. For example, "SYMBOL", "RL12", and "NORT4" are examples of file names. In the RSTS/E and RSX-11M systems, a file name can be up to nine characters long.

A file name extension is a 1- to 3-character alphanumeric name preceeded by a period. The extension can either be assigned by the user or, if unspecified, assigned by the system. The extension generally indicates the format of a file. System-assigned and recognized extensions make it easy for the user and the system to distinguish between different forms of a file. For example, a file having the extension ".FOR" is recognized by the FORTRAN compiler as a source program written in FORTRAN. A file with the extension ".OBJ" is recognized by the Linker as an object program, a legal input file. When in the process of compiling and linking a FORTRAN program, the user has only to specify a file name to the compiler and Linker. The FORTRAN compiler will compile the file whose extension is ".FOR" and produce a file with the same file name whose extension is ".OBJ". The Linker will link the file whose extension is ".OBJ".

In multi-user systems such as RSTS/E and RSX-11M, a distinction must be made between files stored under various accounts on a device. Two different users can have a file named "REFER.OBJ" on a disk. In these systems, therefore, a file specification has an additional component to identify the user file directory or account under which the file is stored. The basic file specification is expanded to use the following format:

> dev:[ufd]filnam.ext

The account number or user file directory is always enclosed in brackets. It consists of the project or group number followed by a comma and a programmer or user number. For example, "[12,4]" is an example of an account or user file directory.

RSTS/E systems also include a protection code as part of the file specification, to indicate the protection that the file receives. A complete RSTS/E file specification could be:

DK1:[200,210]BINFOR.DAT<60>

RSX-11 systems extend the basic file specification format by adding a version number identification after the file name extension. For example, when a file is first created using the editor, it is assigned a version number of 1. If the file is subsequently opened for editing, the editor keeps the first version for backup and creates a new file using the same file specification, but with a version number of 2. A complete RSX-11 file specification could be:

DP0:[15,7]PREPT.MAC;1

In most cases, the user does not have to issue a complete file specification. The PDP-11 operating systems use default values when a portion of a file specification is not supplied. The file name extension defaults, as indicated previously, depend on the kind of operation being performed.

The device name, if omitted, is normally assumed to be the system device. For example, the file specification "FILE.DAT" is equivalent to the specification "DK0:FILE.DAT", if the system device is RK11 drive unit 0. Most systems also allow the user to omit the unit number. If omitted, the unit number is assumed to be unit number 0. For example, DT: is equivalent to DT0:; it signifies DECtape drive unit 0.

If the account number is omitted from the file specification, the system assumes that it is the same as the UIC under which the user logged in or under which the operation is being performed. For example; if the user logged in under UIC 200,200 and issues a file specification "DK3: SAMPL.DAT", it is interpreted as "DK3:[200,200]SAMPL.DAT".

If the version number is omitted from an RSX-11/IAS file specification, the system assumes that the file specification refers to the latest version of the file.

For references to file-structured devices, a file specification must include a file name. The device mnemonics, however, are also used to refer to non-file structured devices. In this case, a file name is irrelevant. For example, an operation to read through a deck of cards and print the information on a line printer is issued in most systems as follows:

#LP:=CR:

The # indicates that an input/output command is being issued; it is printed on the terminal by the program that requests the I/O command. The user types the command LP:=CR:. The = separates the input file specification on the right from the output file specification on the left. The device name LP: signifies that the line printer is to be used as the output device, and the device name CR: signifies that the card

reader is to be used as the input device. A file name, if used, would be ignored, since the system can not symbolically reference data on non-file structured devices.

In addition to relying on defaults in the file specification, the user can also put an asterisk in place of a file name, file name extension, account number, or version number to indicate a class of files. The asterisk convention, also called the wildcard convention, is commonly used in PDP-11 operating systems when performing the same operation on related files. For example, the file specification DP1: [2,1]PROG.* refers to all files on DP1: under account [2,1] with a file name PROG and any extension. The file specification DK:[*,*]FILE.SAV refers to the files under all accounts on RK11 drive unit 0 named FILE.SAV. The file specification DT:*.OBJ refers to all files on the DECtape mounted on drive unit 0 that have the extension .OBJ.

USER INTERFACES

A user interface refers to both the software that passes information between an operator and a system and the language that a system and an operator use to communicate. In the latter sense, a user interface consists of commands and messages. Commands are the instructions that the user types on a terminal keyboard (or gives to a batch processor) to tell the system what to do. Messages are the text that a system prints on a terminal (or line printer) that tells the operator what is going on; for example, prompting messages, announcements and error messages. This section discusses commands, the portion of the user interface that tells the system what to do, and prompting messages, the messages the system prints when it is ready to receive commands.

There are basically four types of commands used in PDP-11 operating systems:

- monitor or command language commands — used to request services from the system as a whole

- I/O commands — used to direct any kind of I/O operation (often a part of monitor commands)

- special terminal commands — these use keys on a terminal for special functions

- system program commands — commands used in system programs that perform operations relevant only for the individual program

Since system program commands are relevant only for individual system programs, and not for operating systems in general, this section discusses monitor and command language commands, I/O commands and special terminal commands only.

Special Terminal Commands

Special terminal commands are a set of keys or key combinations that, when typed on a terminal, are used to perform special functions. For example, a user normally types the carriage return key at the end of an input command string to send the command to the system, which responds immediately by performing a carriage return and line feed on the terminal. The key labeled RUBOUT or DELETE is used to delete the last character typed on the input line.

The most significant special terminal commands are those used with the key labeled CTRL (control). When the CTRL key is held down (like the shift key) and another key is typed, a control character is sent to the system to indicate that an operation is to be performed.

For example, a line currently being entered (whether as part of a command or as text) will be ignored by the system by typing a CTRL/U combination (produced by holding down the CTRL key and typing a U key). The user can then enter a new input line. The CTRL/U function is the same as typing successive RUBOUT keys to the beginning of a line. CTRL/U is standard on PDP-11 operating systems.

Another example is the CTRL/O function. If, during the printing of a long message or a listing on the terminal, the user types a CTRL/O, the teleprinter output will stop. The program printing the output, however, will still continue. The user can type a CTRL/O again to resume output. CTRL/O is a standard function on PDP-11 operating systems.

Physical Device Characteristics
and Logical Data Organizations

One of the most important services an operating system provides is the mapping of physical device characteristics into logical data organizations. Users do not have to write the software needed to handle input and output to all standard peripheral devices, since appropriate routines are supplied with the operating system.

There are generally two sets of routines provided in any operating system, depending on its complexity:

- device drivers or handlers
- file management services

Device drivers and handlers can perform the following operations to relieve the user of the burden of I/O services, file management, overlapping I/O considerations and device dependence:

- drive I/O devices
- provide device independence
- block and unblock data records for devices, if necessary

- allocate or deallocate storage space on the device
- manage memory buffers

These routines may exist in the system as part of the monitor or executive, as in RT-11, MUMPS-11, RSTS/E, RSX-11M or RSX-11S, or they may be provided as separate tasks, as in IAS.

An operating system can also provide a uniform set of file management services. For example, the RT-11 system provides file management services through the part of the monitor called the User Service Routine (USR). The User Service Routine provides support for the RT-11 file structure. USR loads device handlers, opens files for read/write operations, and closes, deletes and renames files.

In summary, an operating system maps physical device characteristics into logical file organizations by providing routines to drive I/O devices and to interface with user programs. Figure 2-6 illustrates the transition between the user interface routines and the I/O devices.
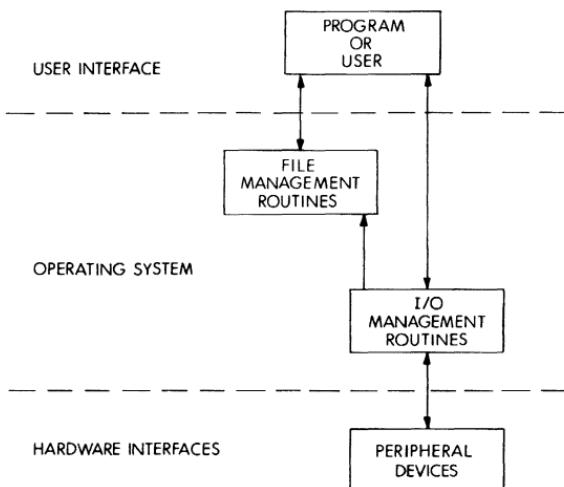
Figure 2-6 Device Control and File Management Services

As an example of the mapping of physical characteristics into logical organizations, the RSX-11 and IAS systems' device drivers and handlers and file management services allow the user application program to treat all file-structured devices in the same manner. All of these devices appear to the user program to be organized into files consisting of consecutive 512-byte blocks which are numbered start-

ing from block one of the file to the last block of the file. In reality, the blocks may be scattered over the device and, in some cases, the device's actual physical record length may not be 512 bytes.

In RSX-11/IAS terminology, the actual physical records on the device (for example, the sectors on a disk) are called physical blocks. At the device driver or handler level, the system maps these physical blocks into logical blocks. Logical blocks are numbered in the same relative way that physical blocks are numbered, starting sequentially at block zero as the first block on the device to the last block on the device. At the user interface level, the operating system maps logical blocks into virtual blocks. Virtual block numbers become file relative values, while logical block numbers are volume relative values.

Figure 2-7 illustrates the mapping between physical, logical and virtual blocks in RSX-11 and IAS systems. The figure shows two disk device types which have different physical record lengths. In this case, the blocks constituting a file are scattered over the disk. The file is a total of 5 blocks long. At the logical block level, the operating system views the file as a set of non-contiguous blocks. At the virtual block level, the user software views the file as a set of contiguous, sequentially numbered blocks.
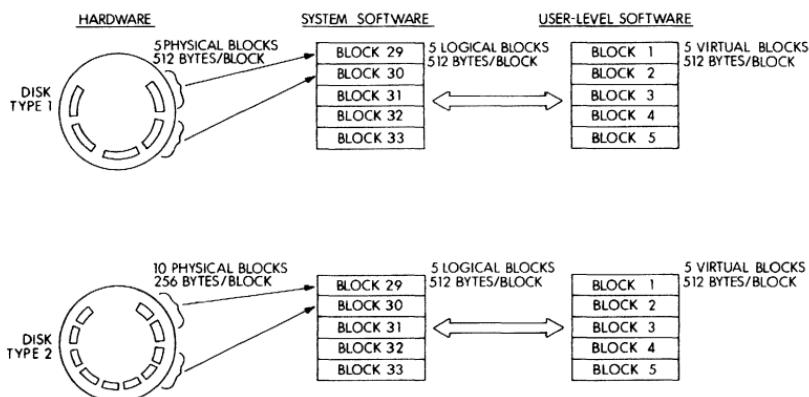
Figure 2-7 Physical, Logical and Virtual Blocks

File Structures and Access Methods

A file structure is a method of organizing logical records into files. It describes the relative physical locations of the blocks constituting a file. The file structure or structures that a particular operating system employs is a product of the way in which the system views the particular I/O devices and the kinds of data processing requirements the system fulfills.

29

File structure is important because a file can be effective in an application only if it meets specific requirements involving:

SIZE Growth of the file may require a change in the file structure or repositioning of the file.

ACTIVITY The need to access many different records in a file or frequently access the same file influences data retrieval efficiency.

VOLATILITY The number of additions or deletions made to a file may affect the access efficiency.

An access method is a set of rules for selecting logical records from a file. The simplest access method is sequential: each record is processed in the order in which it appears. Another common access method is direct access: any record can be named for the access. The non-block replaceable devices, such as paper tape and magnetic tape, can only be processed sequentially. The block-replaceable devices, such as disk and DECtape, can be processed by either access method, but direct access takes greatest advantage of the device characteristics.

PDP-11 operating systems provide a variety of file structures and access methods appropriate to their processing services. All PDP-11 file structures are, however, based on some form of the following basic file structures:

FILE STRUCTURE	ACCESS METHODS
Linked	Sequential
Contiguous	Sequential or Direct Access
Mapped	Sequential or Direct Access

Linked files are a self-expanding series of blocks which are not physically adjacent to one another on the device. The operating system records data blocks for a linked file by skipping several blocks between each recording. The system then has enough time to process one block while the medium moves to the next block to be used for recording. In order to connect the blocks, each block contains a pointer to the next block of the file. Figure 2-8 shows the format of a linked file.
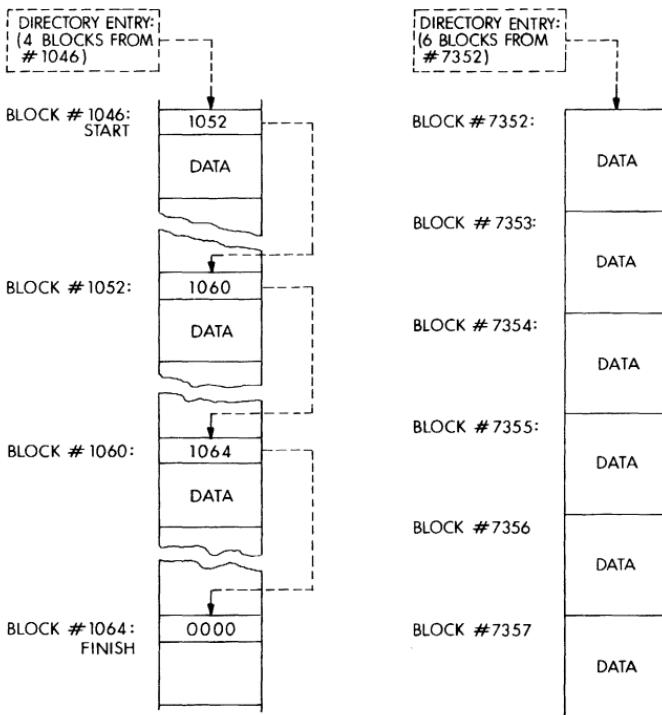
Figure 2-8 Linked and Contiguous File Structures

Linked file structure is especially suited for sequential processing where the final size of the file is not known. It readily allows later extension, since the user can add more blocks in the same way the file was created. In this way, linked files make efficient use of storage space. Linked files can also be joined together easily.

The blocks of contiguous files are physically adjacent on the recording medium. This format is especially suited for random (direct access) processing, since the order of the blocks is not relevant to the order in which the data is processed. The system can readily determine the physical location of a block without reference to any other blocks in the file. Figure 2-8 also shows the format of a contiguous file.

Mapped files are virtually contiguous files; they appear to the user program to be directly addressable sets of adjacent blocks. The files may not, however, actually occupy physically contiguous blocks on the device. The blocks can be scattered anywhere on the device. Separate information, called a file header block, is maintained to identify all the

31

blocks constituting a file. This method provides an efficient use of storage space and allows files to be extended easily, while still maintaining a uniform program interface. Figure 2-9 illustrates a mapped file format.
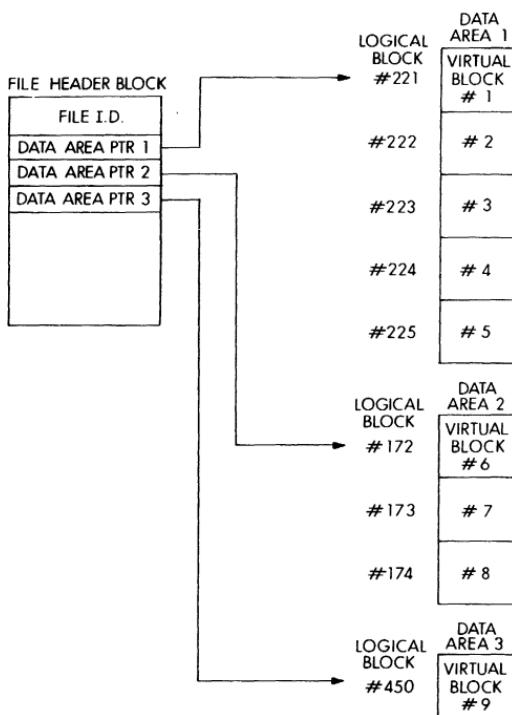
Figure 2-9 Mapped File Structure (Non-Contiguous File)

If desired, a mapped file can be created as a contiguous file to ensure the fastest random accessing, in which case it is both virtually and physically contiguous.

The basic file structures discussed above can be modified or combined to extend the features of each type for special-purpose logical processing methods. Some examples are indexed files and global array files.

Indexed files are actually two contiguous files. One file acts as an ordered map of a second file containing the target data. The index portion or map contains either an ordered list of key data selected from the target data records or pointers to data records in the second

32

file, or both. The target data records can be processed in the order of the index portion, or the target data records can be selected by searching through the index portion for the key data identifying the records. These methods of logically processing the target data are called indexed sequential access and random access by key, respectively.

Global DSM-11 (MUMPS) array files display a special form of linked file structure. The arrays themselves are a logical tree-structured organization consisting of one or more subscripted levels of elements. All elements on a particular subscripting level are stored in a single chain of linked blocks. At the end of each block in the chain is a pointer to the next block in the chain. The levels of the array (all the block chains) are linked together through pointers in the first block of each chain. This file structure ensures that the time it takes to access any element of the array is minimal.

Directories and Directory Access Techniques

Just as file structure and access methods are required to locate records within files, directory structures and directory access techniques are required to locate files within volumes.

A directory is a system-maintained structure used to organize a volume into files. It allows the user to locate files without specifying the physical addresses of the files. It is a direct access method applied to the volume to locate files.

RT-11 supports the simplest kind of file directory. When disk and tape media are initialized for use, the system creates a directory on the device. Each time a file is created, an entry is made in the directory that identifies the name of the file, its location on the device, and its length. When access to the file is requested thereafter, the system examines the directory to find out where the file is actually located. The system can access the file quickly without having to examine the entire device.

In multi-user systems such as RSTS/E, IAS, and RSX-11M, two different kinds of directories are used to enable the system to differentiate between files belonging to different users. They are the Master File Directory and the User File Directories. These directories are maintained as files themselves, stored on the volume for which they provide a directory.

A Master File Directory (MFD) is a directory file containing the names of all the possible users of a particular device. A User File Directory (UFD) is a directory file containing the names of all the files created by a particular user on a device. The system first checks the Master File

33

Directory to locate the User File Directory for the particular user, and then checks the User File Directory to locate the file. Figure 2-10 illustrates the use of the Master and User File Directories.
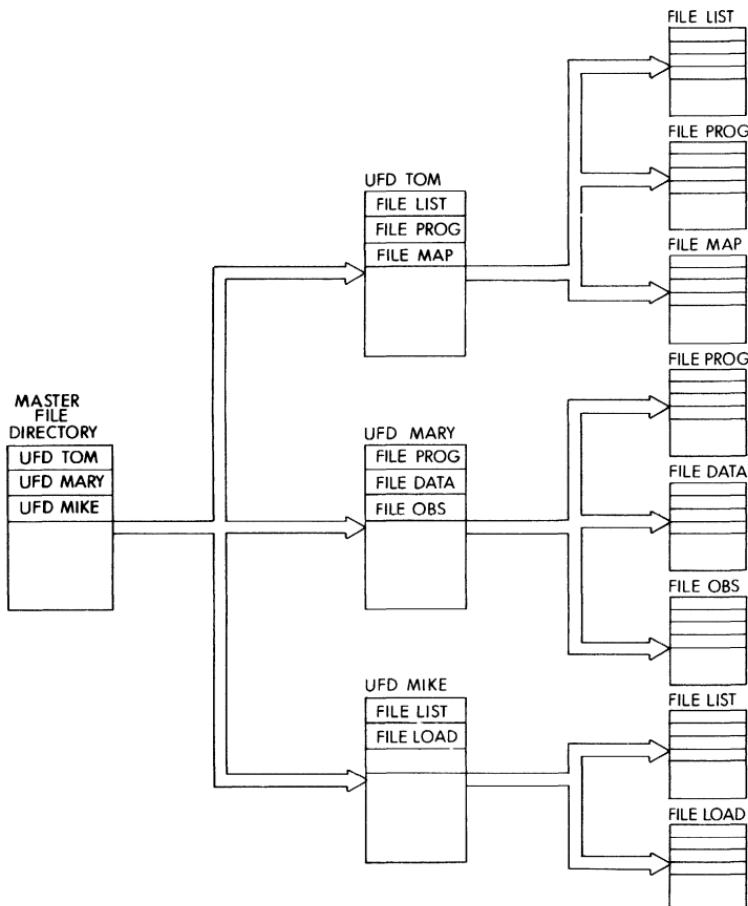
Figure 2-10 Master and User File Directories

RSTS/E creates an MFD on each disk when it is initialized. On all disks except the system disk, the MFD catalogs other user accounts on the disk. The MFD on the system disk has a special purpose, since it maintains a catalog of the accounts under which users can log in, in addition to the user accounts on the disk. A UFD exists on each disk for each account under which files are created. A UFD for an account

is not created until a file is created by the user under that account. DECtape devices are considered to be single-user devices, and the RSTS/E system maintains only a single directory on DECtapes.

The RSX-11M and IAS systems also employ MFD and UFD files on file-structured volumes. As with RSTS/E systems, the number of directory files required depends on the number of users of the volume. For single-user volumes, only an MFD is needed. For multiple-user volumes, an MFD and one UFD for each user are required. An MFD is automatically created when the volume is initialized for use. A UFD is created only by the system manager or priviledged user.

File access in RSX-11M and IAS systems, however, is not limited to using the MFD and UFD files. The basis of file access using the MFD and UFD in these systems is a special file called the index file. Like the MFD, an index file is created on each volume when it is initialized. Files in these systems are mapped files, and the Index File contains the file header for each file stored on the volume, including the MFD. Each file is uniquely identified by a file ID. A file header contains the file's ID and the physical location (logical record number) of each series of contiguous blocks constituting a file. By knowing a file's ID and searching through the index file, a program can locate a file (and any block within the file) without having to use the MFD and UFD directories. Figure 2-11 illustrates how an index file is used to access files on a volume.
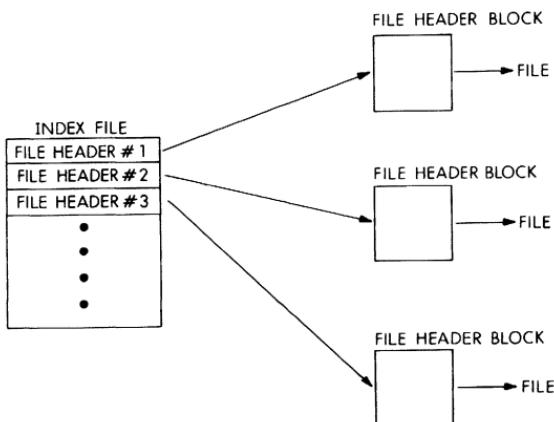
Figure 2-11 Index File Access

All of these operating systems also permit non-block replaceable media, such as cassettes and magnetic tape, to be given a file structure. These media have no directory because a directory could not be updated and replaced. Instead, each file is preceded by one or more header records which contain the directory information such as the file's name. The operating system can locate a file by scanning the volume and reading each file header until the correct one is found. The file can then be processed by a sequential access method.

I/O Commands

As mentioned above, users communicate their intentions to process data files by issuing I/O commands consisting of at least one file specification. Normally, the I/O commands used in a system are standard throughout that system; in addition, most PDP-11 operating systems share the same basic I/O command string format.

For example, in RT-11 systems, the monitor includes a command string interpreter routine that parses and validates I/O command strings. The command string interpreter routine is used both by the monitor and the system programs to obtain a definition from the user of the input file or files to be processed and a definition of the output file or files to be created. User-written programs can also call the command string interpreter to obtain I/O specifications from the operator at a terminal.

A standard I/O command string consists basically of one or more input and/or output file specifications. In all systems except IAS, an I/O command string uses the following general format:

> filespec=filespec

where filespec is a file specification and the equal sign (=) represents a character (usually equal sign or less-than sign) that separates an input file specification on the right from an output file specification on the left. If there is more than one input file specification or output file specification, they are separated from each other by commas. For example, if there are two output file specifications and three input file specifications:

> filespec,filespec=filespec,filespec,filespec

If the program requesting an I/O command string does not need either an input or output file specification, the equal sign (or less-than sign) is not present in the I/O command string.

As an example, the user can run the RT-11 operating system's Linker system utility to link one or more object program files and produce an executable program file and a load map. The I/O command issued to the Linker could be:

*DK:RESTOR.SAV,DK1:RESTOR.MAP=DK:RESTOR.OBJ/B:500

Where:

*	Is the prompting character printed by the Linker program indicating that it wants an I/O command string. After it is printed, the user types the remaining characters on the line.
DK:RESTOR.SAV	Is the name of the executable program file to be created. It will be stored on the disk cartridge mounted on the RK11 drive unit zero.
DK1:RESTOR.MAP	Is the name of the load map file to be created. It will be stored on the disk mounted on RK11 drive unit 1.
DK:RESTOR.OBJ	Is the name of the object module (input file) to be used to create RESTOR.SAV.
/B:500	Is a command string switch indicating that the RESTOR.SAV program is to be linked with its starting address at location 500.

Command string switches are simply ways of appending qualifying information to an I/O command string. The switches used vary from program to program. They are not usually required in an I/O command string, since most programs assume default values for any switch.

Monitor and Command Language Commands

The primary system/user interface is provided in PDP-11 operating systems by either monitor software or special command language interface programs that run under the monitor. The monitor software and command languages allow the user to request the system to set system parameters, load and run programs, and control program execution.

An input command line consists of the command name (an English word that describes the operation to be performed) followed by a space and a command argument. For example, the command to run a program is the word RUN followed by the name of the file containing the program. If the command name is long, it can usually be abbreviated. For example, the command to set the system's date to August 15, 1984 could be DA 15-AUG-84. The system could also accept "DA 27-AUG-75". A command input line is normally terminated by typing the

carriage return key on the console keyboard, although in some systems the key labeled ALTMODE is also used. Typing the carriage return key (or ALTMODE key) tells the system that the command line is ready to be processed.

In the RT-11 system, a monitor component called the keyboard monitor performs the function of notifying the user that the monitor is ready for input by printing a period at the left margin. The user enters a command string on the same line following the period, and terminates the command string by typing the carriage return key.

In the RSTS/E system, the monitor and the BASIC-PLUS language processor share the responsibility for interpreting commands. The system prints the word READY on the terminal and then spaces down two lines. The user then enters a command on the new line and terminates the line by typing the carriage return key. There are three types of commands the user can issue: RSTS/E monitor commands, such as RUN, ASSIGN, or RENAME; BASIC-PLUS immediate mode statements, such as PRINT, INPUT, or OPEN; or Concise Command Language commands.

A Concise Command Language (CCL) command is used to run and pass arguments automatically to designated programs stored in the system library. The programs can be system utilities supplied with the operating system, or can be user-written console routine programs that perform special application operations. For example, RSTS/E includes a system utility called PIP that performs a variety of file manipulation operations, including a file copy operation. The dialog normally used to run the PIP utility and issue a copy command is:

READY	The system prints READY.
RUN $PIP	The user runs PIP.
PIP Vnnn	PIP announces itself.
*FILEB.DAT=FILEA.DAT	PIP prints an asterisk to request
*↑C	an I/O command and the user issues a copy command. PIP prints an asterisk, indicating that the operation was performed and that it is ready to accept another command; the user types a CTRL/C to abort PIP and return to the monitor.
READY	The system prints READY.

The standard RSTS/E system also includes a CCL command named PIP that can be issued to perform any of PIP's normal functions. If

used as a CCL command, the dialog to perform the same copy opera-
tion is:

READY	The system prints READY.
PIP FILEB.DAT=FILEA.DAT	The user issues the CCL com-mand and the argument that tells PIP to copy FILEA.DAT to FI-LEB.DAT.
READY	The system prints READY.

A CCL command not only provides an easy-to-use command inter-
face, it can also provide protection from unauthorized use of certain
programs. For example, if a particular program performs several op-
erations, some of which should not be available to unauthorized users,
the system manager can prevent those users from issuing the RUN
command to run the program, but can allow them to perform safe
operations by using CCL commands.

In the RSX-11 systems, a command interface called the Monitor
Console Routine (MCR) allows the user to perform system level opera-
tions. When MCR is activated, it prints the characters MCR> on the
terminal. The user enters a command on the same line as the prompt,
and terminates the line with a carriage return or an ALTMODE. If the
line is terminated with a carriage return, MCR prints a prompt and is
ready to receive another command. If the line is terminated with an
ALTMODE, MCR does not reactivate. To reactivate MCR at a terminal,
the user types a CTRL/C.

There are two kinds of commands that MCR accepts: general user
commands and privileged user commands. General user commands
provide system information, run programs, and mount and dismount
devices. Privileged user commands control system operation and set
system parameters.

To run a system utility, the user can type the utility's name in response
to an MCR prompt. When the utility is loaded, it prints a prompt to
request a command string. The user can then enter a command string.
When it completes the operation, the user can enter another com-
mand or type CTRL/Z to terminate the program. For example, to run
the PIP utility program:

```
MCR>PIP
PIP>command string
PIP>↑Z
MCR>
```

If the user wants to issue only one command to the utility, the user can type the command string on the same line with the MCR request to run the utility. For example:

MCR>PIP command string
MCR>

In the IAS system, system/user interfaces are provided by programs called Command Language Interpreters (CLI). The standard system includes a CLI called the Program Development System. When it is activated, it prints the prompt PDS> on the terminal to indicate it is ready to accept and process commands. The user has several options for command string formats. If the user is uncertain about a command's syntax, the user can simply type the command name and a carriage return. PDS will ask the user to supply each portion of the command string individually. Users can write their own Command Language Interpreters.

PROGRAMMED SYSTEM SERVICES

All PDP-11 operating systems provide access to their services through requests that programs or tasks can issue during execution.

The RT-11 system provides a variety of programmed requests. There are programmed requests that perform file manipulation, data transfer and other system services such as loading device handlers, setting a mark time for asynchronous routines, suspending a program, and calling the Command String Interpreter. Monitor services are requested through macro instructions in assembly language programs, or through calls to the system library in FORTRAN programs. The basis of the programmed requests in RT-11 are the Emulator Trap (EMT) instructions. When an EMT is executed, control is passed to the monitor, which extracts appropriate information from the EMT instruction and executes the operation requested. When the operation is performed, the monitor returns control to the program.

In the RSTS/E system, users writing BASIC-PLUS programs have access to the monitor's services through system function calls. The function calls allow a program to control terminal operation, to read and write core common strings, and to issue calls to the system file processor. The file processor calls enable a program to set program run priority and privileges, scan a file specification, assign devices, set terminal characteristics, and perform directory operations. A system function is called in a manner similar to normal BASIC-PLUS language calls. When the function operation is performed, the program continues execution.

The RSX-11 and IAS executives include programmed services called

executive directives. Directives can be executed in MACRO programs using system macro calls provided with the system. The FORTRAN or BASIC-PLUS-2 programmer can invoke directives through a subroutine call. The system uses only the EMT 377 instruction to implement directives. The directives allow the program to obtain system information, control task execution, declare signficant events, and perform I/O operations. After the directive is processed, control is normally returned to the instruction following the EMT.

The RSX-11M and IAS systems also include programmed file control services. The file control services enable the programmer to perform record-oriented and block-oriented I/O operations. These services are provided as macro calls.

The IAS system includes a special set of programmed services called Timesharing Control Primitives. These are available for use by any program that is written as a Command Language Interpreter (CLI). They enable a CLI to start or control execution of other timesharing tasks, and share access to devices with other timesharing users.

SYSTEM UTILITIES

PDP-11 operating systems provide, in general, three kinds of system utility programs: program development utilities, file management utilities, and special system management utilities.

Most PDP-11 operating systems include the following kinds of program development utilities:

Text Editor An editor is used for on-line interactive creation and editing of source programs or data files. An editor uses several sets of commands that search for character strings, insert, move or delete characters or lines, and insert, move, delete or append whole buffers of data. Although a text editor is designed for interactive use, it can also usually be run under a batch processor if the operating system supports batch processing.

Assembler An assembler accepts a source program written in PDP-11 machine language and produces an object module as output.

Linker A linker is a program that accepts relocatable object programs created by an assembler or compiler and produces an executable program module. Some linkers provide facilities for overlaid program segments to enable a large program to execute in a small memory area.

Librarian   A librarian is a program that enables a programmer to create, update, modify, list and maintain library files. A library file is an object module (or modules) that is used several times in a program, used by more than one program, or routines that are related and simply gathered together to incorporate easily into a program.

Debugger   A debugger is a program which enables a user to troubleshoot program errors dynamically through a terminal keyboard. It is normally linked with a program and runs as part of the program.

Some of the file management utilities available on many operating systems include:

PIP   The Peripheral Interchange Program (PIP) is a general-purpose file utility package for both the general user and programmer and the system manager. PIP normally handles all files with the operating systems standard data formats. In general, the program transfers data files from any device in the system to any other device in the system. PIP can also delete or rename any existing file. Some operating systems include special file management operations in the PIP utility, such as directory listings, device initialization and formatting, and account creation.

FILEX   The File Exchange program is a special-purpose file transfer utility similar in operation to PIP. It provides the ability to copy files stored in one kind of format to another format. This enables a user to create data on one system in a special format and then transfer the data to a device in a format that another system can read.

DUMP   DUMP displays all or selected portions of a file on a terminal or line printer. In general, DUMP enables the user to inspect the file in any of three modes: ASCII, byte, and octal. In ASCII mode, the content of each byte is printed as an ASCII character. In byte mode, the content of each byte is printed as an octal value. In octal mode, the content of each word is printed as an octal value.

VERIFY   In general, a VERIFY program checks the readability and validity of data on a file-structured device.

42

Most system management utilities included in an operating system are dependent on the function the operating system serves. The RSX-11M, IAS, and RSTS/E systems provide special system management utilities. For example, RSX-11M and RSTS/E include system error logging and report programs. RSTS/E, and IAS and include user accounting programs.

RT-11	RSTS/E	RSX-11M
Is • Foreground/ background (multi-tasking) • Single user • Sensor based • Operating on small CPUs • Protected environment • Easy to install and use • High real-time throughput • Batch processing • Highly reliable • Full development facilities	**Is** • General purpose timesharing • High performance timesharing BASIC • Interactive environment • Multi-language • Batch processing • Basis of most commercial applications	**Is** • Real-time processing • Sensor based • Data base management • Multi-user development • Building block operating system for: - Communications - Commercial applications - Turn-key applications
Is not • Transaction processing • Record management • Data base management	**Is not** • Real-time • High volume transaction processing • Block mode application terminals	**Is not** • Batch processing • Timesharing • Protected environment
	Includes Data Mgr./ Utilities • RMS-11 • SORT-11 • DATATRIEVE-11 • DMS-500	**Includes Data Mgr./ Utilities** • RMS-11 • DBMS • DATATRIEVE-11 • SORT-11
Languages • BASIC-11 • FORTRAN IV • MACRO-11 • FOCAL • APL	**Languages** • BASIC-PLUS • BASIC-PLUS-2 • COBOL • FORTRAN IV • MACRO-11 • RPG II • DIBOL-11	**Languages** • COBOL • FORTRAN IV • FORTRAN IV-PLUS • MACRO-11 • BASIC-11 • BASIC-PLUS-2 • RPG II

Figure 2-12 Operating System Chart

IAS	DSM-11	TRAX-11

Is
• Real-time
• Timesharing
• Batch processing
• Data base management
• Multi-function
• Multi-language
• Extensible executive
• High RSX/VAX/ TRAX compatibility
• Protected environment
	Is
	• Interactive, high-productivity applications development for data base management system
	• Highly approachable
	• Integrated language/command environment
	• Powerful language structure for text processing
	• Large numer of terminals—up to 80
		Is
		• High volume transaction processing
		• Batch processing
		• Protected environment
		• Application development tools:
		  - Debug utility
		  - Terminal screen language
		• Distributed functionality
		• RSX/VAX compatibility
		• Easy systems design

Is not • High capacity (dedicated) timesharing • High capacity (dedicated) real-time • Operating on small CPUs	**Is not** • General timesharing • Real-time • Computational or batch • Multi-language	**Is not** • Timesharing • Sensor based • For smaller CPUs • Large scale batch (IBM)
Includes Data Mgr./ Utilities • DBMS • RMS-11 • DATATRIEVE-11 • SORT-11		**Includes Data Mgr./ Utilities** • RMS-11 • DATATRIEVE-11 • SORT-11
Languages • BASIC-11 • BASIC-PLUS-2 • COBOL • FORTRAN IV • MACRO-11	**Languages** • DSM-11	**Languages** • COBOL • BASIC-PLUS-2 • MACRO-11

Figure 2-12 Operating Systems, cont.

LANGUAGE PROCESSORS

OVERVIEW

DIGITAL's high-level languages let you move freely among operating systems. These languages span the breadth of the PDP-11 operating systems, and conform to the industry standards that have been established. In some cases, more than one version of a particular language is available on a single operating system, each optimized to meet particular requirements.

The basic concepts behind language assemblers and compilers and the common functions and features of PDP-11 language processors are presented in this chapter.

FEATURE TOPICS

- Language Translation Systems Definition
- Assemblers
- Compilers
- Modularity
- Assembly Language Routines
- Library Routines
- Libraries and Object Time Systems
- Assemblers and the Language Compilers
- Program Development Facilities
- COBOL Compiler
- Incremental Compilers
- PDP-11 Assemblers and Language Compilers

LANGUAGE TRANSLATION SYSTEMS

A programming language is a system of symbols and syntax that can be used to describe a procedure that a computer can execute. A language processor is a program that translates one programming language into another. A language processor reads a program written in a language easily understood by people and translates it into a program written in the binary language of a digital computer. The program that the processor reads is called the source program. The program that the processor writes is called the object program.

ASSEMBLERS

An assembler is a language processor written for a particular digital computer. The source language it translates is called assembly language. There is a one-to-one correspondence between most of the mnemonics used as the assembly language operators and the binary instructions of the computer. Some exceptions are macro calls and assembler directives.

During the language translation process, an assembler performs a number of error checking operations. When an error is detected, the assembler notes the error and attempts to continue processing. At the end of processing, the assembler produces an error listing showing all the occurrences of errors, with substantial messages to the programmer. In addition to an error listing, the programmer can obtain an assembly listing in any of several formats and a symbol table listing. In addition, some assemblers can provide a cross reference listing for all symbols used in the program.

Most assemblers produce an object program by making one or more passes over the source program (reading the original source code several times). The resultant object program is in relocatable binary format. That is, the first instruction appears to be located in the first word of processor memory. Since in most cases the program is not to be loaded into the bottom of memory, the object program must be linked to the proper memory addresses before it can be executed.

The linking program is provided as a standard program development utility with an operating system. Figure 3-1 illustrates the fundamental steps in producing an executable program from assembly source code.

SOURCE CODE

```
┌─────────────┐
│  ASSEMBLER  │
│     OR      │
│  COMPILER   │
└─────────────┘
```

OBJECT CODE

```
┌─────────────┐
│   LINKER    │
│   UTILITY   │
└─────────────┘
```

EXECUTABLE CODE
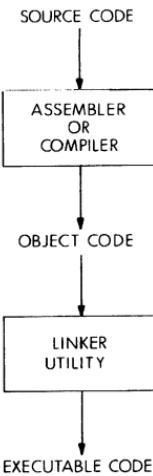
Figure 3-1 Fundamental Assembly or Compilation Procedure

COMPILERS

A compiler is a language processor written to translate a higher-level language whose structure, syntax, and symbols are independent of any particular machine. The higher-level language operators most often do not correspond directly to binary instructions. It is the compiler's job to provide algorithms for their translation.

Most compilers do not translate the source code until the entire source program is read at least once. The translation of the source code into object code takes place during several passes over the source code or, if only one pass over the original source code is made, during several phases of the compilation process. This allows the compiler to examine the code it produces as a whole to eliminate unnecessary instructions (code optimization). In addition, the compiler can perform many levels of error checking and it can produce several kinds of compilation listings, including source code listings, code generation listings, and diagnostics.

An incremental compiler is a compiler that immediately translates source statements into an internal format. Each source statement is translated (and therefore can be executed) before the following statement is translated. Although this method of source translation does not enable possible object code optimization, it allows the compiler to provide program development services not possible in multi-pass or multi-phase compilers. For example, a syntax error detected in a

source statement can be reported to the programmer immediately, and the programmer can correct the statement before proceding.

One significant difference between a general compiler and an incremental compiler is the characteristics of the resulting object program. The object code produced by the general compiler requires a separate step of linking before it can be executed, as shown in Figure 3-1. This approach enables the programmer to combine several object programs into one executable program. This provides several advantages:

Modularity

A source program may be too large to be compiled successfully as a single unit, but, if divided into modular sections, can be compiled as several separate units. The separate sections can be combined at the object level to produce the resultant program. In addition, programs that are extremely complex can be divided into several sections so that they can be easily manipulated, debugged or modified. A change in one module of the program will only require recompilation of that section.

Assembly Language Routines

The compiler's object code can be combined with the object code produced by the operating system's assembler. Algorithms which are most easily written in assembly language, such as user-defined I/O processing, can be incorporated into a program written primarily in a higher-level language.

Library Routines

Libraries of commonly used routines and functions written in either assembly or the higher-level language can be maintained in object format. These routines can be selectively included in the resultant program by the linking utility. This not only eliminates repetitive source coding and associated errors, it also decreases the size of the source and object programs.

The object code produced by an incremental compiler does not require an intermediate step of linking before it can be executed. The incremental compiler actually serves two purposes: it translates the source code into object code and it provides the environment in which to execute the object code. That is, the steps of source code translation, linking, and execution are all provided by the translator. Figure 3-2 illustrates this type of translator operation.
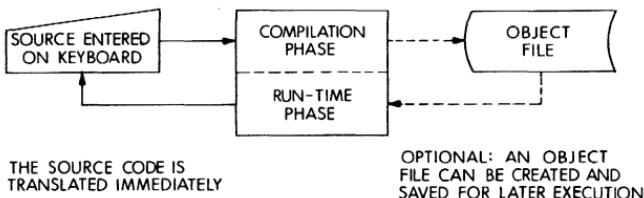
Figure 3-2 Fundamental Incremental Compiler Operation

PROGRAM DEVELOPMENT FACILITIES

A complete language translation system requires facilities for creating and editing source programs, linking object programs into executable programs, and debugging programs. Most PDP-11 operating systems provide an Editor utility for source program creation and editing, and a Librarian utility for library file creation. Operating systems also provide a Linker utility to link and combine object modules produced by multi-pass compilers and assemblers. Finally, operating systems also include debugging utilities.

Some of these facilities may or may not be incorporated into the language translator program itself. For example, an incremental compiler may include an editing facility as part of the language translation code. This allows the programmer to edit the program interactively as it is being compiled and executed.

LIBRARIES AND OBJECT TIME SYSTEMS

Also included in most language translation systems is a library of the most commonly used functions and routines. The system library is generally a part of the language processor's Object Time System (OTS).

A multi-pass or multi-phase compiler does not usually generate all of the machine language code required by the program at run time. Common sequences of code required by the program can be maintained in the OTS file. The compiler then flags the places where the desired sequences are needed. The linker utility, during its pass over the object program, selects those sequences from the OTS file and incorporates them into the executable program module.

An incremental compiler may also have an OTS. In this case, however, the OTS is generally part of the run-time code of the translator. When the object code is executed by the incremental compiler's run-time code, the OTS is used to provide common library code sequences.

51

PDP-11 ASSEMBLERS AND THE LANGUAGE COMPILERS
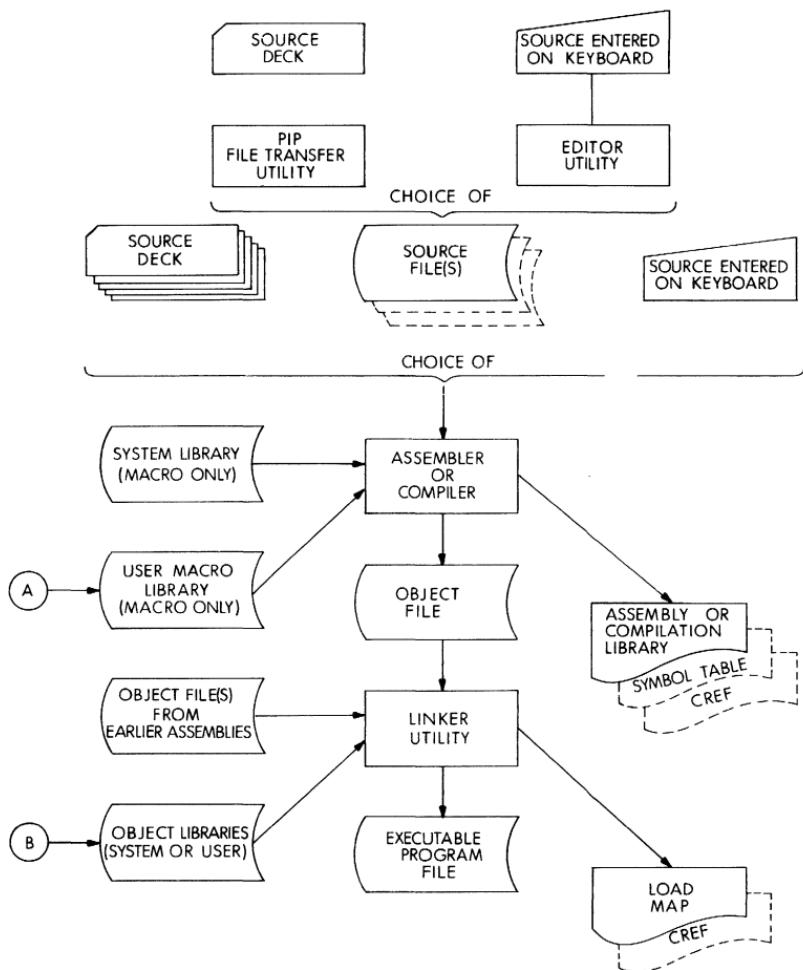
With three exceptions, all the operating systems described in this handbook include the MACRO assembly language. RSTS/E does not include MACRO, but supports it fully; DSM-11 and TRAX do not support any assembly language processor.

Two FORTRAN IV compilers are available: FORTRAN IV and FORTRAN IV-PLUS. FORTRAN IV is available on all the operating systems described in this handbook except MUMPS-11. FORTRAN IV-PLUS is available on the RSX-11 and IAS operating systems.

The MACRO assembler, FORTRAN IV compiler and FORTRAN IV-PLUS compiler display the same external operating characteristics. In general, they accept source code from any valid input device and produce an object file on any valid file-structured device. If the input device is a file-structured device, the assembler or compiler can accept several source files. If desired, an assembly or compilation listing can also be produced as output, either as a file or on a line printer or terminal. MACRO can also generate both a symbol table listing and a Cross Reference Listing (CREF) if desired.

As shown in Figure 3-3, there are several methods for creating sources. A source program can be punched on cards if a card reader is available, or it can, in some cases, be entered directly on the terminal. The common method is to create a file on a file-structured device. The file can be created from a deck of punched cards, using the PIP file transfer utility to copy it onto disk or DECtape. The file can also be created on a terminal, using the operating system's editor utility to store it on disk or DECtape.

Figure 3-3 Building an Executable User Program
Written in MACRO or FORTRAN

In addition to source program files, the MACRO assembler accepts source library files as input. The operating system provides a system library for MACRO containing the macro definitions for the system's monitor calls or executive directives. The assembler selects those macro definitions required by the source program from the system library file.

In RSX-11, IAS and RSTS/E systems, the MACRO assembler can also accept a user-created macro library as input. The sources for the

user-defined macro libraries are created in the same manner as normal source programs. The operating system's librarian utility program is used to create the library files. Figure 3-4 illustrates this procedure.
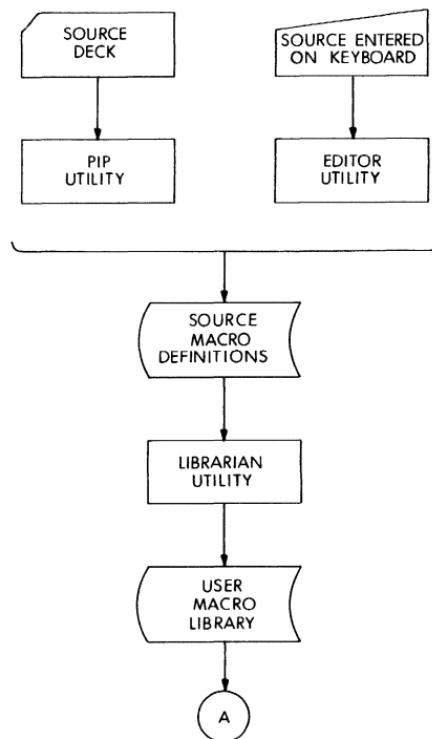
Figure 3-4 Building User MACRO Libraries

Once the assembler or compiler produces an object file, the object file can be linked by the linker utility. The linker can accept several object files as input. In addition, when linking object files produces a FORTRAN compiler, the linker accepts the FORTRAN system object library for the given compiler as input. The linker automatically selects the required routines from the library.

Users can also create their own object library files. The source code is created in the same manner as normal source programs. The librarian utility is used to build the library file. Figure 3-5 illustrates the procedure.

SOURCE CODE
AS IN FIGURE 3-3

ASSEMBLER
OR
COMPILER

OBJECT
FILE

LISTING

LIBRARIAN
UTILITY

OBJECT
LIBRARY
FILE

B
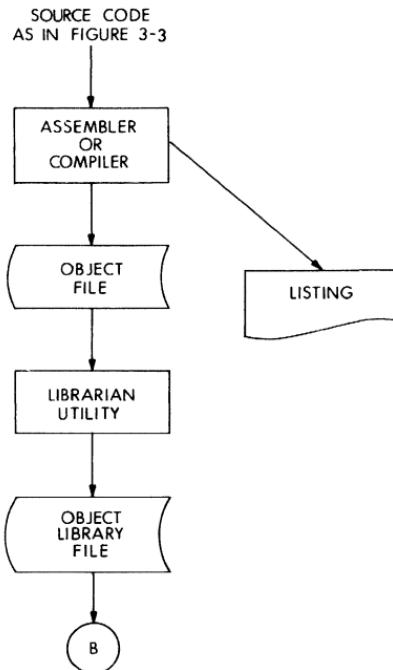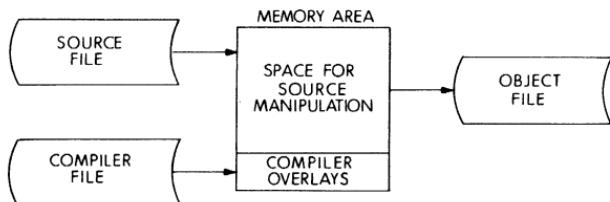
Figure 3-5   Building User Object Libraries from
Sources Written in MACRO or FORTRAN

PDP-11 assemblers and compilers differ in their internal operation. The MACRO assembler is a two-pass assembler. It makes a first pass over the source input to collect the symbol references, expand macros and produce preliminary object code. A second pass is made to resolve symbol references and produce the completed object code and listings.

The FORTRAN IV compiler is a multiple-phase compiler. Instead of making multiple passes over the source program, it reads the source program once and manipulates the source code in memory. The compiler operates in multiple phases. An overlay is read into memory for each phase of the compilation process. This method enables the compiler to compile relatively large programs very quickly.

The FORTRAN IV-PLUS compiler is a multiple-pass compiler. It reads the source program several times, using a work file to build the object code. The work file is deleted when the compilation process is complete. Figure 3-6 illustrates the compilation methods of the two FORTRAN compilers.

MULTI-PHASE COMPILATION

MULTI-PASS COMPILATION
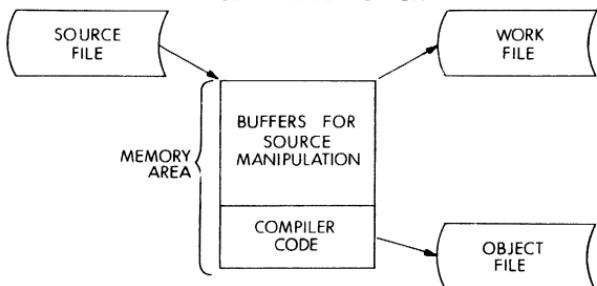
Figure 3-6 Compilation Methods

The FORTRAN IV-PLUS compiler's work file not only makes it possible to compile relatively large programs, but it also allows the compiler to examine thoroughly the object code it produces. The compiler refines the code it produced during the initial compilation to ensure that the program, when executed, will run as fast as possible. Therefore, although the disk accesses required to read and write a work file decrease the speed of the compiler, the work file enables the FORTRAN IV-PLUS compiler to produce highly optimized code.

PDP-11 COBOL COMPILER
The PDP-11 COBOL compiler can accept source input from cards, from the terminal, or from a file created using cards or an interactive editor. The compiler produces both an object file and an overlay description language (ODL) file which describes the overlay tree structure associated with the generated object file.

The compiler is properly termed a multi-phase, multi-pass compiler. The compiler makes several passes over the source program, using a work file to contain various tables built during the compilation process. Utilization of the work file permits compilation of large COBOL programs consisting of approximately 6000 or more source lines. The last pass of the compilation process produces the object and ODL files. Figure 3-7 illustrates the COBOL compiler's external operating characteristics.

Once the compiler produces an object file, the object file can be linked by the Linker utility. However, in the likely event that more than one COBOL-produced object file must be linked to produce an executable task, it is necessary to perform the "ODL merge" operation prior to the linking process. The merge operation is performed by the MRG utility. This utility merges the ODL files from more than one compilation into a single, composite ODL file. Subsequently, the object files, together with the composite ODL file, are linked together by the system linker to produce an executable image. In addition to linking the object files output by the COBOL compiler, the system linker automatically selects the required routines from the COBOL and RMS-11 object libraries.
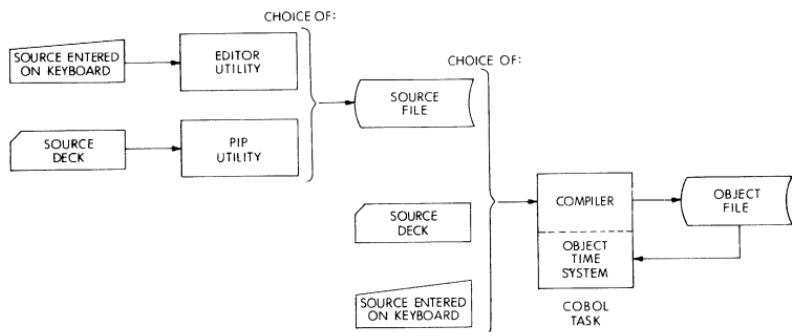
Figure 3-7 COBOL Language Processor Operation

INCREMENTAL COMPILERS

Some of the languages available in the PDP-11 software systems described in this handbook are processed by incremental compilers.

The BASIC language processors can accept source input from a terminal or from a file generated using an Editor utility, as illustrated in Figure 3-8. The most common method of creating a source program is

by giving the source statements to the compiler directly through an interactive terminal. For this reason, the BASIC language processors include an editing facility, which allows the programmer to create, test, and modify the source program interactively.
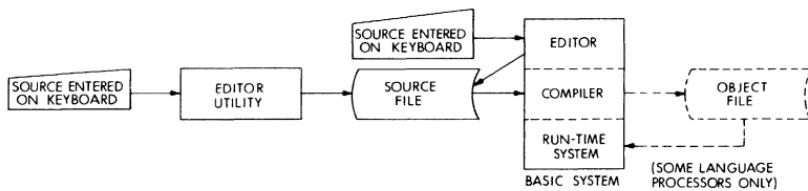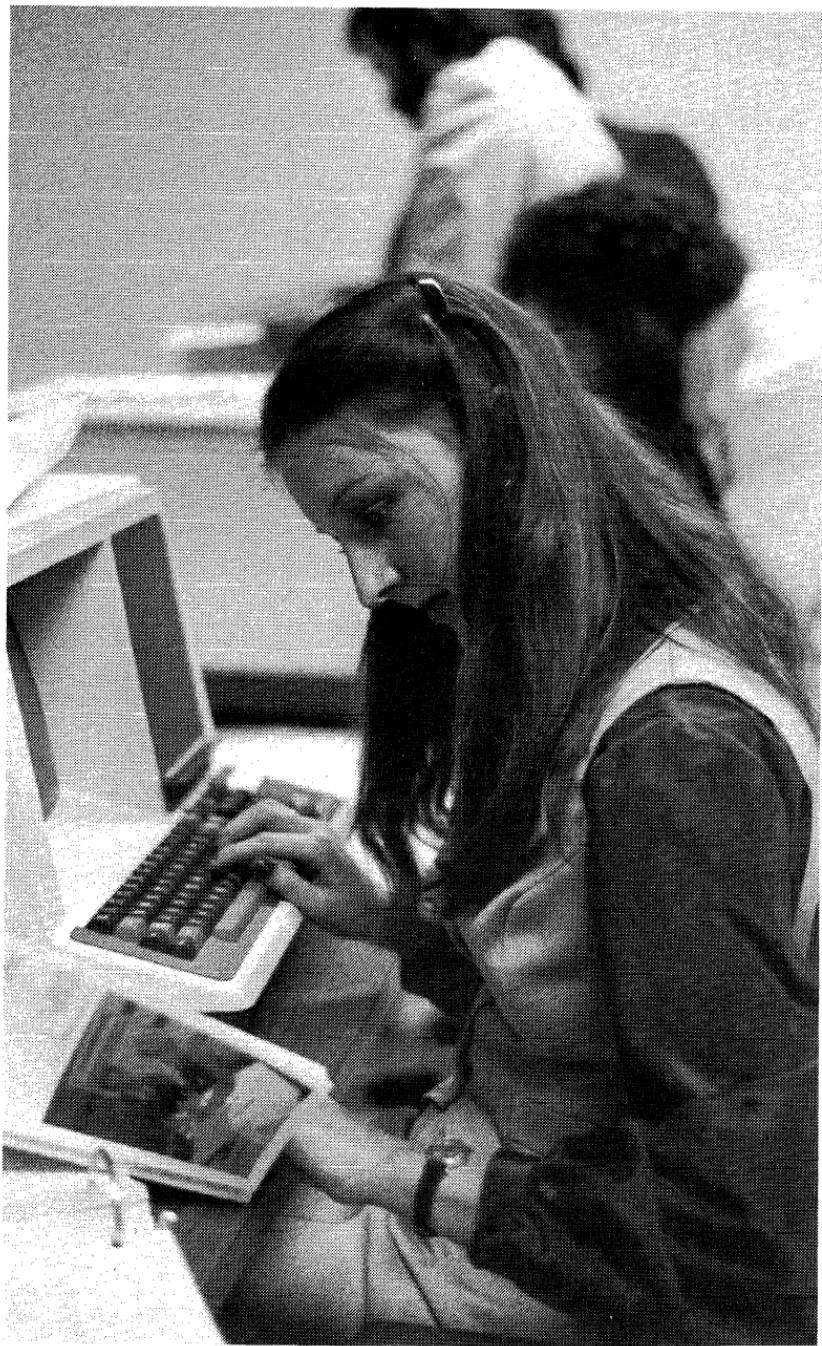
Figure 3-8 BASIC Language Processor Operation

operating systems

RT-11 (V3b)

OVERVIEW

RT-11 is an efficient, single-user, real-time disk operating system for interactive program development and dedicated on-line applications. It supports both single-job and foreground/background monitors. The foreground handles real-time functions and has priority on system resources; program development or batch jobs can operate in the background whenever the foreground is not busy. The system offers optional support for FORTRAN IV, FOCAL, BASIC, APL, and MACRO assembler.

RT-11 is the first of the six main PDP-11 operating systems to be presented in this section.

FEATURE TOPICS

- Functions and Features
- Operating Environments
  - RT-11 Single-Job Monitor
  - RT-11 Foreground/Background Monitor
  - RT-11 Extended Memory Monitor
  - Facilities available in RT-11 FB/XM
- SYSTEM COMMUNICATION
- Indirect Files
  - Keyboard Monitor Commands
  - Programmed Requests
- TEXT EDITOR
- Utility Programs
- Assembled Program Alteration
- System Subroutine Library
- RT-11 System Summary

FUNCTIONS AND FEATURES

RT-11 is an operating system designed to function in a single-user environment. In the commercial environment it can be bundled into variously packaged software known as CTS-300. The system uses a wide range of peripherals and accesses up to 124K words of either solid state or core memory. Three system monitors are provided by RT-11: the single-job monitor (SJ), the foreground/background monitor (FB), and the extended memory monitor (XM).

The single-job monitor allows one program at a time to reside in memory. The program executes until it completes or until it is interrupted with a keyboard command.

The foreground/background monitor allows two independent programs to reside in memory at one time. The foreground program, however, takes priority over the background program. RT-11 allows the background program to execute whenever the foreground program is in a wait state. Typically, the foreground program performs a time-dependent task, such as sampling material every few seconds and then analyzing the resultant data. A background program, on the other hand, usually performs a time-independent task, such as file maintenance or program development. This sharing of resources between two tasks greatly increases the efficiency of the RT-11 system.

The extended memory monitor provides all the features of the foreground/background monitor and, in addition, allows the user to access up to 124K words of memory. The other two monitors are restricted to 28K words of main memory.

The three monitors are upward compatible. That is, the foreground/background monitor provides all the features of the single-job monitor, and the extended memory monitor offers all the features of the foreground/background monitor. Error logging is supported as a SYSGEN option by all three RT-11 monitors.

In addition to the three monitors, RT-11 provides a full complement of system programs that can perform some more specific tasks than the keyboard monitor commands can.

RT-11 also supports a variety of language processors including MACRO-11, an assembly language, and several high level languages, such as FORTRAN IV, BASIC, APL, DIBOL, and FOCAL.

OPERATING ENVIRONMENTS

RT-11 Single-Job Monitor

The RT-11 single-job monitor provides a single-user, single-program system that can operate in as little as 8K words of memory. The SJ monitor is useful for extensive program development; since the monitor itself requires only 2K words of memory, there are at least 6K words left for the program, its buffers and its tables. The SJ environment is also suitable for running programs that require a high data transfer rate, since the SJ monitor services interrupts quickly.

All the system programs can be used under the SJ monitor. Monitor commands and programmed requests are also available to the SJ user. The single-job monitor does not support extended memory.

In summary, the SJ monitor is smaller and faster than the FB and the XM monitors; it is most useful when the user is concerned with program size versus available memory and when a dedicated system is needed.

RT-11 Foreground/Background Monitor

Often, the central processor of a computer system spends much of its time waiting for some external event to occur. Usually, this event is a real-time interrupt or the completion of an I/O transfer. This situation is particularly true of real-time jobs. The foreground/background environment lets the user take advantage of the unused processor capacity to accomplish lower-priority tasks.

In a foreground/background system, the foreground job is the time-critical, real-time job, and the FB monitor gives it priority over the background job. Whenever the foreground job reaches a state in which no useful processing can be done until some external event occurs, the monitor executes the background job, if possible. The background job then runs until the foreground job is again ready to execute. The processor then interrupts the background job and resumes the foreground job.

In effect, the RT-11 foreground/background monitor allows a time-dependent job to run in the foreground while a time-independent job, such as program development, runs in the background. All RT-11 system programs can run as the background job in a FB system. Thus the user can run FORTRAN, BASIC, or MACRO, for example, in the background while the foreground is collecting, storing, and analyzing data. In addition, the FB monitor gives the user the ability to set timer routines, suspend and resume foreground jobs, and send data and messages between the two jobs. The FB monitor is most often used for laboratory work, data acquisition, and real-time applications.

RT-11 Extended Memory Monitor

The extended memory monitor (XM) is an extension of the fore-ground/background (FB) environment. Generally, comments about the FB operation also apply to XM operation. The XM monitor permits either foreground or background jobs to extend their effective logical program space beyond the 32K word restriction imposed by the 16-bit address word of the PDP-11 processors. The XM monitor manages extended memory space as a system resource and dynamically allo-cates it as the user program requests. A program can map selected partitions of its addressing space, called windows, into extended memory areas called regions.

Facilities Available Only in RT-11 FB and XM

1. **Mark Time** — The .MRKT programmed request allows a program to set clock timers for specified amounts of time. When the timer runs out, the system enters the routine that the user has specified. The user can enter as many mark time requests as needed, provid-ing that system queue space has been reserved. The mark time feature is available to SJ monitor users as a SYSGEN option.

2. **Timed Wait** — The.TWAIT programmed request allows a program to "sleep" until the period of time that the user has specified elapses. A foreground program, for example, may need to act on sample data and write it to mass storage once every few minutes. While the foreground program is idle, the background program can run.

3. **Send Data, Receive Data** — The .SDAT and .RCVD programmed requests permit the foreground and background programs to com-municate with each other. The send and receive data functions let one program send messages or data of variable size blocks to the other program. For example, data can be transferred directly from a foreground collection program to a background analysis pro-gram.

4. **Channel Copy** — The .CHCOPY programmed request allows two programs to share the same data file.

5. **Device** — The .DEVICE programmed request allows the user to turn off specific devices upon program termination.

6. **Protect** — The .PROTECT programmed request protects the vec-tors that one program uses from interference by another program.

7. **Channel Status** — The .CSTAT programmed request returns stat-us data about an open channel.

8. **Multi-terminal support** — The multi-terminal support programmed request allows for multi-terminal systems featuring:

.MTATCH
.MTDTCH
.MTGET
.MTIN
.MTOUT
.MTPRNT
.MTRCTO

Facilities Available Only in RT-11 XM

An optional extension of the FB environment is the extended memory monitor (XM), which permits extension of the logical address space for either foreground or background jobs. Some features available to the user only when using the XM monitor are:

1. **Create a Region** — The .CRRG programmed request alows the user to allocate a region in extended memory for the current program.

2. **Eliminate a Region** — The .ELRG programmed request eliminates an extended memory region and returns it to the free list so it can be used by other programs.

3. **Create an Address Window** — The .CRAW programmed request unmaps and eliminates conflicting address windows, creates new windows to address extended memory, and maps new windows to the regions the user specifies. It directs the monitor to find the program a window into the region it has created. This request allows the program to access the physical memory as if it were local to the program.

4. **Eliminate an Address Window** — The .ELAW programmed request unmaps and eliminates address windows.

5. **Map** — The .MAP programmed request lets the user map address windows.

6. **Status** — The .GMCX programmed request returns status data about window mapping.

7. **Unmap** — The .UNMAP programmed request lets the user unmap a window.

SYSTEM COMMUNICATION

The monitor is the center of RT-11 system communications; it provides access to system and user programs, performs input and output functions, and enables control of background and foreground jobs. The user communicates with the monitor through programmed requests and keyboard commands.

Keyboard commands load and run programs, start or restart pro-
grams at specific addresses, modify the contents of memory, and
assign and de-assign alternative device names, to mention only a few
functions. A series of keyboard commands may be placed in a file
(called an indirect command file) if they are to be used frequently. The
series of commands can be invoked with a single keyboard command.

Indirect Files
The user can group together, as a file, a collection of keyboard
commands to be executed sequentially. This collection is called an
indirect command file, or indirect file. Indirect files are best suited for
tasks that require a significant amount of computer time and do not
require user supervision or intervention. Any series of commands that
a user is likely to type often can also run easily as an indirect file. The
indirect file concept is similar to BATCH processing. Although indirect
files lack some BATCH capabilities, they are easier to use, use the
same commands as normal operations, and generally require less
memory overhead than the BATCH processor. RT-11 BATCH is de-
scribed below.

Keyboard Monitor Commands
Table 4-1 shows the RT-11 keyboard commands and their results.

Table 4-1 RT-11 Keyboard Monitor Commands

APL	Invokes the APL language interpreter.
ASSIGN	Associates a logical device name with a physical de-vice.
B	Sets a relocation base.
BASIC	Invokes the BASIC language interpreter.
BOOT	Directs a monitor to take control of the system.
CLOSE	Makes permanent all output files that are currently open after the background job terminates.
COMPILE	Invokes one or more language processors to assem-ble or compile the files specified.
COPY	Performs a variety of file transfer and maintenance operations.

D	Deposits values in memory beginning at the location specified.
DATE	Sets or displays the current system date.
DEASSIGN	Disassociates a logical device name from a physical device.
DELETE	Deletes the files specified.
DIBOL	Invokes the DIBOL compiler to compile one or more source programs.
DIFFER-ENCES	Compares two files and lists the differences between them.
DIRECTORY	Lists information requested about a device, a file, or a group of files.
DUMP	Lists all or any part of a file in octal words, octal bytes, ASCII characters, or Radix-50 characters.
E	Prints in octal the contents of a memory address.
EDIT	Invokes a text editor.
EXECUTE	Invokes one or more language processors to assemble or compile the files specified; it also links object modules and initiates execution of the resultant program.
FOCAL	Invokes the FOCAL language interpreter.
FORTRAN	Invokes the FORTRAN IV compiler to compile one or more source programs.
FRUN	Initiates execution of foreground jobs.
GET	Loads a memory image file into memory.
GT	Enables or disables the use of the VT11 or VS60 display hardware as the console terminal.
HELP	Lists useful information about keyboard monitor commands and other RT-11 capabilities.
INITIALIZE	Clears and initializes a device directory.
INSTALL	Installs the device specified into the system.

LIBRARY   Lets you create, update, modify, list, and maintain library files.

LINK   Converts object modules produced by an RT-11 supported language processor into a format suitable for loading and execution.

LOAD   Makes a device handler resident in memory for use with BATCH or foreground/background jobs.

MACRO   Invokes the MACRO assembler to assemble one or more source files.

PRINT   Lists the contents of one or more files on the line printer.

R   Loads a memory image file into memory and starts execution.

REENTER   Starts the program at its reentry address (the start address minus 2).

REMOVE   Removes a device from the system tables.

RENAME   Changes the name of a specified file or files.

RESET   Resets several background system tables and does a general clean-up of the background area.

RESUME   Continues execution of the foreground job at the point the SUSPEND command was issued.

RUN   Loads a memory image file into memory and starts execution.

SAVE   Writes memory areas in memory image format to the file and device specified.

SHOW   Prints on the terminal all the devices known to the system and any logical names assigned to those devices. It has three options that all call the RESORC program:

                 ● CONFIGURATION
                   Displays information about the monitor status, USR SWAP/NOSWAP, type of processor, special hardware, and SYSGEN options that are in effect.

                 ● DEVICES
                   Displays device information, including current status and vectors.

- TERMINALS

  Displays terminal information, including unit number, type, and SET options enabled for multi-terminal systems.

SQUEEZE   Consolidates in a single area all unused blocks on the device specified.

START   Initiates execution of the program currently in memory at the address specified.

SUSPEND   Stops execution of the foreground job.

TIME   Sets or displays the current time of day.

TYPE   Lists the contents of one or more files on the terminal.

UNLOAD   Makes handlers that were previously loaded nonresident, thus freeing the memory space they occupied.

Programmed Requests

Programmed requests are source program instructions that request the monitor to perform monitor services. These instructions allow assembly language programs to use the available monitor features. A running program communicates with the monitor through programmed requests. FORTRAN programs have access to programmed requests through the system subroutine library. Programmed requests can, for example, manipulate files, perform input and output, and suspend and resume program operations.

Table 4-2 shows the RT-11 programmed requests and their action.

Table 4-2 Programmed Requests

.CDFN   Defines new I/O channels.

.CHAIN   Allows background program to transfer control to another background program that is specified in locations 500-507 (RAD50) without operator intervention; saves words 500-777.

.CHCOPY   Opens a channel for input and logically connects it to a channel on another job open for either input or output.

.CLOSE   Terminates activity on the specified channel and frees it for use in another operation.

.CMKT   Cancels one or more outstanding mark time requests.

.CNTXSW   Specifies locations to be included in context switch; addr is terminated with a 0 word; valid locations are 2-476, user job area, and 160000-177776.

.CRAW   Defines a virtual address window and optionally maps it into a physical memory region.

.CRRG   Allocates a dynamic region in physical memory for use by the current requesting program.

.CSIGEN   Calls Command String Interpreter (CSI) in general mode to accept input and output file specifications; .CSIGEN automatically opens input and output files and loads required device handlers; gets command string from terminal, in core string, or indirect command file; returns command line to the user if the user so specified.

.CSISPC   Calls Command String Interpreter (CSI) in special mode to accept input and output file specifications; works like .CSIGEN, but does not open files; instead, builds a table of file specifications to simplify later file operations; gets command string from terminal, in core string, or indirect command file; returns command line to the user if the user so specified.

.CSTAT   Furnishes 6 information words about an I/O channel:
                 word 1   channel status
                 word 2   file starting block
                 word 3   file length
                 word 4   highest block writtten
                 word 5   device unit number
                 word 6   device name (RAD50)

.DATE   Moves current date word into RO:
                 bits 14-10   month (1-12.)
                 bits 9-5   day (1-31.)
                 bit 4-0   year (72.-99)

.DELETE   Deletes named file from indicated device.

.DEVICE   Sets up list of addresses to be loaded with specified values upon program termination; creates a linked list if specified.

.DSTATUS   Provides information about device characteristics:
                 word 1   device status
                 word 2   handler size
                 word 3   handler entry point
                 word 4   device size

.ELAW   Cancels a defined window and permits redefinition.

.ELRG   Deallocates a dynamic memory region and returns it to the free list.

.ENTER   Allocates space on specified device and creates tentative directory entry for the named file.

.EXIT   Terminates user background program and returns control to the monitor, when used from a background program under FB; when used under SJ, causes KMON to run in background area.

.FETCH   Loads device handler into memory from the system device.

.GMCX   Returns the mapping status of a specified window.

.GTIM   Returns current time of day in ticks past midnight.

.GTJB   Passes certain job parameters back to the user program:
                 gram:
                 word 1   job number (0=B, 2=F)
                 word 2   high memory limit
                 word 3   low memory limit
                 word 4   start of I/O channel space
                 word 5   address of job's impure area
                             with FB and XM monitors
                 word 6-8   reserved

.GTLIN   Obtains a line of input from the console terminal or an indirect command file; allows the user to specify a text string which will be printed on the terminal to prompt the operator for input.

RT-11

.GVAL   Returns a monitor fixed offset value in R0.

.HERR   Disables user error interception and allows the monitor to detect and act on fatal errors.

.HRESET   Resets channels, releases device handlers, and stops all I/O transfers in progress.

.INTEN   Notifies monitor that an interrupt occurred and switches to "system state"; lowers processor priority to device priority level.

.LOCK   Locks USR (system file processor) in memory; the USR normally swaps into memory only when it is needed (to open or close a file), and swaps out when it is not needed (to read or write an open file).

.LOOKUP   Associates specified channel with a device and existing file.

.MAP   Maps a previously defined address window into a region of extended memory.

.MFPS   Reads priority bits from the processor word.

.MRKT   Schedules completion routine to be entered after a specified time interval.

.MTATCH   Attaches a terminal for exclusive use by the requesting job.

.MTDTCH   Detaches a terminal from one job and makes it available for other jobs.

.MTGET   Returns the status of the specified terminal to the caller.

.MTIN   A multi-terminal form of .TTYIN; transfers one or more characters to a buffer.

.MTOUT   A multi-terminal form of .TTYOUT; prints one or more characters from a buffer.

.MTPRN   A multi-terminal form of .PRINT.

.MTPS   Sets priority bits, condition codes, and T bit in the processor status word.

.MTRCTO   Resets CTRL/O for the specified terminal.

.MTSET	Allows the user program to set terminal and line characteristics.
.MWAIT	Suspends execution until all messages are transmitted or received.
.PRINT	Outputs an ASCII string to the terminal.
.PROTECT	Used by a job to obtain exclusive control of a vector pair in the range 0-476.
.PURGE	Deactivates a channel without closing the file (tentative output file is lost).
.QSET	Enlarges I/O queue for the monitor.
.RCTRLO	Enables console terminal printing (resets CTRL/O).
.RCVD	Posts a request to receive message and continues execution.
.RCVDC	Posts a request to receive message and enters specified completion routine when message is received.
.RCVDW	Posts a request to receive message and waits until it is received.
.READ	Initiates transfer of words from specified channel into memory and continues execution.
.READC	Initiates transfer from channel to memory; continues executing user program; enters specified routine when transfer completes.
.READW	Transfers words from specified channel into memory; returns control to user program when the transfer completes or when an error is detected.
.RELEAS	Removes a device handler from memory.
.RENAME	Changes a file name.
.REOPEN	Reassociates a channel with a file on which a SAVE-STATUS was performed.
.RSUM	Resumes execution of a foreground job after it was suspended.
.SAVESTA-TUS	Stores 5 words containing data concerning file definition into memory; frees channel for use:

word 1	channel status
word 2	starting block of file
word 3	length of file
word 4	reserved
word 5	even byte — I/O count;
	odd byte — device unit
	number

.SCCA   Inhibits CTRL/C abort; indicates that CTRL/C was typed at the keyboard; distinguishes between single and double CTRL/C.

.SDAT   Initiates message transfer; returns control to user program immediately.

.SDATC   Initiates message transfers; transfers control to specified routine when message is received.

.SDATW   Initiates message transfer; returns control to user program when message is received.

.SERR   Inhibits monitor from aborting jobs after fatal errors.

.SETTOP   Requests additional memory for program and returns the highest memory address available.

.SFPA   Sets user interrupt for floating point processor exceptions.

.SPFUN   Provides special device-dependent functions to magtape, cassette, diskette, and other mass storage devices.

.SPND   Suspends a foreground job.

.SRESET   Resets certain memory areas, dismisses device handlers, purges currently open files, resets to 16 channels, resets I/O queue to one element.

.SYNCH   Enables the user program to perform certain monitor programmed requests from within an interrupt service routine. Requests requiring the USR may not be issued.

.TLOCK   Attempts to gain ownership of USR; if unsuccessful, returns control with C bit set.

.TRPSET   Allows user job to intercept processor traps to 4 and 10.

74

.TTINR   Inputs a character from the terminal; returns if none available.

.TTOUTR   Outputs a character to the terminal; returns if no room in buffers.

.TTYIN   Inputs a character from the terminal and waits until operation is done.

.TTYOUT   Outputs a character to the terminal and waits until operation is done.

.TWAIT   Suspends the running job for the specified amount of time (number of ticks); requires queue element.

.UNLOCK   Releases USR from memory.

.UNMAP   Unmaps a window and flags that portion of address space as being inaccessible.

.UNPROTECT   Cancels a protected vector pair in the range 0-476.

.WAIT   Suspends program execution until I/O completes to the specified channel.

.WRITC   Transfers words from memory to specified channel; when complete, passes control to specified routine.

.WRITE   Initiates transfer from memory to channel; returns control to user program immediately.

.WRITW   Transfers words from memory to channel; when transfer is complete, returns control to user program.

TEXT EDITOR

The text editor (EDIT) is a program that creates or modifies ASCII source files for use as input to other system programs such as the MACRO assembler or the FORTRAN compiler. EDIT, which accepts commands from the user at the terminal, reads ASCII files from any input device, makes specific changes, and writes on any output device. EDIT allows efficient use of VT11 or VS60 display hardware, if they are part of the system configuration.

EDIT considers a file to be divided into logical units called pages. A page of text is generally 50-60 lines long (delimited by form-feed characters) and corresponds approximately to a physical page of a program listing. The editor reads one page of text at a time from the input file into its internal buffers, where the page becomes available for editing. EDIT is used to:

- Locate text to be changed.
- Execute and verify the changes.
- List an edited page on the console terminal.
- Output a page of text to the output file.

Normally, the editor operates in either command mode or text mode. In command mode, the editor interprets all input typed on the keyboard as commands to perform some operation. In text mode, the editor interprets all typed input as text to replace, insert into, or append to the contents of the text buffer.

UTILITY PROGRAMS
The following sections describe the RT-11 system programs available. The user can take advantage of nearly all of the capabilities of the RT-11 system by using the keyboard monitor commands. However, it is the system utility programs (and not the monitor itself) that actually perform many of the system's functions.

Command String Interpreter
The Command String Interpreter (CSI) is the part of the RT-11 system that accepts a line of ASCII input, usually from the user at the console terminal, and interprets it as a string of input specifications, output specifications, and options for use by a system utility program.

The Peripheral Interchange Program (PIP)
The peripheral interchange program (PIP) is a file transfer and file maintenance utility program for RT-11. PIP is used to transfer files between any of the RT-11 devices and to merge, rename, and delete files.

Device Utility Program (DUP)
The device utility program (DUP) is a device maintenance utility program. DUP creates files on file-structured RT-11 devices. It can also extend files on certain file-structured devices (disks and DECtape), and it can compress, image copy, initialize, or boot RT-11 file structured devices. DUP does not operate on non-file structured devices (line printer, card reader, terminal, and paper tape).

Directory Program (DIR)
The directory program (DIR) performs a wide range of directory listing operations. It can list directory information about a specific device, such as the number of files stored on the device, their names, and their creation dates. DIR can list details about certain files, too, including their names, their file types, and their size in blocks. DIR can also

print a device directory summary, and it can organize its listings in several ways, such as alphabetically or chronologically.

Linker (LINK)

The RT-11 linker (LINK) converts object modules produced by an RT-11 supported language translator into a format suitable for loading and execution. The linker processes the object modules of the main program and subroutines to:

- Relocate each object module and assign absolute addresses.

- Link the modules by correlating global symbols that are defined in one module and referenced in another.

- Create the initial control block for the linked program that the GET,R,RUN, and FRUN commands use.

- Create an overlay structure if specified and include the necessary run-time overlay handler and tables.

- Search libraries specified by the user, to locate any unresolved globals.

- Automatically search a default system library to locate any remaining unresolved globals.

- Produce a map showing the layout of the executable module.

- Produce a symbol definition file.

The RT-11 linker requires two passes over the input modules. During the first pass it constructs the symbol table, including all program section names and global symbols in the input modules. After it processes all non-library files, the linker scans the library files to resolve undefined globals. It links only those modules that are required into the root segment (that part of the program that is never overlaid). During the final pass, the linker reads the object modules, performs most of the functions listed above, and produces a load module (which is memory image format for background jobs or for jobs that run in the single-job environment, relocatable format for foreground jobs, and formatted binary for use with the Absolute Loader).

The linker runs in a minimal RT-11 system of 8K words of memory; the linker uses any additional memory to facilitate efficient linking and to extend the size of the symbol table. The linker accepts input from any random access device on the system; there must be at least one random-access device (disk or DECtape) for memory image or relocatable format output.

Librarian (LIBR)

The librarian utility program (LIBR) lets the user create, update, modify, list, and maintain object library files. A library file is a direct access file (a file that has a directory) that contains one or more modules of the same module type. The librarian organizes the library files so that the linker and MACRO assembler can access them rapidly. Each library contains a library header, library directory (or global symbol or macro name table), and one or more object modules or macro definitions.

The object modules in a library file can be routines that are repeatedly used in a program, routines that are used by more than one program, or routines that are related and simply gathered together for convenience. The contents of the library file are determined by the user's needs. Object modules in a library file from another program can be accessed from another program by making calls or references to their global symbols; the user then links the object modules with the program that uses them, producing a single load module.

DUMP

DUMP is the RT-11 program that prints on the console or line printer, or writes to a file, all or any part of a file in octal words, octal bytes, ASCII characters, or Radix-50 characters. DUMP is particularly useful for examining directories and files that contain binary data.

File Exchange Program (FILEX)

The file exchange program (FILEX) is a general file transfer program that converts files from one format to another so that they can be used with other operating systems. Transfers between any block-replaceable RT-11 directory-structured devices can be initiated.

Source Compare (SRCCOM)

The RT-11 source compare program (SRCCOM) compares two ASCII files and lists the differences between them. SRCCOM can either print the results or store them in a file. SRCCOM is particularly useful when it is necessary to compare two similar versions of a source program. A file comparison listing highlights the changes made to a program during an editing session.

Format Program (FORMAT)

The format program has several functions. It can be used to write headers for an RK05 disk as well as write single or double density sectors for an RX02 diskette.

Resource Program (RESORC)

The resource program allows for the display of information about the system configuration. This command is accessed through the SHOW monitor command.

ASSEMBLED PROGRAM ALTERATION

Three RT-11 programs help the user debug programs and make changes to programs that are already assembled. They are: the on-line debugging technique (ODT), PATCH, and the object module patching utility (PAT).

RT-11 On-Line Debugging Technique

RT-11 on-line debugging technique (ODT) is a program (supplied with the system) that aids in debugging assembly language programs. From the terminal, the user can direct the execution of programs with ODT. ODT performs the following tasks:

- Prints the contents of any location for examination or alteration.
- Runs all or any portion of an object program using the breakpoint feature.
- Searches the object program for specific bit patterns.
- Searches the object program for words that reference a specific word.
- Calculates offsets for relative addresses.
- Fills a single word, block of words, byte or block of bytes with a designated value.

PATCH

The PATCH utility program is used to make modifications to any RT-11 file. PATCH can be used to examine and then to change words or bytes in the file.

Object Patch Utility (PAT)

The RT-11 object module patch utility (PAT) allows the user to patch or update any code in a relocatable binary object module. PAT does not permit the examination of the octal contents of an object module; that is a function of PATCH. An advantage to using PAT is that relatively large patches can be added to an object module without performing any octal calculation. PAT accepts a file containing corrections or additional instructions and applies these corrections and additions to the original object module.

BATCH

RT-11 BATCH is a complete job control language that allows RT-11 to operate unattended. RT-11 BATCH processing is ideally suited to frequently run production jobs, large and long-running programs, and programs that require little or no interaction with the user.

RT-11 BATCH permits the user to:

- Execute a RT-11 BATCH stream from any legal RT-11 input device.

- Output a log file to any legal RT-11 output device (except magtape or cassette).

- Execute the BATCH stream with the single-job monitor or in the background of the foreground/background monitor or with the extended memory monitor.

- Generate and support system-independent BATCH language jobs.

- Execute RT-11 monitor commands from the BATCH stream.

SYSTEM SUBROUTINE LIBRARY

The RT-11 FORTRAN System Subroutines (SYSF4) are a collection of FORTRAN-callable routines that allow a FORTRAN user to utilize various features of RT-11 foreground/background(FB)and single-job(SJ) monitors. SYSF4 also provides various utility functions, a complete character string manipulation package, and a 2-word integer support. This collection of routines is usually placed in a default system library, which is an object module library file called SYSLIB.OBJ. This library file is the default library that the linker uses to resolve undefined globals and is resident on the system device (SY:). This concatenated set of routines is in a file called SYSF4.OBJ. The installation procedures describe how to make these routines into a library.

The following are some of the functions provided by SYSF4.

- Complete RT-11 I/O facilities, including synchronous, asynchronous, and completion-driven modes of operation. FORTRAN subroutines may be activated upon completion of an input/output operation.

- Timed scheduling of asynchronous sub jobs (completion routines). This feature is standard on FB and optional on the SJ monitor.

- Complete facilities for interjob communication between foreground and background jobs (FB and XM only).

- FORTRAN interrupt service routines.

- Complete timer support facilities, including timed suspension and time-of-day information. These timer facilities support either 50 or 60 cycle clocks.

- All auxiliary input/output functions provided by RT-11, including the capabilities of opening, closing, renaming, creating, and deleting files from any device.

- All monitor-level informational functions, such as job partition parameters, device statistics, and input/output channel statistics.

- Access to the RT-11 Command String Interpreter (CSI) for acceptance and parsing of standard RT-11 command strings.

- A character string manipulation package supporting variable-length character strings.

- INTEGER*4 support routines that allow 2-word integer computations.

SYSF4 allows the FORTRAN user to write almost all application programs completely in FORTRAN with no assembly language coding.

LANGUAGES
Languages that run under RT-11, such as MACRO, FORTRAN, APL, FOCAL, and BASIC, are described in individual chapters in this handbook.

RT-11 SYSTEM SUMMARY

Is

- Foreground/background (multi-tasking)
- Single user
- Sensor based
- Operating on small CPUs
- Protected environment
- Easy to install and use
- High real-time throughput
- Batch processing
- Highly reliable
- Full development facilities

Is not

- Transaction processing
- Record management
- Data base management

Languages

- BASIC-11
- FORTRAN IV
- MACRO-11

RESOURCE-SHARING TIMESHARING SYSTEM RSTS/E (V6C)

OVERVIEW

RSTS/E is a resource-sharing timesharing system supporting many language processors. This system can also support general purpose timesharing as well as batch processing. Each RSTS/E user can have virtually the entire system's processing power, utilities and peripherals at his command during program development or execution. Its dynamic scheduling algorithm allocates processor time, memory space, file space and peripherals to continually keep processing efficient.

FEATURE TOPICS

- Functions and Features
- System Configuration and Operation
  - System Code
  - Language Processors (BASIC-PLUS)
  - Timesharing Operations Overview
  - SYSGEN
- System Management Utility Programs
- Device and File Conventions
- User Interface
  - System and Installation Defined (CCL) Commands
  - General System Utility Programs
  - Batch Processing
- SYS System Functions and the PEEK Function
- RSTS/E System Summary

FUNCTIONS AND FEATURES

The RSTS/E operating system allows multiple users to interact with the system and its data structures. RSTS/E supports up to 63 users simultaneously processing data using the BASIC-PLUS, COBOL, BASIC-PLUS-2, FORTRAN IV, APL, or RPG II language processors. BASIC-PLUS jobs may vary in size up to 16K-word programs, and can use chaining and interjob communication features to execute even larger programs. Programs using other languages can have a maximum size of 28K words. RSTS/E also includes a comprehensive set of easy-to-use system utilities for the system manager and timesharing users. The system also supports line printer spooling and execution of up to 8 batch streams. It may also be offered as part of variously packaged commercial hardware/software systems as CTS-500. When RSTS/E is packaged as CTS-500, certain otherwise optional software is bundled into the system.

RSTS/E can support a maximum of 63 concurrent jobs. The actual number of jobs a configuration can support depends on the characteristics of the application(s), the processor, processor options, disk drives, and the amount of memory available. As a general rule, RSTS/E can support up to 63 jobs on a PDP-11/70. On a PDP-11/34, 11/35, 11/40, 11/45, 11/50, 11/55, or 11/60 processor, RSTS/E can be expected to support less than the 63 job maximum. In most cases, the practical maximum is 32 jobs or fewer on these processors.

RSTS/E requires at least 64K words of memory for most configurations. Installations making use of the commercial features of COBOL, BASIC-PLUS-2, RPG II, DIBOL with CTS-500, and/or RMS-11K will generally need at least 80-96K words of memory. For those installations that need only BASIC-PLUS and a limited subset of the RSTS/E utilities, it may be possible to operate with as little as 48K words of memory. To support more than a few multiusers simultaneously, additional memory may be required.

A minimum peripheral complement includes a console terminal and a disk system. The system device can be a single RP02, RP03, RP04, RP05, RP06, RM02, or RM03 disk system or a dual drive RK05, RK06, RK07, or RL01 disk system. A TE10, TE16, or TS03 magnetic tape system is required for system generation and back up unless the disk system includes three RK05s, two RL01s, two RK06s, or two RK07s.

On a PDP-11/34, PDP-11/35, PDP-11/40, PDP-11/45, PDP-11/50, PDP-11/55, or PDP-11/60, memory can expand up to 124K words; on a PDP-11/70, up to 1920K words. In addition, RSTS/E can support multiple disk, DECtape, magnetic tape and floppy disk drives, multiple line printers, a card reader, a paper tape reader/punch, and a variety

of terminal interfaces. All of these devices can be available to any terminal user. The terminals can be accessed under program control for input and output. A single program can control any number of terminals up to a maximum of 127.

RSTS/E users can expect efficient operation because the operating system dynamically allocates processor time, memory space, file space and peripherals to best suit changing demands. The system manager and designated privileged users have access to the monitor's system management commands either interactively using system utilities or under program control. Additional system commands and utility programs are also available to all users.

The RSTS/E file system provides a wide range of on-line processing capabilities. Files can be accessed randomly or sequentially, either through BASIC-PLUS, or through the RMS-11 (Record Management Services) subsystem. Single and multi-key ISAM is optionally available with RMS-11K software. Files can contain alphanumeric string, integer numeric, floating point numeric or binary data. Files can be created, updated, extended or deleted interactively either from the user's terminal or under program control. Files can be sorted by the SORT-11 program. Files can be protected from access on an individual, group or system basis. Files can also be accessed by many users while being updated on-line.

RSTS/E provides the ability to back up files selectively or totally. Back-up can be done on-line without disrupting users or it can be done off-line.

DECnet/E provides the RSTS/E system with communications software for programs written in BASIC-PLUS and BASIC-PLUS-2. This software offers point-to-point, task-to-task, and network file transfer communications facilities.

RSTS/E also supports an emulator for the IBM 2780 terminal and permits communication between DIGITAL systems and IBM Remote Job Entry programs supporting OS/HASP, OS/ASP, DOS POWER and OS/RJE or a second DIGITAL-supplied 2780 emulator.

DATATRIEVE-11 provides RSTS/E with a file maintenance, report generation and query facility.

Table 5-1 summarizes the components of RSTS/E. A complete list of the hardware and software supported by RSTS/E is provided in the RSTS/E Software Product Description.

Table 5-1 RSTS/E System

System type	General timesharing system using the BASIC-PLUS interpreter with optional language support of COBOL, FORTRAN IV, RPG II, BASIC-PLUS-2, and APL. (DIBOL is also available in CTS-500 systems.)
CPUs supported	PDP-11/40 with Extended Instruction Set and Memory Management; PDP-11/34, PDP-11/45 or PDP-11/60 with Memory Management Unit; PDP-11/70
Memory ranges	Minimum: 64K words memory for most installations, though it is possible to configure an adequate 48K system if only the BASIC-PLUS interpreter is to be used.
	Maximum: 124K words on PDP-11/34, 35, 40, 45, 50, 55, 60 1920K words on PDP-11/70
Minimum peripherals	Console terminal
	Disk system: RP02, RP03, RP04, RP05, RP06, RM02, RM03 Dual drive RK05, RK06, RK07, or RL01 TS03, TU45, TE10, or TE16 magnetic tape systems (not required if at least three RK05s, three RL01s, two RK06s, or two RK07s are included)
Additional CPU hardware	PDP-11/34,45,60 or PDP-11/70 FP11 Floating Point Processor PDP-11/40 KE11-F Floating Instruction Set PDP-11/34 KK11-A Cache Memory Option

Additional peripherals	Up to eight line printers (LP11, LP05, LS11, LV11 or LA11). Up to a total of eight RK05s; up to four RL01s; up to eight RK06s and RK07s; up to a total of eight RP02s, RP03s, RP04s, RP05s, or RP06s; up to a total of eight RM02, or RM03 disk drives. Up to a total of eight RS03 or RS04 disk drives, or up to eight RS-11 disk drives. Up to a total of eight TS03, TE10, TE16, and TU45 magnetic tape drives. Up to four RX11 floppy disk systems for a maximum of eight RX01 drives. Card reader (CR11 punched, CM11 marked or CD11 high-speed punched card reader). PC11 paper tape reader/punch. Up to a total of 127 terminal line interfaces, of which up to 16 can be single-line (KL11, DL11, LC11, DC11, or DJ11), and the remainder multi-line interfaces (DZ11s or DH11s with or without DM11s). Up to a total of 127 terminals: LA30, LA36, LA180, VT05, VT50, VT52, VT55, LT33, LT35, RT02, or IBM 2741-compatible terminals.
Optional software	COBOL, RPG II, BASIC-PLUS-2, APL, FORTRAN IV Language Processors (DIBOL/DECFORM is also available on CTS-500 systems); SORT-11 File and Index Sort Program; RMS-11K Record Management Services. DMS-500 RSTS/E Commercial Extensions Package, RSTS/2780 Remote Job Communications Package, DECnet/E Communications System. DATATRIEVE-11 Query and Report Generator.

SYSTEM CONFIGURATION AND OPERATION

RSTS/E system software exists as system code, language processing code, and system program code. The system code and language processing code are tailored at system generation time according to the hardware configuration on which the system runs and the software features which are chosen by the system manager. Once the system is generated, the system code language processing code are frozen and alterable only by patching or generating new code. The system program code exists in a library of programs executable by the system software or by individual users on the system. The library of programs is alterable and expandable during timesharing without requiring regeneration of the system.

System Code

The RSTS/E system code is stored on the system disk as a save-image library (SIL). A save-image library, when loaded into memory, is immediately executable by the PDP-11 computer. The system code comprises many distinct elements which are either resident in memory or on disk during timesharing. Permanently resident elements are the following:

- interrupt and trap vectors
- small and large system buffers
- system information and data tables
- disk and device drivers
- file processor modules

Optionally, the following are also resident modules:

- RJ2780 — Remote Job Entry handler
- DECnet/E — Network Communications handler

The following are either permanently resident or disk resident (overlay) elements, the choice to be selected at system generation time.

- file processor modules
- infrequently used utility routines

The following is loaded only at system start-up time.

- system initialization code

RSTS/E operations start when the system disk is bootstrapped. The bootstrap routine loads the initialization code which determines the hardware configuration and performs many consistency checks to ensure the integrity of the software. When checking is completed, the initialization code remains resident and allows many options, some of which are described below.

When timesharing operations are started, the initialization code is overlaid by the permanently resident system code and the system default run-time system. As timesharing operations proceed, infrequently used overlay code and system and user programs are loaded from disk as needed.

Language Processors
The BASIC-PLUS language processor generally serves as the system default run-time system. However, any of the languages mentioned above may also be used for applications programs. The language processors reside on the system disk in machine executable form and can be either permanently resident in memory or temporarily resident (swappable). If, for example, BASIC-PLUS were the system default RTS operating under RSTS/E, the system disk would contain the following sets of elements:

1. Permanently resident elements
   - BASIC-PLUS text editor and analyzer
   - BASIC-PLUS incremental compiler
   - BASIC-PLUS run-time system

2. Temporarily resident elements
   - auxiliary run-time systems for other language processors
   - object time systems
   - language processors
   - object time systems run as user jobs

In this example, the BASIC-PLUS code is loaded into memory at the start of timesharing operations and remains resident during the session. The code analyzes all BASIC-PLUS statements and generates and executes intermediate (compiled) code. Many monitor services are available to a BASIC-PLUS program through system function calls.

The auxiliary run-time system associated with a given language processor or object time system is loaded into memory only when a request is made to execute that language compiler or to execute a compiled program written in that language. The language compiler is swapped out to disk as required, just as any normal user job would be.

The run-time system may vary in size from 2K words to 16K words, and is generally shared among users.

System Program Code
A library of programs is produced and stored on disk during the system library build procedures of system generation. Both the system and users execute these programs to perform system housekeeping and common utility functions. The system manager can use the pro-

grams to monitor and regulate system usage. Some library programs can be tailored by altering the source statements supplied by DIGITAL and recompiling to replace the current copy on the system disk.

Timesharing Operations Overview

To begin a timesharing session, a user logs in to the system by entering an account number and password at a terminal. The user is assigned an account number and password by the system manager.

Immediately after the user logs in, his terminal is under the control of the keyboard monitor of the system default run-time system. The terminal is also in edit mode, and is returned to edit mode when any program execution is completed or whenever a CTRL/C is typed at the terminal. If, for example, BASIC-PLUS is the main language processor, the terminal edit mode would be the BASIC-PLUS command level. In edit mode, the system examines each ASCII text line entered by the user and determines whether that line is a system or an installation-defined command, an immediate mode statement, or a program statement. (Installation-defined commands are made possible by the concise command language (CCL) facility described below.) System and installation-defined commands are executed immediately after being entered. Immediate mode statements are first translated into an intermediate code, which is placed in the user's job area, and are executed immediately by the run-time system. Program statements (lines of ASCII text preceded by line numbers) are stored in their ASCII form in a temporary disk file under the user's account. Each program statement is also compiled into its intermediate code representation, which is placed in the user's area of memory.

A user job area is initialized at log-in time and set to a size of 1K or 2K words, depending on the run-time system being used. When BASIC-PLUS is used, the user job area is initially 2K words. The job area can grow in increments of 1K words to a maximum size set by the system manager at the start of timesharing operations. Intermediate code created in the user's job area upon entry of program statements in edit mode is not executed automatically. The related program statements being created can be changed. A copy of the intermediate code of the program can be transferred to disk storage or to an external storage medium.

A user changes from edit mode to run mode by typing the RUN system command or the CHAIN immediate mode statement. In run mode, the run-time system interpretively executes the intermediate code stored in the user's job area. When a program finishes execution, the terminal is returned to edit mode, signaled by the printing of the READY message. The user can interrupt the run-time system by typing CTRL/C,

which also returns the terminal to edit mode. Note: "edit mode" is so named because at that point the current BASIC-PLUS program can be edited by retyping any line. When a language other than BASIC-PLUS is used, an editing program must be run to make any changes in the programs.

When the terminal is in run mode, a privileged user can detach the running job from the terminal. This allows the user to login again, open up another job area, and run another job. The detached job runs unattended, but is still associated with the account under which the user logged in. To regain control of a detached job running under account, the user can log in on any free terminal and attach the job to that terminal.

The RSTS/E system allows jobs to run (in either edit mode or run mode) one at a time. A job runs until it either enters an I/O wait state or exhausts the time quantum which either the system or the system manager has assigned to it. At the point when the currently running job ceases to run, the scheduler finds the next job that is ready to run and begins running that job. Meanwhile, the interrupt-driven I/O device handlers are processing requested data transfers. Upon completion of a transfer, the scheduler marks the job that requested the transfer as ready to run again and starts it from the point at which execution ceased.

RSTS/E attempts to keep as many jobs in memory as possible. When more memory is required to run a job than is available, the system temporarily swaps some jobs out of memory and stores them in one of the swap files defined by the system manager.

When it is again their turn to run, the jobs in the swap files are swapped back into memory. Jobs waiting for keyboard input and jobs waiting for device I/O completion are most likely stored in the swap files, while jobs currently running or involved in disk or magtape data transfers are necessarily in memory.

As the system processes each job, it maintains accounting information in memory concerning that job. When the job is logged off the system, this information is used to update the accounting information stored on the disk for that account.

System Generation
System generation is normally a one-time operation in which the system manager defines the hardware configuration and selects the basic software options. The system manager needs to perform a system generation only when the system is first installed or when the hardware configuration changes. Both the monitor and the BASIC-PLUS

code can be generated in one operation or either can be generated separately. Software options can be included in the system to increase processing power or can be excluded from the system to conserve memory.

In addition to defining the number and kinds of peripherals and processing hardware during system generation, the system manager defines special configuration options. Some of these options are discussed below.

Pseudo Keyboards

The system manager can define the system to have one or more pseudo keyboards. A pseudo keyboard is a non-physical device that has the characteristics of a physical terminal but that has no terminal associated with it. As such, a pseudo keyboard has both input and output buffers from which a program can extract output and to which a program can force input. Using a pseudo keyboard as a communications device, a user can write a program to control other jobs. In addition, each copy of the BATCH system program requires one pseudo keyboard to run jobs in a batch stream. If the installation plans to run several copies of BATCH simultaneously, at least that number of pseudo keyboards must be defined.

Multiple Terminal Service

The multiple terminal service option allows one BASIC-PLUS program to interact with several users simultaneously by servicing their terminals on one I/O channel. This eliminates the need to run separate copies of the same program when several terminals must perform a similar function.

Maximum Number of Jobs

With sufficient hardware, RSTS/E can support up to 63 simultaneous jobs. The maximum number of jobs that can be run efficiently depends on the available memory space and the number and types of disks and processor options on the system. When a job is started, it is given a number by the system. Jobs are numbered sequentially from one to the maximum number of jobs the system can handle. Jobs include both attached jobs and detached jobs. The maximum number of jobs must be specified at system generation since it determines the size of some monitor tables. The number can be lowered during system initialization to adjust to changing requirements, but it can not be increased above the configured maximum unless the system is regenerated.

Floating Point Precision and Scaled Arithmetic

The system manager can select either single precision (2-word) or double precision (4-word) floating point numeric format. If the system

has floating point hardware, the system manager can select a floating point math package that will increase processing speed by using the hardware instructions. The scaled arithmetic feature is included in all 4-word floating point math packages. Scaled arithmetic avoids loss of precision in floating point calculations; it is therefore very useful in calculating sums of money that cannot be manipulated easily as integer quantities.

System-Wide Logical Names
RSTS/E allows the system manager to assign up to 50 logical names on a system-wide basis. Any user can type a system-wide logical name to access the device (and, optionally, the account) it represents.

File Processor Buffering
The optional file processor (FIP) buffering module accelerates file processing on the RSTS/E system. The module reduces the number of accesses to disk by maintaining more than one disk directory block in memory. The system manager can enhance FIP buffering by allocating additional memory to extended buffer space for use as a cache for disk directory blocks.

System Initialization
After generating the system, the system manager bootstraps the RSTS/E system to load the initialization (INIT) code into memory. The INIT code is a collection of routines used to create the file structures, system files, and start up conditions required for normal operation of the RSTS/E system. The INIT code is essentially one large stand-alone program with many functions. Immediately after a system generation, several options must be used before the RSTS/E system can be brought up for timesharing. Thereafter, the initialization code provides the mechanism for altering critical system files and parameters as installation requirements change. INIT includes routines which ensure the integrity of disk file structures and perform many checks on the hardware configuration. Options are provided which enable the system to function even when certain hardware elements are inoperative. Finally, the initialization code is responsible for loading the RSTS/E Monitor into memory for normal timesharing operations.

Once the default system initialization and start-up parameters are set up, the system manager does not have to repeat manual start-up each time the system is started. Using the automatic restart feature, the RSTS/E system can recover and restart the timesharing session automatically after a system malfunction or power failure. When the system is started in automatic restart mode, control by-passes all parts of the start-up code that call for operator intervention.

After system generation, however, or if the system manager chooses to reset system parameters, the system manager must run the initialization code options. Some of these options are:

HARDWARE   Reports on the survey of the hardware system taken when the system disk was booted. Also lets the system manager provide more information about the hardware (e.g., non-standard addresses of devices).

DSKINT   Initializes a disk cartridge or disk pack to contain a RSTS/E file structure and removes bad blocks from the user available space on the volume.

REFRESH   Creates or rebuilds the system files in account [0,1] on any initialized RSTS/E disk; rebuilds the storage allocation table for a disk; adds blocks to the BADB.SYS system file, the system list of bad blocks.

DEFAULT   Establishes or changes the default start up conditions such as the maximum number of jobs which can be run and the maximum size of a job. It also allows the user to change the system default run-time system or to put it into high-speed semiconductor memory if it is available.

START   Brings the RSTS/E system up for normal timesharing operations. In addition, START allows the user to set the maximum number of jobs, maximum job size and memory relocation parameters to override the DEFAULT specifications for this timesharing session only.

System Management Utility Programs
RSTS/E includes system utility programs for both the system manager and general user. Some system management utilities are privileged programs and can be run only by the system manager. Other utilities are not privileged and can be run by the general user, but have privileged features that can be executed only by the system manager.

System management utilities include: initialization and maintenance programs, resource management and accounting programs, system error logging and analysis programs, operator services and spooling programs, and user communication programs.

System Initialization and Maintenance

INIT   Controls system startup operations. This includes mounting disks, adding system files, defining CCL commands, establishing auxiliary run-time systems, setting terminal characteristics, enabling LOGINs, starting system programs, and sending messages to terminals. The system manager can create control files that perform timesharing start-up automatically.

SHUTUP   Performs an orderly system shut down operation.

UTILTY   Allows the system manager to: enable/disable LOG-INs, broadcast messages, kill, suspend, or detach a job; reset system date and time; enable and disable disk caching; mount and dismount private disks; add and remove system files; lock and unlock disks; clean disks; zero user accounts; control run-time systems; add and delete CCL commands; and add and remove system logical names.

TTYSET   Sets terminal characteristics.

SYSTAT   Monitors system status, including active jobs, device assignments, auxiliary run-time systems and detached jobs.

VT5DPY   or   Displays the system status on a VT05, VT50, or VT52
VT50PY   DECscope and updates the status at given intervals.

PRIOR   Reports and allows the system manager to change the priority, run burst and maximum size assigned to an existing job.

Resource Management and Accounting

DSKINT   Initializes a disk for use on a RSTS/E system.

REACT   Creates or deletes user accounts on disks.

UMOUNT   Allows the user to mount or dismount disk packs and magtapes.

SYSCAT   Prints a current directory listing of any disk.

MONEY   Extracts system accounting information for any selected account or all accounts; accounting information includes amount of CPU time used, the KCT factor (use of 1K words of memory for one-tenth of a second), amount of connect-time, device usage time, and disk storage usage.

Operator Services and Spooling Programs

OPSER   Establishes interjob communications on which the controlled (on-line) programs QUEMAN, SPOOL, BATCH and BACKUP depend. Provides the means by which an operator can interact with the controlled jobs. Defines a terminal (the operator services console — OSC) on which OPSER broadcasts information.

QUEMAN   Manages the queuing of jobs to spooling programs. Collects queue requests, maintains a file of all pending requests and a table of all on-line spooling programs.

SPOOL   Handles requests made for line printer output and maintains communications paths with both OPSER and QUEMAN.

BATCH   Executes files containing batch job commands that have been queued on a batch device and maintains communication paths with both OPSER and QUEMAN.

Error Logging and Analysis

ERRCPY   Retrieves error-related data logged automatically by the RSTS/E monitor. Upon occurrence of a hardware error, monitor routines save the contents of the device registers and send a message to ERRCPY to retrieve the data and store it in a specially formatted disk file (the error logging file).

ERRINT   Initializes and validates the error logging file.

ERRDIS   Produces summaries of error-related data and formats them for output to a terminal or line printer. Allows the system manager to obtain a summary or detailed report of the error-related data preserved by the ERRCPY program; to zero the contents of the error logging file; or to obtain a list of potentially bad disk blocks.

ANALYS   Retrieves and reports on the critical contents of memory obtained when a system crash occurs.

ODT   Allows the system manager to open a file, a device, or memory as an address space and examine or change word or byte contents. The user can also list the contents of system table locations.

User Communication

GRIPE   Allows the general user to communicate comments about the system to the system manager.

PLEASE   Communicates directly with the operator services program OPSER and, when OPSER is not running, sends text to the system console terminal (KB0:). Operators may run PLEASE to send commands to OPSER. Users who are not valid operators may run PLEASE to send text to the operator services console (OSC).

TALK   Enables users to broadcast messages to other users terminals.

DEVICE AND FILE CONVENTIONS

RSTS/E provides a device access structure that allows many users to share the resources of the system in a consistent manner. This section describes the device and file naming conventions, the public and private disk structures, and the account system used by RSTS/E.

File Specifications

The file specification for any user-identifiable collection of data is completely described by some or all of the following information:

   dev:[proj,prog]filnam.ext<prot>/option(s)

where "dev:" is a physical or logical device name, "[proj,prog]" is a user account number, "filnam" is a user-specified file name, ".ext" is a file name extension, "<prot>" is a file protection code and "/option(s)" is one or more file specification options.

For non-file structured devices such as paper tape, line printer, or terminal devices, only the device designator is required in a file specification. For file-structured devices such as disk, DECtape or magnetic tape, RSTS/E requires that the user at least specify a file name in addition to the device designator. File name extension, account number and protection code all have system defaults, and need only be specified if the system default is not to be used to identify the file.

RSTS/E recognizes the following default extensions:

.B2S	BASIC-PLUS-2 source file
.BAC	BASIC-PLUS compiled program (binary format)
.BAS	BASIC-PLUS source program file (ASCII format)
.CBL	COBOL source program file (ASCII format)
.CMD	Indirect command input file for running a system program
.CTL	Batch control file containing batch commands
.DAT	Data file
.DIR	Directory file
.FOR	FORTRAN source program file (ASCII format)
.LOG	Batch output log file
.LST	Listing file
.MAC	MACRO source subprogram file (ASCII format)
.OBJ	Compiled or assembled object program file (binary format)
.ODL	Overlay Description Language input file
.SAV	Executable program file (binary format)
.TMP	Temporary file created by a system program
.TSK	Executable program file (binary format)
.WRK	Utility program work file

The account number field (containing the project and programmer numbers) identifies the owner of the file. If it is omitted, the owner is assumed to be the current user. This field is meaningful only for disk and magtape files; it has no significance for DECtape files or files on non-file structured devices.

The account number can be represented by special characters to indicate special system or user-defined accounts. For example, use of the $ character (dollar sign) in the project-programmer field indicates that the file is stored under the system library account ([1,2]), where all standard utility programs are stored. Other special account number characters are:

!	Account [1,3] or installation-defined account
%	Account [1,4] or installation-defined account
&	Account [1,5] or installation-defined account
#	Account [n,0] where "n" is the current account project number
@	Assignable account

The accounts associated with the !, % and & characters can be changed during system installation. The # character is unique because the system interprets it according to the account under which the user is running. For example, if the user is running under account [10,20] and specifies the # character, the system interprets it to mean account [10,0]. This feature allows each project on the system to have its own library of files.

When creating or renaming a file, a protection field can be specified. Files can be protected against reading, writing, and deleting for three classes of users where distinctions are made on the basis of the project and programmer number of the user attempting to access the file. The three classes of users are:

owner	the individual user
project group	all users having the same project number as the owner (termed the owner's project group)
others	all other users not in the owner's group

The protection code assigned to a file consists of a selected sum of the following numbers:

1	Read protect against owner
2	Write protect against owner
4	Read protect against owner's project group

8	Write protect against owner's project group
16	Read protect against all others
32	Write protect against all others
64	Executable program; can be run only
128	Program with temporary privileges (normally occurs only when file's protection includes <64>)

For example, in creating a compiled BASIC-PLUS file, a default pro-tection code of <124> is supplied. This permits only the owner to access the file, since 124 = 64 + 32 + 16 + 8 + 4.

However, when any of the above protection codes are combined with code 64, they may take on different meaning. If code 64 is combined with code 1, for example, the new meaning is "Read/Write protected against owner."

A file specification option or options may be included as the final element of the specification. These options may specify the size to which a disk file is pre-extended, the minimum number of contiguous disk blocks forming a cluster, and the read/write mode in which the file's data is passed to the device driver.

System Accounts and Libraries

RSTS/E systems have three system accounts that are integral to the operation of the system and have auxiliary accounts for more efficient operation of the system. The MFD account is used on the system device and other disk devices in the system to control system access. The system library account is used by the RSTS/E system to manage a library of generally available and restricted use system programs and message and control files. A third special system account contains RSTS/E Monitor files and routines which are critical to the operation of the system.

Of particular interest to the system manager is the accounting infor-mation maintained on each user account in the MFD on the system device. This accounting information is normally accessed through the system accounting utility programs. The system manager or privileged users can also access and change this information in BASIC-PLUS using the SYS monitor functions. Table 5-2 summarizes the account-ing information maintained in the MFD.

Table 5-2 Account Information Stored in the MFD on the System Device

TYPE	DESCRIPTION	EXPLANATION
Identification	Project-program-mer number	Account number under which a user logs in and creates files.
	Password	Password required to gain access to the system.
Accumulated Usage	CPU time	Processor time the account used to date
	Connect time	Number of minutes the user has been connected to the system via a terminal or remote line.
	Kilo-core ticks	Core use factor. One KCT is the usage of 1K words of memory for one tenth of a second.
	Device time	Peripheral device time the account has used.
	Disk Storage Quota	Number of 256-word blocks the user is allowed to retain at logout time.

Privileged Capabilities and System Operation

Privilege is a special condition for a user job. With privilege, a job has capabilities not available to other, nonprivileged jobs. These capabilities are:

• unlimited access on the system

• ability to designate privileged programs

• use of privileged aspects of system programs

- use of privileged SYS system functions and the PEEK function

A job has privilege under one of the following conditions:

- It is a logged-out job (a job without an account).
- It is running under a privileged account.
- It is running a privileged program.

A logged-out job has privilege because the system must perform certain privileged operations to log a job in to the system. The privilege remains in effect as long as the job remains logged out.

A job running under a privileged account has privilege. A privileged account is one whose project number is 1. The system library account [1,2] is an example of a privileged account. Such a job running under a privileged account has permanent privilege. The privilege remains in effect until the job is logged out or the job changes to a nonprivileged account (one whose project number is not 1).

A privileged program is an executable file with a protection code of <192> (the sum of the privileged protection <128> and the compiled file protection <64>) or greater. A job running such a privileged program has temporary privilege unless it is running under an account which has permanent privilege. The job gains the temporary privilege when it runs a privileged program. The privilege remains until the pogram exits or until the program drops its temporary privilege.

This last type of privilege is necessarily temporary because users of both privileged and nonprivileged accounts may be able to run a privileged program. If the privilege were not temporary, an unexpected halt in the job would leave the system vulnerable to unwarranted tampering.

A temporarily privileged job can rely on the normal protection mechanisms built into the system. Under programmed control, the job can either permanently or temporarily relinquish (drop) its temporary privilege. This ability allows a job to perform privileged operations selectively. For example, a job could set itself up initially using privileged capabilities and then drop its privilege permanently because further processing does not require privilege. Alternatively, a job could temporarily drop and later regain its temporary privilege depending on the type of processing required.

The following paragraphs summarize privileged capabilities.

Unlimited Access
No file in the RSTS/E system can be protected against a privileged job. A privileged job can create and delete files under any account

number on any disk. Such unlimited access does not generate the normal PROTECTION VIOLATION error.

Ability to Designate Privileged Programs

A program is privileged when it is an executable file and has a protection code of <192> or greater. Only the system manager or other users running under privileged accounts can create or modify privileged programs.

Use of Privileged Features of System Programs

If a program is designated privileged and is not protected against execution, any user can run the program with temporary privilege. Temporary privilege means that system operations normally reserved to a user of a privileged account can be executed while running under a nonprivileged account.

The ability to designate a program as privileged allows the system manager to extend use of privileged functions to non-privileged users. For example, the program TTYSET allows general users to change characteristics of their terminals. Such an action is a privileged system function executable only by owners of privileged accounts. With temporary privilege, however, execution of the function by the owner of a nonprivileged account does not generate the normal PROTECTION VIOLATION error.

The same TTYSET program additionally allows a privileged user to change characteristics of other terminals. A check is built into the program to ensure that a user attempting to change the characteristics of a terminal other than his own is indeed a permanently privileged user. In effect, the execution of some privileged functions is made available to the nonprivileged user but other privileged features are available only to those users logged into the system under privileged accounts.

USER INTERFACE

This section describes the system facilities available to the general user, including the system and installation-defined commands, the system utility programs and the batch processing commands.

System and Installation-Defined (CCL) Commands

The RSTS/E system commands issued by the user at a terminal are easy-to-use English words or abbreviations. The system accepts both long and short command formats for inexperienced and experienced users. It responds with understandable statements and, if a command does not supply complete information, prompts the user for remaining data.

RSTS/E system commands include the following:

Table 5-3 lists the standard system commands.

Table 5-3 RSTS/E System Commands

Login/logout Commands

HELLO LOGIN LOG I ATTACH, ATT	These commands allow the user to log in to the system by specifying an account number and password. The system also notifies the user what job number is assigned, whether any other users are logged into the system under the same account, and what, if any, jobs are running detached under the account. The user can choose to attach to a detached job. If the user is already logged in and issues a HELLO command, the user can change accounts or attach to another job without logging off the system.
BYE	Allows the user to log off the system. Checks the user's disk quota to ensure that the user does not exceed the limit allowed by the system manager. Closes and saves any files remaining open.

Device Assignment Commands

ASSIGN	Allows the user to reserve a device for use by a single job, associate one or more logical names with a particular device, assign a specific account to the assignable account number "@," or change the default protection code given to files created under an account.
DEASSIGN	Allows the user to release a device or all devices previously reserved for user by a job, cancel a logical name for a device, or cancel the association between the @ account and a specific account.
REASSIGN	Transfers the control of a particular device to another job.

Program Execution and File Manipulation Commands

RUN	Executes a specified compiled program. If the program does not exist as a compiled program, RUN loads the BASIC-PLUS source program into the job area, compiles and runs it.
RUNNH	Executes the program currently in the job area.

CONT   Restarts execution of the program currently in the
                    user's job area where it was interrupted (either by a
                    STOP statement in the program or a CTRL/C issued
                    from the terminal).

CCONT   Same as CONT but detaches job.

In addition to the standard commands, some system programs can be
run by typing a unique system command called a Concise Command
Language (CCL) command. CCL commands allow a user to enter one
command that runs a system utility and specifies a single command
for the utility to execute. The number of CCL commands which can be
defined varies from system to system, depending on the number of
"small buffers" configured into the system. An average system prob-
ably includes a fairly standard set of CCL commands for certain
RSTS/E utility programs. The system manager has the option of freely
adding to, deleting from, or modifying the standard set of CCL com-
mands.

The precedence of CCL commands is above that of RSTS/E com-
mands and BASIC-PLUS immediate mode statements. As a result, the
system manager can control the use of a command or immediate
mode statement. For example, the system manager could define a
CCL command named BYE that performs certain operations before
allowing a user to log off the system. As another example, the system
manager could define a PRINT command that performs operations
different from those of the BASIC-PLUS immediate mode PRINT
statement. The CCL command has no effect on a BASIC-PLUS state-
ment preceded by a line number since numbered lines can contain
only valid BASIC-PLUS statements.

The user types the CCL command and the program command on one
line and enters it to the system. For example, the user can run the PIP
system utility to print a copy of a file on the line printer in either of two
ways:

1. RUN $PIP   The user issues the RUN command for the PIP pro-
                    gram stored in the system library account ($ = [1,2]).

LP:=FILE   PIP requests a command by printing "". The user
                    issues the request to print a copy of FILE on the line
                    printer. When PIP finishes the request, it prints
                    another "*" to prompt another command.

*↑C   The user types a CTRL/C to terminate PIP and re-
                    turn to system command level.

READY	The system prints READY on the terminal to indicate that it is ready to accept a system command.
2. PIP LP:= FILE	The user issues the CCL command PIP to run the PIP program and issues the request to print a copy of FILE on the line printer.
READY	When PIP finishes executing the request, the system prints READY on the terminal to indicate that it is ready to accept a system command.

Although CCL commands are installation dependent, DIGITAL defines a standard set of commands which are listed below. Note that the UMOUNT system program is designed to be run only through the MOUNT and DISMOUNT CCL commands.

CCL Command	Associated Program
ATTACH	LOGIN
BYE	LOGOUT
CREATE	EDIT
DISMOUNT	UMOUNT
DIRECTORY	DIRECT
EDIT	EDIT
HELLO	LOGIN
HELP	PIP
LOGIN	LOGIN
MOUNT	UMOUNT
PIP	PIP
PLEASE	PLEASE
QUEUE	QUE
SET	TTYSET
SYSTAT	SYSTAT
UTILTY	UTILTY

In addition to the system commands and CCL commands, RSTS/E supports the following special control character commands:

CTRL/C	Stops any current program execution and returns the system to command mode.
CTRL/O	Suppresses or enables output to the user terminal.
CTRL/S	Suspends output on a terminal until a CTRL/Q is received.
CTRL/Q	Resumes output interrupted by a CTRL/S.
CTRL/U	Deletes the current line entered.

CTRL/Z   Used as an end-of-file character.

CTRL/R   Retypes current terminal input line.

General System Utility Programs

In addition to the system management utility programs, RSTS/E includes several utility programs available to the general user. These programs include system information and terminal utility programs, file utility programs, and special service programs. Like the system management utilities, they are stored in the system library account and are called and executed by issuing the RUN system command or, if it is available, the appropriate CCL command.

General system utilities include the following:

System Information Programs

SYSTAT   Provides current system information concerning job, device, and buffer status. This includes identifying the active jobs in the system, the accounts under which they are running, their size, their associated keyboard if attached, and their current activity. It also identifies which devices are assigned and to which job they are assigned.

QUOLST   Provides current system information, including the number of free blocks remaining on the system structure, the number of blocks used by an account, the number of free blocks remaining in an account, and its disk quota.

MONEY   Prints the current account status, including the amount of CPU time, connect time, kilo-core ticks and disk blocks used.

GRIPE   Allows the user to communicate comments to the system manager.

TTYSET   Allows a user to establish terminal characteristics for the terminal. The user can call a macro command that establishes the standard characteristics for a selected type of terminal or select an individual combination of characteristics.

INUSE   Prints the message "IN USE" at a terminal to allow a user to leave the terminal momentarily.

File Manipulation Programs

EDIT   Allows the user to create or modify text or program files.

PIP   Allows the user to transfer files from one device to another, merge files, delete files, zero a device directory or list a device directory.

COPY   Copies all the information on a disk, DECtape or magtape device.

BACKUP   This comprises a package of programs which allow the user to preserve and recall files stored under one or more user accounts by transferring multiple files from the private or public disk structure to a private disk, a DECtape or a magtape.

DIRECT   Prints directories of selected file-structured devices.

FILCOM   Compares two text files line by line and prints any differences found.

Special Service Programs

MAC
MACRO   Assemble MACRO-11 source code into object format. MAC operates under the RSX-11 run-time system; MACRO operates under the RT-11 run-time system.

LINK   Links object modules produced by FORTRAN or MACRO into an executable image which runs under the RT-11 run-time system.

TKB
(Task Builder)   Builds an executable image by linking object modules produced by the MAC assembler or language processors other than FORTRAN. The resulting task image runs under the RSX run-time system specified by the user.

QUE   Creates jobs that are to be executed by spooling programs such as BATCH and SPOOL. It also lists pending requests and kills pending requests.

RUNOFF   Generates a formatted listing of a text file containing special RUNOFF text format commands.

Batch Processing

The capability to execute a batch of commands allows the user to submit jobs to be run without terminal dialog. Batch processing is

particularly useful in executing large data processing operations for which interactive requirements are not a factor.

Batch input can be submitted from standard job control files on a random access file-structured device or from an I/O device such as the card reader. Such input consists of elements of the batch control language and is collectively referred to as a batch stream. It is possible to execute multiple streams simultaneously by running multiple copies of the BATCH program. The capability to run more than a single batch stream is controlled by the system manager.

To request the running of a batch job, the user runs the library program QUE and specifies the batch control file or files as in the following example:

```
RUN $QUE
QUE Vnnnn - RSTS Vnnnn
#Q BA:BATJOB=FILE1,FILE2,FILE3.DAT
#
```

Or, if QUE is available as a CCL command:

```
QUE BA:BATJOB=FILE1,FILE2,FILE3.DAT
```

The user normally queues a batch job to device BA:. The job and log files in this example will be named BATJOB, and the files FILE1.CTL, FILE2.CTL, and FILE3.DAT will be concatenated to form the batch control file. The log file BATJOB.LOG will be printed after the job is complete.

The BATCH command set consists of the following control commands:

$JOB	Marks the beginning of a job and assigns a job name.
$EOJ	Marks the end of a job.
$BASIC	Calls the BASIC-PLUS compiler to compile a source program.
$RUN	Executes a specified program.
$CREATE	Creates a file consisting of data in the input stream.
$DATA	Marks the beginning of an input stream.
$EOD	Marks the end of an input stream.
$DELETE	Deletes a specified file.

$COPY   Copies a specified file.

$PRINT   Prints a specified file on the line printer using the
                spooler.

$DIRECTORY   Produces a directory listing.

$MESSAGE   Logs a message on the system console terminal.

$MOUNT   Requests system operator to logically mount a de-
                vice.

$DISMOUNT   Cancels a logical device assignment and requests
                operator to dismount a device.

Data Formats

Under BASIC-PLUS, RSTS/E allows users to store data in any of three
formats.

STRING   A sequence of ASCII characters treated as a unit.
                One ASCII character is stored in one byte and
                strings are normally variable length.

INTEGER   A number in the range -32768 to +32767. Integers
                are stored in two bytes in 2's complement represen-
                tation. Integer operations provide economies in
                space as well as increases in processing speed over
                floating-point operations.

FLOATING   A number approximately in the range of 10^{-38} to 10^{38}.
POINT   Floating point numbers can be stored either in 2-
                word format, which allows up to seven decimal digits
                of precision, or 4-word format, which allows up to 17
                decimal digits of precision.

To perform decimal calculation on a system having 4-word floating
point numeric storage, the user has an option to scale the numbers
stored in the system. The user can specify the number of decimal
places in fractional numbers by use of the SCALE system command.

With the scaled arithmetic feature, the scale factor can be set to an
integer value between 0 and 6. The system uses the scale factor to
preserve the accuracy of fractional numbers to the selected number of
decimal places. The value 0 is used to disable the scale factor, and
allow the system to perform calculations using standard double preci-
sion floating point arithmetic.

With a scale factor between 1 and 6 in effect, the system, upon input of a floating point number, internally moves the decimal place the selected number of places to the right and rounds it to an integer. The system performs all subsequent calculations with the floating point integers and, in turn, translates the result of each arithmetic operation into a floating point integer with the selected scale factor. On output the system moves the decimal point to the left of the selected number of places and passes the result to the output format routines.

Scaled arithmetic conversion thus avoids the loss of precision inherent in representing fractional numbers in binary notation, since the system can represent the integer accurately in floating point format.

File Access Techniques

Under BASIC-PLUS, RSTS/E provides three methods of file access:

Formatted ASCII   For standard sequential I/O operations.

Virtual Arrays   For random access of large data files. A virtual array is stored on disk and can contain string, integer and floating point matrices.

Record I/O   Allows the user to have complete control over I/O operations.

Formatted ASCII data files are the simplest method of data storage, involving a logical extension of the BASIC-PLUS PRINT and INPUT statements. The INPUT statement allows data to be entered to a running program from an external device, for example; the user's keyboard, a disk, DECtape, or paper tape reader. The PRINT statement causes the output of a specified string of characters to a selected device.

The PRINT-USING statement allows the user to control output formatting. A special set of formatting characters allows the user to format strings and numeric fields with tabs, special characters and punctuation. For example, the user can format check amounts with asterisk-fill for protection.

The RSTS/E virtual array facility provides the means for a BASIC-PLUS program to operate on data structures that require fast random access processing yet are too large to be accommodated in memory at one time. To accomplish this, RSTS/E uses the disk file system for storage of data arrays, and maintains only portions of these files in memory at any given time.

All references to virtual arrays are ultimately located via file addresses

relative to the start of the file. No symbolic information concerning array names, dimensions, or data types is stored within the file. Thus, different programs may use different array names to refer to the data contained within a single virtual array file.

Virtual arrays are stored as unformatted binary data. This means that no I/O conversions (internal form to ASCII) need to be performed in storing or retrieving elements in virtual storage. Thus, there is no loss of precision in these arrays, and no time wasted performing conversions.

Any data element in a virtual array is completely contained within a single element (256 words) of disk storage. This restriction has no effect on integers and floating-point items, where the size of data items is fixed (1-word integer, 2- or 4-word floating point numbers), but does limit the maximum length of a virtual string to 512 characters (512 bytes). The number of data elements stored in each disk segment is a function of the size of each element.

Strings in virtual storage occupy pre-allocated space in the virtual file, and thus differ from strings in memory, where space is allocated dynamically. A disk segment containing virtual strings can be considered to be a succession of fields, each of the maximum string length. When a virtual string is assigned a new value, it is stored left-justified in the appropriate field. If the new string value is shorter than the maximum length, the remainder of the field is filled with zeros. When the string is retrieved, its length is computed as the maximum string length minus the number of zero-filled bytes.

The third type of I/O, record I/O, permits a program to have complete control of I/O operations. Record I/O is the most flexible and efficient technique of data transfer available under BASIC-PLUS, although it is less simple to use than formatted ASCII I/O or virtual array I/O.

Input and output to record I/O files is performed by the BASIC-PLUS GET and PUT statements. These statements allow the user to read or write specific blocks (physical records) of a file, where the block size is dependent on the type of device being accessed. For example, disk file blocks are always 512 bytes long, while records from a keyboard device are one line long, where a line is delimited by a carriage return or similar terminating character. With disk files, the program has the capability of performing random access I/O to any block of the file. Furthermore, using record I/O operations, the user can create a logical organization for file formats by controlling record length.

Normally, the system permits only one user at a time to have write privileges on any given file, to prevent loss of data if two users try to

write the same block of a file. However, in certain applications (for example, sales order-entry applications) it might be normal for several users to be updating a single master file. For this reason, a special UPDATE option is available with RSTS/E Record I/O operations that permits multiple users to have write access to a file while guarding against simultaneous writing of a single physical record. In this case, write privileges are gained on a record-by-record basis, and no two users can have write access to the same record simultaneously, although multiple users can open the file for write operations.

Logical Disk Structures

Access to all executable code and to system and user data on the RSTS/E system is accomplished through a logical structure of files.

The logical disk structure is divided into two types: public and private. The file structure on a disk, whether it is designated public or private, is the same.

A public disk is a disk on which any user can create files. Every user has an account on a public disk. There is always at least one public disk on the system, which is called the "system disk." All public disks together on a system are called the "public structure" because the system itself treats all of the public disks together as a unit. For example, when a program creates a file in the public structure, that file is placed on the public disk with the most space available. This is done to ensure proper distribution of files across the disks in the public structure. The actual determination of which disks on a particular system are public and which are private is left to the system manager. Therefore, this allocation will vary from system to system.

The system disk contains the system code. Language processors and the library of system programs are contained on the public structure. Storage of active user jobs which are temporarily swapped out of memory are in swapping files, at least one of which is on the system disk. When a system includes one or more fixed head disks in its configuration, it is frequently advantageous to put some swapping files on a fixed head disk. Remaining space on the system disk and all space on other public disks is available for general storage of user programs and data files.

Any remaining disk drives in the RSTS/E disk structure can be devoted to private disk packs or disk cartridges. A private disk is one that belongs to a few user accounts, conceivably to a single user account. Files can be created only under these accounts, and can be read (or written) by other users only if the protection code of the file permits. A user who does not have an account on a private disk cannot create a file on it.

113

Private disks are always referenced by a physical or logical device name, for example, "DK1:" for the RK05 disk drive unit 1, or "CREDIT:" for the device assigned the logical name CREDIT. The public structure is normally referenced by default; when no device name is given, the system assumes the public structure. It also has the specific name "SY:". The system will not allow two files of the same name to exist in the public structure for a single user.

All public disks must be physically on-line and logically mounted whenever the system is running and must be accessible to all users during timesharing operations. Private disks can be logically mounted and dismounted and interchanged as needed during timesharing operations.

Control of and access to files in the RSTS/E system is accomplished by two structures called a Master File Directory and a User File Directory. A Master File Directory, or MFD, exists on each disk initialized for use on the RSTS/E system. The MFD is treated as an account on the disk, has a project-programmer number [1,1], and catalogs other accounts on the disk. The MFD on the system disk is a special case, since it maintains a catalog of the accounts which can be used to log in to the system. MFD accounts on other disks contain entries of accounts which can create files on that disk. Any user gains access to any file on a private disk if the protection code of the file permits. However, only those users whose accounts are entered in the MFD of the private disk can create files on the disk.

A user File Directory, or UFD, exists for each account under which files are created. The UFD contains accounting and retrieval information for each file stored under that account. A UFD for an account on a public disk is not created until a file is created under that account on that disk.

SYS SYSTEM FUNCTIONS AND THE PEEK FUNCTION
SYS system function calls allow a user program written in BASIC-PLUS to perform special I/O functions, to establish special characteristics for a job, to set terminal characteristics, and to request execution of special monitor operations. The function calls are available in the BASIC-PLUS and BASIC-PLUS-2 languages. They are system dependent and their format allows a variable number of parameters through the use of concatenated strings of binary values.

There are twelve SYS functions. With one exception, all the functions can be called by nonprivileged user programs. A special SYS function can be used to issue calls to FIP, the file processor. SYS calls to FIP allow the user to select a FIP function. Some of the FIP functions can be called only by privileged user programs.

The twelve SYS functions are listed below:

Cancel CTRL/O	Cancels the effect of the user's typing a CTRL/O on a specified terminal.
Enter tape mode	Disables the terminal echo feature (useful when reading a paper tape with the low speed teletypewriter paper tape reader).
Enable echoing	Reverses the effects of an enter tape mode function call or a disable echoing function call.
Disable echoing	Prevents the system from echoing information typed on a specified terminal. For example, information such as a password is not displayed but is accepted as input by the system.
Enable delimiterless character input mode	Allows less than a full line to be accepted as input from the terminal. Normally, the system waits until a line terminated by a carriage return, line feed, form feed, CTRL/D combination or escape character has been typed before accepting input. In delimiterless character mode, one or more characters typed at the terminal are passed immediately to the program by the next keyboard input request statement without waiting for a delimiting character.
Exit to Editor with no prompt message	Exits from the program but does not clear the program from memory, does not print a prompting message and does not close files. Thus, this exit allows the user to continue running the program.
Get core common	Allows a program to extract a single string from a data area loaded by another program previously run by the same job. The data area is called the core common area.

Put core common | Allows a program to load a single string in a string common data area called core common. This string can be extracted later by another program, running under the same job and called by the CHAIN statement. This function allows a program to pass a limited amount of information when a CHAIN statement is executed.

Exit and clear program | Clears the current program from memory and returns control to the user's private run-time system. Optionally, transfers control to a specified run-time system and establishes it as the job's private default run-time system. Cancels all type-ahead. Returns information on last open file.

Call to FIP | Causes a dispatch call to the system file processor.

Cancel all type-ahead | Allows a program to clear all unsolicited input from a terminal's buffers. This is particularly useful for screen-oriented applications where the echoing of unsolicited input would ruin the visual effect of "painted" templates.

Return information on last opened file | Allows a program to determine the device and account on which it is stored, or to determine where the most recently opened file resides.

FIP calls allow the user program to perform a variety of file, device, job and system operations. Nonprivileged user programs can issue the following FIP function calls:

Monitor Information Calls

Read Accounting Data | Reports the following accounting data for the program's account:

Account number — project number, programmer number
CPU time — amount of processor time used
KCT use — one KCT (kilo-core tick) is the use of 1K words of memory for one-tenth of a second

116

	Connect time — amount of time the terminal has been connected
	Device use time — amount of time spent using devices excluding the public disks
	Disk storage — number of disk blocks allocated
	Logout quota — number of disk blocks allowed to retain at logout time
Get Monitor Tables	Reports Monitor information such as the number of configured terminals, maximum number of jobs, address of the memory allocation table, address of the job status table, etc.
Return Error Message	Extracts the error message text corresponding to an error code.

Device Assignments

Assign/Reassign Device	Reserves an I/O device for use by a job, if it is available. Reassign transfers device control to another job.
Deassign a Device	Releases a device to the device pool for use by other jobs.
Deassign all Devices	Releases all devices previously assigned to a job.

Directory and File Control Calls

Filename String Scan	Determines whether file naming syntax is valid. For example, it can check whether a given file name is valid.
Directory Lookup on Index	Searches for and reports a directory entry by its index position in the directory.
Magtape Directory Lookup	Searches for and reports a directory entry on a magnetic tape device.
Disk Directory Lookup	Searches for and reports a directory by filename entry on a disk device for a specified file.
Disk Wild Card Directory Lookup	Searches for and reports directory entries on a disk device for all files with (a) specified character(s) occurring in the file name or extension.

Job Control Call

CTRL/C Trap Enable   Allows a program to control processing when a CTRL/C is typed on the terminal.

Communications Call

Send a Message   Allows a job to send a message to an eligible receiving job.

FIP calls that can be used by privileged programs are:

Monitor Information Calls

Read or Reset Accounting data   Allows a program to reset accounting data for any job after reading the data.

Accounting Dump   Allows a program to dump accumulated accounting data.

Change Date and Time   Changes the date and time values maintained by the system.

Job Control Calls

Change Password or Quota   Allows a program to change a user's password or logout disk space quota.

Change priority, run burst or maximum size   Allows a privileged user to give a running job an increased or decreased chance of gaining run time in relation to other running jobs, and to determine how much CPU time the job can have if it is compute bound.

Set Special Run Priority   Allows a program to raise the priority of a job slightly above that of other jobs in its priority class.

Lock/Unlock Job in Memory   Prevents unnecessary swapping by forcing the job executing the call to remain in memory. The call eliminates swapping time between run bursts.

Drop and Regain (Temporary) Privileges   Allows an executable program to either temporarily or permanently drop temporary privileges. A program normally issues this call after it has used temporary privilege to set itself up. If a program temporarily drops its temporary privilege, it can use this call to regain the privilege.

Create Job	Creates a new job and causes it to run a specified program.
Kill Job	Terminates a job under program control.
Login	Logs a job in to the system.
Logout	Logs out a job that was initiated by a user at a terminal.
Detach	Dissociates the calling job or another specified job from its terminal. This frees the terminal for other use and makes the noninteractive job immune from interruption by someone typing a CTRL/C at the terminal.
Reattach	Attaches a detached job to a terminal.
Set Terminal Characteristics	Performs the same functions as the system program TTYSET. Allows a user to set lower case, baud rate, scope operation, etc., on a specified terminal.

System Control Calls

Set Logins	Sets the number of allowable logins to a specified number.
Enable Logins	Sets the number of logins allowed to the maximum number possible.
Disable Logins	Sets the number of logins allowed on the system to one. If no jobs are active on the system, one user can log in. Once one user is logged in, no other users can log in to the system. The exception is the console terminal, from which it is possible to log in despite having the number of logins restricted.
System Shutdown	Logs the current (and only) job off the system and bootstraps the initialization code from the system disk.
Broadcast	Allows the user program to print a message on another user's terminal.
Force Terminal Input	Allows the user program to force data entry on another user's terminal. The forced data is seen as input by the system.

File Management Calls

Create User Account — Allows the user program to create an entry in the MFD on a disk for an account.

Delete User Account — Allows the user program to remove a MFD entry for an account on a disk.

Change File Statistics — Allows the program to change a file's creation date or time or date of last access in the UFD entry for the file.

Set Disk Access — Allows the program to logically mount or dismount a disk pack and to lock or unlock a disk pack (allow or prevent access).

Clean a Disk — Rebuilds the Storage Allocation Table on a disk.

Communication Calls

Declare a Message Receiver — Notifies the system that an eligible receiver job is ready to receive messages. The system sets up a message queue and relays messages sent from other jobs when the program asks for a message.

Remove a Receiver — Notifies the system that a receiving job is no longer eligible to receive messages.

The PEEK function allows a privileged user to examine any word location in the monitor part of memory. The program can examine words in small or large buffers, in the resident portion of the file processor, and in the low core and tables section of memory. The function does not allow a user program to examine the contents of another user's program.

The PEEK function is normally used to examine either addresses returned by Get Monitor Table calls or addresses of fixed monitor locations.

RSTS/E SYSTEM SUMMARY

Is
- General purpose timesharing
- High performance timesharing BASIC
- Interactive environment
- Multi-language
- Batch processing
- Basis of most commercial applications
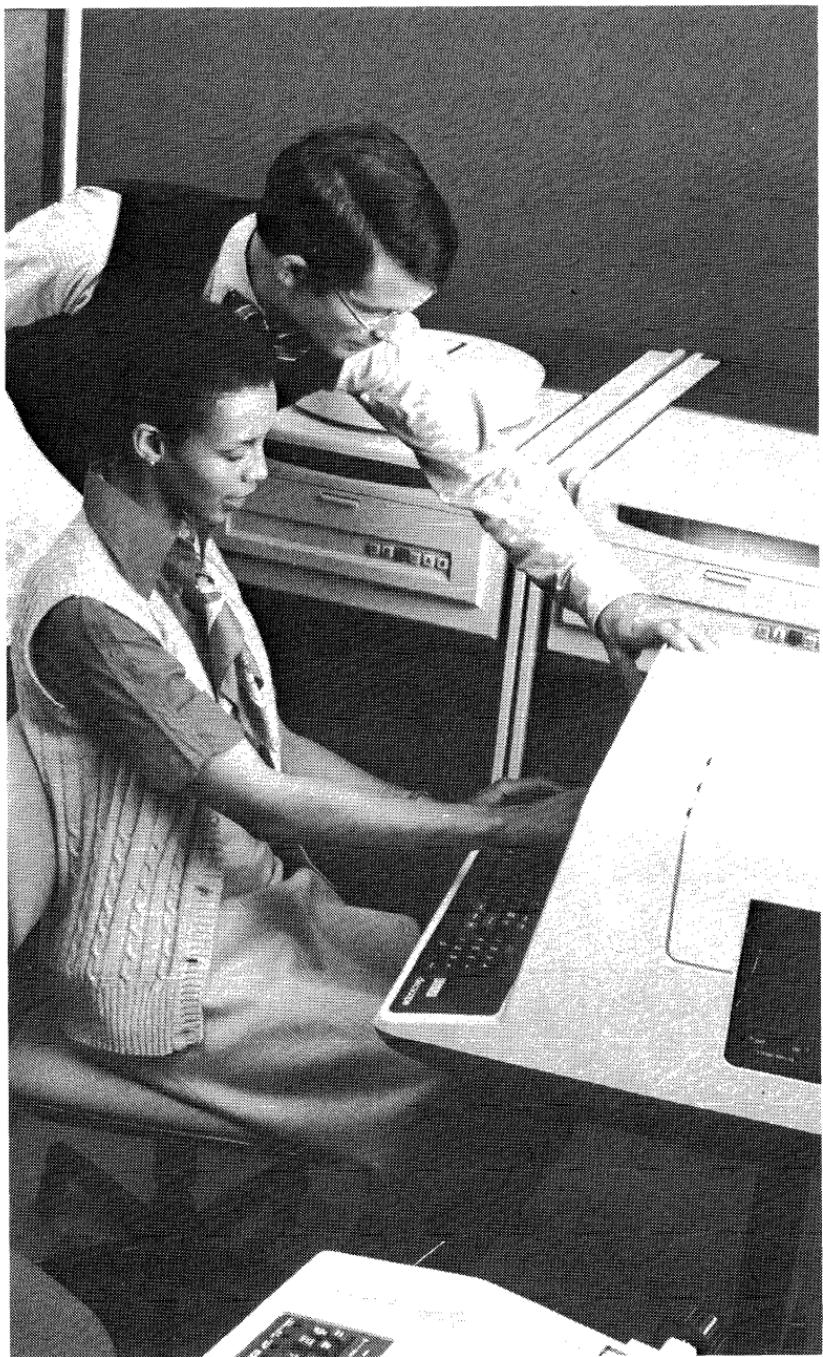
Is not
- Real-time
- High volume transaction processing
- Block mode application terminals

Includes Data Management/Utilities
- RMS-11
- SORT-11
- DATATRIEVE-11
- DMS-500

Languages
- BASIC-PLUS
- BASIC-PLUS-2
- COBOL
- FORTRAN IV
- MACRO-11
- RPG II
- DIBOL-11

REAL-TIME MULTI-PROGRAMMING SYSTEMS RSX-11M (V3.1) AND RSX-11S (V2.1)

OVERVIEW

RSX-11M is the primary PDP-11 real-time operating system. It supports multi-tasking, dynamic memory management, multiple programming languages, interactive program development and a wide range of equipment interfaces. Task scheduling in RSX-11M is event driven, in contrast to systems which use a static scheduling mechanism to determine a task's eligibility to execute. RSX-11S, a subset of RSX-11M, provides a dedicated execute-only environment for monitoring and controlling many real-time processes concurrently.

FEATURE TOPICS

- Functions and Features
  - Common RSX-11 Operating System Concepts
  - Multiprogramming
  - Priority Scheduling
- System Organization
  - RSX-11M Executive and Memory Structures
- RSX-11S System Components
- System Conventions
- Devices
- File Structures
- File Specifiers
- RSX-11 MCR Commands (Table 6-1)
- Indirect Files (Command Files)
- System Directives
- File Control Services (FCS)
- RMS-11 Record Management Services
- System Utility Programs
- RSX-11M System Summary

FUNCTIONS AND FEATURES

RSX-11 is a unique family of compatible real-time multiprogramming operating systems for the PDP-11 computers. The RSX-11 family includes RSX-11M, a compact, efficient operating system, and RSX-11S, a small, execute-only operating system for dedicated application environments. The RSX-11 operating systems comprise a compatible hierarchy. RSX-11S is a memory-based proper subset of RSX-11M, fully compatible internally. A program written to execute as a task under RSX-11S will execute under RSX-11M without change.

RSX-11M includes an executive, MCR services, FCS or RMS file system, and a complete set of system utility programs. Under RSX-11M, programs can be written in MACRO, FORTRAN IV or FORTRAN IV-PLUS, COBOL-11, BASIC-11, or BASIC PLUS II.

RSX-11M is a multi-user system. More than one terminal user can interface with the Monitor Console Routine (MCR) services simultaneously. An MCR facility allows users to create a file containing executable commands to control common sequences of operations. The MCR facility in RSX-11M systems also allows the user to create indirect command files using a procedure control language to effect a multi-stream batch capability.

RSX-11S requires a host RSX-11M, RSX-11D or VAX system for program development and system generation. Tasks can be written in MACRO, FORTRAN IV, or FORTRAN IV-PLUS, assembled or compiled, subsequently linked on the host system, and then transported to an RSX-11S system for execution. The minimum RSX-11S system includes an executive (with incorporated device drivers) and a special FCS that contains no support for file-structured devices. The user can also add a subset of RSX-11M's MCR services if the hardware configuration includes a terminal. If on-line task loading is desired, the user can include an On-line Task Loader (OTL) utility. If the user wants to save a system image for subsequent re-booting, the user can include the System Image Preservation (SIP) utility.

Since RSX-11S is a memory-only system, it does not support a file system, non-resident tasks, task checkpointing, dynamic memory allocation or program development. It does, however, support data storage on all devices supported by RSX-11M. Its purpose is to provide a run-time environment for the execution of tasks on a small system with a very modest complement of peripherals.

RSX-11M runs on any of the PDP-11 processors except the LSI-11. The minimum system requires a console terminal and either one of the larger disks plus a magnetic tape system, or an RK05 disk system with a secondary storage device. Without a Memory Management Unit, the

system can support between 16K and 28K words of memory. With memory management, memory can range between 24K and 124K words or up to 1920K words with a PDP-11/70. At least 24K of memory is required for concurrent applications execution and program development. A program or shared data area may be anywhere from 32 words in size to as large as the system memory size minus the size of the operating system.

The minimum configuration for an RSX-11S system is a PDP-11 processor (including the LSI-11) with at least 8K words of memory and one load device. At least 16K words are required for on-line task loading or the execution of tasks written in FORTRAN IV-PLUS. With a Memory Management Unit, memory can expand up to 124K words or 1920K words with a PDP-11/70.

The operating systems support a broad range of peripherals including card readers, line printers, fixed-head disks and a variety of laboratory, industrial control, and communications equipment. Note that although the maximum configuration for an RSX-11S system is the same as that for an RSX-11M, RSX-11S is a memory based system and does not support disks or magnetic tape as file-structured devices.

COMMON RSX-11 FAMILY OPERATING SYSTEM CONCEPTS

The RSX-11 family of operating systems is designed to provide a resource-sharing environment ideal for multiple real-time activities. The basic facilities that the RSX-11 family provides for handling multiple requests for services while maintaining real-time response to each request are:

- multiprogramming
- priority scheduling
- contingency exits
- power-fail shutdown and auto-restart

In addition, RSX-11M provides:

- disk based operation
- checkpointing
- dynamic memory allocation (optional)

The basic unit of work which these operating system facilities service is called the task. A task consists of one or more programs written in a source language such as MACRO or FORTRAN, assembled or compiled into an object format, and then built into a task image by the linker utility called the Task Builder. In addition to the normal linkage

functions of combining object modules or creating overlays, the Task Builder sets up the basic task attributes that determine the task's resource requirements and relationship to other tasks in the system. The significant task attributes that affect a task's operation in a real-time multiprograming environment are:

- Partition — the section of memory where the task will reside when it executes.

- Priority — the task's relationship to other tasks competing for system resources.

- Checkpointability — the task's ability to be swapped out of memory when it is not executing to make room for a task of higher priority that is ready to run.

Once a task is built, it can be installed in the system and executed. Task installation simply registers a task's attributes with the system. The task is not in memory, nor is it in competition for system resources. An installed task can be put in active competition for system resources by the operator or by another active task in the system.

When an installed task is activated, the system will allocate necessary resources, bring the task into memory for execution, and place it in competition with other active tasks. Task installation is the basis for efficient task operation. An installed task uses very little memory resource; yet, when the task is needed to service a real-time event, it can be introduced into the system quickly since its basic parameters are already known to the system.

Tasks can also share code and data among themselves through the common partition facility. A common partition is made accessible to the system and to tasks by installing the common partition and the tasks which intend to use it.

The following paragraphs describe how task execution is handled by the RSX-11 systems.

Multiprogramming
Multiprogramming is the concurrent execution of two or more tasks residing in memory. In a single processor, only one task can have control of the CPU at a time. When that task does not need CPU time (for example, when it is waiting for input from a terminal), another task that needs CPU time can execute. In the RSX-11 family, the multiprogramming of tasks is accomplished by logically dividing available memory into a number of named partitions. Tasks are built to execute out of a specific partition, and all partitions in the system can operate in parallel.

In general, RSX-11 systems can have two kinds of partitions: system controlled and user controlled. System controlled partitions are intended for the execution of tasks where the user wishes the system to implicitly handle the allocation of memory. User controlled partitions are intended for the execution of tasks where the user wants to handle the allocation of memory.

A system controlled partition is dynamically allocated by the system to contain as many tasks as will fit simultaneously in the partition. Tasks are allocated a contiguous region in the partition, and are relocated using the hardware Memory Management Unit. The Memory Management Unit provides the facilities necessary for memory management and task relocation and protection. Systems using the Memory Management Unit are called mapped systems because the hardware allows the system to map virtual memory addresses into direct physical addresses. Only mapped RSX-11M systems can have system controlled partitions.

A user controlled partition is allocated to only one task at a time. The user has complete control over system activity in this type of partition. As a result, it provides an ideal environment for a real-time task's execution.

In RSX-11M or RSX-11S systems, a user controlled partition can be subdivided into as many as seven non-overlapping subpartitions. The subpartitions occupy the identical physical memory occupied by the main partition. Tasks built to execute in the subpartitions can execute in parallel. Tasks cannot, however, be resident in a main partition and its subpartitions simultaneously. If a main partition is occupied, the subpartitions can not be. All subpartitions can have tasks residing in them; therefore, up to seven potentially parallel task executions can exist within a pre-empted user-controlled main partition. The goal of subpartitioning is to reclaim large memory areas when a task requiring a main partition is no longer active.

Furthermore, RSX-11M and RSX-11S systems can be mapped or unmapped systems. If the hardware configuration does not include a Memory Management Unit, the RSX-11M system is an unmapped system. If a Memory Management Unit is available, the RSX-11M or RSX-11S system can be a mapped system. Mapped systems can have both system controlled and user controlled partitions. Unmapped systems can have only user controlled partitions.

From the operator's point of view, almost no differences exist between mapped and unmappped RSX-11 systems. One difference exists, however, in installing tasks into a partition. In unmapped systems, a task is linked to be installed and run in a partition with a specific base

address. It can not run in any partition whose base address is not the same. In mapped systems, a task can be installed into any partition large enough to contain it.

Mapped RSX-11M or RSX-11S systems provide automatic memory protection. The memory area assigned to a task is protected from other tasks executing in the system. Each task has an absolute address range in which to execute. A task can reference and alter memory only within that specific task area which it owns.

Priority Scheduling

Task scheduling in the RSX-11 family is primarily event-driven, in contrast to systems which use a time slice mechanism for determining a task's eligibility to execute. The basis of event-driven task scheduling is the software priority assigned to each active task. A task's default priority is set when the task is built. It can be altered once it is installed by an MCR command from the console. Priorities can also be changed dynamically from within a task.

Tasks are run at a software priority level ranging from a low of 1 to a high of 250. The executive grants central processor resources to the highest priority task capable of execution. That task retains control of the central processor until it declares a significant event.

A significant event occurs when a task issues a system directive that implicitly or explicitly suspends a task's execution, or when an external interrupt occurs that can affect a task's execution. For example, a task can issue a directive that indicates it wants to wait until an I/O operation is complete before continuing execution; a significant event is declared when the I/O operation is complete. A special system directive also exists that allows a task to stimulate the event-driven task-scheduling mechanism explicitly.

When a significant event is declared, the executive interrupts the executing task and searches for a task capable of executing. The highest priority task that has all the resources it needs to run and can make use of the resources it needs will be the task that gains control of the CPU.

Event flags are associated with significant events. When a significant event occurs, the event flag indicates the specific cause of the interrupt.

There are 64 event flags: 1 through 32 are local to the task, while event flags 33 through 64 are common to all tasks. A task can set, clear, test, and wait for any event flag or combination of event flags to achieve efficient synchronization between itself and other tasks in the system.

For example, upon completion of I/O requests, the executive normally sets a requester-indicated event flag and declares a significant event. If a requesting task instructs the system that it cannot run until an event flag is set (signaling task I/O completion), other eligible tasks of lower priority may run. In the scan of the active task list, a task that is awaiting I/O completion is bypassed until a significant event is declared, usually upon task I/O completion.

Although event-driven scheduling is the primary RSX-11 task-scheduling mechanism, it is not the only mechanism available. As an option during system generation, RSX-11 systems allow the user to supplement event-driven task scheduling with time-based round robin scheduling for some or all tasks.

In RSX-11M and RSX-11S systems, round robin scheduling is based on a priority range specified by the user during system generation. All tasks that have priorities within the specified range are scheduled using a time-slice algorithm. Tasks with higher or lower priorities than the specified range receive service in an event-driven manner. As a whole, the task range also receives service in an event-driven manner, but CPU time is shared among the tasks within the range.

Traps

Subroutines entered automatically as the result of an unanticipated synchronous condition (for example, an attempt to execute an illegal instruction) or as the result of an asynchronous condition anticipated or unanticipated (for example, an I/O completion) are called task trap routines.

Task traps are another means of governing task execution. While significant events have a system-wide scope, traps are local to a task. Traps interrupt the sequence of instruction execution in the task and cause control to be transferred to a pre-specified point in the program. In this way, traps provide the ability to service certain conditions without continuously testing for their existence.

When a task plans to use the trap facility, it must contain a trap service routine. This routine is automatically entered when the trap occurs using the task's normal priority and privilege. If a service routine is not supplied, the action taken by the executive is dependent upon the type of trap.

There are two types of traps: Synchronous System Traps (SSTs) and Asynchronous System Traps (ASTs).

SSTs provide a means of servicing fault conditions within a task, such as memory protection violation and floating point unit exceptions. These conditions, which are internal to a task and are not significant

events, occur synchronously with respect to task execution. In these cases, if an SST service routine is not included in the task, the task's execution is aborted.

ASTs commonly occur as the result of a significant event and thus occur asynchronously with respect to a task's execution. A task does not have direct or complete control over when ASTs occur. ASTs are for information purposes, such as signifying an I/O completion that a task wants to know about immediately.

If an AST service routine in not provided, a trap does not occur and task execution is not interrupted.

It should be emphasized that SSTs are only initiated by the executive; no further action is taken. That is, they appear to the executive just like normal task execution. The executive, having initiated an SST, cannot determine that the task is in the SST service routine. Thus, an SST service routine can be interrupted by another SST or an AST. SSTs can be nested.

SSTs are caused by activities internal to the task, while ASTs occur as a result of an external event. The executive keeps track of all ASTs, queues them first-in, first-out, and is aware that a task is executing an AST.

Power Failure Restart

Power failure restart is the ability of a system to smooth out intermittent short-term power fluctuations with no apparent loss of service and without losing data, all the while maintaining logical consistency within the system itself and the application tasks. Power failure affects absolute response time and peak load capacity differently from the facilities previously discussed, since it applies to the aggregate system performance rather than to increasing performance when the system is actually in operation. A system is not performing when it is shut down, and if the executive can reduce the shutdown periods with power failure restart, aggregate performance is increased.

1. When power begins to fail, the processor traps to the executive which stores all register contents.

2. When power is restored, the executive again receives control and restores the previously preserved state of the system.

3. The executive then informs any tasks that have requested power failure restart notifications through the Asynchronous System Trap mechanism that a power failure has occurred. These tasks can then, if required, make the restorations of state they deem necessary.

4. The executive schedules all device drivers that were active at the time the power failure occurred at their powerfail entry point. Drivers have the option of always being scheduled on power recovery, or of being scheduled only when the driver has outstanding I/O.

These drivers can then, if required, make those restorations of state (for example, repeating I/O requests) that they deem necessary. This approach is quite efficient because the repeating of I/O is placed nearest the source most likely to contain instructions on how to make the restoration.

Disk-Based Operation (RSX-11M)
Except in some dedicated applications, the total code in a system always exceeds the available main memory. A disk-based system uses random access peripherals both as an extension of executive main memory and as the principal data interchange medium. The use of disk as the system data storage medium provides the base for program development facilities, a common file system, checkpointing, and rapid initiation of tasks. The Task Builder makes it possible for the user to build overlaid tasks and call these overlays from disk. The total effect is to extend significantly the achievable peak load while still maintaining system response time requirements.

Task Checkpointing (RSX-11M)
Effective multiprogramming is achieved when many tasks reside in memory simultaneously, spending some of their residency waiting for I/O completion, waiting for synchronization with other tasks, or being unable in some way to continue execution. While one or more tasks are waiting, another task can utilize the central processor's resources.

This multiprogramming scheme normally applies only to memory-resident tasks. Once a task is in memory, the executive allows it to run to completion in a multiprogrammed fashion even if its memory is required for the execution of a higher priority, non-resident task. However, if it is desirable to free memory for execution of a higher priority task, a task can be declared checkpointable when it is task built or installed.

A checkpointable task can be swapped out of memory when a higher priority task requests the partition in which it is active. Checkpointing is another way of making it possible to load the processor with as much work as it can possibly absorb, and still meet its real-time commitments.

In RSX-11M systems, task priority normally determines which tasks can checkpoint other tasks. A checkpointable task currently active in a partition, but of a lower priority than another task requesting the partition, can be pre-empted and rolled-out to disk. Later, after the higher

131

priority task has completed its execution, the lower priority task can be rolled-in and restored to active execution at the point where it was previously interrupted.

The system extends the checkpoint capability by disregarding the priority of a task in cases where the task currently active in a partition is waiting for terminal input. A task requesting a partition can checkpoint a task of higher priority if that task is waiting for terminal input.

Dynamic Memory Allocation (RSX-11M)

Dynamic memory allocation is an extension of the RSX-11 multiprogrammed partition structure. Dynamic memory allocation allows the system to respond rapidly to changing requirements for system resources.

RSX-11M allows the user to load and execute more than one task in a system-controlled partition. If a task loaded into a system-controlled partition does not fill the entire partition, another task can be loaded into the space either above or below it, as long as the remaining contiguous physical space is large enough to contain it.

The executive keeps an internal list of the available areas of memory in the system-controlled partitions, together with a list of all tasks requesting to run in those partitions. Tasks are brought in from the disk on a priority basis and are loaded into the first available memory area in the partition. The executive continues to load tasks as long as there is sufficient contiguous physical memory available in the partition. When a task terminates, the memory it occupies becomes available again.

If the dynamic memory allocation option is included in an RSX-11M system, the user can also include the automatic memory compaction option. Normally, a task can not be loaded into a system-controlled partition unless there is sufficient contiguous space for it between other tasks loaded in the partition. When a task terminates, it can leave a space which is insufficient to load another task, but, considered together with other unused areas, can be used to contain a task. If automatic memory compaction is included in the system, the tasks in a system-controlled partition will be moved to obtain a large enough area in the partition to load another task.

SYSTEM ORGANIZATION AND COMPONENTS

The following sections discuss the basic design elements of RSX-11M and RSX-11S operating systems. In RSX-11 systems, total system structure is essentially dependent on the decisions that the user makes during system generation. The user defines the system organization and chooses the executive services appropriate for the

particular applications environment. This procedure is referred to as system generation (SYSGEN).

There are three basic functional uses for which memory is allocated. The amount of memory allocated to each function is specified by the user during system generation. The the functional memory spaces are for:

• the RSX-11 executive and system dynamic memory

• the partition space for tasks and shared commons

RSX-11M and RSX-11S systems are designed to provide the most efficient use of system resources during system operation. To be useful to a wide range of applications and still obtain maximum system performance for a given operating environment, RSX-11M/S systems require the user to become reasonably involved in system generation.

System generation for RSX-11M/S systems provides the user with absolute control over system features and capabilities. Users concerned about size can eliminate the executive services that are not essential to a particular applicaton.

RSX-11M Executive and Memory Structures
For RSX-11M/S system generation, the user specifies the sizes and base addresses of the partitions, and selects the executive services and amount of dynamic memory needed for the particular application. System generation is performed in two phases: the first phase defines the hardware configurations and software options, the second phase builds the complete system. Some system generation parameters can be changed on-line, for example, partition configuration. If executive services are to be changed, however, the user must regenerate the system.

RSX-11M system generation requires the user to allocate partitions. In RSX-11M systems, the user can define and delete partitions on-line. RSX-11M systems have two kinds of shared commons: libraries and global common blocks. The shared commons require their own partitions, and are not loaded automatically when tasks require them. Commons are fixed and must be explicitly loaded before a task requiring them.

MCR Command Buffer
The MCR command buffer holds the data for a requested MCR function task. The buffer is set up by the MCR dispatch task. The dynamic memory required for the buffer is returned to the pool after the GET MCR COMMAND LINE directive passes the command line to the MCR function task.

The RSX-11M basic executive organization is illustrated in Figure 6-1. The individual regions are explained below; most of the regions are directly affected by system generation parameters.

Trap Vectors
This region contains the hardware trap and interrupt vectors and requires 128 words. This region is expandable during system generation to a maximum of 256 words.

System Stack
Used for nesting interrupts and internal calls made by the executive. Forty words are required.

System Common Data
Contains pointer filled in during system generation.

System Tables
Contain the data used to control system operation. Included are partition descriptions, the system task directory, and device tables. The total size of the table region is established by system generation configuration selections.

Dynamic Storage Region
The executive has continuing needs for temporary storage. Such storage is acquired, used, and returned to the available pool. If a given executive service requests dynamic storage, and it is unavailable, the executive will inform the user task, which usually waits for some storage to become available. The size of this region is important, for if it is too small, waiting periods will be induced; if it is too large, system effectiveness is lowered, since fewer tasks can fit in memory. The size of the region is a system generation parameter.

```
16K ┌─────────────────────────┐
    │      USER  TASKS         │  ◄──── USER DEFINED
 8K ├─────────────────────────┤        PARTITIONS AREA
    │      FILE SYSTEM/        │        EXECUTIVE SYSPAR
    │  TASK TERMINATION ROUTINE / ◄──── PARTITION AREA (CAN
    │   MONITOR CONSOLE ROUTINE│        BE MOVED DURING
    │                          │        SYSTEM GENERATION)
    ├─────────────────────────┤        CAN BE EXPANDED
    │  DYNAMIC STORAGE AREA    │  ◄──── DURING SYSTEM
    ├─────────────────────────┤        GENERATION OR VIA
    │    TASK LOADER FOR       │        AN MCR COMMAND
    │  NON-RESIDENT TASKS      │
    ├─────────────────────────┤
    │     SYSTEM TABLES        │
    ├─────────────────────────┤
    │    TERMINAL DRIVER       │
    ├─────────────────────────┤
    │    LOADABLE DEVICE       │        DEVICE DRIVERS
    │  DRIVERS  PARTITION      │  ◄──── CAN BE ADDED
    ├─────────────────────────┤        DURING SYSTEM
    │     DISK DRIVERS         │        GENERATION
    ├─────────────────────────┤
    │   EXECUTIVE SERVICES     │
    ├─────────────────────────┤
    │    BASIC EXECUTIVE       │
    ├─────────────────────────┤
    │   SYSTEM COMMON DATA     │
    ├─────────────────────────┤
    │   SYSTEM STACK  SPACE    │
    ├─────────────────────────┤
    │     TRAP  VECTORS        │
  0 └─────────────────────────┘
```

Figure 6-1 Basic RSX-11M Executive 16K Word System

The Basic Executive

The basic executive includes the code that controls the multiprogram-ming environment, performs task checkpointing and power fail re-start, and handles system traps. During system generation, the user has the option of including or omitting the following services:

• task checkpointing

• task checkpointability during terminal input

• Memory Management Unit support

• dynamic memory allocation

• automatic memory compaction

• I/O rundown (automatic system clean-up after a task aborts and leaves files in an indeterminate state)

135

- asynchronous system trap support
- external (user-written) MCR function support
- task termination and device-not-ready messages
- power failure recovery
- GET PARTITION PARAMETERS directive support
- GET SENSE SWITCHES directive support
- EXTEND TASK directive support
- GET TASK PARAMETERS directive support
- ALTER PRIORITY directive support
- SEND/RECEIVE directives support
- Memory Management directives support
- automatic install, request, and remove-on-exit support (RUN command option)
- logical device assignment support
- setting upper/lowercase conversion for terminal input
- multi-user protection support
- transparent terminal READ/WRITE support
- RMS record locking
- executive-level round-robin scheduling
- executive-level disk swapping
- user-written device driver support
- executive debugging tool
- panic/crash dump and system failure reporting
- device error and timeout logging
- loadable device drivers
- ANSI magtape support
- direct connect to user tasks of hardware interrupts directive

The following processor options support can be included or omitted:

- Floating Point Processor support
- FIS support
- programmable clock support
- watchdog timer support
- parity memory support

Executive Directive Services

This region contains the service routines which respond to the directives issued by users to request executive services. These programs make use of the basic executive.

Device Drivers

Three fixed drivers can be included in the basic 8K executive:

- disk

- cassette, DECtape, magnetic tape, line printer or floppy disk

- terminal (basic DL11 driver only)

These are multi-unit drivers that can service up to the maximum devices controlled by the respective hardware interfaces. Drivers can be either fixed in memory with the executive or they can be loadable, allowing for more efficient memory use.

Task Loader for Nonresident Tasks

This loader is a task and operates out of its own partition. Thus, it can run in parallel with system and user tasks. The loader, which is device independent:

1. Loads tasks on initial load requests.

2. Writes checkpointable tasks to disk when required.

3. Returns previously checkpointed tasks to active competition for processor resources.

File System, Monitor Control Routine (MCR) and Task Termination (TKTN)

These three routines function as tasks. In the minimum system, they execute out of the same partition.

As distributed, the RSX-11M system generates a file system that runs in 2K words. The user has the option of building a larger file system with greater processing speed and increased function.

Panic Dump and Crash Modules

These two routines respond to system software failures, providing core dumps and selective analysis. They are not included (or shown) in the basic 8K system, but are mentioned because of their fundamental importance in error analysis. Most program development systems (as opposed to dedicated on-line systems) will likely include these routines.

In a 16K system with an 8K executive, the remaining 8K words are available for user task partitions. In 16K-word systems, partition definitions cannot be altered without regenerating the system. In systems with more than 16K words of memory, the user can re-define partitions

on-line using an MCR console command. Figure 6-2 illustrates a typical memory organization for a large mapped RSX-11M system.

```
124K  ┌────────────────────────┐      ◄─── SYSTEM CONTROLLED
      │      USER  TASKS        │           GEN PARTITION
34.5K ├────────────────────────┤
      │   LOADABLE  DEVICE      │      ◄────── DRVPAR
26.5K ├────────────────────────┤
      │    PRINT  SPOOLER       │      ◄────── PAR4K
      │    AND  PMD  TASK       │
22K   ├────────────────────────┤
      │    BIG  FILE  SYSTEM    │      ◄────── FCPPAR
20K   ├────────────────────────┤
      │  MCR AND  TKTN TASKS    │      ◄────── SYSPAR
      ├────────────────────────┤
      │   DYNAMIC  STORAGE      │
      │       REGION            │
      ├────────────────────────┤
      │ TERMINAL  AND  SYSTEM   │
      │        DISK             │
      │   DISK  DRIVERS         │
      ├────────────────────────┤
      │     EXECUTIVES          │
      │ (INCLUDES  ALL SERVICES)│
0     └────────────────────────┘
```
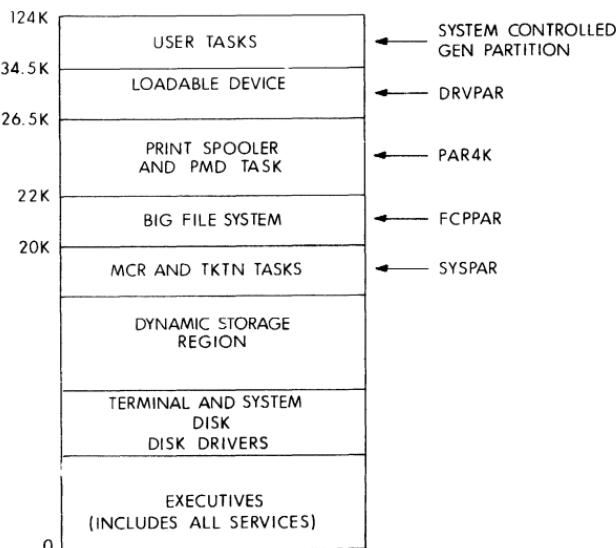
Figure 6-2 Memory Organization for a Large Mapped RSX-11M System

RSX-11S System Components

RSX-11S requires an RSX-11M, RSX-11D, or VAX system for system generation and program development. An RSX-11S system is generated from the RSX-11M system using the standard system generation process. The maximum hardware and software configuration is the same as that of an RSX-11M system with the exceptions of file system support, non-resident tasks, task checkpointing, and dynamic memory allocation.

Since it is based on RSX-11M, RSX-11S enjoys most of the inherent features and generation capability of that system. For example, RSX-11S automatically supports all of the peripheral devices that RSX-11M supports, including hardware features such as floating point processors, parity memory, and memory management. All are selectable at system generation and can be included in an RSX-11S system at the cost of memory use.

The basic software building blocks for an RSX-11S system are:

1. The generatable features of the RSX-11M Executive (2.5K to 4K)

2. A special File Control Services (FCS) (1.25K) that contains no support for directory devices.

3. All RSX-11M I/O device drivers

4. Subset MCR (2K)

5. On-line Task Loader (2.5K)

6. System Image Preservation Program (1.5K)

The minimum software system is an executive. The smallest executive that can be generated requires 2.5K words of memory. Services that are omitted from the 2.5K executive include:

- address checking

- Asynchronous System Traps (required for FORTRAN)

- I/O rundown

- task termination and device-not-ready notification

- external MCR functions (user-written functions)

- install, request, and remove-on-exit support

- SEND, RECEIVE, GET TASK PARAMETERS, GET SENSE SWITCH-ES and GET PARTITION PARAMETERS directives

- parity memory support

- network support

- all I/O drivers

Although omitted from the minimum executive, these features can be generated into an RSX-11S system at the cost of memory use.

The minimum RSX-11S software system must include the executive and the I/O device drivers. For example, two to four small I/O device drivers could be added to the minimum executive at the cost of an additional 1.5K words of memory. In an 8K word system, approximately 4K words would be available to application tasks.

If operator communication is required, subset MCR can be included in a system at a cost of 2K of memory. In an 8K system this still leaves approximately 2K for application tasks.

The On-Line Task Loader (OTL) can be included in an RSX-11S system if the on-line loading of tasks is desired.

Tasks are created on a host RSX-11M system, transferred to the load medium using RSX-11M's File Exchange Utility (FLX), and then loaded into a running RSX-11S system using OTL. The minumum size for OTL is 2.5K words. In 2.5K words, however, OTL supports only one load device. On-line task loading requires a 16K-word system, since approximately 8.5K words will be required for system software (2.5 executive, 2K MCR, 1.5K device drivers, and 2.5K OTL).

The System Image Preservation Program (SIP) is an on-line utility task that provides the capability to save the image of a running system into a load device medium in bootstrapping format. The saved system can subsequently be restored by bootstrapping it from the load device medium. The minimum size for SIP is 1.5K words. In 1.5K words, it can support only one load device.

The standard RSX-11M File Control Services (FCS) record I/O package contains a large amount of code to support file-structured devices. (RSX-11S contains no file support and this code is therefore unnecessary.) The special version of FCS provided with RSX-11S is the standard FCS without the file support code. This provides a significant size reduction.

SYSTEM CONVENTIONS

To simplify operations, RSX-11 systems observe certain conventions with respect to devices, file structures, file naming, operator commands, and indirect files.

Devices

The RSX-11 systems support a variety of peripheral devices. They are referred to by a 2-letter name and an optional 1- or 2-digit unit number followed by a colon. For example, TT12: represents user terminal number 12. Peripheral devices can be referred to by mnemonics, by pseudo-device names, or, in task references, by logical unit numbers. In addition, RSX-11M systems support logical device name assignments.

Pseudo device names are associated with normal device mnemonics assigned by the system manager. They permit the system manager to dynamically determine the physical devices that will send or receive information. RSX-11M supports the following pseudo devices:

SY: System device: indicates the device on which the system disk is mounted.

TI: Terminal interface: indicates the terminal with which a particular task is associated. Each terminal has a unique TI. The TI of each task is assigned to the requesting terminal.

CL: Control log: indicates the device normally used for the listing of files. The CL device is normally redirected to the line printer.

CO: Console output: indicates the device by which the system can communicate with the system manager. The CO device is normally redirected to the system console.

Logical unit numbers (LUNs) provide the mechanism for programs to maintain device independence. The logical unit numbers used in a program can be assigned by means of device mnemonics to any available peripheral device that performs the desired function. LUNS can be assigned by the programmer at task-build time, or by the task itself at run time. Because the system provides LUN assignments, it is not always necessary to assign a LUN to a task. Furthermore, LUNs can be changed by an MCR function for any installed, inactive, non-fixed task.

RSX-11M has an additional facility for associating a logical name with a physical device, called logical device assignment. Logical device assignments are a convenient way to associate logical names with physical devices. There are two types of logical device assignments: local and global. Local assignments apply only to commands and tasks initiated from the terminal on which the assignment was made. Global assignments apply to all commands or tasks. If a logical name is defined as both global and local, the local assignment overrides the global assignment. Logical device names can be the same as physical device names or can be any character string using the syntax for device names.

File Structures
RSX-11M supports a common file structure for disk called Files-11. In addition, RSX-11M supports ANSI Standard Level 3 format for single or multi-volume magnetic tape files.

Files-11 is a general purpose file system that provides a facility for the dynamic creation, extension, and deletion of files on disk. It includes a scheme for volume and file protection which allows the owner of a volume or file to deny all access or certain kinds of access to all users, groups of users, or particular users in the system. This scheme for volume and file protection provides the key to the system protection, in that only users with access privileges are allowed access.

A Files-11 volume is a collection of files which reside on a single disk. The system can directly address each file on the volume by means of file pointers which reside in the volume's directory files.

Each Files-11 volume has two kinds of directory files that are used for file management: the Master File Directory (MFD) file, and User File Directory (UFD) files.

The Master File Directory (MFD) file is automatically generated by the file system when a volume is initialized as a Files-11 volume, and is used to store pointers to all of the User File Directory (UFD) files on the volume.

141

User File Directory (UFD) files are created as needed. They are used to store pointers to all of the files belonging to, or associated with, the user whose account number (User Identification Code or UIC) corresponds to the UFD file name.

All Files-11 files, whether MFD, UFD, or user files, have the same basic format. All files have a file header area, and one or more data area(s). Figure 6-3 illustrates the Files-11 file format.

FILE HEADER AREA

| FILE OWNER FIELD (UIC) |
| FILE NAME FIELD (FROM 1-9 ALPHANUMERIC CHARS.) |
| FILE TYPE FIELD |
| VERSION NUMBER FIELD |
| FILE PROTECTION FIELD |
| DATA POINTER FIELD(s) |
| DATA AREA A POINTER |
| DATA AREA B POINTER |
| DATA AREA N POINTER |

DATA AREA A

DATA AREA B
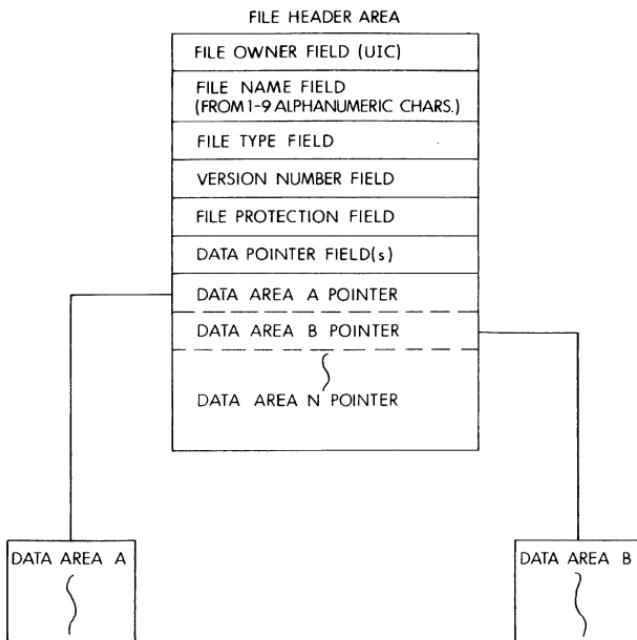
Figure 6-3 Sample Files-11 File Structure

The file header area contains all the pertinent information required by the file system to process the file. For the purposes of this introduction, the user need be familiar with only the following fields:

File Owner Field   The file owner field contains the account number (UIC) of the user who created the file.

File Name Field	The file name field contains the name assigned to the file when it was created. File names can be a maximum of nine alphanumeric characters long.
File Type Field	The file type field contains the mnemonic that identifies the file by its functionality; for example, FTN defines a FORTRAN source file.
Version Number Field	The version number field identifies the particular version or generation of the file.
File Protection Field	The file protection field contains a code that describes who is allowed to access the file: system, owner, group, or world. It also describes the type of access allowed; read, write, extend or delete.
Data Pointer Field	The data pointer field describes the physical allocation of the file on the volume. Each data area pointer describes a physically contiguous portion of the file.

By establishing pointers to blocked data in the file's header area, as opposed to storing the data immediately following the file header, the system accomplishes two things: all files on the volume have the same structural format regardless of functionality; and all fragmented or non-contiguous areas of the volume can be put to use, that is, a file can be expanded merely by attaching another pointer to a blocked data area in its file header.

Users always address data in a file-relative manner. The translation of file-relative address into physical addresses is performed by the file system and is completely transparent to the user.

File Specifiers

Any system component that needs to refer to files does so using a standard file command string with the following general format:

Output file specifications = input file specifications

There can be several file specifications on either side of the equal sign. Optional switches are used to indicate desired operations other than default operations. File specifiers have the following format:

dev:[uic]filename.type;version/switch

where:

dev: is the physical device on which the volume contain-
                      ing the desired file is mounted (for example DB0: or
                      DK1:).

[uic]   is the User Identification Code that specifies the user
                      file directory containing the desired file.

filename   is the name of the file. File names can be up to nine
                      alphanumeric characters in length. File name and
                      type are always separated by a period.

type   is the designator distinguishing among various
                      forms of files. For example, a FORTRAN source file
                      might be named COMP.FTN, while the object file
                      associated with that program might be named
                      COMP.OBJ.

;version   is a number used to differentiate among versions of
                      a file. For example, when a file is first created using
                      the text editor, it is assigned a version number of 1. If
                      the file is subsequently opened for editing, the editor
                      keeps the original file for back-up and creates a new
                      file with the same file-name and type designations,
                      but with a version number of 2.

/switch   is usually an optional qualifier. Switches are normal-
                      ly used either to direct the execution of a task, or to
                      qualify an input parameter.

If any of the file specifier elements except the file name is omitted from
the file specifier, the system can use a default value. A task can also
establish defaults for a file. The system default for the device name is
the system device. The default for the user file directory specification
is the UFD that corresponds to the UIC under which the task is run-
ning. The default for the version specification is the latest version
number. For RSX-11M systems, the defaults for the type specification
vary according to the operation to be performed. The common set of
file types is:

.CMD   An indirect file containing a list of task or MCR com-
                      mands for a task. (In RSX-11M the commands can
                      also be MCR commands.)

.DAT   A data file, as opposed to a program file

.DIR   A directory file, for example, a UFD directory

.FTN   A FORTRAN source program

.LST   A listing file

.MAC   A MACRO-11 source program

.MAP   A task builder memory allocation listing file

.MLB   A user macro library

.ODL   An overlay description file

.OLB   An object module library

.SML   The system macro library

.STB   A symbol table file

.TSK   A task image file

.CRF   A cross reference file that will be appended to the map file

.SYS   A system image file

MCR Operator Commands and Terminal Control

The Monitor Console Routine (MCR) is the terminal interface between the user and the RSX-11 operating system. In the system, terminals can have either of two functions: command or slave. The system does not accept any unsolicited input from a slave terminal; its I/O is completely under task control. A command terminal is used to activate MCR and interface with the system using MCR system commands. The SET command can be used to characterize a terminal as a slave or command terminal.

MCR's system commands enable the general user to perform the following functions:

- gain access to the system
- initiate and terminate execution of system or user programs

In addition, the privileged user can perform the following additional functions:

- adjust, modify, and control the system environment

The privileged MCR user has complete control over the system's operation.

In RSX-11M, the privileged characteristic is associated with certain terminals, as determined by the system manager initially during system generation and subsequently by setting terminal characteristics using the SET command. Non-privileged commands can be invoked at any command terminal. Privileged commands can be invoked only at privileged terminals.

In RSX-11M, the MCR task itself processes most of the standard MCR commands, but will call independent command tasks to process some commands. The MCR organization makes it possible for users to add operator console services to meet their application needs.

The RSX-11 systems include four different kinds of MCR commands; initialization commands, informational commands, task control commands, and system maintenance commands. Table 1 lists the RSX-11 MCR commands, and indicates the systems in which they are available.

Table 6-1 RSX-11 MCR Commands

Initialization Commands

COMMAND	SYSTEMS	FUNCTION
BOOT	M	Boostraps a system that exists as a task image file on a file-structured volume. It provides a convenient means for terminating one system and starting another. For example, BOOT can be used for terminating a real-time system and starting a program development system.
TIME	M & S	Lists the time and date maintained in the system clock calendar. A privileged user can change the time and date.
MOUNT	M	Declares that a volume is logically on-line for access by the system.
DISMOUNT	M	Declares that a volume is logically off-line and cannot be accessed by the system.
INITVOLUME	M	Initializes a volume for use by the system.

COMMAND	SYSTEMS	FUNCTION
INSTALL	M	Installs a task in the system by making an entry in the System Task Directory; this allows the operator to subsequently run the installed task. This function is performed by the On-line Task Loader utility in RSX-11S systems.
SET	M	Allows the user to establish or alter a variety of parameters, including terminal device characteristics, command or slave terminal characteristics, and default UIC for a terminal.
UFD	M	Creates a User File Directory on a volume and enters its name in the Master File Directory.
HELLO	M	Allows the user to log in to the system and be identified as a valid user.
BYE	M	Logs a user off the system. The only valid command that MCR recognizes after BYE is HELLO.

Informational Commands

COMMAND	SYSTEMS	FUNCTION
ACTIVE TASK LIST	M & S	Lists the active tasks in the system, indicating the tasks' current status, for example, task suspended, waiting for I/O, etc.
BAD	M	Locates any unusable blocks on a disk.
DEVICES	M	Prints the symbolic names of all device units known to the system. Indicates if a device handler is resident, a volume is mounted, or to what device a symbolic device name is assigned.

COMMAND	SYSTEMS	FUNCTION
LUNS	M	Prints a list of the physical device units and corresponding logical unit numbers for an indicated task. It is used to determine which physical devices a task requires.
PARTITIONS	M	Lists a description of each memory partition including partition name, base address and use. It also lists a description of each memory-resident sharable library and global command block.
TASK LIST	M&S	Lists a description of each task installed in the system, including task name, version number, default partition name, priority and size.
HELP	M	Displays contents of HELP file.
BRO	M	Broadcasts a message to one or a set of terminals.

Task Control Commands

COMMAND	SYSTEMS	FUNCTION
ALTER	M	Allows the user to change the priority of a task.
FIX	M	Allows the user to fix a task in its partiton in memory. A fixed task gets faster response to requests for execution. (A function exists in OTL for RSX-11S systems to fix a task when it is loaded).
UNFIX	M	Frees fixed tasks from memory.
ASSIGN	M	Assigns a logical device name to a physical device.
REASSIGN	M	Reassigns a Logical Unit Number (LUN) from one physical device to another.
REDIRECT	M & S	Redirects all I/O requests from one physical device to another.

COMMAND	SYSTEMS	FUNCTION
LOAD	M	Makes a specified device driver resident in memory and ready to honor I/O requests.
UNLOAD	M	Unloads a specified device driver from memory.
RUN	M & S	Initiates the execution of an installed task. An installed task can be started immediately, started a specified time from when the command is issued, started a specified time from the next time unit, or started at an absolute time of day. A special option of the RSX-11M RUN command allows the user to run a task that has not been installed; when issued, the task is installed, run and removed on exit. In all cases except the latter, the user can specify a reschedule interval for the task.
CANCEL	M & S	Cancels any pending periodic rescheduling for a task.
ABORT	M & S	Terminates execution of a specified task.
CLQUEUE	M	
RESUME	M & S	Continues execution of a previously suspended task.
REMOVE	M & S	Removes a task name from the system task directory (opposite of INSTALL).

System Maintenance Commands

COMMAND	SYSTEMS	FUNCTION
SAVE	M	Copies the memory image of the system to the system disk so that a bootstrap can reload it and start up the system. (In RSX-11S systems, the System Image Preservation utility performs this function.)

149

COMMAND	SYSTEMS	FUNCTION
OPEN	M & S	Allows the privileged user to examine or modify a word in memory.
BRK	M	Breaks to the Executive Debugging Tool (XDT).

In addition to the MCR commands available to control system execution, an RSX-11 system provides the following special terminal control characters:

CTRL/C   Activates MCR at a terminal. The system types the prompt "MCR>". Note that, unlike most other PDP-11 systems, the RSX-11 family does not use CTRL/C to affect the execution of any currently running tasks other than MCR.

CTRL/Z   Logical end-of-file; when typed in response to a prompt from most utility programs, CTRL/Z causes the program to exit.

CTRL/I   Causes a horizontal tab.

CTRL/K   Causes a vertical tab of four lines.

CTRL/L   Causes eight line feeds.

CTRL/U   Cancels the current input line.

CTRL/O   Enables or disables output to a terminal.

CTRL/S   Temporarily suspends output to the terminal. This feature enables users with high-speed terminals to fill the display screen, stop output with a CTRL/S and then continue with a CTRL/Q.

CTRL/Q   Resumes printing of characters on the terminal from the point at which printing was interrupted using CTRL/S.

CTRL/R   Causes the system to reprint the current line entered in the terminal buffer and allows the user to view exactly what has been entered so far.

Indirect Files (Command Files)
An indirect file is a sequential file containing a list of commands. Rather than typing commonly used sequences of commands, the user can

type the sequence once and store it on a file using the Editor utility program. To execute the sequence, the user types an "at" sign (@) and then the command file name. The affected task locates the indirect file and executes the command it contains.

There are two types of indirect files, indirect task command files and indirect MCR command files.

The commands contained in an indirect task command file are task specific. They can be interpreted only by a specific task such as the MACRO assembler, the Task Builder, or another utility program. The indirect file is specified in place of the command line normally given to the task when it is run. For example, to give an indirect file to the MACRO assembler to execute, the user types:

MCR>MAC@MDFIL.CMD

which causes MACRO to read and execute the file CMDFIL.CMD for all of its commands.

RSX-11M supports an indirect command file processor for MCR command processing. In this case, the indirect file contains commands to the MCR console interface. To execute a series of MCR commands using the indirect MCR command file processor, the user types the "at" sign followed by an indirect file's name in response to the MCR prompt. For example, to execute a series of MCR commands contained in the file name BEGIN.CMD, the user types:

MCR>@BEGIN.CMD.

In addition to the standard MCR commands, the RSX-11M indirect command file processor can accept special commands that allow the user to control command file processing. These special commands provide the following capabilities:

INITIATE PARALLEL TASK EXECUTION
It is possible to request initiation of a task and not wait for the task to terminate before having the next command line processed. Normally, the indirect file processor passes a task initiation command line to MCR and then waits until the command is executed before continuing. In this case, however, the indirect file processor can initiate a task, pass a command string to it, and continue processing the indirect file command lines in parallel with the initial task's execution.

WAIT FOR A TASK TO FINISH EXECUTION
Indirect command file processing can be suspended until a particular task has terminated.

TEST IF A TASK IS INSTALLED OR NOT INSTALLED
A test can be made to determine whether a particular task is installed

in the system or not. If the task is installed, the remainder of the command line is ignored.

TEST IF A TASK IS ACTIVE OR NOT ACTIVE
A test can be made to determine whether a task is active or not. If the task is active, the rest of the command line is processed. If the task is not active, the rest of the command line is ignored.

SUSPEND EXECUTION FOR A SPECIFIED TIME INTERVAL
Indirect file processing can be suspended for a specified number of clock ticks, seconds, minutes, or hours. When the interval is exhausted, indirect file processing continues at the point where it was interrupted.

PROVIDE COMMENTARY
Comments can be included in the command file. Comments are displayed on the entering terminal and are convenient to provide explanation or to give instructions to the user who issued the command file.

PAUSE FOR OPERATOR ACTION
It is possible to suspend indirect file processing until the user at the entering terminal performs some action. The file processor prints a message on the terminal to notify the user. To continue indirect command file processing, the user types a RESUME command.

ASK A QUESTION AND WAIT FOR A REPLY
It is possible to print a message on the entering terminal, suspend indirect command file processing until input is received, and then set a specified symbol true or false depending on the input contents. If the symbol is not already defined, an entry is made in the symbol table and its value set.

DEFINE A SYMBOL
A symbol can be defined or its value can be changed. A symbol can represent either a true or false value. When a symbol is first defined, a symbol table entry is made and set to a specified value. A symbol can have any alphanumeric name up to six characters long.

TEST IF A SYMBOL IS TRUE OR FALSE
The value of a symbol can be tested at the beginning of a command line. If the test is true, the rest of the command line is processed. If the test is false, the remaining part of the command line is not processed.

TEST IF A SYMBOL IS DEFINED OR NOT DEFINED
A test can be made to determine whether a symbol has been defined or not. If it is defined, the rest of the command line is processed. If is is not defined, the rest of the command line is ignored.

DEFINE LABELS
A command line in the command file can be labeled.

BRANCH TO A LABELED LINE

Control can be transferred from one line in an indirect file to another line in an indirect file by an unconditional branch to a labeled line. A branch can transfer processing to a labeled line before the branch command line or after the branch command line.

BRANCH TO A LABELED LINE ON DETECTING AN ERROR

Control can be transferred from one line in an indirect file to another line if an error occurs. If the conditional branch on error line is processed, control is passed to a specified command line if one of the following errors is detected: undefined symbol reference, symbol table overflow, undefined label, or syntax error. This feature enables the user to gain control to clean up before aborting execution.

COMBINED LOGICAL TEST

Tests can be combined using Boolean AND and OR directives. In addition, an implied logical AND is effected if multiple tests are placed on the same line; the command on the line is executed only if all tests are true.

MCR indirect files can reference other MCR indirect files. Up to four levels of indirect MCR command files can be specified. Each time a new level is entered, all symbols previously defined are masked out of the symbol table and only symbols defined in the current level are available. When control returns to a previous level, the symbols defined in that level are available again.

RSX-11M can execute multiple MCR indirect files simultaneously. Several users at MCR command terminals can initiate MCR indirect command file processing. This effectively provides multiple-stream "batch" processing in RSX-11M systems.

FILE CONTROL SERVICES

RSX-11 file control services enable the user to perform record oriented and block oriented I/O operations and to perform additional functions required for file control, such as open, close, wait, and delete operations. To invoke FCS functions, the user issues macro calls to specify desired file control operations. The FCS macros are called at assembly time to generate code for specified functions and operations. The macro calls provide the system-level file control primitives with the necessary parameters to perform the file access operations requested by the user. Figure 6-4 illustrates the file access operation.
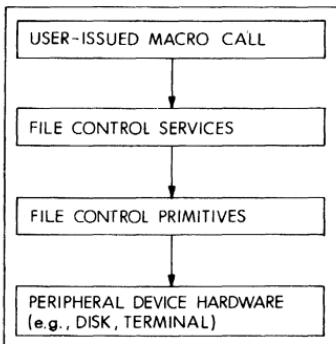
Figure 6-4 File Access Operation

FCS is a set of routines that is linked with the user program at task-build time from a resident system library or a system object module library. These routines, consisting of pure, position-independent code, provide a user interface to the file system, enabling the user to read and write files on file-structured devices and to process files in terms of logical records.

Logical records are regarded by the user program as data units that are structured in accordance with application requirements, rather than existing merely as physical blocks of data on a particular storage medium.

FCS provides the capability to write a collection of data (consisting of distinct logical records) to a file in a way that enables the data to be retrieved at will. Data can be retrieved from the file without having to know the exact form in which it was written to the file.

FCS thus provides a sense of transparency to the user so that records can be read or written in logical units that are consistent with an applications requirement.

File Access Method
RSX-11 supports both sequential and direct access to files. The sequential access method is device-independent, that is, it can be used for both record-oriented and file-structured devices (for example, card reader and disk, respectively). The direct access method can be used only for file-structured devices.

Data Formats for File-Structured Devices
Data is transferred between peripheral devices and memory in blocks. A data file consists of virtual blocks, each of which may contain one or

154

more logical records. In FCS, a virtual block in a file consists of 512 bytes.

Records in a virtual block can be either fixed or variable in length. The first two bytes of a variable-length record contain a value defining the length of that record (in bytes), excluding the record length bytes.

Virtual blocks and logical records within a file are numbered sequentially, starting with one. A virtual block number is a file-relative value, while a physical block number is a volume-relative value. For example, the first virtual block in a file is always virtual block number 1, but at the same time it could also be physical block number 156.

Block I/O Operations

The READ and WRITE macro calls allow the user to read and write virtual blocks of data from and to a file without regard to logical records in a file. Block I/O operations provide a very efficient means of processing file data, since such operations do not involve the blocking and deblocking of records within the file. Also, in block I/O operations, the user can read or write files in an asynchronous manner; control can be returned to the user program before the requested I/O operation is completed.

When block I/O is used, the number of the virtual block to be processed is specified as a parameter in the appropriate READ and WRITE macro call. The virtual block so specified is processed directly in a buffer reserved by the program in its own memory space.

As implied above, the user is responsible for synchronizing all block I/O operations. Such asynchronous operations can be coordinated through an event flag specified in the READ and WRITE call. The event flag is used by the system to signal the completion of the I/O transfer, enabling the user to coordinate those block I/O operations which are dependent on each other.

Record I/O Operations

The GET and PUT macro calls are provided for processing record-oriented files. GET and PUT operations perform the necessary blocking and deblocking of the records within the virtual blocks of the file, allowing the user to read or write individual records.

In preparing for record I/O operations, the user program must specify the format of the records. For example, it must specify whether the records are fixed or variable in length, or whether records that are to be output to a carriage-control device are to contain carriage-control information, which can be either at the beginning of the record or embedded within the records.

For sequential access files, I/O operations can be performed for both fixed and variable length records. For direct access files, I/O operations can be performed only for fixed length records.

In contrast to block I/O operations, all record I/O operations are synchronous; control is returned to the user program only after the requested I/O operation is performed.

Because GET and PUT operations process logical records within a virtual block, only a limited number of GET or PUT operations result in an actual I/O transfer, that is, when the end of a data block is encountered. Therefore, all GET and PUT I/O requests will not necessarily involve a physical transfer of data.

The File Storage Region
The file storage region (FSR) is an area allocated in the user program as the working storage area for record I/O operations. The FSR consists of two program sections which are always contiguous to each other. The first program section of the FSR contains the block buffers and the block buffer headers for record I/O processing. The user determines the size of the area at assembly time. The number of block buffers and associated headers is based on the number of files that the user intends to open simultaneously for record I/O operations.

The second program section of the FSR contains impure data that is used and maintained by FCS in performing record I/O operations. Portions of this area are initialized at task-build time, and other portions are maintained by FCS. This program section is intentionally isolated from the user to preserve its integrity.

The size of the FSR can be changed, if desired, at task-build time.

The data flow during record I/O operations is depicted in Figure 6-5. Note that blocks of data are transferred directly between the FSR block buffer and the device containing the desired file. The blocking and deblocking of record during input is accomplished in the FSR block buffer during output. Note also that FCS serves as the user interface to the FSR block buffer pool. All record I/O operations initiated through GET and PUT calls are totally synchronized by FCS.

Data Transfer Modes
When record I/O is used, a program can gain access to a record in either of two ways after the virtual block has been transferred into the FSR from a file:

MOVE MODE   Individual records are moved from the FSR
                             buffer (as shown in Figure 6-5). Move mode
                             simulates the reading of a record directly
                             into a user record buffer, thereby making
                             the blocking and deblocking of records
                             transparent to the user.

LOCATE MODE   The user program accesses records direct-
                             ly in the FSR block buffer. Program over-
                             head is reduced in locate mode, since rec-
                             ords can be processed directly within the
                             FSR block buffer.
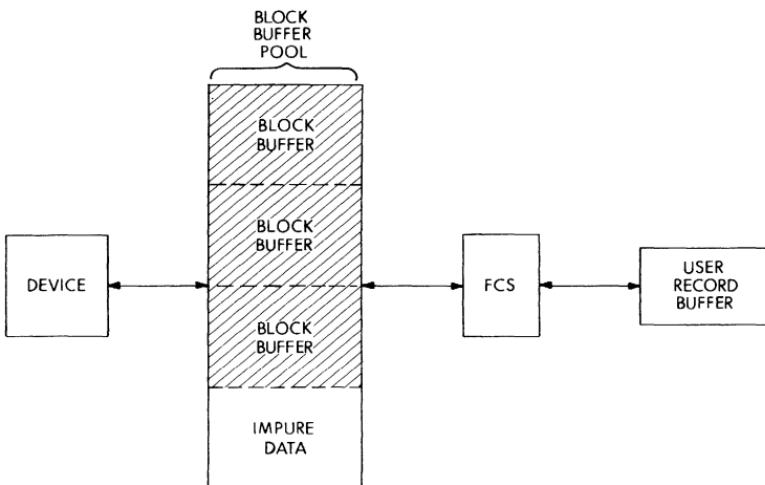
Figure 6-5 Record I/O Operations

Shared Access to Files

FCS permits shared access to files according to established conven-
tions. Two macro calls, among several available in FCS for opening
files, can be issued to invoke these functions. The OPNS macro call is
used specifically to open a file for shared access. The OPEN call, on
the other hand, invokes generalized open functions which have shared
access implications only in relation to other I/O requests then issued.

OPNS allows several active read-access requests and one write-ac-
cess request for the same file. OPEN allows multiple read-access
requests for the same file, but does not permit concurrent write ac-

cess. Note that shared access during reading does not necessarily imply the presence of read requests from several separate tasks. The same task can open the same file using different logical unit numbers.

Spooling Operations
FCS provides facilities at both the macro and subroutine level to queue files for subsequent printing. A task issues the PRINT macro call to the queue a file for printing on the system line printer.

FCS Macros and Macro Use
FCS includes four basic kinds of macro that simplify the user's interface to the system's file control primitives. The four kinds are:

- initialization macros
- file-process macros
- command line processing macros
- the CALL macro

The initialization and file-processing macros are used to establish the data base description and the necessary temporary storage areas needed to perform I/O operations. The command line processing macros are used to dynamically process I/O commands entered from a terminal. The CALL macro is used to invoke file control routines.

The initialization and file-processing macros set up the following structures to define the data base:

- A file data block (FDB) that contains execution-time information necessary for file processing. It defines the basic characteristics of a file, i.e., record type, record, size, access privileges, etc.
- A data set descriptor that is accessed by FCS to obtain the file name, type, version number, and location which are necessary to open a specified file. The data set descriptor is used when a program accesses a given set of known or pre-defined files.
- A default file name block that is accessed by FCS to obtain default file information required to open a file. This is accessed when complete file information is not specified in the data set descriptor. It is used by programs written to access a general set of files.

There are two types of initialization macros: assembly-time macros and run-time macros. Data supplied during assembly of the source program establishes the initial values in the FDB. Data supplied at run-time can either initialize additional portions of the FDB or change values established at assembly time. Furthermore, the data supplied through the file-processing macros can either initialize portions of the FDB or change previously initialized values. The user not only has a

broad range of control over defining the data base characteristics, but also has control over when the definitions are made.

File processing macros also determine the way in which files are processed. These macro calls are invoked and expanded at assembly time. The resulting code is then executed at run time to perform the following operations:

OPEN	Opens and prepares a file for processing.
OPNS	Opens and prepares a file for processing; allows shared access to the file (depending on the mode of access).
OPNT	Creates and opens a temporary file for processing.
OFID	Opens an existing file using the file identification provided in the filename block.
GET	Reads logical records from a file.
GETR	Reads fixed-length records from a file in random-access mode.
GETS	Reads records from a file in sequential access mode.
PUT	Writes logical records to a file.
PUTR	Writes fixed-length records to a file in random mode.
PUTS	Writes records to a file in sequential mode.
READ	Reads virtual blocks from a file.
WRITE	Writes virtual blocks to a file.
DELETE	Removes a named file from the associated volume directory and deallocates the space occupied by the file.
WAIT	Suspends program execution until a requested block I/O is performed.
PRINT	Queues a file for printing on a special terminal or line printer.

In summary, the file-processing macros allow the user to specify random access or sequential access to files, and perform block oriented

or record oriented file processing. In addition, the PRINT macro allows the user to spool files to a line printer or terminal device.

The command line processing macros allow the user to access special routines available in the system object library. The Get Command Line (GCML) routine accomplishes all the logical functions associated with the entry of a command line from a terminal, an indirect command file, or an on-line storage medium. The Command String Interpreter (CSI) routine takes command lines from the GCML input buffer and parses them into appropriate data set descriptors required by FCS for opening files.

The CALL macro allows the user to access a special set of file control routines. These routines allow a MACRO program to perform the following operations: find, insert, or delete a directory entry, rename a file, extend a file, mark a temporary file for deletion, and delete a file, among other operations.

RMS-11 RECORD MANAGEMENT SERVICES
Digital Equipment Corporation's Record Management Services provides a set of general purpose file handling capabilities. RMS-11 allows user-written application programs to create, access, and maintain data files with efficiency and economy.

RMS-11's variety of file organizations and access modes gives the user the ability to choose those methods best suited to the application. RMS-11 files can be organized sequentially, relatively, and by the indexing method. Based upon these file organizations, RMS-11 records can be accessed in a number of ways:

- Sequentially
- Randomly by relative record number or by indexing on one or more keys or by a unique Record's File Address (RFA).
- By dynamic access, a mixture of sequential and random access modes.
- Directly by physical location of data.

RMS-11 complements DBMS-11, DIGITAL'S data base management system, by providing file and record handling capabilities for those applications whose size and data structures do not suggest the need for central data administration and the complete data base management services of DBMS-11. Thus, DIGITAL customers are provided with a growth path from keyed access data management to data base management.

RMS-11 includes a set of utility programs for the creation and maintenance of files, and a set of operating system routines through

which records are transmitted to and from user programs. Under RMS-11, records are regarded by the user program as logical data units that are structured and accessed in accordance with application requirements. As a result, programmers can retrieve data from a file without having to know the exact format and internal structure maintained by operating system routines. Thus, the RMS-11 user has a sense of transparency with the file system interface because RMS-11 handles much of the data buffering and shared access control responsibilities. Other major features of RMS-11 are:

- Collections of record entries are organized by RMS-11 so that no pre-sorting is required in the creation of indexed files.

- When new records are inserted into an indexed file, RMS-11 incrementally reorganizes the file, thereby retaining the ablility to access records efficiently and eliminating the need for overflow areas.

- Support of both fixed length and variable length records furnishes efficient space utilization and access. Additional space/performance controls give the user flexibility in the configuration of buffer space and storage areas.

- For applications that require data file organizations supporting access by one or more user-specified key values, RMS-11's file handling capabilities can be extended by adding the multi-keyed access option. This option provides both generic and approximate key searches to maximize data retrieval capabilities.

- Concurrent access features enable data sharing in multi-user, multi-application environments, and reduce redundant data occurrences. System controlled record locks provide data integrity during concurrent access.

- Multi-level file private control provides protection against unauthorized data access.

- Application programming is simplified through multiple high-level languages.

- RMS-11 used with ANS-74 COBOL programs provides compatibility across systems for decreased training costs and maximum program portability.

- RMS-11 includes a comprehensive set of utilities for file creation and maintenance.

  DEFINE: creates and defines the attributes of a file.

  DISPLAY: displays the attributes of a single file or a group of files.

  CONVERT: provides for easy transfer and conversion of data from any specified input file to a given output file.

BACKUP: protects against the loss of data due to hardware failure or software error by creating a back-up copy on an alternate storage device.

RESTORE: restores previously backed-up files to their original state.

SYSTEM UTILITY PROGRAMS

The RSX-11M system provides a wide variety of system utility programs. RSX-11S provides two utility programs, the On-Line Task Loader and the System Image Preservation program. This section describes the RSX-11M system utilities.

There are two sets of system utilities: those used primarily for program development and debugging, and those used for general purpose file manipulation. The common set of program development utilities is:

EDI and EDT Editors

The editors are used to enter source programs or data files into the system and to modify them as needed. A large set of easy-to-use commands makes the editors effective program development tools. EDI is the traditional editor and EDT is the newer line-oriented editor which has video terminal features.

SLP Source Input Program

SLP is an editing program used to create and maintain source language files on disk.

TKB Task Builder

The task builder creates loadable memory images from assembled or compiled tasks. It links relocatable object modules and resolves any references to global symbols, common areas, and shared libraries. The task builder is used to specify a task's attributes, such as check-pointability, priority, etc. The task builder is also used to create sharable commons. The task builder provides an overlay descriptor language to construct task overlays. The overlay descriptor language simplifies the process of dividing a task into overlaid segments and specifying load methods. Finally, if it is requested by the task-build command, the user can obtain a cross reference of all global symbols defined or referenced in the task. There are three different task builders supplied: a small one for small systems, a larger one which supports all of the features and a third one in which certain features have been omitted in order to produce very fast executions.

LBR Librarian

LBR provides the capability to create and maintain disk-resident libraries of object modules and user-defined macros.

ODT On-Line Debugger

ODT aids the user in debugging programs that have been assembled or compiled and task built. From the keyboard, the user interacts with ODT to:

- print or change the contents of a location in the task
- run the program using the breakpoint features to halt the program at specified points
- search the program for a specific bit pattern
- calculate offsets for relative addresses

Trace capability is also provided to aid in the debugging of FORTRAN programs.

ZAP Task Patch

ZAP provides a facility for examining and modifying task image files and data files. With ZAP, permanent patches can be made to task image or data files without having to re-create the file.

PAT Object Module Patch

PAT allows the patching or updating of code in a relocatable binary object module.

XDT Executive Debugging Tool

XDT, a subset of ODT, is an interactive debugging tool for executive modules, I/O drivers and interrupt service routines.

BAD Block Locator

BAD is a utility which determines the number and location of bad blocks on disks and records this information on the last good block of the disk. RSX-11M will read this information when the disk is initialized.

PMD Post-Mortem Dump

PMD is a debugging tool that can generate a memory dump for a task that terminates abnormally (called a post-mortem dump) or an edited memory dump for a running task (called a snapshot dump). Snapshot dumps can be requested any number of times during task execution. In general, both post-mortem and snapshot dumps provide the following information: contents of the CPU registers, including the stack pointer, program counter, processor status word, and floating point registers (if the program used the latter), the task status code, event flags set, overlays that were in memory at the time, outstanding I/O requests, LUN assignments, etc.

The set of general purpose file manipulation programs includes:

PIP Peripheral Interchange

PIP is used to copy files from one device to another, for example, from disk to printer, to rename files, to list files, and to delete files.

FLX File Exchange

FLX is a special purpose file copy utility. It can copy a file in Files-11 format and covert it to either RT-11 or DOS/BATCH format on another device, or copy a file in either RT-11 format or DOS/BATCH format on another device, or copy a file in either RT-11 or DOS/BATCH format and convert it to Files-11 format. It can also copy files in Files-11, RT-11 or DOS/BATCH format to another device in the same format. Valid RT-11 devices for copy operations are RK11 disk, cassette, and DEC-tape. Valid DOS/BATCH devices for copy operations are RK11 disk, DECtape, magnetic tape, and cassette. RSX-11M also supports RT-11 format on floppy disks and both RT-11 and DOS/BATCH formats on paper tapes. (Note that RSTS/E can use the DOS/BATCH format for DECtapes. It is therefore possible to use FLX to transfer files to and from RSTS/E systems as well.) Furthermore, FLX can initialize a cassette in RT-11 or DOS/BATCH format, list RT-11 or DOS/BATCH cassette directories, and delete files on RT-11 or DOS/BATCH volumes.

DMP File Dump

DMP enables the user to obtain the listing of Files-11 files or volumes in either ASCII or octal format.

DSC Disk Save and Compress

DSC enables the user to backup/restore disk volumes to magnetic tape or other disks and to combine unused blocks on disks to create contiguous blocks. DSC comes both as a stand-alone and an on-line program.

VFY File Verification

VFY checks the consistency and accuracy of the Files-11 file structure on a Files-11 device, for example, a disk. It also prints the number of available blocks in a volume, locates files that could not otherwise be accessed, and lists the names of files on the volume.

CMP File Compare Utility

CMP is a utility which will compare, line by line, two ASCII files. Its output can be either a new file with all the differences encountered, a listing of one with change bars marking the differences, or an output suitable for input to the SLP utility.

RSX-11M SYSTEM SUMMARY

Is

- Real-time processing
- Sensor based
- Data base management
- Multi-user development
- Building block operating system for:
  - Communications
  - Commercial applications
  - Turn-key applications

**Is not**

- Batch processing
- Timesharing
- Protected environment

Includes Data Management/Utilities

- RMS-11
- DBMS
- DATATRIEVE-11
- SORT-11

Languages

- COBOL
- FORTRAN IV
- FORTRAN IV-PLUS
- MACRO-11
- BASIC-11
- BASIC-PLUS-2
- RPG II

INTERACTIVE APPLICATION SYSTEM
IAS (V2)

OVERVIEW

IAS supports concurrent real-time, timesharing, and batch process-
ing, making it the ideal multi-purpose operating system. Since an IAS
system also offers powerful easy-to-use program development, it is a
natural host for smaller RSX-11 based systems in a distributed com-
puting arrangement. It also offers its users data base management
capabilities.

FEATURE TOPICS

- Functions and Features
- IAS Executive Organization and Services
  - Active Task List
  - The Timesharing Schedule
  - Batch Processing
  - System Generation and Initializaton
- Command Language Interpreters
  - Program Development System (PDS)
  - PDS Commands (Table 7-2)
- System Control Interface (SCI)
- Timesharing Control Primitives
- IAS System Summary

OPERATING SYSTEM FUNCTIONS AND FEATURES

IAS is a large general purpose operating system that runs on a PDP-11/70, PDP-11/60, PDP-11/55 or PDP-11/45 processor. It is a multi-user timesharing system that supports concurrent interactive, batch and real-time applications. It includes the MACRO assembler. As options, FORTRAN IV, FORTRAN IV-PLUS, COBOL, SORT, CORAL-66, BASIC, and BASIC-PLUS-2 language processors can be added, as well as the RMS and DBMS data and record management facilities. IAS features:

- multiple programming languages: FORTRAN, BASIC, COBOL, CORAL-66, SORT, and MACRO

- a single, easy-to-learn and use interactive command language

- priority scheduling for real-time tasks

- submission of batch jobs from interactive terminals

- timesharing services for development of interactive applications programs

- a simple internal software interface for the development and use of special-purpose, multi-user interactive applications

- a sophisticated file system providing device independence; file protection; sequential, random, and relative file access; and, optionally, multi-keyed ISAM

- dynamic allocation of system resources

- use of shared, reentrant code to minimize memory requirements

- system management facilities for system configuration, generation and control

- facilities to account for and restrict the use of system resources

IAS supports a variety of peripherals useful in batch and real-time applications, including line printers, card readers and laboratory peripherals.

As an interactive time-sharing system, IAS presents an easy-to-use system interface. The program development system, PDS, provides a computing environment that supports most application processing requirements of IAS users. As such, it presents to IAS terminal users a standard interface which requests and processes valid passwords and user names before making system facilities available to the user. The interface allows the user to create programs, submit jobs to the batch stream, and issue commands to create and manipulate program and data files.

As a batch system, IAS services multiple queues of batch jobs. FORTRAN, MACRO, and COBOL jobs can be submitted to batch. The

user interface for batch processing is the same as the PDS interactive interface. Therefore, programs can be developed in interactive mode and run in production in batch mode. The system manager can control the amount of service that batch jobs receive from the processor. In particular, the system manager can guarantee a minimum processor time for batch processing.

As a generalized, flexible base for executing interactive applications, IAS provides support for application-specific user interfaces for applications such as data entry, bank teller terminals or engineering computation, where it is necessary or desirable to present a customized interface to terminal users (operators, for example).

Further, IAS supports the concurrent execution of multiple interactive applications. Thus, a data processing application and the program development system can execute concurrently and be serviced jointly by the timesharing facilities of the system.

The interactive application facility is further enhanced by the capability of the FORTRAN IV-PLUS compiler and IAS to develop and support sharable programs. For the user, this means that system overhead (memory occupancy and swapping time) is minimized. Also, the user can allocate specific application interfaces and deallocate them as required. This facility is flexible and extendable. The system is easily modified and additional applications are easily added.

The special-purpose interfaces can be written and checked out using the IAS program development system and then installed by the system manager for use on specific terminals. IAS provides a number of system services that can be called from the application program to enhance the function of these special-purpose interfaces.

IAS was built from the RSX-11D operating system. It provides, therefore, the RSX-11D real-time processing facilities of multiprogramming, priority scheduling, power-fail restart, contingency exits, disk-based operation and task checkpointing of real-time tasks. Real-time, interactive, and batch operations can occur concurrently and, normally, in that order of priority.

IAS system operations are managed by two executives. The real-time executive schedules real-time activities according to their priorities and manages the system resources not allocated to the timesharing activities. The timesharing executive schedules timesharing users on the basis of a time-slicing algorithm when real-time activities do not take precedence. Batch processing normally uses processor time available after interactive users are serviced. Both batch tasks and interactive tasks run under control of the timesharing scheduler.

Table 7-1 provides a summary of the IAS system's features.

Table 7-1 IAS System Summary

System type	Concurrent interactive, batch, and real-time processing system with multi-language support
CPUs supported	PDP-11/70, PDP-11/60 (in 11/40 mode),PDP-11/55, and PDP-11/45 with memory management
Memory size range	Minimum: 64K words Maximum: 124K words, 1920K words on a PDP-11/70
Batch processing	Standard spooled queue and optional uns-pooled card reader
Languages	MACRO and optionally FORTRAN IV, FORTRAN IV-PLUS, COBOL, BASIC, BASIC-PLUS-2, and CORAL-66
System tasks and special utilities	VFY File and Media Verification Program, and special BAD Bad Blocks Reporting Program CDA Core Dump Analyzer, PRE Media Preservation Program, Error Logging and Diagnostics Package
File system	RSX-11 family's Files-11

IAS EXECUTIVE ORGANIZATION AND SERVICES

To provide system flexibility, the IAS operating system is controlled by a system monitor consisting of a real-time executive kernel and a timesharing executive. The primary functions of the kernel include memory and disk management, supervision of privileged tasks (including real-time tasks and device handlers), file management, and maintenance of the general integrity of the system. The kernel maintains the Active Task List (ATL) to control task dispatching.

The timesharing executive controls both interactive and batch processing. It controls the execution of timesharing tasks by time slicing and by swapping tasks in and out of memory.

The Active Task List

The kernel coordinates the dispatching of all tasks on the system by scanning the entries in the Active Task List (ATL). The ATL is a priori-

ty-ordered list of all resident active tasks in the system. Because of their requirement for immediate service, the I/O device-handler tasks are put at the top of the ATL. For the same reason, any user-designated real-time tasks are assigned to high-priority levels. The timesharing executive, which runs at a lower priority than I/O and real-time tasks, controls the scheduling of user timesharing tasks by inserting tasks in the ATL. Figure 7-1 illustrates the priority structure of the ATL.
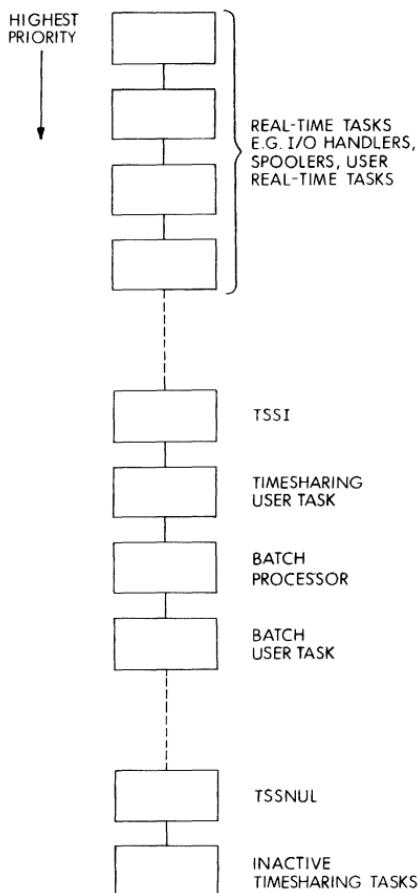

Figure 7-1 Schematic Diagram of ATL Structure

The timesharing scheduler uses two tasks, TSS1 and TSSNUL, to control the dispatching of tasks. The timesharing scheduler task TSS1 selects a task for executing by placing its entry in the ATL at a priority of one less than itself. The scheduler task then gives up control (for example, waits for an event flag such as time slice complete) to allow the kernel to dispatch the user task. TSSNUL is the null job and runs continuously in a loop so that tasks below it on the ATL can never execute. When a timesharing task is not executing, TSS1 places the ATL entry for that task below that of TSSNUL. TSSNUL always executes at priority 1.

The Timesharing Scheduler

The prime objective of the scheduler is to reduce as far as possible the average response time to all user demands. In order to do so, the scheduler distinguishes between various levels of user importance and urgency of service. The scheduler maintains a number of round-robin queues, or levels, of tasks to be scheduled. The scheduler scans each level (high to low) in a round-robin fashion until it finds a memory-resident runnable task. A non-resident ready-to-run task will cause the swapping system to be activated.

A task which uses a full time quantum is transferred to the next lower level unless it is already at the lowest level. Tasks at lower levels are not scheduled as often as tasks at higher levels. Tasks allocated to a lower priority level are given a longer time quantum when next activated. Thus, large jobs are run and swapped less frequently, but in compensation, receive more processor time once activated.

To prevent tasks from being starved of processor time because the scheduler is continuously scheduling higher priority tasks, a means of promoting tasks from one level to the level above is provided. If, over a given period of time, no scheduling has been performed at a given level, then a task at that level is moved to the bottom of the level above.

If the scheduler finds a runnable task that is not resident, then the task must be loaded into memory to receive its quantum of CPU time. Space is created in memory by moving resident tasks to create the required contiguous space, and, if necessary, by writing inactive tasks to the swap area on disk(s).

Two time factors are associated with every task. The quantum determines the amount of CPU time a job may have before it is swapped out of main memory. The time slice is the maximum CPU time a task is allowed to use before a rescheduling operation is performed.

The time quantum for a particular task is determined by:

$$Q = At + C$$

where:

A   is a factor (in clock ticks) assigned to a task when it is
                loaded: for example, 1 tick per 1K words in the task.

t   is a time factor associated with the scheduling level; t
                increases as the level number increases.

C   is the minimum guaranteed quantum for the system.

The quantum for a task at a low scheduling level may be quite large. In order not to block other higher priority tasks awaiting service, the scheduler calculates the quantum of the task, and then allocates the task a number of time slices. At the end of each time slice, the scheduler will try to run higher priority tasks. However, the task will not be swapped until its quantum has expired. If the task enters a wait state, however, the quantum will be set to zero. In this case, therefore, it will be made available for swapping.

The time slice parameter can be adjusted to achieve the desired compromise between responsiveness and system throughput. If the time slice is set to its maximum value, all tasks will execute without interruption for their entire quantum. The time slice should never be smaller than the maximum quantum for a Level 1 task. All the parameters of the scheduling algorithm can be adjusted by the system manager to tailor IAS scheduling to the needs of the local installation.

Batch Processing
Batch runs as if it were another timesharing terminal. The batch command language is the same as the general purpose interactive program development command language, and it is processed by the same command language interpreter (see below).

The batch processor obtains its command input from a queue of commands. The batch queue is maintained independently, thus enabling jobs to be submitted to the queue at any time. The processor can service two types of queues. The system can maintain a spooled queue which consists of: 1) batch job files submitted from interactive terminals, and 2) command input from the card reader (if the card reader is designated as a spooled device). The batch processor can also service a queue of commands directly from the card reader if it is designated as an unspooled device.

Batch processing is initiated and terminated by the system manager. The batch processor executes at the batch scheduling level where it is serviced by the timesharing scheduler. Batch processing shares CPU time with interactive tasks, but its priority for service is always below that of the active tasks.

To assure that batch processing receives adequate service, the system manager can specify the percentage of CPU time to be made available to it, and the length of time (quantum) batch should run when it does receive service. For example, the system manager could direct IAS to devote ten percent of the available time to batch jobs in 2-second quanta. Tasks in the batch level are not subject to the promotion/demotion mechanism of the timesharing scheduler; that is, tasks remain in the batch level for as long as they are executing.

Batch user tasks share space with the interactive tasks (if any) currently executing. While any space not currently in use by batch is used for interactive processing, batch can be guaranteed space so that requirements up to that maximum will always be satisfied by swapping interactive tasks out of memory if necessary.

Executive Data Structures
The IAS system maintains a number of common areas in which the various executive tasks store information and communicate with each other. SCOM contains system tables, the kernel node pool, lists, and some servicing routines. SYSRES, the System Resident Library, contains common routines which will be used by most tasks. IASCOM is a library containing timesharing nodes, lists, tables, and common routines for manipulating the timesharing data structures. IASBUF is a buffer area used for communication between the timesharing control primitives, IASCOM, and the timesharing executive.

I/O Services and Device Independence
Input and output constitute a significant part of all programmed activity. Thus, IAS provides a variety of services to perform these operations.

The IAS file system is a collection of system services that permits the user to view I/O as a transaction between a program and a named, protected collection of records known as a file. The file system manages all data transfers and provides the mechanism whereby a file intended for a record-oriented device, such as a line printer, can be dynamically directed to an area on magnetic storage.

Access to a user's files stored on a disk, DECtape or labeled magnetic tape is controlled by a protection specification on each file. When creating a file, a user can specify whether other users may have access to the file and, if so, whether they may modify the file or merely read it.

One of the goals of any file system is to make the user program independent of the I/O hardware. Thus, while the storage characteris-

tics of a medium are organized around physical records, the user deals only with logical records.

To provide greater device independence, the IAS user will in general use logical units instead of referring directly to physical devices. IAS provides a set of logical unit numbers (LUNs) which are not associated with specific physical devices or files until run time. In the source program, all device and file references use LUNs. These LUNs may be assigned to particular devices by a command issued before the program is executed.

Sharing of Common Routines
In a system designed to support many users, there is a high probability that many tasks will use the same code sequences, such as mathematical routines and specialized I/O routines.

The common code could be built directly into each task requiring it, but this might result in several copies of the same code occupying memory space at the same time. The alternative employed by IAS is to put the common code where all users can share it, so that only one copy of the code is required. The IAS system uses shared code heavily.

Under IAS, shared areas may be data areas (global common), sets of common routines (libraries), or the pure (read-only) areas of complete tasks (shared tasks). Global common areas allow simultaneously active tasks to share data. A sharable library consists of routines which may be interrupted to service another request, then resume execution later at the point of interruption. Users who write reentrant routines can include their own sharable libraries in the IAS system. Shared code does not need to be permanently resident; it can be loaded at the time a task which uses it is run. Programs written in either FORTRAN IV-PLUS or MACRO can be shared.

System Generation and Initialization
System generation is the process by which a collection of system services is tailored to meet local physical constraints and performance requirements.

IAS consists of a set of independent program segments which can be linked selectively to eliminate services not required at a given installation. For example, a system manager might eliminate the device handlers for devices not included in the hardware configuration.

During system generation, the system manager also defines and names the partitions in which programs will execute. This normally includes defining a timesharing partition and any special partitions dedicated to the execution of real-time applications.

After generating the system, the system manager runs a special start-up task to initialize the timesharing system. The start-up task prompts for a series of parameters that specify the system configuration for the current session. The parameters can be entered at the terminal, or read from a predefined file. The values specified for the start-up parameters override the defaults specified at system generation.

The start-up parameters include:

- Terminals to be allocated for timesharing use
- Devices to be made available for timesharing users
- Devices to be used for swapping
- System control parameters
- Partition names for timesharing
- Partition names for executive tasks

Once the start-up parameters are established, the system operator enables general timesharing users to access the system by allocating timesharing terminals to the system's command language interpreters.

COMMAND LANGUAGE INTERPRETERS

A command language interpreter (CLI) is a task which interfaces with a person who uses the IAS system. PDS, the program development system CLI supplied with IAS, allows the general user to access all non-privileged facilities of the system. Another CLI called the system control interface (SCI) allows the system operator to alter the state of the system, to designate user interfaces (CLIs), and to allocate facilities to each user.

Normally, PDS is the standard CLI to which a general terminal in the IAS system is allocated. Using the SCI interface, the system operator can designate a specific task other than PDS as the CLI for a terminal. For example, the system operator might set aside one terminal to be used solely for program editing. When EDIT is designated as the only CLI for that terminal, EDIT will be invoked when CTRL/C is typed, and a user at that terminal will not be able to issue commands to anything except the editor.

Users can write their own CLI tasks. The CLI tasks can be installed and allocated timesharing terminals. This means that the system can present a number of different terminal interfaces. A user-written CLI task can define its own command language, which can be as simple and understandable as required. It can be specifically designed for a particular application operation. Application terminal users do not, therefore, have to learn a generalized command language such as PDS to perform their subset of daily activities.

A CLI is written as a normal, non-privileged user task which can use, in addition to the standard system directives and file system facilities, the IAS system's timesharing control primitives (see below). A CLI can be written in any language which provides the facilities it requires; for example, a CLI that wishes to use the system QIO directive must be written in FORTRAN, MACRO or BASIC (with user-defined functions).

After a task has been installed as a CLI, IAS automatically provides certain task execution controlling functions. For example, when CTRL/C is typed on a terminal allocated to a CLI, a copy of the non-shared part of the CLI is activated. If the task specifically requests the information, IAS will inform the task of any events happening at its terminal or terminals.

The following two sections describe the two standard CLI tasks provided with the IAS system: PDS, the program development system, and SCI, the system control interface.

Program Development System (PDS)

A typical timesharing user interfaces with IAS through the program development system (PDS) command language interpreter. Under PDS, users can create, compile, link, load, and run programs. They can submit jobs to the batch stream, use various peripheral devices, and obtain system information.

PDS is a prompt-oriented system. After PDS is activated at a terminal, either by the autostart mechanism or by typing a CTRL/C, PDS invites the input of a command by issuing the prompt "PDS>". The user replies by typing a command name and its parameters, if any, followed by a carriage return. If a user does not supply all the parameters required in a command, the system will prompt the user for them. Additionally, the user can issue the HELP command to display the commands available.

As an example, the user can log in to the system, using the LOGIN command, in two ways. If the user desires the prompts, the user can simply type the command LOGIN in response to the PDS prompt.

PDS> LOGIN	The user issues the command.
USERID? SMITH	The user ID is a 1- to 12-character user name which identifies a person to the system. PDS requests a user ID.
PASSWORD?	PDS requests a password. The password is not displayed.

USER SMITH UIC[200,200] TT07: TASK 25 22:30:07 27-SEP-77

> PDS validates the user name and password
> and accepts the user to the system by print-
> ing information relevant to the user's job.

If the user types a user name after issuing the LOGIN command, PDS does not prompt for a user name, it prompts only for the password.

PDS> LOGIN SMITH
PASSWORD?
USER SMITH UIC[200,200] TT07: TASK 25 22:30:52 27-SEP-77
PDS>

As another example, the user can issue a command to rename a file in any of three ways. If the user simply types the command name RE-NAME, PDS prompts for the old file specification and the new file specification parameters.

PDS> RENAME
OLD? MATRIX.FTN
NEW? BACKUP.TMP
PDS>

If the user does not want the prompts, the user can enter the entire command on one line.

PDS> RENAME MATRIX.FTN BACKUP.TMP
PDS>

If the user issues the command name followed by a carriage return, but does not need the second prompt, it is also acceptable to enter the command parameters on the line with the first prompt.

PDS> RENAME
OLD? MATRIX.FTN BACKUP.TMP
PDS>

The user can supply PDS commands in a file rather than typing them in one at a time on the terminal. The user creates a file containing the commands PDS is to execute, called an indirect file. To execute the commands in the file, the user replies "@filename" to a PDS prompt, where "filename" is the name of the indirect file. PDS processes the file in the same manner that it processes commands typed individually on the console. The commands, as well as any error messages that occur during the execution of the commands, will be displayed on the user's output device.

For example, suppose the user creates an indirect file named PERF.CMD containing the PDS commands to compile, link and run

the source program PERF.FTN. By typing the command "@PERF.CMD" in response to a PDS prompt, PDS will execute the command file.

PDS> @PERF.CMD   The user issues the indirect file command.

FORTRAN/LIST:PTEST PERF   The first command requests the FORTRAN compiler to compile the source program named PERF.FTN (the .FTN extension is assumed by default) and produce an object program (named PERF.OBJ by default) and a listing file named PTEST.LST (the .LST extension is assumed by default).

22:34:17 TASK TERMINATION CORE SIZE 20K CPU TIME 01.05

                                   PDS prints a message when compilation is complete.

LINK PERF   The second command requests the linker to link the object program named PERF.OBJ. (The .OBJ extension is assumed by default.)

22:35:49 TASK TERMINATION CORE SIZE 15K CPU TIME 14.41

                                   PDS prints a message when linking is complete.

RUN PERF   The third command requests PDS to execute the program PERF.TSK (the extension .TSK is assumed by default).

22:35:58
22:37:12 TASK TERMINATION CORE SIZE 10K CPU TIME 00.13

                                   PDS prints messages regarding program execution.

PDS>   After processing the command file, PDS indicates that it is ready to accept another command.

There are several types of PDS commands; commands that provide access or system information, commands that allocate resources,

commands that manipulate files, and commands that control task execution. The system manager can designate certain PDS commands as privileged or non-privileged for any particular user. That is, when defining the user accounts, the system manager specifies which PDS commands each user can issue. For example, some PDS commands control real-time task execution. Only those users who have been given real-time execution privileges can issue the real-time execution control commands.

Except for the LOGIN, LOGOUT, JOB, and EOJ commands, all non-privileged commands can be issued in either interactive or batch mode. When a command is issued in batch mode, it requires a dollar sign ($) preceding the first character of the command name.

Table 7-2 lists the general PDS commands. Commands that contain the term "real-time" in their description are available only to the users with real-time execution privileges. All other commands listed are non-privileged.

Table 7-2 PDS Commands Summary

System Initialization Commands

SET	Used to change system and device defaults for a particular user.
PASSWORD	Changes password.
LOGIN	Initiates an interactive session at a terminal.
LOGOUT	Ends an interactive session at a terminal.
JOB	Denotes start of a batch job.
EOD	Denotes end of data in a batch job.
EOJ	Denotes end of a batch job.

System Informational Commands

HELP	Displays a list of all available commands.
SHOW	Displays system and device defaults and various types of system status information.
MESSAGE	Sends a message to the system operator.

Job Control Commands

ALLOCATE	Reserves a device for single user access.
DEALLOCATE	Releases a device.
ASSIGN	Associates a user-specified logical name or a physical device with a logical unit number.
DEASSIGN	Disassociates a logical name or physical device with a logical unit number.
CANCEL	Cancels the periodic scheduling of requests for a real-time task.
MOUNT	Requests mounting of a volume or volume set.
DISMOUNT	Requests operator to dismount a volume.
GOTO	Transfers control in an indirect command file or batch command file.
ON	Allows for testing of errors in an indirect or batch command file.
STOP	Prevents all further processing in a batch or indirect command file.
INSTALL	Install a real-time task.
RUN	Initiate execution of a user program (used also for real-time task execution).
ABORT	Kills a suspended user program or command.
CONTINUE	Restarts a previously suspended program or command.
QUEUE	Queues a file for printing or queues a batch job.
SUBMIT	Submits a job for batch execution.
ALTERPRIORITY	Changes priority of a real-time user task.
FIX	Inhibits the checkpointing of a real-time task.
UNFIX	Allows a real-time task to be checkpointed.

CANCEL   Cancels periodic rescheduling of a real-time task.

REMOVE   Removes an installed real-time task.

File Manipulation Commands

APPEND   Appends one or more file(s) to another.

COMPARE   Allows the line-by-line comparison of two input files.

COPY   Copies one or more files.

CREATE   Creates a disk file from card or terminal input or creates a directory file.

DELETE   Deletes one or more files.

MERGE   Takes records from a sequential, indexed, or relative file and merges them with an indexed or relative file.

DIRECTORY   Displays the names of files in the indicated directory.

DUMP   Produces a printed listing in ASCII (or octal) of the contents of a file.

PRINT   Prints a file or files on the system's printer.

RENAME   Changes the name of a file.

SORT   Sorts a file into a specified sequence.

TYPE   Types a file at the user's terminal.

INTIALIZE   Initializes a foreign (DOS and RT11) volume.

UNLOCK   Unlocks a file that was locked.

PDS Program Development Commands

EDIT   Invokes the interactive editor or, optionally, the line editor.

MACRO   Invokes the MACRO assembler.

LINK   Invokes the linker to link together
                            FORTRAN and/or MACRO object modules.

LIBRARIAN   Invokes the librarian to create object pro-
                            gram libraries.

FORTRAN   Invokes either the FORTRAN IV or
                            FORTRAN IV-PLUS compiler.

COBOL   Invokes the COBOL language processor.

BASIC   Invokes the BASIC subsystem.

CORAL   Invokes the CORAL-66 compiler.

In addition to the PDS commands, IAS supports special terminal con-
trol commands issued from the terminal. These control commands
are:

CTRL/C   Returns control to PDS (suspends a running program).
CTRL/U   Deletes current line.
CTRL/I   Skips to next tab position.
CTRL/K   Vertical tab.
CTRL/L   Form feed.
CTRL/O   Enables/disables terminal output.
CTRL/R   Retypes the current input line.
CTRL/S   Suspends current output until CTRL/Q is typed.
CTRL/Q   Resumes current output.
CTRL/Z   Generates an end-of-file.

In addition to the general PDS commands, IAS includes special PDS
commands available only to the system manager. The system manag-
er must be logged in under the system management account to gain
access to these privileged PDS commands. There are three types of
privileged PDS system management commands:

• accounting commands to authorize users and report system use

• real-time system control commands

• volume and file control commands

System Control Interface (SCI)
The system operator communicates with IAS through the system con-
trol interface (SCI) command language interpreter. The SCI command
language uses the same syntax and conventions as the PDS com-
mand language, including prompting for missing parameters. Indirect
SCI command files are also supported.

SCI commands enable the operator to monitor the system in four different areas:

- command language interpreter control
- overall system and task control
- peripheral device control
- system information

The command language interpreter (CLI) commands allow the operator to install and remove CLI tasks, allocate and deallocate resources (e.g., terminals) to a CLI task, and abort a CLI task at a particular terminal. These commands are used both to initialize a timesharing system and to modify the system's characteristics during system operation.

The system and task control commands enable the operator to: load and unload device handlers which are not permanently resident; mount and dismount volumes; set the system parameters to suit the current workload; and shut down the system. These commands also enable the operator to have ultimate task execution control. For example, the operator can terminate any task in the system. This can be useful when, for example, a batch task loops indefinitely because of internal errors,.

Peripheral device control commands provide the operator with the facility to service user requests for access to disk packs, magnetic tapes or other removable media. Additionally, the operator can control the output spooling mechanism and the type of printer forms being used.

The system information commands allow the system operator to display system information such as the active task list, CLI allocations, partition names and sizes, date and time, and device status.

TIMESHARING CONTROL PRIMITIVES

IAS provides an installation with a convenient mechanism for implementing special-purpose interactive applications systems such as inventory control, order entry, on-line file update, etc. Programs written in either FORTRAN IV-PLUS or MACRO are ideally suited for this purpose since the program, if reentrant, can be shared by multiple users. The programmer writes a program as if it were responding to only one terminal, thus eliminating many of the problems associated with interactive multi-user applications. If the application requires special system services or interlocks between users, it can use the system directives, the file system, and the IAS timesharing coordination facili-

ties to perform these functions. The timesharing facilities are provided through a set of routines called timesharing control primitives (TCP).

A task designated as a CLI obtains service from TCP by issuing calls to specify desired operations. TCP runs at a higher priority level than the timesharing scheduler to provide a high service level to CLI tasks and to ensure that up-to-date system information is always available to the timesharing executive. TCP presents a kernel handler interface to the system; the basic method is through a QIO (Queue I/O) directive.

There are eight kinds of TCP routines available to the user writing a CLI. For protection purposes, the system operator can control the privileges of a CLI task. When installing the CLI task, the operator specifies which TCP facilities the CLI can use. The following sections describe the TCP routines.

CLI Control Primitives

These primitives provide the necessary CLI authorization and allocation to terminals required to establish the timesharing environment. This group provides the facilities for dynamically controlling the CLI population of the system and the allocation of terminals to those CLIs. The CLI control primitives are:

Initialize a CLI

A task is initialized either at system start-up by the IAS executive or at any subsequent time by the system manager. The CLI must already be installed. It can be initialized as a batch subsystem (the default is interactive).

Allocate Terminals to a CLI

Allocates or starts up a CLI task for specified terminals or processing spooled input.

Relinquish Terminals for a CLI

Releases the terminals that the calling CLI is servicing.

Task Initiation and Control

Task initiation and control primitives enable a CLI to submit tasks to timesharing either on behalf of the terminal user or for the CLI's own purposes. They also provide the facilities for the CLI subsequently to control the execution of those tasks by suspending, continuing, or aborting the tasks. They are:

Task String Parse

Parses a command string, identifying a task name. It also identifies whether or not a parameter string is present in the command.

Set Up a Job Node
Sets up the information in a job node to enable successful scheduling of a task on submission of the job node to the scheduling queue.

Queue Job Node
Queues a specified job node to be run under IAS. The scheduling level at which the task is to run initially can be specified.

Set Task Termination
Enables a CLI to abort a task previously submitted for scheduling.

Set Task Continue
Enables a task to be continued after being suspended, either as a result of a suspend request or a CTRL/C issued from the terminal.

Set Task Event Flag
Allows a CLI to communicate with a task which it has previously initiated by setting a local event flag for that task.

Buffer Management
A CLI can claim and relinquish buffers by using this group of primitives. They are:

Claim IAS Buffer
Allows a single buffer to be picked from the IAS buffer pool.

Relinquish IAS Buffer
Relinquishes a buffer by returning it to the buffer pool.

Terminal Event Control Primitives
The terminal event control primitives enable TCP to communicate to the CLI events that occur asynchronously with the current CLI activity. For example, TCP can notify the CLI of task termination for the CLI's terminal user, of a CLI exit requested, or that a CTRL/C was received on a terminal. This mechanism allows the CLI to make decisions as to its subsequent actions. The primitives are:

Declare Terminal Event
Declares a terminal event for the CLI to service.

Service Terminal Event
Invoked by the CLI task to service asynchronous events occuring for its terminal user.

Job Node Management
Every task submitted for scheduling under IAS timesharing must have an associated job node. The job node management primitives enable a calling CLI task to control the allocation of job nodes to its user and therefore the user's ability to run tasks.

Claim Job Node
Enables a job node to be picked from the job node pool for the requestor. Normally the job node is claimed on behalf of a terminal belonging to the requesting CLI task, but a CLI can claim a job node for its own use.

Assign a UIC
Assigns a user identification code to a job node. The assigned job node usually belongs to a terminal node but it could also be a floating node claimed by the CLI for some other purpose.

Relinquish a Job Node
Releases a job node currently allocated to the terminal serviced by the calling CLI or releases a specified job node previously claimed by the calling CLI.

System Management
The system management primitives enable the system manager to obtain information about the system and reset the tuning parameters. Access to the system tuning parameters (batch and interactive quanta, maximum number of interactive jobs, etc.) is available only to the system manager.

Set or Report the Timesharing Task Promotions Period
Reports and optionally changes the timesharing promotion period. During timesharing tasks execution, the system allocates tasks among the scheduling levels according to their activity. A task that uses a full time quantum in a high level is transferred to the next lowest level, where the quantum size is greater. The goal is to move highly interactive tasks to high levels, while CPU-bound tasks move to low levels. To avoid having tasks in low levels becoming starved for CPU time, tasks are periodically promoted. If, during the promotion period, no task in a level has been scheduled, the task at the top of that level is promoted to the bottom of the next highe level.

Set or Report the Batch Quantum
Reports the current values of the batch time quantum, the time between batch schedules, and optionally gives new values to one or both of the parameters.

Set or Report the Timesharing Quantum
Reports the current value of and optionally changes the time quantum allocated to a specific scheduling level. The IAS scheduler relies on the quantum values increasing with level number.

Set or Report the Timesharing Quantum Constants
Reports and optionally gives new values to the constants used in cal-

culating the time quantum given to each task in each scheduling level. The quantum given to a task is determined by the relationship:

$$Q = At + C$$

C represents the minimum quantum given to any task and t represents the timesharing quantum allocated to a specific scheduling level. A, the allocation factor, is a function of task size, where $A = N/M$. N is the number of clock ticks allocated for M number of 1000-byte blocks of task size. Using this TCP, the task can modify the values for C, N and M.

Report Task Characteristics
Reports the limits associated with user tasks which run under the timesharing system:

- maximum allowed number of timesharing tasks
- maximum task size
- maximum number of user LUNs assigned
- maximum number of currently assigned devices

Report Time Statistics
Reports the following system statistics:

- elapsed time since start-up
- elapsed time since start of statistics collection
- total time given to timesharing
- total time given to executing timesharing tasks
- total time when no execution occurred
- total time given to batch

Report Task Information
Reports information about the active timesharing user tasks in the system. Any one of the following can be requested: report all user tasks in the system; report all tasks initiated from a specific terminal.

Device Management Primitives
This group of primitives controls the allocation of devices to timesharing users. It enables the control of multiple users of the system who wish to make use of peripheral devices. It allows any number of interactive users (and optionally one batch user) to have simultaneous access to Files-11 volumes or directory devices and exclusive use of foreign volumes and non-directory devices. The primitives also allow the assignment and deassignment of LUNs to devices which are effective for all subsequent timesharing user tasks run on the CLI.

Assign LUN to Device

Assigns a terminal user's LUN to: 1) a specified volume mounted on a given device unit; 2) a specified device; 3) a specified volume mounted on any of the devices of the specified device type. In all cases the device must be one which is allocated as available to timesharing.

Deassign LUN

Deassigns the device assigned to a given LUN for a terminal user or deassigns all LUNs for a given device for a user.

Check Device Allocation

Checks whether a device is in the timesharing user's device map and is on-line.

Record On-Line Volume

Records the information about volumes mounted for the timesharing users. The number of timesharing users using a device is incremented and a device table entry is made for the requestor's terminal, signifying that the volume has been mounted.

Check Device For Mount

Called by the MOUNT program when the MOUNT utility is being run on behalf of a timesharing user. It ensures that the device can be mounted successfully for the user by checking for a free device map entry and checking that no other user has the device's exclusive use.

Relinquish Volume

Makes a device available to other users if the current user has exclusive control. If the user does not have exclusive control of the device, the system is notified that the device is no longer needed by the user (that it can be dismounted as far as the current user is concerned).

CLI Service

These primitives provide service functions to CLI tasks. Information about the user task currently running for the CLI (name, size, CPU time used so far) and devices currently assigned to the CLI's terminal user are provided. Additionally, the CLI can control the terminal context for its user terminal—the CLI can inhibit or allow the action of CTRL/C on the terminal via this mechanism, as well as using it to record the CLI's own context information.

Set or Report Terminal Context

Reports and optionally changes the context of the terminal. Control context governs whether a CTRL/C is recognized at a terminal.

Give Job Statistics

Returns the task time and CPU time used for the task currently running for the terminal being serviced by the calling CLI task. It also returns the device and LUN information for the terminal user.

Report Terminals for a CLI

Reports the terminals in the system for the requesting CLI task.

SYSTEM TASKS AND SPECIAL UTILITIES

IAS provides a common command language for all standard system program development utilities such as the editor, linker and librarian.

In addition to the standard program development utilities, IAS also provides two special system tasks called VERIFY and BAD BLOCKS. These tasks are available only to the system manager. VERIFY is used verify the consistency and validity of the files on a Files-11 volume. BAD BLOCKS is used to locate any unusable blocks on a disk and is normally run prior to disk volume initialization.

The system manager or operator also has available a special utility called CDA. CDA (Core Dump Analyzer) is a task that executes on-line with other tasks to capture system information at the time of crash. It provides the capability to later analyze the state of the system at the time the crash occurred.

General users have access to a special utility called PRESERVE. PRE-SERVE is a multi-user task that creates copies of disk, magnetic tape or DECtape volumes. PRESERVE can also be booted into memory as a stand-alone program.

IAS SYSTEM SUMMARY

Is

- Real-time
- Timesharing
- Batch processing
- Data base management
- Multi-function
- Extensible executive
- High RSX/VAX/TRAX compatibility
- Protected environment

Is not
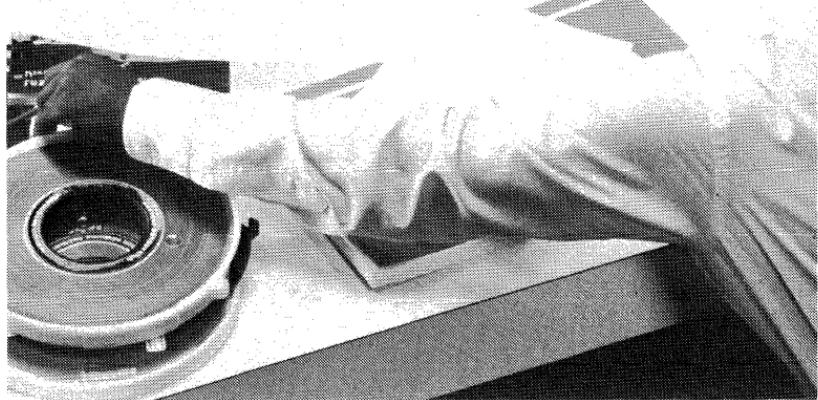
- High capacity (dedicated) timesharing
- High capacity (dedicated) real-time
- Operating on small CPUs

Includes Data Management/Utilities
- DBMS
- RMS-11
- DATATRIEVE-11
- SORT-11

Languages
- BASIC-11
- BASIC-PLUS-2
- COBOL
- FORTRAN IV
- MACRO-11

DIGITAL STANDARD MUMPS
DSM-11 (V.1)

OVERVIEW

DSM-11 is a multi-user data base management system that includes both an operating system and a high-level language. The DSM-11 language has text handling capabilities that facilitate the inspection of any piece of data for content (such as keywords) or for any format. Other text-handling capabilities permit several pieces of text to be combined into one, and divided into segments. Since DSM-11 is an on-line program development and data storage and retrieval system with effective string manipulation capabilities and an M-tree file structure, a programmer can write, debug, or modify a program to develop a working application quickly.

FEATURE TOPICS

- Functions and Features
- Executive and System Features
  - Job Scheduling
  - I/O Monitor
- User Interface
- Terminals and Ancillary I/O Devices
- Data Management
- Data Storage Elements
- DSM Disk Structure and Global Arrays
- Language and Utilities
- The MUMPS Language
  - Expressions
  - DSM-11 Commands Summary
- DSM-11 System Summary

INTRODUCTION — FUNCTIONS AND FEATURES

DSM-11 is an interactive multi-user data base management operating system. The capabilities of the system are heavily oriented toward string manipulation using the high-level Standard MUMPS language, DIGITAL Standard MUMPS, ANSI STD X11.1-1977. The system relieves the user of any concern for programming peripheral devices or for structuring data bases in the traditional sense.

Language processing by the system is interpretive. This greatly facilitates program development by eliminating the need to load editors, assemblers, linkers, etc. The DSM application programmer is relieved of assembly language programming. The major concerns of the application programmer are developing the proper logical hierarchy for a data base and developing efficient logic for the data processing requirements.

The DSM language is provided with its own stand-alone operating system. In addition to supporting the Standard MUMPS language and providing all operating system capabilities, the system affords the user a unique data base structure and access method. Data which is referred to symbolically is automatically stored and linked in sparse, hierarchical structures called M-trees. The physical and logical allocation of mass storage for the tree-structured data base is handled completely by the operating system so that the programmer can concentrate on application data relationships. The data base thus created can either be made available to all system users or be restricted to a class of users.

The DSM-11 operating system runs on any of the PDP-11/34, 11/60, and 11/70 central processors. The system permits up to 63 simultaneous users, operating on any of up to 80 terminals, to interact with a common data base. The system is specifically designed to manipulate strings of data and to expand or contract the data storage areas through dynamic, problem-oriented procedures.

The operating system is highly modular and resides permanently in memory. The system uses between 20K and 32K words of memory, depending on the hardware configuration and system software options selected during system generation. During system generation, the remaining memory is subdivided into user partitions. Machines with no more than 32K words of memory can have 2 to 4 user partitions. Machines with more than 32K words can have a maximum of 63 partitions.

A partition holds one active user's program, local data, and system overhead data. There is no fixed correspondence between terminals and partitions. Indeed, jobs can run without having terminals associated with them, and multiple terminals can be attached to one job.

Partition assignment is performed dynamically at log-in time, and is also permitted during execution. The recommended size for partitions is approximately 4K bytes each, but they do not all have to be the same size; the maximum partition size is 16K bytes. When logging in, a user is assigned the next available partition. If the requested size is not available, the next largest partition will be assigned.

Each active user requiring CPU time obtains a time slice in turn. A checkpoint form of timesharing is used whereby a program is allowed to execute until its time slice has expired, plus any additional time required to complete a current operation. Control then passes to the next job (in priority order) requiring service. The software is entirely memory resident; there is no swapping to disk.

Additional features include:

- variable-sized data elements and logical records
- random access of data using multiple keys
- a variety of terminal and peripheral devices
- system utilities for backup, validation, and reporting
- easy writing, storing, and debugging of programs
- on-line modification of system configuration, system utilities, and system library
- inter-task memory-to-memory communication facilities
- choice of ANSI standard and EBCDIC magnetic tape labeling
- journaling at system level
- spooling
- data base access by more than one CPU
- stand-alone backup program (fast, flexible, error tolerant)

Table 8-1 summarizes the supported hardware of the DSM-11 operating system.

Table 8-1 DSM-11 Supported Hardware

CPUs supported	PDP-11/34, PDP-11/45, PDP-11/60, or PDP-11/70
Memory ranges	Minimum: 32K words (allows 2 to 4 users) Maximum: 124K words on 11/34, 11/60; 1 megabyte on 11/70

Disk systems	Both fixed-head and removable pack disk systems can be used for on-line storage of user programs, the data base, and system utility programs. The maximum size system can provide more than 1.4 billion bytes of on-line storage. RK11, RK06, RK07, RP04, RP05, RP06, RM02, RM03.
Minimum peripherals	Console terminal A disk system (RK11, RK06, RK07, RP04, RP05, RP06, RM02, or RM03) A tape system (TS03, TU10, TU16, or TU45 magnetic tape system)
Additional peripherals	Maximum 16 single-line controllers (DL11) Maximum multiplexers: 5 DH11s (16 lines) or 6 DZ11s (8 lines) with a maximum of 80 terminals total DMC11 Synchronous Communications Interface Industry compatible magtape (TS03, TU10, TU16, TU45; up to 4 drives each) CR11 card reader LP11 line printer

EXECUTIVE AND SYSTEM FEATURES

Job Scheduling

The executive implements the timesharing aspects of the system and permits partitioned multiprogramming using dynamic assignment of memory-resident user partitions. In a timesharing environment, jobs are generally highly interactive and normally require little processing time between I/O requests. The executive passes control from one user to another in order to use the central processor as much as possible. Because jobs are resident in memory partitions, the executive can switch from user to user in minimum time.

The executive uses a set of priority-weighted queues to administer its scheduling algorithm. Jobs waiting to run can be placed in either one of two sets of wait queues, depending on the priority set by the application system designer. These queues are the Wait 1, Wait 2, and Wait 3 queues, and the Wait A, Wait B, and Wait C queues. The priorities are Wait 1, Wait A, Wait 2, Wait B, Wait 3 and Wait C queues. Initially, a job starts in the highest priority wait queue. When a job reaches the front of this queue, it is placed in the run queue, where it executes for the duration of its time slice. The number of DSM commands executed is incremented with the interpretation of each command. The counter is reset upon the completion of an input message. When a job is swapped out, if the command count is less than 20, the job is placed in

the Wait 1 or Wait A queue. If the command count is greater than 20 and less than 8192, the job is placed in the Wait 2 or Wait B queue. Otherwise, the job is placed in the Wait 3 or Wait C queue.

If the job is still executing, and has not issued an I/O request (which would change its priority) by the end of this time slice, it is placed in the lowest priority wait queue. When it reaches the front of this queue, it is allocated another time slice and is once again placed in the run queue. After this point, the executive circulates the job between the lowest priority wait queue and the run queue. When the job becomes I/O bound, the executive places the job at the end of the highest priority wait queue (unless the I/O was disk I/O, in which case the job is placed at the front of the highest priority wait queue). Note that the time slice given to any job in any wait queue always remains the same, regardless of the wait queue in which the job is placed. This queueing algorithm gives priority to the most highly interactive jobs in the system.

I/O Monitor

When a job becomes I/O bound, the executive places the job in the appropriate hung state that signals the I/O monitor to start its processing. The I/O monitor initiates and processes the I/O activity through its interrupt handlers.

The DSM interpreter and the I/O monitor communicate through buffers for terminal I/O character processing, but the I/O monitor supervises the asynchronous filling and emptying of these buffers to overlap output with that program's processing whenever possible.

The I/O monitor creates a terminal-independent environment in which an application program can run with any terminal of the hardware system regardless of its specific speed and formatting characteristics. At terminal log in, a partition initially "owns" one terminal. It may subsequently acquire other terminals in the system, or it may release the original terminal and continue as a detached job.

The I/O monitor also supervises the peripheral I/O devices of the system, including the magtape drive, card reader and line printer.

User Interface

Most users of the DSM-11 system gain access to the system's programs using a special log-in sequence which involves one or two access codes (depending on the privileges of the user). These codes, provided by the system manager, are the User Class Identifier code or UCI, and the Programmer Access Code or PAC.

The DSM-11 system can have up to 10 UCIs (classes of user). A UCI

allows access to the programs and globals listed in the program and global directories for that UCI. A user who is permitted simply to run programs needs to know only the UCI and the name of the programs for that UCI.

Users who are allowed to create or modify programs and global files must know the system's PAC. This code permits system operation in direct mode. In direct mode, a programmer can issue DSM commands at the keyboard, as well as create, modify and delete global data and programs associated with the UCI under which the user logged in.

If the user intends to program, the partition is initialized and control is passed to the interpreter for the subsequent programming session. If the user desires activation of a service program, the requested program is loaded from the disk into the partition and execution of that program commences. In either case, the user retains the partition until logging off the system or until the requested program finishes executing.

DSM-11 also employs a concept known as "tied terminals." An attempt to log in at a tied terminal activates the task to which the terminal is tied and limits the user to the resources associated with that task. Normally, the user gains access to the system by typing a CTRL/C, entering a UCI or UCI and PAC code, and then selecting a program or command to execute. When the user types a CTRL/C at a tied terminal, the task to which the terminal is tied is immediately activated. This capability gives the system manager an effective control mechanism for system access.

To log in to the system, the user types the CTRL/C keys or the BREAK key on the terminal. If the terminal is not tied, DSM responds by requesting a UCI code. The terminal user can respond in one of two ways. If the user is not a privileged user, the response consists of a UCI code followed by the name of the program to be executed. In this case, DSM logs in the user if the UCI is valid, executes the named programs and logs off the user.

If the user is a privileged user, the response consists of a UCI code followed by the PAC. In the latter case, DSM enters direct mode, indicated by its printing a greater than character (>) on the terminal. In direct mode, the programmer can:

- execute DSM commands immediately
- enter program code
- run programs and access global files listed in the UCI directories
- run library utility programs

All application programs, system utilities and library programs are written in Standard MUMPS language. This language allows an application programmer to write a program and debug, edit, run, and modify it in a single interactive session at a terminal. This minimizes the programmer's time in solving a problem, the computer time needed in checking it out, and the elapsed time required to obtain a final running program. The interpreter is that part of the operating system responsible for these services. The executive and the I/O monitor serve to enable the interpreter to operate efficiently.

The interpreter examines and analyzes all Standard MUMPS language statements, executing in turn the desired operations. Each Standard MUMPS language statement undergoes identical processing each time it is executed by the interpreter. Intermediate code is not generated. Comprehensive error checking is also performed to ensure proper language syntax.

In addition, the interpreter stores and loads programs through the disk storage system. During program execution, the interpreter can overlay external program segments invoked by an active program. Proper linkages are set up to return to the invoking program when execution of the segments terminates.

A number of major advantages are obtained from the use of the interpreter as the major component of the DSM system. First, programs written in an interpretive language do not require any compiling or assembling. Error comments during execution are printed at the programmer's terminal and allow quick recovery, program modification and re-execution. All program debugging and modification operations are performed in the DSM language directly at the terminal. This makes modification convenient, particularly in an environment where the troubleshooting necessary to interface a program with an application area is a time-consuming process. The DSM environment allows a programming session to take the form of a conversational dialog between the programmer and the terminal device.

Almost any DSM command or function can be executed from the keyboard in direct mode. When a command is entered, the DSM language interpreter executes the command immediately and gives the appropriate response to the programmer. A command line can consist of several Standard MUMPS commands and arguments, comments, and data. For example, the programmer can enter the command line:

>WRITE "7+5=",7+5

This command tells DSM to print the characters 7+5= on the terminal, evaluate the arithmetic expression 7+5 and print the result on the

terminal. DSM therefore responds by immediately printing:

7+5=12
>

To create a program, the programmer enters a paragraph of code which may consist of one or more lines. Each paragraph begins with a label and a TAB character; subsequent lines begin with just a TAB and are addressed via the label plus an offset. For example:

>A   WRITE "7+5=", 7+5,!
                WRITE "THIS IS A TEST",!
>

In order to print the second line of this program on the terminal, one would type P   A+1.

Entering lines of code in this manner signals the system to store the line in the program buffer of the user's partition rather than to execute it immediately. DSM responds only by printing the > character. The programmer must explicitly request DSM to execute the stored command line. For example:

>DO A
7+5=12
THIS IS A TEST
>

The DO command tells DSM to begin executing at the line labeled A of the stored program, and it will continue to execute until it encounters a control command such as GOTO or QUIT, or arrives at a point where there is nothing else to interpret.

Once a program has been created, the programmer can store the contents of the partition's program buffer on disk or on a secondary storage device such as magnetic tape. The program can then be reloaded into the program buffer from the disk or secondary storage. A program can be modified when it is loaded in the program buffer by adding new lines or by replacing, deleting, or modifying existing lines of code.

Terminals and Ancillary I/O Devices

In addition to the disk devices reserved for use by the DSM data base supervisor, DSM allows users to have access to terminals and ancillary I/O devices such as the card reader and magnetic tape devices. Each I/O device has a unique identification number in the system.

Ownership of terminals and ancillary I/O devices is established using the OPEN command. Once ownership is established, I/O may proceed

using the I/O commands available. In general, the programmer need not be concerned with specific characteristics of I/O devices, since data transfers consist of ASCII strings not greater than 255 characters. There are, however, certain physical operating characteristics of these devices which may be of interest to the programmer: for example, rewinding a magtape or a form feed on the line printer. There are also logical characteristics such as use of special characters to indicate end of a logical record or end-of-medium (EOM). The omission of such characters can result in logical records of unlimited length (except for physical device limitations such as length of tape).

The unique identification number of each I/O device always represents the same device regardless of the hardware configuration of the particular system. For example, the console terminal is always device #1 and the line printer is always device #3. If a particular system does not have a line printer, then device #3 is non-existent, and any attempt to reference it generates an error.

The commands which affect input and output operations to the terminals and ancillary devices are: READ, PRINT, WRITE, WRITE* and ZLOAD. The WRITE command is used to output both local and global data, as well as literals, constants and format control characters. The WRITE* command is used primarily to take advantage of special features of I/O devices, which are specified, generally, by non-printing ASCII codes. The WRITE* command accepts numeric arguments, the low-order seven bits of which are taken as the decimal representation of the ASCII code. For example, the command W*10 is used to output a line feed character.

In addition to the standard I/O peripheral devices such as the line printer and magnetic tape drives, DSM has two special "devices." They are the Sequential Disk Processor and the CPU-CPU device.

The Sequential Disk Processor (SDP) allows the user to access the disk physically as an assignable sequential I/O device. The SDP can access only the disk space that is explicitly set aside for its use. Other disk space, including the global data base structure, can not be accessed. Sequential disk processing allows the user to impose any file structure on the SDP space.

The CPU-CPU device is a DMC11 synchronous interface, full- or half-duplex, that connects the DSM-11 CPU to another CPU. The other CPU does not necessarily have to be a DSM-11 system, but does have to recognize DMC-11 protocols. This device allows a DSM program to communicate with a program running on another central processor.

In an attempt to connect more users to a common data base, DSM-11 allows a system to be connected syntactically to up to four other sys-

tems. Functionally, these routines will allow a user on one system to lock global nodes or read or write global data from the other systems.

Spooling

DSM-11 also includes the ability to spool output to line printers. The spooling device is a file-structured mechanism used for temporary storage of information. Typically, one would direct the output of several programs to separate files on this device. These files would then be processed one at a time by a de-spooling program which would write them to an output device such as the line printer. After a file had finished printing, it would be erased from the structure.

In order to aid in classifying these spool files, a destination code accompanies each file, in addition to its own unique file index number. The destination code is a value which is in the range of (1) to (255). This code is recorded in the directory entry for each file for easy access. By using this code, a file can easily aid in retrieving a particular group of files.

For instance, let us say that there are two de-spooling routines running on a DSM system, each handling one printer (devices 3 and 4). The user may choose to designate files which have a destination code of 3 to be written to device #3 and those with a destination code of 4 to be printed on device #4.

Each de-spooling routine would attempt to access any existing file with an associated destination code (3 for device #3, 4 for device #4). If no file exists with this destination code, an error code is returned in the system variable $ZA. The de-spooling routine would then recognize that and would sleep for a specific period and then try again.

If the open was successful, the de-spooler would read from that file and output it to its associated printer. Upon completion of this (end-of-file is indicated by a $ZA error code), the file would be erased by use of an option on the CLOSE command.

In this way, actual file numbers become relatively unimportant, and several files with the same destination may be open simultaneously.

In the released version of DSM-11, the despooling routine (% DSPOOL) uses this destination code as the intended output device and will process all spool files in a serial manner. It is suggested that the user examine this routine and use it as a model for customizing the spooling facilities to the particular installation environment.

The system global ↑%SPOOL is used by %DSPOOL and %DEVIL for communication with other routines. An entry point QUIT↑%DSPOOL is provided to shut down the de-spooling routines. The routine %DEVIL

is a device error monitor routine which scans for printer errors and indicates them on the console device. It then re-enables the printer for a re-try.

Journaling

DSM-11 also supports the technique known as journaling. Journaling is a technique whereby an additional copy of any data that is modified on the disk is made on another device. In the DSM system, any item that is changed on the data base is also written on to the magnetic tape for a journal record. Should a catastrophic failure to the disk occur, it is always possible to bring back the journal tape entries and restore to the previous backup copy, bringing the system right up to date as of the time of the failure. This journaling is transparent to the MUMPS application programmers. It runs at the system level, built in to the operating system so that MUMPS programs need not be modified or specially written to handle journaling. All desired changes to the data base are recorded automatically on to the journaling system. Journaling also has the capability of writing transactions delimiters.

It is frequently important in data base systems to be sure that all of a particular grouping of items, or a particular transaction, is updated on the device. With DSM-11, it is possible to write transaction delimiters onto the journal so that the restore program can be sure that it has an entire transaction before it does the restore.

The journaling in the DSM-11 system is double buffered. This is a performance enhancement that means the system should very rarely have to wait for the magnetic tape to catch up with the data base changes. All entries being made to the tape are written into one buffer. While that buffer is being written out on to the tape, the updated transactions are being written to the second buffer. This ping-pong effect can keep the magtape moving at optimum speed, and not bog the system down waiting for the tape transport activity.

The journaling is optional either by the entire system or by specific job. It is important to note that, once included in a system during system build, journaling is assumed. A user must take an overt action to stop the journaling, rather than one to start it. When a particular operation doesn't need journaling, such as a batch process that can easily be repeated, then the journaling can be stopped to enhance system performance.

DATA MANAGEMENT

Data Base Supervisor

The data base supervisor consists of a group of routines which pro-

vide physical as well as logical control of the various disk systems which store the data base.

In DSM, all file information is referenced symbolically, in the context of hierarchical global variables and arrays. This replaces the standard technique of sequentially accessing the blocks constituting files on secondary memory devices. Instead, the content and structure of the tree-structured symbol tables are logically mapped into the physical storage medium of the system. The data base supervisor maps logical information from global arrays into directories of fixed-size blocks. Maps of unused disk blocks are maintained to facilitate the dynamic allocation of disk storage space to files. These storage allocation maps are bit maps in which there is a correspondence between the map address and the bit position within the map, and the disk address of the block.

Whenever a file needs a block, the system references a table which governs the allocation of data for that particular user. This table has entries in it which indicate the block number where a scan for an empty block is to be started. Types of blocks allocated in this way are: global directory, global pointer, routine directory, routine pointer and global data blocks. Given a starting location, the system references a master allocation table to determine the availability of blocks in the desired area. (This map is known as the master map and is kept in main memory.) Having thus found the region where an available block is to be found, the appropriate map block is referenced for the specific block number.

DSM utilizes a data retrieval method known as disk cache. Once a block of data accomodating a given level of subscripting is referenced, its address is placed in the partition's overhead area and the block remains in memory until a reference to a different block is made. When a level is reached, often no further disk access need be made to reference associated information. At system generation, the system manager has the option to establish a buffer pool of up to the equivalent of 64 disk block buffers. Disk data blocks will be kept in the buffer pool as a function of frequency of use. Frequently used blocks will tend to remain in memory, thus reducing the number of disk accesses. Furthermore, when data are updated, care is given to repacking, and sometimes reorganizing, the individual data elements within a chain, to ensure maximum use of space for variable length data.

When a part of a global structure is deleted, it is attached to a garbage chain. The garbage collector routine removes blocks from the tree-structured chain and updates the storage allocation maps according-ly.

Data Storage Elements

All user data, whether numeric or string, are stored in the system as ASCII character strings. DSM interprets these strings in one of two ways: as numbers, such as those used in calculations, or as strings, such as names and addresses.

Numbers in DSM are signed numbers which can be up to 27 significant decimal digits long. Examples of numbers are:

2.08
151.95
403,222
.6379465

A data value has the form of a number if it satisfies the following restrictions.

1. It contains only digits and the characters '-' and ',' (the + character is not necessary. The number +403,222 is equivalent in value to 403,222.

2. At least one digit is present.

3. A decimal point (.) occurs no more than once.

4. The number zero is represented by the one-character string '0'.

5. The representation of each positive number contains no hyphen (-) character.

6. The representation of each negative number contains the hyphen (-) character followed by the representation of the positive number which is the absolute value of the negative number. (Thus, the following restrictions describe positive numbers only.)

7. The representation of each positive integer contains only digits and no leading zeros.

8. The representation of each positive number less than 1 consists of a decimal point (.) character followed by a non-empty digit string with no trailing zero.

9. The representation of each positive non-integer greater then 1 consists of the integer part of the number followed by the fractional part of the number.

String data in DSM is any contiguous series of legal DSM characters that are to be considered a single data entity. Strings in DSM can be up to 255 characters long. Examples of strings are:

HELLO, MY NAME IS
55 SECONDS
2,564,843,485,076,193

FRIENDS, ROMANS, COUNTRYMEN,...
FROP%X10.CF

Program data values can be expressed as literals, constants or variables. Three types of variables can be created in Standard MUMPS programs: simple variables, subscripted variables and global variables. Variables can be created, modified and deleted using the SET, READ, and KILL commands.

System variables are a fourth type of variable. These variables, maintained by the operating system, contain general information for use by all Standard MUMPS programs. With one exeception, system variables are read-only and cannot be altered as can normal variables.

A subscript is a value enclosed in parentheses which is appended to a variable name to identify uniquely a number of data elements which are to reside under that variable name. All the subscripted variables residing under a common name are collectively referred to as an array. An array can consist of variables which have more than one level of subscripting, and when more than one level is used for global array subscripts, they are separated by commas.

A sparse array is an array in which only those elements which are explicitly defined or which are required to support the array structure actually exist. Unlike other languages which may require a declaration of the maximum size of an array to preallocate space, DSM dynamically allocates storage for all arrays only as needed, thus conserving storage space.

Local variables are variables that reside in the same partition as the commands which created them and are used as scratch or transient data. These variables are accessible only to programs running in the same partition. Simple variables have no subscript, for example, ABC, R45, X, %D. Subscripted variables can have multiple levels of subscripting, with numeric or string subscripts. For example: ABC(2), R49("LIST"), ABC(4+B(C*D)/0.89).

Global variables are multi-subscripted arrays. Unlike local variables, they are external to a program's partition, providing a common data base available to all programs in a given user class. There is no logical limit to the number of subscripts that can be used. The physical limit is 63 characters for a complete global reference. Like subscripted local variables, global arrays also reside in sparse arrays and are created simply by reference in a program. Each global array is identified by a unique name which is similar to a local variable name in a program, but is always preceded by an up-arrow character (↑).

Array elements, which are often called nodes, can contain either a numeric or string data value. Nodes may be either pointer nodes or data nodes. Pointer nodes are stored as required by the system at the higher levels of the tree. All data are stored in a well-ordered form in data blocks—regardless of the number of subscripts.

A global variable node can be referenced in a program using a special abbreviated syntax called naked syntax. The naked syntax facility permits the programmer to abbreviate the global reference.

In form, only the up-arrow and subscripts are explicitly stated. The global name is assumed from the last global reference made. Thus, if a reference to ↑ABC(2) is to be made after referencing ↑ABC(1), only the subscript is specified: ↑(2). The first subscription in the naked reference replaces the last subscript in the previously completed global reference. Thus if ↑ABC(2,3,4) is referenced sucessfully, then a reference to ↑(1,2,3) would refer to ↑ABC(2,3,1,2,3).

In addition to storing global data files, the disk is also used to contain Standard MUMPS language programs, which include both user-created programs and system utility programs.

The availability of programs and global data to users is controlled by the system's protection scheme. Up to 10 classes of user can be defined within the system. Each user class has access only to those programs and globals residing in that class. In addition, specially named library programs residing in UCI #1 (the system UCI) can be accessed by all users.

The DSM Disk Structure and Global Arrays

The primary devices used by the DSM-11 system are the disk units allocated to the storage of DSM globals and DSM programs. Each UCI defined by the system manager has two directories associated with it: the global directory (that is, the file directory) and the program directory.

Directories for programs and globals are normally stored on the system disk. Storage area for programs and globals usually begins on the same disk unit as the associated directories. As programs and globals increase in size and number, storage area will ultimately flow across physical disk unit boundaries. This is completely transparent to the user. The general user does not have to be concerned with any DSM-11 disk device unit naming to retrieve globals or programs from any of the disks allocated for this purpose.

The system manager can locate the directories on any disk unit in the system. The system manager can also limit program and global storage to specific disk units in the system.

Globals are logically organized as multidimensional tree-structured arrays. An element of an array has a logical name consisting of the global name and the subscript(s) uniquely identifying the element. For example, ↑ABC(2,3.4,JONES) is the name of the element in the global called ABC whose first subscript is 2, whose second subscript is 3.4, and whose third subscript is JONES. The elements of a global array are called nodes.

The user's global directory contains the names of all the globals it can reference, together with the pointers to the tree structures for each of the globals.

The basic new data structure is organized along the lines of "multiway B-trees." A general discussion of multiway trees can be found in Knuth, "Sorting and Searching," Volume 3, Chapter 6.2.4.

Essentially, the new structure consists of three types of disk blocks: directory, pointer, and data blocks. The organization and growth of data blocks will first be described, and, in so doing, the others will be explained.

All of the data are stored in blocks called data blocks. Each piece of data is stored with the set of subscripts that is required to access the data. The subscripts are concatenated to form the "node name," and the associated data in the node's "value." Thus a data block may look as follows:

A_1_3, HELLO	A_5, YES	A_19_3_2, BYE
1st NODE	2nd NODE	3rd NODE

In this example, ↑A(1,3) = "HELLO." All of the nodes are kept in order by node name, and ↑A(5) would preceed ↑A(5,0).

Let us assume that this is the only data block, and that in attempting to add ↑A(2), we find that the block is not large enough. A new block is allocated, and part of the block is placed in the new data block. Another block is also allocated, and it contains the first node name of each of the data blocks with an associated pointer to each data block. Our structure may now look as follows:

	A_1_3	A_5	POINTER
BLOCK			
A_1_3, HELLO	A_2, THERE	A_5, YES	A_19_3_2, BYE

The top block in this diagram is a pointer block. Other nodes will be inserted in the data blocks, and other splits will occur causing new entries to be inserted in the pointer block. Eventually, the pointer block will become so full that it too must split. When this split occurs, a

higher level pointer block is allocated, and the process repeats itself. Every time the bottom pointer level splits, a new node is inserted in a higher level pointer block until it, too, splits.

Note that all of the data are stored at the bottom level, and that, although deletions may result in the collection of certain data and pointer blocks, the same number of pointer block levels will always exist between any data block and the top pointer block.

The number of pointer block levels depends upon the file size (number of elements and length of subscripts). In small files, the number of pointer levels may be one or two. Large files will require three levels.

The complete segregation of pointer and data blocks permits the system considerable freedom in block allocation. All of the pointer blocks may be stored on a couple of cylinders, thus reducing the head motion which is the major factor in access time.

LANGUAGE AND UTILITIES

A set of DSM language utility programs provides the user with the tools to maintain and service the system efficiently. All these utilities are written as Standard MUMPS language programs, and as such can be easily modified and extended to suit the needs of a particular installation.

The utility programs consist of two operationally distinct groups: system utility programs and library utility programs. The system utility programs provide functions for use by the system manager. They reside on the disk under the control of the system UCI (UCI #1), and are accessible only to those individuals possessing the system UCI code.

Library utility programs provide general services which are available to all system users, regardless of UCI. These programs also reside under the system UCI but employ a naming convention which distinguishes them from system utilities.

Tables 8-2 and 8-3 briefly describe the system utility and library programs.

Table 8-2 System Utility Program Summary

BBD	Bad Block Deallocator
STU	System Startup
STA	System Status

SSD   System Shut Down

RJD   Restore Job/Devices

PAN   Partition Analysis

DDR   Device Descriptor Rpt

CTK   Caretaker

KTR   Caretaker Rptr

JPC   Job Priority Change

BCS   Broadcaster

DGAM   Disk Growth Area Management

DAT   Sets date in $HOROLOG

TIM   Sets time in $HOROLOG

DSM Backup and Utility System

The Standard MUMPS Backup and Utility System (SMBU) is a bootable, stand-alone system. SMBU allows the user to back up or save the significant data from DSM disks so that in the event of a system failure, the disks can be restored to their former states. In addition, SMBU performs the following important utility functions:

1. Labels disks and magtapes for identification purposes and for prevention of inadvertent distruction of important data.

2. Formats and tests disks; initializes disks to be used in a DSM environment.

3. Makes exact image copies of magtape and disk volumes to back up non-MUMPS data.

4. Allows the direct allocation or deallocation of individual blocks on a DSM disk.

Once loaded into memory, SMBU begins executing automatically. Thereafter, the user need only answer the question it asks in order to have it perform the operations desired. If in doubt as to the way to answer a particular question, the user can type the single character 'H' (followed by a carriage-return) and receive a list of sample answers to the question, showing the format in which the answer is expected.

Note that if the user types 'H' in response to the first question SMBU asks, he will receive an "SMBU Command Summary"—a brief description of all SMBU commands, explaining the use of each.

Table 8-3 Library Utility Program Summary

%BDLMP	Block Dump — dumps requested global blocks.
%CP	Character Print — turns echoing of characters on and off.
%D	Writes out the date equivalent of whatever is in $HOROLOG.
%DO	Decimal to octal subroutine
%DOC	Decimal-octal converter
%ED	Routine Editor
%FL	Routine first line list
%GBA	Gobal Block Analysis
%GD	Global Directory
%GL	Global Lister
%GP	Global Place
%GR	Global Restore
%GS	Global Save
%GSEL	Global Name Selector
%GU	Global Utilization
%GUCI	Get current UCI name and number
%H	Takes care of all date and time conversions.
%IOS	I/O device selector
%JRNL	Turns journaling off and on.

%LT   Prints out lock table

%OD   Octal to decimal subroutine

%PROTECT   Global Protect — changes protection codes of differ-
                 ent globals.

%RBA   Routine Block Analysis

%RD   Routine Directory

%RLST   Routine List

%RS   Routine Save

%RSE   Routine Search

%RSEL   Routine Name Selector

%RR   Routine Restore

%T   Writes out the time equivalent of whatever is in $HO-
                 ROLOG.

The MUMPS Language
ANSI STANDARD MUMPS X11.1-1977 contains a large repertoire of
capabilities; its basic orientation is procedural, much like FORTRAN or
COBOL. Its capabilities are primarily directed toward the processing
of variable-length string data, although mixed mode operations are
expressly permitted. In addition, standard algebraic and Boolean op-
erations are available.

Language processing is in every sense interpretive. Each line of
Standard MUMPS code undergoes identical processing each time it is
executed. The language interpreter has two operating modes: pro-
gram execution mode (indirect mode) and program creation mode
(direct mode). In direct mode, programs can be created, modified,
debugged, stored, and executed in whole or in part. Indirect mode
permits the execution of these programs.

The following paragraphs discuss some of the major elements of the
Standard MUMPS language.

Expressions
An expression is a value description that can be made in the Standard
MUMPS language. An expression is any legal combination of oper-

ands and operators. Expression elements include such basic language elements as literals, constants, simple variables and subscripted variables. Also included are function references and subexpressions, which are simply legal expressions enclosed in parentheses. The following are examples of expression elements:

123.34	constant
ABC	simple variable
"ABCD"	literal
MX(5)	local subscripted variable
↑XYZ(2,5)	global variable
$LENGTH(Z)	function reference
(A+B-(C/D))	subexpression

The operators in an expression serve to represent the various arithmetic and logical computations of the Standard MUMPS language. Table 8-4 lists the Standard MUMPS expression operators.

Table 8-4 Summary of Expression Operators

Type	Symbol	Function
Arithmetic	+	Addition
	−	Subtraction or Unary Minus
	*	Multiplication
	/	Division
	#	Modulo
	\	Integer divide
Relational	<	Less than
	>	Greater than
	=	Equality
	<=	Less than or equal to
	>=	Greater than or equal to
	'=	Not equal to
Boolean	&	AND
	!	OR
String Relational	[	Contains
	]	Follows
	?	Pattern verification
	=	Equality
	'	Not
String Concatenation	_	Concatenation
Indirection	@	Indirection*

213

*Indirection allows data values to be executed as Standard MUMPS code.

Indirection is denoted by the character @ followed by an atomic expression. The value of the expression is substituted for the occurrence of indirection before the rest of the line is interpreted. There are three basic types of indirection.

1. Argument indirection, where the indirection occurs in place of a command argument, and the value must be one or more complete command arguments.

2. Name indirection, where the indirection occurs in any context where a named variable can occur and the value of an indirection must be a complete variable name.

3. Pattern indirection, where the indirection occurs in place of a pattern and the value must be a pattern.

Of special importance are the relational string operators. They provide facilities for determining the characteristics of string data. The operators return true or false results. They are:

String Contains ([) — The string specified by the left operand is examined for the occurrence of the string specified by the right operand. If a match is found, the result is true.

String Follows (]) — The string specified by the left operand is compared character-for-character with the string specified by the right operand to establish relative position according to the ASCII collating sequence. If the string specified by the left operand follows that specified by the right operand, the result is true.

Pattern Verification (?) — The string specified by the left operand is examined for the occurrence of the character patterns specified by the pattern specification codes. If a matching condition exists, the result is true. The pattern specification codes can be preceded by a single decimal integer to specify the number of occurrences of a particular character type. The pattern specification codes are:

A   Verify upper and lowercase alphabetics
C   Verify 33 control characters
E   Verify entire set of 128 characters
L   Verify 26 lowercase alphabetics
N   Verify 10 numerics
P   Verify punctuation
U   Verify upper case alphabetics or any
         string of characters delimited
         by punctuation marks

Commands

A command is the basic unit of expression in the Standard MUMPS language. A command is a mnemonic which symbolizes the action to be performed, for example GOTO or SET. The command name can be abbreviated to one letter. It usually takes one or more arguments which specify the objects of the action to be performed. Several Standard MUMPS commands can be present on a command line. Program comments can be appended to any command line using a semicolon to separate the command line from the comment text.

Standard MUMPS uses alphanumeric tags plus offsets to identify lines of code, and programs are arranged in a paragraph concept. Standard MUMPS commands are executed from left to right within a line and sequentially from one line to the next (assuming no control commands are encountered). For example, INIT is a tag which identifies the first line of the INIT paragraph; INIT + offset addresses subsequent lines of the INIT paragraph without the need to tag every line of code.

Certain commands permit the optional use of an argument or argument list. The indirection syntax operator, symbolized by @, provides dynamic argument definition. In form, the command argument is replaced by the indirection syntax operator immediately followed by a variable name. During execution, the contents of that variable name are taken as the argument. For example:

1.15 S ARG = "15+3/6" ;variable ARG is set to value
1.20 W @ ARG ;contents of ARG are evaluated and output

An optional Boolean-valued expression preceded by a colon can be used as part of an argument to specify conditional execution. For example:

GOTO 3:A>B ;control is transferred to paragraph 3 if A is greater than B

Commands can be issued in either indirect mode or direct mode unless specified otherwise. Table 8-5 summarizes the DSM commands.

Table 8-5 DSM-11 Commands Summary

BREAK   Stops a routine at a specified point to allow examination of routine variables. BREAK allows an argument which returns a Boolean value (1 = true, 0 = false). A true argument allows a given job to be interrupted, whereas a false argument prevents a job from being interrupted.

CLOSE   Releases one or more designated devices from ownership.

DO   Initiates execution of the MUMPS code at the specified argument.

ELSE   Provides the means for testing the sense of the previously executed IF command. When the sense of the preceding IF is false, commands following the ELSE on the line are executed. Otherwise, control passes to the next program step. This command can be issued in indirect mode only.

FOR   Produces efficient looping by repeating commands residing on the same line for a specific set of variable values. QUIT terminates a FOR loop.

GOTO   Program control is permanently transferred either to a line of code in the same routine, to the start of a new routine, or to a particular line of code in a new routine. Note that if return of control is required, the DO command should be used. This command can be issued in indirect mode only.

HALT   First LOCK (see below) with no arguments and CLOSE of all devices opened by this job are executed (although these are not stated explicitly). Then execution of the current process is terminated. HALT does not take an argument.

HANG   Suspends program execution for a specified time interval. When the interval is up, program execution resumes at the command following the HANG.

IF   Effects a change in a program's operation based on the validity of one or more Boolean-valued expressions. If all expressions are true, the remainder of the command line is processed. If any expression is false, the next step is executed. It can be used without arguments; the condition tested is the value of system variable $T, which is set by the last IF statement. The ELSE command is used to test the logical reverse of an IF.

KILL   Used to delete both local and global variables, with specified arguments or exclusive arguments.

LOCK   Program convention for notifying other users that a particular node of a global and all nodes to which it

points (all its descendants) are not to be referenced. This allows the program to protect global data which may be accessed by several programs simultaneously for updating. LOCK without arguments releases all nodes previously locked.

OPEN   Obtains ownership of one or more devices.

QUIT   Terminates the execution of a logical process, including the execution of a line or program. QUIT is often used to terminate prematurely operations which are executed within the range of the DO, FOR, and XECUTE commands.

READ   Reads one or more lines of characters into specified local variables. Additional optional arguments are a message to be written and the format control characters, and timing information. A timed READ enables the program to continue processing if the time interval elapses before any input is received. This is particularly useful in applications where terminals are either infrequently attended or unattended.

SET   Assigns the result of an expression to a specified variable.

WRITE   Specifies the output of data and format control to the current device. When an argument includes an asterisk followed by an integer value, one character whose code is the number represented by the integer is sent to the current device. The effect of this character at the device is device dependent.

USE   Designates a specific device as the current device for input and output. Before a device can be named in the argument of a USE command, its ownership must have been established through execution of an OPEN command.

VIEW   Permits the reading and writing of memory locations and disk storage blocks in the system's data base. The use of the VIEW command is restricted by several levels of protection.

XECUTE   Provides a means of interpreting Standard MUMPS code which arises during program execution. Each argument of the XECUTE command is interpreted as if it were a line of Standard MUMPS code.

217

Z Commands

The following commands are known as Z commands. These commands are the DSM-11 extensions to the DSM language, and can be used only under DSM-11.

The Z commands may be abbreviated to their first two characters, i.e., ZINSERT may be abbreviated to ZI.

ZG(O)   Resumes execution of a routine after a BREAK command. If an error occurs while in the BREAK state or if CTRL/C is typed, continuation is not possible using the ZG0 command, because system error processing removes the break state from the user's partition.

      Example:

      > S X=0 F I=1:1 S X=X+I B:X=15
      <BKERR> B:X=15
      >ZGO

ZI(NSERT) ZI(NSERT) "stringexpr":label

      where "string expr" is the line the user wishes to insert in his routine. Note that the contents of "stringexpr" should have a space, not a TAB after the label (if a line label is specified within "stringexpr").

      where :label is the label of the line just previous to where "stringexpr" is to be inserted; :label can also be a line label plus offset, when no label exists on the line previous to the insert.

      Inserts one or more routine lines into the user's routine. May be used in two basic ways:

      1. To insert one or more new lines into a routine

      2. To replace an existing line or lines in a program

      To insert or replace one or more lines in a routine, ZINSERT starts inserting lines after the line pointer. ZINSERT can insert only one line at a time; a new ZINSERT command must be given for each line to be inserted. To insert a line in the following routine,

      >ONE<TAB>;This is test line one
      >THREE<TAB>;This is test line three

      the user simply types:

      >ZINSERT"TWO;This is test line two":ONE

218

This will insert the line labeled TWO after line ONE.

To replace line THREE, the user types

>ZINSERT "THREE W ""HELLO"""; This is test line three":TWO

Notice that HELLO has two sets of double quotes. This will cause HELLO to print out as "HELLO". One set of double quotes would print HELLO.

At this point, ZREMOVE can be used to delete the last line in the routine.

Example:

To insert a line at the beginning of the previous routine, type:

>ZINSERT "FIRST ;this routine prints out test lines": +0

The +0 inserts this line as the first line of the routine.

ZJ(OB)   ZJ(OB) entryref["MGR"]:8

where entryref is of the form:

[label]↑rnam

Starts a new partition executing the job ↑rnam, starting at label. This will occur only if a partition is available. If the job is successfully started, $TEST is set to true (1). If a partition is not available, $TEST is set to false (0). An if without arguments can be used by the starting job to determine whether the desired job was started. The optional square brackets allow the manager to start programs in any user area specified by the literal "MGR", and the optional colon allows specification of a particular partition size, in a number of 512-byte increments.

Examples:

1. >ZJOB ↑AA:8

Starts job ↑AA in a new 4K-byte partition.

>W $T
  1

To confirm that ↑AA was sucessfully started, check $T. A result of 1 means succesful start.

2. >ZJOB CHKR↑AA

Starts job ↑AA in a new partition, starting at label CHKR.

ZL(OAD) ZL(OAD) rnam

Loads routines from disk, sequential disk area or magtape into the user's partition.

Examples:

1. >ZLOAD CALC

Loads a routine named CALC from disk.

2. >0 47 U 47 ZL

Loads a routine from magtape unit into your partition.

ZR(EMOVE) ZR(EMOVE) [zremovearg,...]

where zremovearg has the form label or label1:label2

where label, label1, and label2 are labels of existing lines in a routine.

In the argumentless form, deletes all lines in a routine. With arguments, deletes all lines from label1 to label2 inclusive.

Examples:

1. ZREMOVE

Deletes all routine lines currently in user's partition.

>ZR BREAK0:BREAK2

Deletes all lines starting with BREAK0 and continuing through BREAK2.

ZS(TORE) ZS(TORE) rnam

Without an argument, stores the routine buffer under the name that is in the partition vector. If the partition vector does not contain a routine name (that is, when the special variable $ZNAME is equal to the null string), the routine must be stored with a name (pnam).

If the partition vector does not contain a routine name, and the user attempts to store a routine without specifying a name, a <NOPGM> error is generated.

To delete a routine from a disk or tape file, use ZRE-MOVE to clear the partition, and then file that empty partition under the name of the routine to be deleted.

Example:

>ZR

ZREMOVE clears the user's partition.

>ZSTORE ROU

ZSTORE then stores the user's empty partition under the name ROU, a routine to be deleted. ROU is now deleted.

ZU(SE)   ZU(SE) dev#

Allows temporary use of a terminal device which another job actually owns. This command is primarily intended for use in a broadcast utility routine and for error response queries to the console device.

The user need not OPEN or CLOSE this device. If the device specified in this command is valid, the service is temporarily set as the user's current device. The IOD package arbitrates as to which job accesses the device at any one time. READs should always be preceded by a prompt that identifies the requested information in the same READ command. This will always guarantee that the prompt will appear directly in front of the input, identifying the request. Whenever output is performed by the read command, the input buffer is flushed so that typed-ahead information will not mistakenly be processed.

The legal device values for this command are:

1   4--19   64-11

Functions

A function performs an operation and returns a value based on the outcome of that operation. A function name is always prefixed by a dollar sign ($). Functions are listed in alphabetical order, and may be abbreviated to their first two characters, with the $ counting as the first character. Table 8-6 lists and defines the currently specified functions.

Table 8-6 Functions

$ASCII Selects a character of an ASCII string and returns the code of that ASCII string as a decimal integer.

$CHAR Translates a string of decimal integers into a string of ASCII characters.

$DATA Returns an integer indicating whether the named variable (specified as an argument to $DATA) either contains data, or has 'descendants,' or both.

$EXTRACT Returns a character or substring of a string expression, selected by position number.

$FIND Returns an integer specifying the end position, plus one, of a specified substring within a given string.

$JUSTIFY Returns the value of an expression, right-justified within a field of a specified size.

$LENGTH Returns the length (number of characters) of a string.

$NEXT Returns the lowest numeric subscript value on the same level but numerically higher than the last subscript of the named global or local variable.

$PIECE Returns a substring of a specified string that is delimited by a specific character.

RANDOM Returns a pseudo-random integer uniformly distributed in the closed interval (0, intexpr-1).

SELECT Returns the value of one of several expressions in a list, selected by the truth values in a second list of expressions.

$TEXT Returns the text content of a specified line of the routine in which the function appears.

$VIEW Returns an integer between 0 and 65535, equal to the contents of the memory location specified in the argument.

$Z Function Descriptions

There are certain functions, called $Z functions, which are DSM-11 specific. These functions are provided as extensions to DSM, making many more options available to the user. These functions may be abbreviated to their first three characters. (The $Z counts as the first two characters).

$Z(SORT) $Z(SORT) (glvn(subscript ,...))

Identifies the next subscript at the same level as the given global or local variable. Identical to $NEXT, except for the following differences:

$NEXT
● numeric collating sequence

● starting point is −1

● failure condition is −1

$ZS
● string collating sequence

● starting point is the null string

● failure condition is the null string

System Variables

A number of special reference-only variables are defined within the system to control the flow of information and to provide system information to Standard MUMPS programmers. These variables are maintained and updated by the system for each job partition. They can be examined by various Standard MUMPS commands (WRITE, SET, etc.) but, with the exeception of the $E variable, can not be altered by the program. All special variables can be abbreviated to their first two characters, except for $Z variables, which can be abbreviated to their first three characters. Table 8-7 lists the special variables.

Table 8-7 Special Variables

$HOROLOG	Contains the current date and time.
$IO	Identifies the current I/O device.
$JOB	Contains the job number (positive integer) of each executing DSM job.
$STORAGE	Returns an integer number of characters of free space available for use in the current partition.
$TEST	Contains a truth value computed from execution of the most recent IF command containing an argument, or an OPEN, LOCK, or READ with a timeout.

$X   Contains a non-negative integer value that points to the next column position to be output.

$Y   Points to current line number.

$Z   Reserved for DSM-11 specific extensions:

$Z Special Variables

$ZA   Used with device I/O.

When magtape is the current device, $ZA contains an integer whose bit pattern displays the magtape hardware status register (drive status register for the TJE16). When the sequential disk processor is the current device, $ZA contains either the current disk block address or the error status. When a terminal is the current device, $ZA contains the error status. When another processor is the current device, the low order byte of $ZA contains a count of unsuccessful I/O transmissions (message state only) and the high order byte describes error conditions (message and terminal state).

Refer to the DSM-11 Programmers Guide for bit assignments.

$ZB   Used with terminal type devices; returns the last word of the DDB containing information on the status of a particular device. For the sequential disk processor, $ZB returns the byte offset into the current block. For the DMC-11, $ZB returns, in the low byte only, the current message number (in message mode). For a remote device, $ZB returns the modem timer value.

$ZE(RROR)   Aids error detection in routines. Can be set to enable error trapping. To enable the error trap, $ZE is set equal to a line reference. The line reference may include a reference to a routine, such as SET $ZE= "label + intexpr↑rnam". When an error occurs, control passes to the line and/or routine referenced by $ZE; $ZE is then reset to indicate both the error code and the line and routine which were executing at the time of the error.

The error trap is disabled by setting $ZE to the null string.

Example:

S $ZE="ERR+2↑STAT"

On error, control passes to routine STAT at line ERR+2 (the line whose position is +2 from the line labeled ERR).

$ZN(AME)   Contains the name of the current routine, unless $ZN was set by an argumentless ZREMOVE, in which case $ZN contains the null string. To reset $ZN, a ZLOAD DO, GOTO or an argumentless ZREMOVE command must be made; $ZN can also be reset by transfers to other routines.

DSM-11 SYSTEM SUMMARY
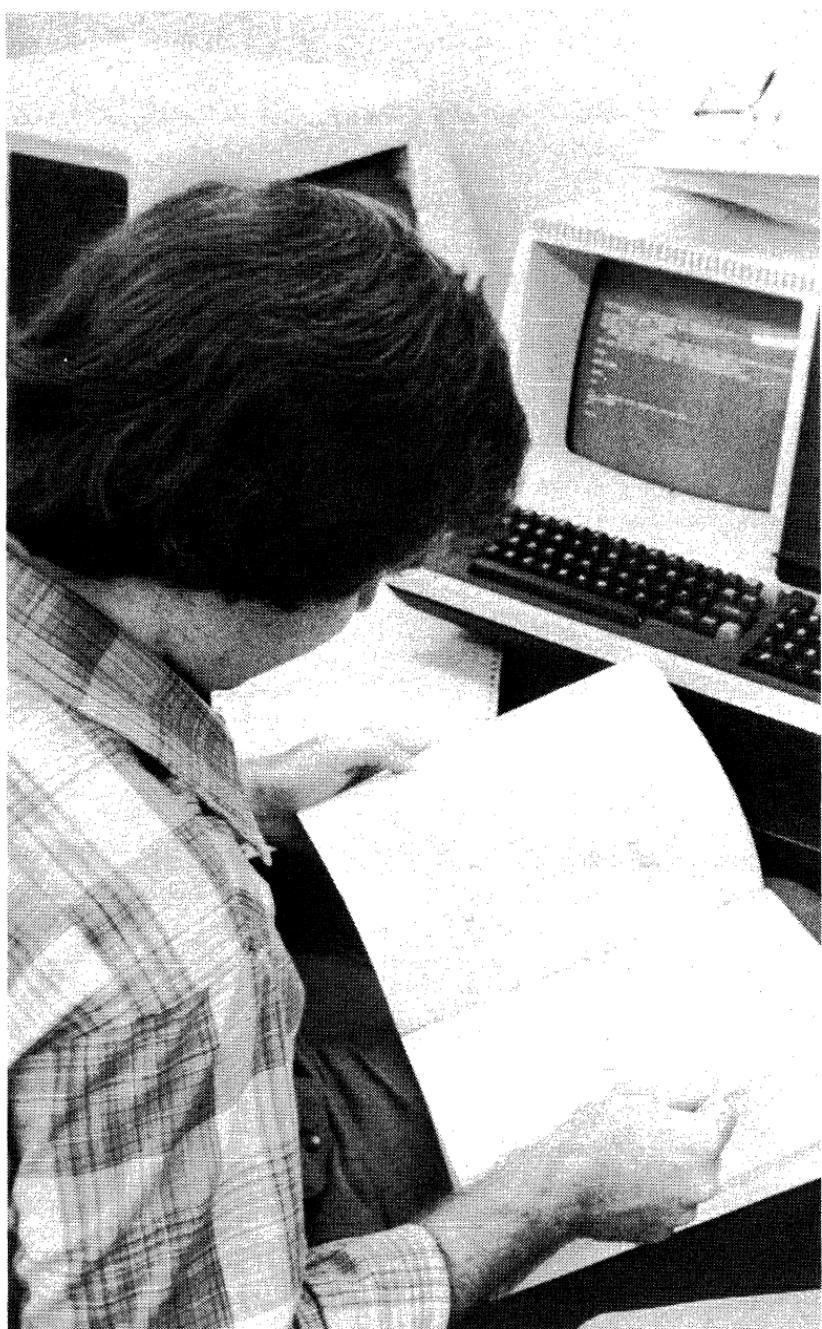
Is

- Interactive, high-productivity applications development for data base management system
- Highly approachable
- Integrated language/command environment
- Powerful language structure for text processing
- Large numer of terminals—up to 80

Is not
- General timesharing
- Real-time
- Computational or batch
- Multi-language

Languages
- DSM-11

TRANSACTION PROCESSING SYSTEM TRAX (V.1)

OVERVIEW

TRAX is specially designed to handle high volume transaction processing in a commercial environment. The system provides for easy interactive application development. It utilizes its own "smart" terminal and terminal language, provides for distributed processing and programming in high-level languages, and offers high data throughput.

FEATURE TOPICS

- Introduction
- Trax System Organization
- VT-62 Application Terminal
  - Screens
  - Application Terminal Security
- Application Terminal Language/Forms Control
- BASIC TRAX-11 Terminology
- Support Environment Features
  - Program Development/Text Preparation
  - Compilation and Linking
  - Batch and Spooling
- System Generation
  - On-line Diagnostics
  - Software/Hardware Error Logging
- TRAX Station Structure
- Station Message Processing
- Exchange Structure
- File Access/Recovery Methods
- System Utilities
- TST Library
- TRAX Communications
- TRAX Languages/Data Manager Options
- TRAX System Summary

INTRODUCTION — WHAT IS TRAX?

TRAX is a high-volume transaction handling data processing system designed for easy application development. It has been designed by DIGITAL as a dedicated system to optimize interactive commercial transaction processing through use of an efficient multitasking architecture, a simplified programming style, and a wide array of program development features.

TRAX is a totally integrated hardware and software system. It employs its own forms-oriented terminal, the VT62, a smart terminal with built-in data formatting capabilities, driven exclusively by TRAX software. Data entered into forms at application terminals provide the basic input to TRAX transaction processors.
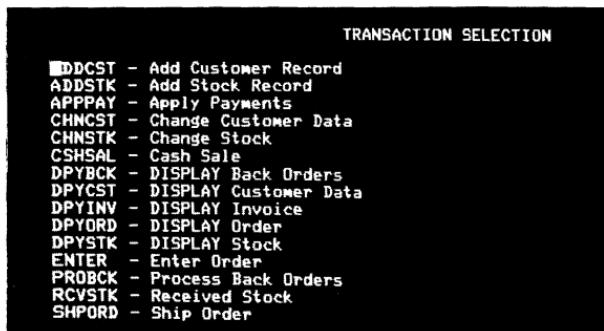
AN APPLICATION EXAMPLE

An order entry application provides an example of the benefits of TRAX on-line transaction processing.

When an order is entered in any system, inventory or customer credit may be insufficient. In a batch transaction processing system, these insufficiencies are usually discovered only after batches of orders are keypunched and processed. Since credits and debits, and shipments sent and received, are also applied to the data base in batches, the credit and inventory records against which batched customer orders are checked may be out of date: recent payments may not yet be processed, so credit or inventory problems reported during batch order processing may not really exist—or a problem may not be discovered until the order is picked (and stock found insufficient) or the credit department holds the shipment.
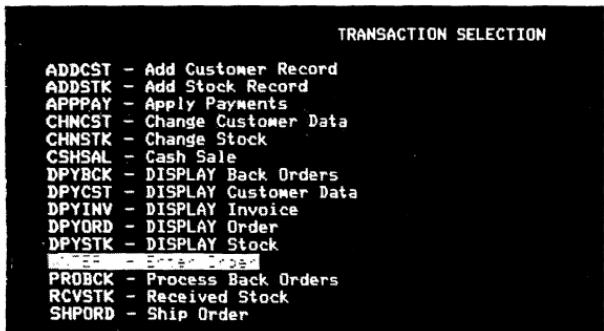
With TRAX, on-line access to the data base permits order entry applications to report credit or inventory problems to the terminal user while the customer is still on the phone. Shipments made, stock received, and debits and credits are applied on-line, as they occur. So credit and inventory checks during order entry are made against records that reflect the moment's reality, not the status of files updated periodically in batch runs. Should a credit problem be discovered, the TRAX terminal user can do a customer status or credit lookup transaction and discuss the problem while the customer is still on the phone. If inventory is insufficient, the customer can be offered a partial shipment or a substitute item recommended by the TRAX application program.

Typically, the VT62 terminal user selects the desired transaction from a transaction selection form, as shown.

TRAX

```
                              TRANSACTION SELECTION
    ■DDCST - Add Customer Record
    ADDSTK - Add Stock Record
    APPPAY - Apply Payments
    CHNCST - Change Customer Data
    CHNSTK - Change Stock
    CSHSAL - Cash Sale
    DPYBCK - DISPLAY Back Orders
    DPYCST - DISPLAY Customer Data
    DPYINV - DISPLAY Invoice
    DPYORD - DISPLAY Order
    DPYSTK - DISPLAY Stock
    ENTER  - Enter Order
    PROBCK - Process Back Orders
    RCVSTK - Received Stock
    SHPORD - Ship Order
```

Screen 1 Transactions available are displayed on the terminal.

Moving the terminal's cursor to the desired transaction on the transac-
tion-selection form, the terminal user presses the SELECT key. The
terminal confirms the transaction chosen, the Enter Order transaction,
by highlighting it in reverse video, as illustrated.

```
                              TRANSACTION SELECTION
    ADDCST - Add Customer Record
    ADDSTK - Add Stock Record
    APPPAY - Apply Payments
    CHNCST - Change Customer Data
    CHNSTK - Change Stock
    CSHSAL - Cash Sale
    DPYBCK - DISPLAY Back Orders
    DPYCST - DISPLAY Customer Data
    DPYINV - DISPLAY Invoice
    DPYORD - DISPLAY Order
    DPYSTK - DISPLAY Stock
    ENTER  - Enter Order
    PROBCK - Process Back Orders
    RCVSTK - Received Stock
    SHPORD - Ship Order
```

Screen 2 Terminal confirms selected transaction.

When the ENTER key is depressed, a form will appear with the cursor
positioned at the beginning of the "customer ID number" field. A cus-
tomer can be identified either by ID number or by name, as the screen
below illustrates.

Customer Identification

Customer #:
Customer Name:

Screen 3   Terminal displays first form of Enter Order transaction.

Customer Identification

Customer #:
Customer Name:

Screen 4   Operator types customer name where prompted by form.
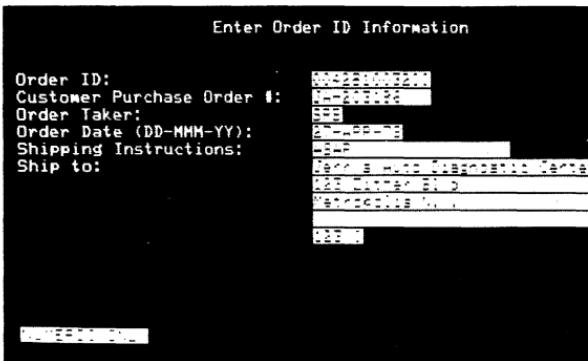
If the ENTER key is pressed at this point, another form will appear that allows the operator to enter general information about the order.

Common typing errors are detected locally in the terminal. For example, if the letter 'T' is entered into the system in place of the number '6' in a Zip Code, an error message will be displayed. The error detection is made without CPU involvement. The screen below illustrates this process.

Enter Order ID Information

Order ID:
Customer Purchase Order #:
Order Taker:
Order Date (DD-MMM-YY):
Shipping Instructions:
Ship to:

Screen 5   Operator types ordering information.

Screen 6 Terminal responds instantly to typing errors.

If the ENTER key is depressed at the completion of this step, a message will appear that reads "transaction complete," as illustrated below.

Screen 7 Operator enters order.

Once the customer order information has been sent to the CPU by depressing the ENTER key, TRAX immediately supplies a form, as shown below, for entry of the item description and quantity ordered information.

231

```
TRANSACTION COMPLETE
  Stock    Quan Description of Item          Each      Available  Extended
                Gasket                       $2.00            23  $12.00
```

Screen 8 Successful completion of transaction is quickly confirmed.

TRAX SYSTEM ORGANIZATION

The TRAX *applications environment* consists of transaction processing support services and a *transaction processor*. A transaction processor is a collection of data tables and software capable of processing an application's transactions. It and the accompanying TRAX support services are the elements that make TRAX a uniquely efficient system.

The TRAX *support environment* provides support services for all non-transaction processing activities, including application batch processing, application development (both transaction processing and batch processing), and management and control of the entire system. The batch processors and support terminals shown in Figure 9-1 can be used interchangeably for any of these activities, because batch processors are the full equivalent of support terminals. In other words, the batch language is the same as the support terminal language.
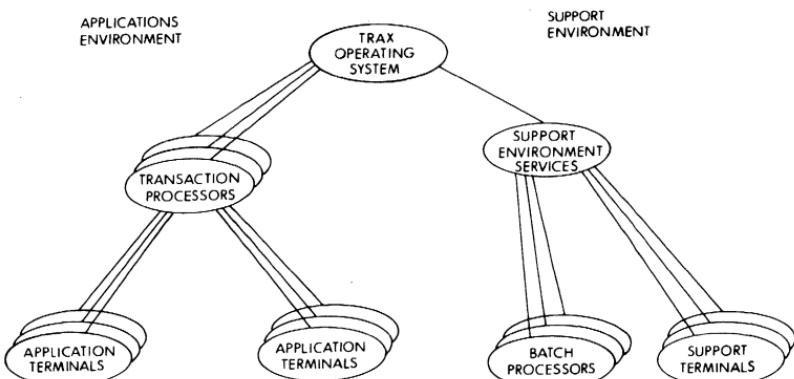
Figure 9-1 TRAX System Structure

232

VT62 APPLICATION TERMINAL

The VT62 is an *applications-only* terminal designed to optimize TRAX transaction processing. It optimizes system performance with block mode transmission and locally buffered menus, forms, and error-checking. For example, messages specific to the form displayed are loaded by the host with the form into the terminal, for local error detection without CPU intervention.
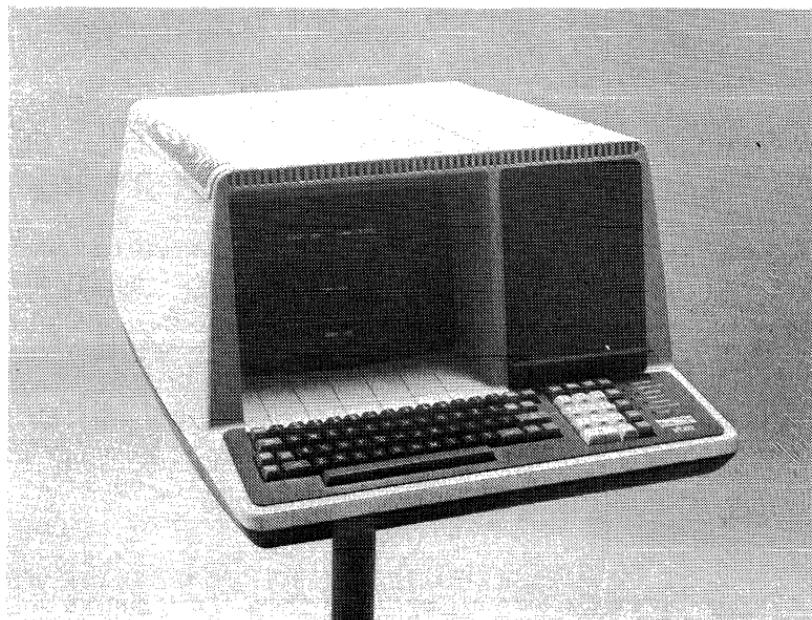
Figure 9-2 VT62 Terminal

A block-mode terminal, rather than communicating character-by-character with the host, transmits whole screen loads of data at once, considerably reducing CPU involvement. The VT62 is also a multidrop terminal. Up to eight VT62s can be hung on a line and polled by the processor.

The VT62 communicates as specified by DIGITAL's DECnet standard DDCMP protocol. It operates asynchronously at speeds up to 9600 baud and synchronously up to 4800 baud. It can operate in full or half duplex using either dedicated lines for local use or standard data set modems over dial up or leased lines. It can be connected point to point or multidropped with up to eight terminals per line.

There is provision for adding a hard copy terminal (LA180P) to the VT62. This output-only device shares the VT62 microprocessor and line interface, but otherwise looks like a separate and independent output-only device to the applications programmer. This option can be used as a low cost, error free, remote forms-oriented printer.

Screen(s)

The display screen, which has 24 80-character lines, can be logically divided under control of a form definition into three areas: the Display Area, the Form Area, and the Error Line.

The top part of the screen is the Display Area. This area can be from 0 to 23 lines long, software selectable. Any area between the Display Area and the bottom line, the Error Line, is the Form Area.

The Display Area is for menu selection and system or operator messages. The Form Area is for user data input in transaction processing. When typed errors are detected in the terminal, the Error Line displays a locally stored message.

Because error messages are loaded into the terminal, foreign language and special purpose messages can be used.

Display Area

The Display Area is divided into menu fields, display fields and unused (blank) areas. Menu fields are usually names of transactions that can be selected; VT62 users can move the cursor from one menu field to another by means of control keys. They can (de)select a menu item (a transaction) by pressing the (DE)SELECT key. A selected menu item is displayed in reverse video and is returned to the host when the user presses the ENTER key.

In the Display Area, display fields are used by the system to present text to the user. The text can be presented in normal or reverse video. The cursor cannot be positioned in these fields and the user can not alter them.

```
TRANSACTION  SELECTION

[A]DDCST  —  Add  Customer  Record
ADDSTK  —  Add  Stock  Record
CSHSAL  —  Cash  Sale
DPYCST  —  DISPLAY  Customer  Data
DPYBCK  —  DISPLAY  Back  Orders
DPYINV  —  DISPLAY  Invoice
DPYORD  —  DISPLAY  Order
DPYSTK  —  DISPLAY  Stock
ENTORD  —  Enter  Order
CHNCST  —  Change  Customer  Data
CHNSTK  —  Change  Stock
APPPAY  —  Apply  Payments
PROBCK      Process  Back  Orders
RCVSTK  —  Received  Stock
SHPORD  —  Ship  Order
```

Screen 9 Display Area, Form Area, Error Line

```
Customer  Master  File  Subsystem  —  Change  Customer  Transaction

—►STATE  CODE  INCORRECT

Customer  Number      000003
Customer  Name        ROBERT  T  SMITH

Address               DIGITAL  EQUIPMENT  CORP.
                      146 MAIN STREET
City, State           MAYNARD                      MX
Zip  Code                                          01754
Telephone             (617)  493-8874

Attention             BOB  SMITH

Credit  Limit  ($)         500.00

Function  Keys:   ENTER  to  refile  customer  record ,  CLOSE  to  quit  without  filing
```
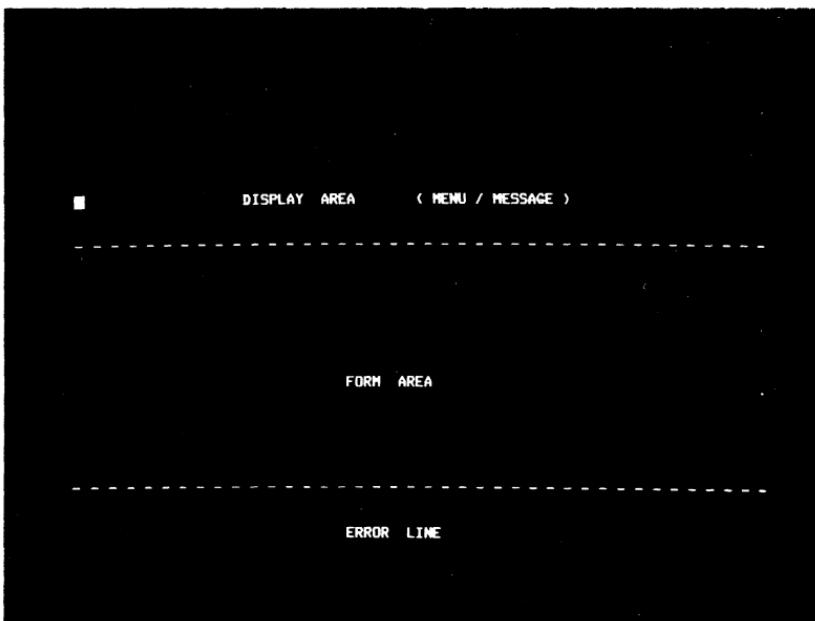
Screen 10 Display Area showing menu selection.

Customer Master File Subsystem — Add Customer Transaction

Customer Number   000003
Customer Name   ROBERT T SMITH

Address   DIGITAL EQUIPMENT CORP.
                      146 MAIN STREET
City, State   MAYNARD   MA
Zip Code   0175X
Telephone   (617) 493-8874

Attention   BOB SMITH

Credit Limit ($)   500.00

Function Keys: ENTER to Add customer record, CLOSE to quit function
→NUMERIC ONLY

Screen 11 Form Area and Display Area showing system message.

DISPLAY AREA   (MENU / MESSAGE)

FORM AREA

ERROR LINE

Screen 12 Error Line below Form Area with local error message resulting from user-entered typing error.

236

Application Terminal Security
Application terminal security is provided by a sign-on/sign-off facility and by work classes and user authorizations. A work class is a list of transactions stored as part of the transaction processor. When associated with a terminal (message processor) or a user, a work class determines the transactions that station or user is allowed to invoke. A list of valid transactions is required to define a work class. A given transaction can be included in any number of work classes, and be dedicated to a terminal.

If a work class is to be associated with a terminal station, the work class name is specified when the station is defined. Only those transaction types included in the work class can be invoked from the terminal station. The sign-on and sign-off transactions can be included in a terminal's work class. If a user signs on, the transactions in the work class associated with the user's identifier can be invoked. After sign-off, the terminal work class again defines what transactions can be invoked at the terminal.

For each user authorization record, the following is specified:

- user identifier
- password
- valid work class names

ATL — APPLICATION TERMINAL LANGUAGE/FORMS CONTROL
The Application Terminal Language (ATL) provides a straightforward method for the application programmer to define a form.

The form definition, when incorporated into a transaction processor, is used to control all aspects of the interaction between the terminal user and the TRAX system.

Using ATL, the application programmer specifies:

- The layout of the form presented to the user at the application terminal, including the size of the Display and Form Areas and the sizes of each field (display, prompt, input, print, or menu).

- How the user must fill out the form, including special restrictions on the kinds of characters that are permitted in various input fields.

- Initial contents of each field. Initial contents can be literal text, the current date, time, transaction number, or transaction name, the terminal station name, text supplied by a program (via a message), or blank.

- Enabling of special terminal control keys (function keys) by which the terminal users can indicate one of several responses to the form.

- The formats of the replies that are to be used within the context of the form. Up to 64 replies can be associated with a form and these replies are program selectable by number. For a reply, the programmer specifies which fields on the screen are to be overwritten and their new contents. The programmer has available all the types of data that were available for initializing the field, that is, literal strings, system-supplied data, and data contained in the message that requested the reply. The required data fields in the message are defined as part of the reply specification. In addition, the programmer can specify the position of the cursor and enable or disable selected function keys.

- Special form options, such as designating a form to be a transaction selection form. If the form is to be used to select the next transaction, the programmer specifies which field or menu will contain the transaction name, whether authorization checks are to be made, and associated error messages.

- Typed screen entries are formatted into a message for processing by one or more application programs. The format of the message is also defined as part of the form. In addition to the contents of the input and selected menu fields, the programmer can specify all the system-supplied text (e.g., date, time) or literal text strings.

For maximum transaction processing efficiency, ATL is a compiled language. The application programmer prepares a separate ATL source for each form definition, using the DIGITAL standard editor (EDT). This source file contains English-like ATL language statements and compiler directives such as default specifications and statement repetition.

The statements, as a file, are read by the forms definition file utility program, ATL, which produces form definition records. The utility checks the form definition for errors and produces a detailed description of the form and its associated messages. The compiled output is saved as part of the transaction processor's forms definition file.

The example below illustrates an ATL screen format definition.

```
!ACCOUNT NUMBER FORM

ENTER                                    !SET THE SCREEN
      SPLIT = 3                          !3 LINE DISPLAY AREA
PROMPT = 1,1                             !SET UP THE PROMPT
      VALUE = "ACCOUNT #:"               !WITH THIS TEXT
INPUT = ...,+2                           !SET UP THE INPUT FIELD
      LENGTH = 6                         !6 CHARACTERS LONG
      CLEAR = "_"                        !INDICATE WITH UNDERSCORE
      ATTRIBUTES = NUMERIC,RIGHT,REQUIRED
      LABEL = ACCOUNT                    !LABEL THE FIELD
MESSAGE = 1                              !SET UP THE FQM
      VALUE = ACCOUNT                    !SEND THE INPUT DATA




END

■
```

Figure 9-4 ATL Screen Format Definition

Figure 9-5 shows all of the processing paths which can result from the interaction between exchange control processing and forms. The proper choice of processing paths by the application designer will result in straightforward conversational processing as shown in Figure 9-6.
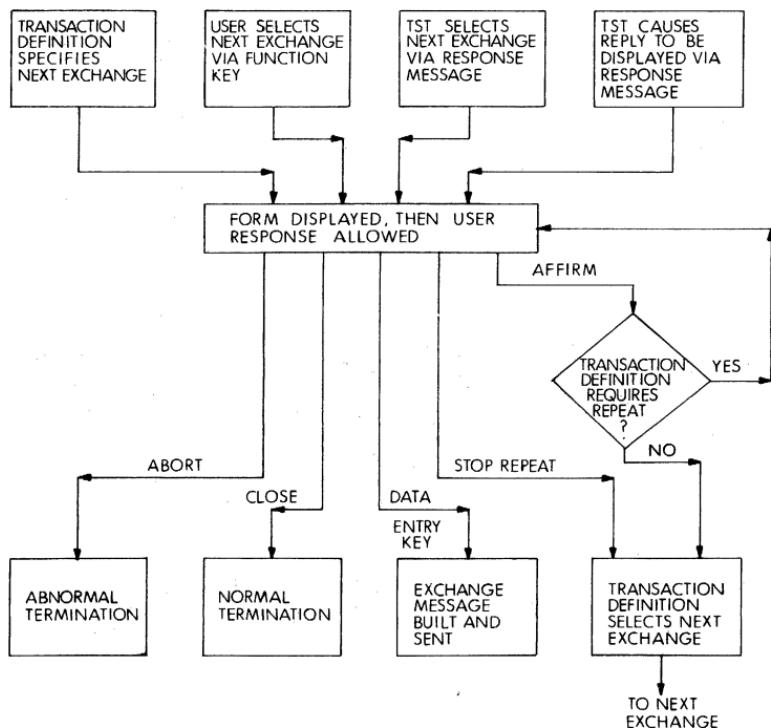
Figure 9-5 Processing Paths for Forms
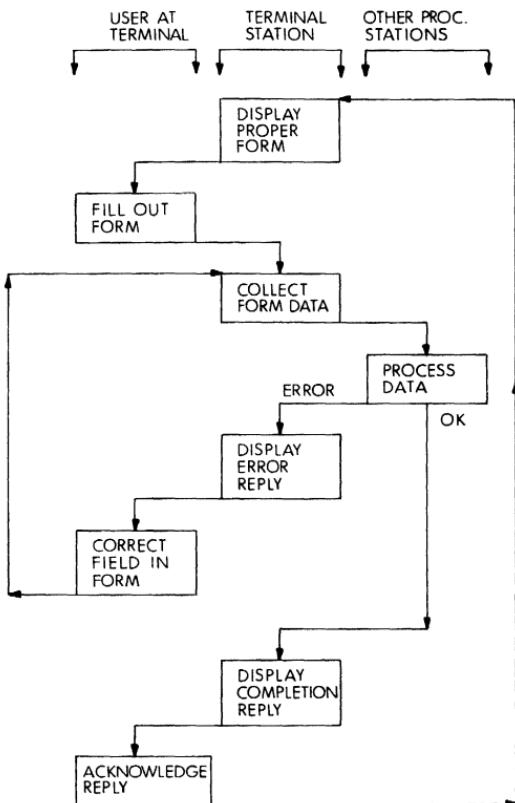
Figure 9-6 Typical Exchange or Message Processing Paths

BASIC TRAX TERMINOLOGY

Application

An application is a logically related set of data processing operations which support a specific business activity. It is a set of transactions in a transaction processor. Examples of applications include:

- Order Processing
- Accounts Receivable
- Accounts Payable
- Purchase Orders
- Inventory Control
- Student Records

241

The term "application" is also used as an adjective to describe the elements of a data processing system which were specifically developed to support an application. For instance:

- application program
- application file
- application terminal

Transaction
A transaction is a pre-defined unit of data processing performed by a transaction processor. The processing of the forms associated with an order entry is an example of a transaction. There are generally several forms associated with each transaction. The processing of one form is an exchange.

Transaction Instance
Every invocation of a data processing operation (i.e., a transaction) is defined in TRAX as a transaction instance. In TRAX, transactions are usually initiated by the actions of an application terminal user, and the result of the specified operation is made available to the user.

Transaction Processor
A transaction processor has the following elements:

- TRAX transaction processor executive services
- A set of forms definitions
- A set of transaction definitions
- A set of station definitions
- A set of Transaction Step Tasks (TSTs)
- A set of work classes (optional)
- A set of user authorizations (optional)
- A set of specifications for each file accessed, including journaling requirements
- A set of support services

A single transaction processor is defined to process the transactions associated with one or more applications.

Once it has been constructed, the transaction processor functions as a unit. It may be installed, started, stopped, and removed from the system with simple operating system commands. While it is running, the applications terminals assigned to it are available for transaction processing.

The management services which TRAX provides to a transaction processor are extensive. Only application routines to handle application-specific processing steps are programmed: the rest of the specifications needed to construct a transaction processor are parameter table entries. These tables drive the TRAX transaction processor management services, producing in the desired transaction processor behavior. This organization results in:

- Consistency of transaction processor structure
- Much less application code to debug
- Consistently applied system optimization techniques
- Easier documentation

Transaction Slot
Every transaction instance has associated with it a data structure called the transaction slot, containing:

1. the current exchange message
2. the transaction workspace
3. the system workspace

The first item is the data exchanged between the user who has just filled in a form and the application program(s) that must process the data the user provided.

The transaction workspace is the area where an application program can save data for use by another program, providing both programs are performing processing for the same transaction instance.

The system workspace is used by the TRAX transaction processing support services to administer the execution of the transaction instance. Like the transaction workspace, it is allocated when the transaction instance is initiated and is kept until the transaction instance terminates.

The exchange message, transaction workspace, and system workspace are directly associated with the transaction instance and are stored together within the transaction processor. The aggregate data structure containing these three elements is called the transaction slot. The transaction slot contains all available transaction instance context.

Transaction Processor Cache
When a TRAX transaction processor is started, it has assigned to it a contiguous area of memory referred to as a cache. The transaction processor uses this cache for two major purposes:

1. To satisfy requirements for temporary buffer space.
2. To store in memory disk-resident data such as RMS index and data blocks, forms definitions, application programs, etc.

Every time a request is made to read a specific disk block, the transaction processor looks first in its cache. If the block is already there, a physical disk access is avoided.

When the cache is full, a new block is read into the cache area that contained the least recently used block. Thus, frequently used blocks such as the first level index of an RMS index-sequential file or a popular transaction definition record will tend to remain in the cache. Of course, if a block in the cache is updated, the transaction processor also updates the disk itself.

The net effect is a decrease in the number of real disk accesses and increased system performance.

TRANSACTION PROCESSOR

```
        ┌──────────────┐      FORM 1
        │    READ      │◄─────────────┐
        │   MASTER     │              │
        └──────────────┘              │
               │ TST 1                │
               │                      │
        ┌──────────────┐              │
        │   DISPLAY    │        EXCHANGE 1
        └──────────────┘              │
               │ TST 2                │
               │                      │
        ┌──────────────┐      FORM 2  │
        │   UPDATE     │◄─────────────┘
        └──────────────┘
               │ TERMINAL
               │ INPUT
        ┌──────────────┐
        │   VALIDATE   │◄─────────────┐
        └──────────────┘              │
               │ TST 3          EXCHANGE
               │                      │
        ┌──────────────┐      FORM 3  │
        │   WRITE      │◄─────────────┘
        └──────────────┘
               │ TST 4
```
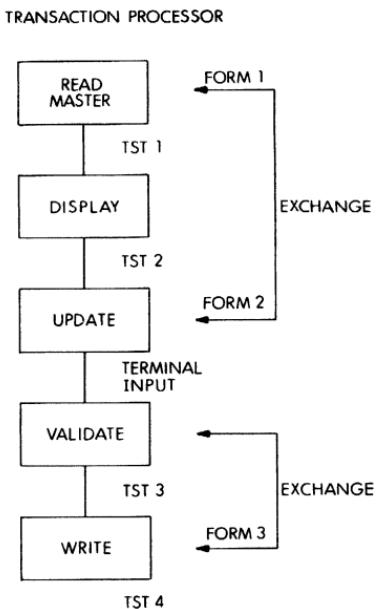
Figure 9-7 Transaction Processor

Transaction Step Task (TST) Structure
A transaction is processed by a set (one or more) of subroutine-like application modules called TSTs. The TST is initiated upon receipt of a station exchange message at the station associated with each TST.

The TST, being a subroutine, accepts a set of parameters which are passed by the TRAX executive. These parameters are:

- the address of the exchange message
- the address of the transaction workspace

  (The maximum size of the workspace is defined by the designer via the TRADEF utility. The combined size of the station exchange message and workspace cannot exceed 8 K bytes.)

The TST processes the exchange message, and optionally performs any data file accesses, manipulates the workspace or exchange message, sends any independent station messages and then exits.

All data file access is via native language I/O statements. The TRAX system does the actual processing of the RMS I/O operations.

All other interactions between the TST and any other element in the system are performed by routines in a TST library supplied with the TRAX system.

SUPPORT ENVIRONMENT FEATURES

A TST is coded in the support environment (SE) with the standard editor (EDT) provided with TRAX. The TST is assembled or compiled with a language processor, the COBOL compiler, for example. The resulting object module is then built into a task image with the TSTBLD utility.

In addition to its special transaction-processing features, TRAX provides support for typical batch processing and utility programs. The services provided by the TRAX support environment (SE) are similar to those provided by other DIGITAL multi-user operating systems.

Whereas TRAX transaction processors always work as a unit, programs run in the TRAX support environment stand alone. These programs may be initiated from a support terminal or by a batch processor.

The TRAX support environment can be used to run:

- Application batch processing.
- Program development utilities, including text editors, compilers, task builders, debuggers, and documentation preparation utilities.
- Utility programs that generate transaction processors.
- Utility programs that manage a TRAX system, including console operator's utilities, batch processors, and spoolers.
- Utility programs that back up and restore system data files.

Because no specialized system services are provided in the support

environment, equivalents of transaction processors are not required. Programs in the support environment stand alone, and have the same system services available to them. Programs may be started or stopped as desired, and interaction between a program and its support terminal (or batch processor) is conversational, rather than forms oriented.

Program Development/Text Preparation

Source programs and documentation are prepared on-line in the support environment with the DIGITAL standard editor, EDT. Incorporation of screen mode display, cursor control, and word processing commands provides an efficient means to enter, display, and edit text. A text formatting program is available for documentation files.

Compilation and Linking

The DCL command language has simple commands to translate a source program into a runnable program or task.

DEBUG

Individual TSTs are debugged in the support environment using the DEBUG utility. DEBUG emulates the entire transaction environment yet gives the programmer access to interactive debugging facilities such as the BASIC-PLUS-2 debug monitor. In addition, it allows the programmer, either interactively or via a file, to provide the input messages for the TST to process. All external calls made by the TST are logged in a file or on the support terminal.

When all TSTs are individually tested, a debug transaction processor can be constructed to run in the application environment. This incorporates a full trace facility as well as software error logging for the operator's use.

Software Performance Monitoring

TRAX provides a set of utilities which can produce extensive transaction processor statistics. These utilities keep track of various system parameters and provide a picture of the transaction processor operation over chosen periods of time. They can be used for system tuning. The statistics collected via the report generation facilty include station, cache, transaction, and data file statistics as well as general system statistics. The report generation utility has been designed to run independently of the sample gathering so that reports can be generated during off hours or as a low priority background job.

Batch and Spooling

TRAX provides a centralized QUEUE facility within the support envi-

ronment. This facility allows the generation of named work queues and the subsequent submission of batch or line printer spooling jobs to named queues. The submission can occur from a support terminal, from a program running in the support environment, or from a transaction application program (TST) running in a transaction processor. Multiple batch processors and line printer spoolers can operate concurrently. The QUEUE management commands include:

- INITIALIZE a queue (generates a name).

- DELETE a queue, the complementary operation to the INITIALIZE option. All elements in a removed queue are purged.

- START a queue, i.e., commence de-spooling activities on the particular queue.

- STOP a queue, the complementary operation to START; all elements remaining in the queue are preserved for future de-spooling operations.

- SHOW the contents of a queue.

- Hold an element in a queue, i.e., prevent de-spooling of the job even if priority and after times would allow such a de-spooling.

- Release a held job for normal de-spooling.

- DELETE an element from a queue.

- Modify the submit/time characteristics of an element in a queue. For example, move it to the top of the queue by changing its priority to be greater than any other element.

- STOP/START the curently active job in a queue in order to:
  - abort it
  - restart it (characteristics allowing)
  - requeue it (characteristics allowing)
  - space (backward or forward) a number of pages for print jobs

- INITIALIZE a de-spooling task or batch processor.

- DELETE a de-spooling task or batch processor.

- START a de-spooling task or batch processor.

- STOP a de-spooling task or batch processor.

- ASSIGN a queue to a de-spooling task or batch processor.

- DEASSIGN a queue from a de-spooling task or batch processor.

- START the queue manager.

- STOP the queue manager immediately or at completion of jobs currently running.

SYSTEM GENERATION

TRAX system generation is simple, interactive, and straightforward. All TRAX options are fully installed by answering "yes" to a question and mounting the option tape; no special installation instructions are required. When the generation is complete, a copy of the system is stored on magnetic tape so system generation doesn't have to be repeated.

Patching is fully automated; all control and patch data reside on a single magnetic tape. This tape is issued at regular intervals.

Transaction Processor Generation Utilities

Transaction Processor generation involves identifying and defining all components of a transaction processor. Eight interactive utilities are provided to simplify the task. They are:

TPDEF   Defines the general attributes of a transaction processor.

TRADEF   Defines all transactions.

WORDEF   Generates and names sets of transactions required to limit system access by terminal number (see STADEF) or by operator identification (see AUTDEF).

AUTDEF   Establishes operator identification and association of operator to one or more transaction sets (work classes) defined previously by WORDEF.

ATL   Generates all form definitions.

TSTBLD   Generates task images of all TSTs.

FILDEF   Identifies and logically names all application data files belonging to this transaction processor.

STADEF   Defines all stations.

Dialog Conventions

The transaction processor generation utilities run as interactive dialogs. That is, the utility will type a prompt and take appropriate action based on the response. The format of a prompt is:

> question <default>?

The value enclosed by angle brackets is the utility's default value. If there is no default, the angle brackets are omitted. The utility checks

each reponse as it is received. If the response is invalid, the prompt is repeated. In most cases, the utility provides an error message before re-prompting.

In general, the operator types a response, terminating it with a carriage return. The form of the answer should be clear form the prompt. Unless a literal type of answer is expected, the utility will convert all input to upper case and compress all blanks. If the expected response is a Yes/No, the utility will accept Y, Ye, Yes, N, or No as valid answers. Unless otherwise stated, all input numbers are expected to be decimal. If the user expected to select an item from a set (e.g., select the next function), the response need contain only the number of characters required to obtain a unique answer.

All prompts recognize several special responses. If the operator types a carriage return without typing other characters before it, the utility assumes the default value. This is the value that was specified between the angle brackets. If the prompt is not understood or the valid answers are not known, a ? carriage return should be typed. The utility will display an explanation of the prompt and the prompt will be repeated.

The ESCAPE key is used to escape from a particular dialog step and return to the previous step. If the step is a step in a dialog loop, the utility will return to the first step of that loop.

If a CTRL/Z is typed in response to a prompt, the utility will do an orderly exit.

On-line Diagnostics

TRAX provides a set of diagnostics for disks, tapes, and application terminals that can be run from the support environment while normal system operation continues. This allows investigation of possible hardware problems without terminating normal system operation.

Software Error Logging

TRAX software error logging writes all transaction processor-detected errors to a disk file and optionally to a support terminal. A utility can be used to select and format the software error entries into an error log report.

Software error logging detects malfunctioning applications software and provides a unique tool to debug an on-line transaction system. The software error log also reports terminal status errors and hardware errors that were detectable only by software.

Hardware Error Logging

The TRAX hardware error logging facility monitors the hardware reliability of the system.

It detects and records information about hardware errors as they occur whether or not the error is recoverable.

The system operator (or programmer) can use the analysis and report utilities to detect increased error frequency and schedule maintenance without affecting normal system operation.

COMMON FEATURES

Although TRAX makes a significant demarcation between its transaction processing services and its support environment, there are several operating system features which are shared by all TRAX system components:

1. Multitasking Operating System

   TRAX is an advanced multitasking operating system. Many different tasks can be active simultaneously within a single CPU. Support environment programs can be active at the same time as transaction processors, and multiple transaction processing steps can be active simultaneously within a given transaction processor.

2. RMS-11 File Management

   The TRAX operating system supports RMS-11 file access for both the transaction processing and the support environments. For transaction processing, TRAX provides RMS-11 file management as part of the transaction processor executive services. In the support environment, where each program is independent of any other, RMS-11 support must be linked into each program. The two methods of using RMS-11 are compatible, allowing files created in either environment to be used in the other.

TRAX STATION STRUCTURE

A station is a logical location within a transaction processor where formatted data (station messages) are received and processed.

A station also controls a defined system resource (terminal, link, applicaton program, etc.).

This control encompasses the formatting of data to and from the resource, the control of all transaction instances initiated at the station, and the translation of resource input data into system messages for further processing by a pre-defined set of stations.

Each station has an active processing element associated with it: either a set of TRAX transaction processing services or a specially

programmed application routine. The station's active processing element is responsible for processing all station messages that are directed to its station. In the course of this processing, the active element may generate new messages and send them to other stations. When the active element has finished with the original station message, TRAX system services forward the message to its next designated destination.

There are seven types of stations within a transaction processor. They are:

1. TERMINAL STATION
   A station that controls one application terminal.

2. TST STATION
   A station that controls the execution of a named transaction step task (application program). Each TST has its own station.

3. MASTER LINK STATION
   A station that controls a communication link to another transaction processor or to an IBM CICS system.

4. SLAVE LINK STATION
   A station that receives transaction initiation requests from a master link in another transaction processor (in the same or a separate system) and initiates and controls the requested transactions.

5. SUBMIT BATCH STATION
   A station that submits batch job requests received from transaction instances within the same transaction processor.

6. SLAVE BATCH STATION
   A station that receives transaction initiation requests from a support environment program and subsequently initiates and controls the requested transaction(s).

7. MAILBOX STATION
   A station that adds received messages to a sequential disk file and allows transactions to retrieve such messages in a first-in/first-out manner.

Station Message Processing

The vehicle for data transfer between stations is referred to as a station message. There are four types of station messages:

1. Exchange Message
   An exchange message is generated by the station which initiates a transaction and is routed sequentially to a pre-defined list of stations. Each station receiving the exchange message may read or modify the message contents. The exchange message travels as part of the transaction slot.

2. Response Message

A response message is a station message returned from a TST processing the current exchange message back to the station which inititated the transaction instance.

Response messages are used for the following purposes:
- returning data from the exchange message processing to the source station
- exchange control (discussed later in this chapter)
- forms control (discussed previously)

3. Mailbox Messages

A mailbox message is a station message sent from a TST to a mailbox station. The message data is stored on a disk file by the mailbox station.

4. Report Message

A station message sent to an output-only terminal station, containing a form name and the data to be displayed or printed.

Exchange Structure

An exchange is a cycle of transaction processing consisting of the processing of an exchange message at one or more stations.

The system begins an exchange by placing the exchange message (containing the data to be processed) into the transaction slot. An exchange ends when the last station in the exchange message route list has finished processing the message.

To define an exchange, the applications designer must allow for:

1. A six character exchange name.
Each exchange in a transaction definition has a unique name.

2. A forms definition (form) name.
Each exchange that will be initiated from an application terminal has a unique name.

3. Route list
The route list provides the name(s) of one or more station(s) to which the exchange message is to be sequentially routed.

4. Subsequent action
Subsequent action is a set of parameters that define the action to be taken by the originating station after the exchange message has been composed and dispatched to the first processing station.

EXCHANGE STRUCTURE
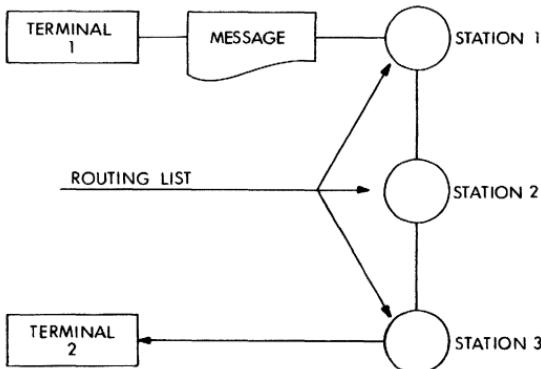
Figure 9-8 Exchange Structure

Exchange Control

Exchange control is the specification of the next exchange to be exe-
cuted when the current exchange terminates. Exchanges are con-
trolled three ways: with exchange definition options, response mes-
sages, and terminal function keys.

Interactive terminal stations manage the flow of a transaction instance
so that there is never a conflict in exchange control. At any instant a
terminal station may be awaiting a response message to a previously
sent exchange message, or it may be awaiting a user response to a
previously dispatched form or reply. But it is never awaiting both si-
multaneously. Response messages are ignored when awaiting user
responses and user responses are ignored when awaiting response
messages. Both a response message and a function key may exert
exchange control, but only if they are employed in the proper
sequence and at the proper times. They will then both have an effect,
and the result will be successive applications of the relevant exchange
control rules.

Station Sharing

It is possible for many transaction instances to be active at the same
time in a given transaction processor. Each will have a station mes-
sage waiting or being processed at a station. A station can be used by
different transactions. For example, two TSTs that might arrive at a
station for processing at the same time are

• a look-up on a customer file

• a look-up on an inventory file

For the most part, the application designer is not affected by these multiprocessing issues. The TRAX transaction processing support services manage each station so that extra messages are queued up and released as soon as the appropriate station is free. The exchange messages and workspaces are managed automatically, so that each time a station is activated, the proper exchange message and work-spaces are attached for its use.

FILE ACCESS/RECOVERY METHODS

A transaction processor is constructed with specific file access permissions. Each accessible file must be identified during the construction process.

The transaction processor will open each of the files as specified by the applications designer. The files will then be available to all transaction instances within that transaction processor. The transaction processor opens each file with either update or read-only access. If the transaction processor has update access to a file, any of its transaction instances can update or read records in the file. Of course, if the transaction processor has read-only access, none of its transaction instances can update the file.

Only one transaction processor can have update access to a given file. All others accessing that file at the same time must be restricted to read-only access.

A transaction processor provides full file sharing and record locking facilities for its transaction instances. File sharing and record locks are managed on a transaction instance basis, rather than on a station basis. Thus, a station does not lock a record for itself but for the transaction instance which it is processing at the time. The record lock remains associated with the transaction instance, and the locked record is available to all other stations which subsequently process the transaction instance. In fact, once the original locking station ceases to process the transaction instance, it will be unable to access the locked record for another transaction instance.

At the option of the application designer, file updates done by a transaction processor can be staged and journaled. This option is selected individually for each file accessed by the transaction processor; files can be journaled and staged, just staged, or neither.

Staging

Staging is the delay of each update to a file until the end of the transaction instance requesting the update. Its value is that the update will never occur if the transaction instance is aborted, and the file will

remain intact without cleanup operations. Of course, the delay of the update also prolongs the period that the records are locked; a lock which was exercised when a record was read for updating will not be released until the actual update occurs after the transaction instance terminates. In some situations, this may prolong the period of record locking to an unacceptable degree from a system performance viewpoint, and staging in such situations is impossible.

The application designer and his programming staff must be alert for possible record lockout situations when staging is used. The application programmer cannot cause premature file operations on a staged file.

Staged Records

Staged records are stored in the transaction instance's system workspace to await the end of the transaction instance. Storage of staged records uses the bulk of the system workspace, and is a major determining factor in the size of the workspace required for a given transaction definition.

Journaling

Journaling is the parallel writing of updated records to a second medium in addition to the updating of the original file. The resulting journal can be used to reconstruct the file in case it is damaged or lost.

TRAX transaction processors journal by writing the transaction slot onto the journal medium at the end of the transaction instance. This transaction slot contains the current exchange message for the transaction instance, as well as the transaction workspace and the system workspace. Files which require journaling will also be staged automatically, thus guaranteeing that updated records in journaled files will appear in the system workspace, and hence will be journalized.

The journaling operation is done at the end of any transaction instance which has updated at least one record in a journaled file.

Recovery

TRAX error detection and recovery methods allow applications design to include system integrity and recovery capabilities acquired at some cost in system overhead.

TST LIBRARY (USER INTERFACE)

The following is a list of system functions available to a TST—all are invoked by an external call specifying parameters using the standard higher-level language calling sequence.

CALL   SYSTEM RESPONSE

SNDMBX   Send to Mailbox Station — causes message to be added to specified mailbox station queue.

REPORT   Send report to output-only terminal station.

REPLY   Send Reply Message — allows initiating station to go on to new exchange message.

PROCEED   Send proceed message — directs initiating station to go on to new exchange.

STPRPT   Send stop repeat message — directs initiating station to disregard any existing repeat options.

TRNSFR   Send transfer to named message — permits the initiating station to begin execution of a specified exchange definition.

CLSTRN   Send close transaction message — causes a normal termination of the current transaction instance.

ABORT   Send response message and abort transaction — automatically deletes any remaining routes for the current exchange message.

RESTRT   Restart exchange — abort will occur if transaction not defined with exchange recovery.

AROUTE   Add route — the specified station (6-character ASCII) is added to the end of the route list for current exchange message.

DROUTE   Delete route — the specified station will be deleted from the route list of the current exchange message.

DALLRT   Delete all routes — all remaining routes are deleted.

GETMBX   Get from mailbox.

MBXNUM   Get the number of message currently queued to mailbox.

TSPAWN	Spawn a transaction — allows a TST to spawn a single exchange transaction or a new transaction instance.
TABORT	Abort a transaction — causes abort of a spawned transaction, or if ABORT call cannot be executed.
GETIME	Get Time — fills specified buffer with current date and time.
GETSTN	Get current station name — returns in buffer the station name associated with the current TST.
GETSRC	Get transaction source station name — returns name of source station for current transaction instance.
GETRAN	Get transaction name — returns the transaction name for the current transaction instance.
LOGTRN	Log transaction — writes user specified log data to journal file.
GETFIL	Get file specification — used to return a physical file specification to a TST for further processing.

TRAX LANGUAGES/DATA MANAGER OPTIONS

TRAX includes either COBOL or BASIC-PLUS-2; the other is optional. TRAX COBOL is functionally similar to COBOL-11 V3.5. TRAX BASIC-PLUS-2 is the same as Version 1.5 offered on DIGITAL systems. The TRAX TST philosophy of programming small modules of TST code to handle specific steps within an application makes memory utilization more efficient through use of a shared object time system. In addition, the DATATRIEVE-11 data management option is a standard part of the system.

TRAX COMMUNICATIONS

TRAX Communications are designed to allow an active transaction within a TRAX transaction processor to initiate communications with a transaction processor in another system by using an IBM 3271 emulator. This other system may be another TRAX system or an IBM System/360 or System/370 running an IBM transaction processor (CICS).

The communication facility isolates the application program from detailed knowledge of protocols, message flow control, and link multiplexing. Access to either TRAX or IBM remote systems is via the same simple user interface.
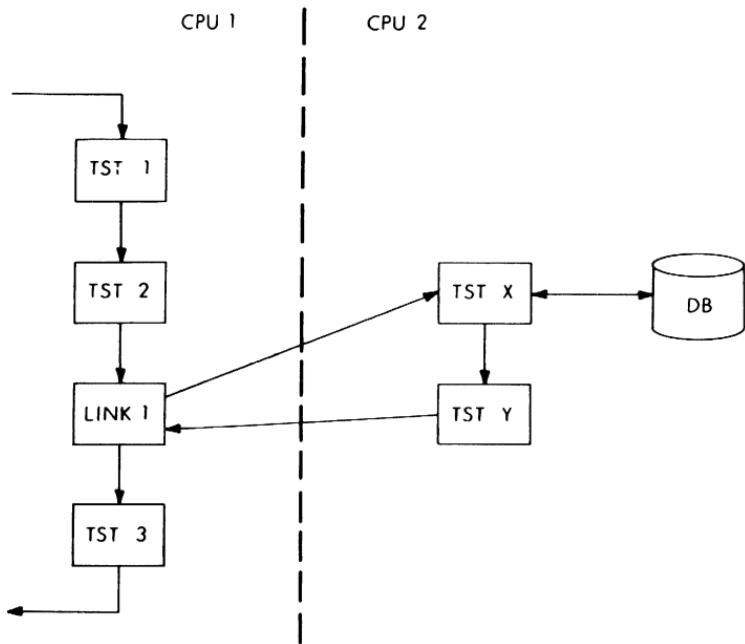
Figure 9-9 TRAX Communications

KMC11-A

The KMC11-A auxiliary processor is used within TRAX for the charac-ter-by-character interrupt processing required for application terminal line interfaces and the TRAX/TL-3271 interface. The KMC11-A high speed microprocessor as used by TRAX is capable of controlling up to 48 DZ11 lines or 16 DUP11-DA lines. If both interfaces are present, then two KMC11-As are required.

The KMC11 connects directly to the PDP-11 UNIBUS and accesses both main memory and the I/O device registers to significantly reduce the CPU processing load.

TRAX/TL

Communication between two transaction processors, each on a separate TRAX system, requires use of the following hardware:

• DMC11-AL plus DMC11-MD — Used for local operation over coaxi-al cable. The TRAX systems may be up to 18,010 ft. (5,487 m) apart. The units include built-in modems.

• DMC11-AR plus DMC11-DA — Used for remote operation over common carrier faciities. Interfaces to EIA/CCITT synchronous modems (Bell Series 200 or equivalent), capable of operation up to 19,200 bps.

Multiple transaction processors within one TRAX system may communicate with one or more transaction processors over one physical link.
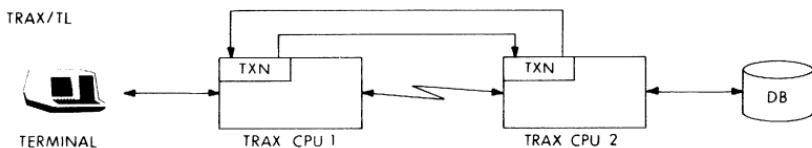
Figure 9-10 TRAX/TL

Figure 9 11 ΓRAX Supported Devices

Max. Memory	Disks		Tapes	Terminals	Line Printers
	PDP-11/34				
256 Kb	RK07 —	28 Mb	TE16 — 1600 bpi, 45 ips	LA36	LP11 —
	RM02 —	67 Mb	TU16 — 1600 bpi, 45 ips	VT52	96 char.,
	RP04 —	88 Mb	TU16 — 1600 bpi, 75 ips	VT62	132 col.
	RP05 —	88 Mb		LA180	
	RP06 —	176 Mb			
	PDP-11/60				
256 Kb	RK07 —	28 Mb	TE16 — 1600 bpi, 45 ips	LA36	LP11 —
	RM02 —	67 Mb	TU16 — 1600 bpi, 45 ips	VT52	96 char.,
	RP04 —	88 Mb	TU16 — 1600 bpi, 75 ips	VT62	132 col.
	RP05 —	88 Mb		LA180	
	RP06 —	176 Mb			
	PDP-11/70				
4 Mb	RM03 —	67 Mb	TE16 — 1600 bpi, 45 ips	LA36	LP11 —
	RP04 —	88 Mb	TU16 — 1600 bpi, 45 ips	VT52	96 char.,
	RP05 —	88 Mb	TU16 — 1600 bpi, 75 ips	VT62	132 col.
	RP06 —	176 Mb		LA180	

TRAX SYSTEM SUMMARY

Is

- High volume transaction processing
- Batch processing
- Protected environment
- Application development tools:
    - Debug utility
    - Terminal screen language
    - Distributed functionality
    - RSX/VAX compatibility
    - Easy systems design

Is not
- Timesharing
- Sensor based
- For smaller CPUs
- Large scale batch (IBM)

Includes Data Management/Utilities
- RMS-11
- DATATRIEVE-11
- SORT-11

Languages
- COBOL
- BASIC-PLUS-2
- MACRO-11

DECNET PHASE II

OVERVIEW

DECnet Phase II is the collective name for the set of software products that extend various PDP-11 operating systems by enabling the user to interconnect these systems with each other to form computer networks. The DECnet Phase II products discussed here include DECnet-11M Version 2, DECnet-11S Version 2, DECnet/E Version 1, and DECnet-RT Version 1.

FEATURE TOPICS

- DECnet-11 Introduction
- DECnet/PDP-11 Systems Summary
- DECnet Phase II and the PDP-11 Products
  - Communication and User/Program Functions (chart)
  - Comparative Analysis DECnet/PDP-11 Operating Systems (chart)

DECNET PHASE II INTRODUCTION

DECnet products create distributed networks from DIGITAL computers and their interconnecting data links by creating a general mechanism for sharing resources and providing interprogram communications within a distributed data processing environment. DECnet implementations adhere to a common network architecture that defines the structure and protocols each must use to communicate through the network.

DECNET PHASE II AND THE PDP-11 PRODUCTS

DECnet products contain several capabilities that can be shared in common with different PDP-11 operating systems. These common capabilities can be described in terms of specific communications and user/program functions as illustrated below.

Communications Functions

Point-to-Point communications describe the ability to communicate between two nodes that are connected over a previously established physical link.
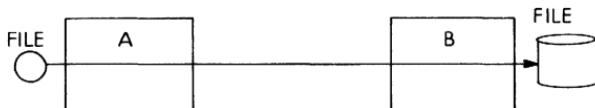
User/Program Levels
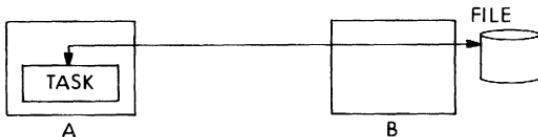
Task-to-Task user/program functions describe the ability of a program running on one node to interact with a program running on another node.

File transfer user/program functions describe the ability to transfer data from a file on one node to a file on another node.

Resouce access user/program functions describe the ability of a program on one node to access resources (such as files), or to request services (such as a load request) physically located on another node.

In addition, DECnet products have a range of capabilities which may operate selectively on PDP-11 operating systems, The table below offers a comparative analysis of DECnet/PDP-11 systems capabilities.

DECNET/PDP-11 SYSTEMS SUMMARY

DESCRIPTION:
DECnet/RT Version 1.0 allows a suitably configured RT-11 system to participate as a Phase II DECnet node in point-to-point computer networks. DECnet/RT offers task-to-task communications, network file transfer and network resource-sharing capabilities, using the DIGITAL Network Architecture (DNA) protocols. DECnet/RT communicates with adjacent nodes over synchronous and asynchronous communication lines, and parallel interfaces. Access to DECnet/RT is supported for RT-11 user programs written in MACRO-11 and FORTRAN.

DECNET/RT FEATURES
- Transmits and maintains data integrity between two adjacent nodes of a network.
- FORTRAN and MACRO-11 tasks can interact with other tasks executing in the DECnet environment; transfer data on a record-by-record basis to remove peripheral devices and files; request the execution of programs in other systems in the network; and cause programs executing on remote systems to be terminated.
- Inter-system file transfer
- Includes software utilities to monitor network activity, provide inter-system operator communications and aid network maintenance.

DECnet/E, Version 1

DESCRIPTION:
DECnet/E allows a suitably configured RSTS/E system to participate as a Phase II DECnet node in point-to-point computer networks. DECnet/E is a Phase II network product and is warranted for use only with Phase II DECnet products supplied by DIGITAL.

DECnet/E offers task-to-task communications and network file transfer capabilities using the DIGITAL Network Architecture protocols.

DECnet/E communicates with adjacent nodes over synchronous communication lines interfaced with DMC11 microprogrammed controllers.

DECNET/E FEATURES

- Transmits and maintains data integrity between two adjacent nodes of a network.
- BASIC and BASIC-PLUS-2 tasks can interact with other tasks executing in the DECnet environment.
- Inter-system file transfer
- Includes software utilities to monitor network activity, provide inter-system operator communications, and aid network maintenance.

DECnet-11M, Version 2.0

DESCRIPTION:

DECnet-11M Version 2.0 allows a suitably configured RSX-11M system to participate as a Phase II DECnet node in point-to-point computer networks. DECnet-11M offers task-to-task communications, network file transfer, and network resource-sharing capabilities, using the DIGITAL Network Architecture (DNA) protocols. DECnet-11M communicates with adajacent nodes over synchronous and asynchronous communication lines, and parallel interfaces. Access to DECnet-11M is supported for RSX-11M user programs written in MACRO-11 and FORTRAN.

DECnet-11M is a Phase II network product and is warranted for use only with Phase II DECnet products supplied by DIGITAL.

DECNET-11M FEATURES

- Transmits and maintains data integrity between two adjacent nodes of a network.
- FORTRAN and MACRO-11 tasks can interact with other tasks executing in the DECnet environment; transfer data on a record-by-record basis to remote peripheral devices and files; request the execution of programs in other systems in the network; and cause programs executing on remote systems to be terminated.
- Inter-system file transfer
- Down-line system loading
- Down-line program and task loading
- Includes software utilities to monitor network activity, provide inter-system operator communications and aid network maintenance.

DECnet-11D, Version 2.0

DESCRIPTION:

DECnet-11D Version 2.0 allows a suitably configured RSX-11D system to participate as a Phase II DECnet node in point-to-point computer networks. DECnet-11D offers task-to-task communications, network file transfer and network resource-sharing capabilities, using the DIGITAL Network Architecture (DNA) protocols. DECnet-11D communicates with adjacent nodes over synchronous and asynchronous communication lines, and parallel interfaces. Access to DECnet-11D is supported for RSX-11D user programs written in MACRO-11 and FORTRAN.

DECNET-11D FEATURES

- Transmits and maintains data integrity between two adjacent nodes of a network.
- FORTRAN and MACRO-11 tasks can interact with other tasks executing in the DECnet environment; transfer data on a record-by-record basis to remote peripheral devices and files; request the execution of programs in other systems in the network; and cause programs executing on remote systems to be terminated.
- Inter-system file transfer
- Down-line system loading
- Down-line program and task loading
- Includes software utilities to monitor network activity, provide inter-system operator communications, and aid network maintenance.

DECnet-11S, Version 2.0

DESCRIPTION:

DECnet-11S Version 2.0 allows a suitably configured RSX-11S system to participate as a Phase II DECnet node in point-to-point computer networks. DECnet-11S offers task-to-task communications, network file transfer and network resource-sharing capabilities, using the DIGITAL Network Architecture (DNA) protocols. DECnet-11S communicates with adjacent nodes over synchronous and asynchronous communication lines, and parallel interfaces. Access to DECnet-11S is supported for RSX-11S user programs written in MACRO-11 and FORTRAN.

DECNET-11S FEATURES

- Transmits and maintains data integrity between two adjacent nodes of a network.

- FORTRAN and MACRO-11 tasks can interact with other tasks executing in the DECnet environment; transfer data on a record-by-record basis to remote peripheral devices and files; request the execution of programs in other systems in the network; and cause programs executing on remote systems to be terminated.

- Inter-system file transfer

- Down-line system loading

- Down-line program and task loading

- Includes software utilities to monitor network activity, provide inter-system operator communications, and aid network maintenance.

DECnet-IAS, Version 2.0

DESCRIPTION:

DECnet-IAS Version 2.0 allows a suitably configured IAS system to participate as a Phase II DECnet node in point-to-point computer networks. DECnet-IAS offers task-to-task communications, network file transfer and network resource-sharing capabilities, using the DIGITAL Network Architecture (DNA) protocols. DECnet-IAS communicates with adjacent nodes over synchronous and asynchronous communication lines and parallel interfaces. Access to DECnet-IAS is supported for IAS user programs written in MACRO-11 and FORTRAN.

DECNET-IAS FEATURES

- Transmits and maintains data integrity between two adjacent nodes of a network.

- FORTRAN and MACRO-11 tasks can interact with other tasks executing in the DECnet environment; transfer data on a record-by-record basis to remote peripheral devices and files; request the execution of programs in other systems in the network; and cause programs executing on remote systems to be terminated.

- Inter-system file transfer

- Down-line system loading

- Down-line program and task loading

- Includes software utilities to monitor network activity, provide inter-system operator communications, and aid network maintenance.

Table 10-2

	DECnet-11M Version 2	DECnet-11S Version 2	DECnet-11D Version 2	DECnet-IAS Version 2	DECnet/E Version 1	DECnet-RT Version 1	DECnet-VAX Version 1
Task-to-Task	YES	YES	YES	YES	YES	YES	YES
Intersystem File Transfer	YES	NO	YES	YES	YES	YES	YES
Command/Batch File Submission	YES[1]	NO	YES[1]	YES[1]	YES	YES	YES[1]
Command/Batch File Execution	YES	NO	YES	YES	YES	NO	YES
Remote File Access	YES	YES[2]	YES	YES	NO	YES	YES
Down-Line System Loading	YES	NO	YES	YES	NO	NO	YES
Down-Line Task Loading	YES	NO	YES	YES	NO	NO	NO
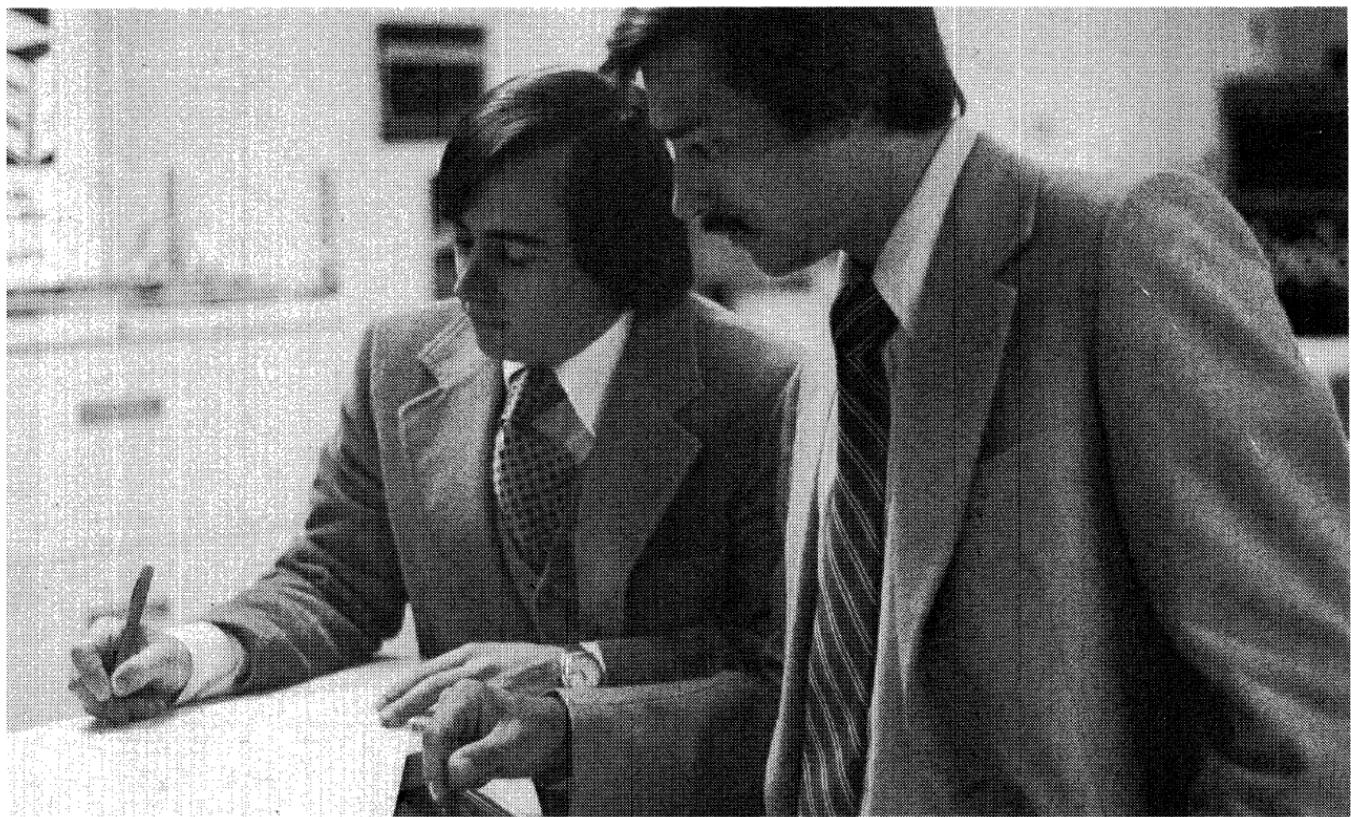
[1] Cannot submit files to DECnet/E systems. Can tell DECnet/E to execute batch files already at the DECnet node.

[2] Offers local users network access to remote file systems. Does not allow users on remote systems to access local files.

DECnet PHASE II

data
managers

SORT-11 (V.2)

OVERVIEW

SORT-11 allows for the reordering of data based upon control or key fields within the input data records. SORT-11, RMS-11K, DATA-TRIEVE-11, and DBMS are the four major Data Managers in PDP-11 software. All of these systems may run on a variety of PDP-11 operating systems as illustrated below.

DATA MANAGEMENT TABLE

RMS-11K	RSX-11M, IAS, TRAX-11, RSTS/E
DATATRIEVE	RSX-11M, IAS, TRAX-11, RSTS/E
DBMS	RSX-11M, IAS, TRAX-22
SORT	RSX-11M, IAS, TRAX-11, RSTS/E

FEATURE TOPICS

- Functions and Features
  - Record Sort (SORTR)
  - Tag Sort (SORTT)
  - Address Routing Sort (SORTA)
  - Index Sort (SORTI)

- DATA files

- Command String and Specification File

- SORT Processing Options

FUNCTIONS AND FEATURES

The SORT utility program allows the user to reorder data from any input file into a new file in a sequence that is based upon control or key fields within the input data records. SORT runs under any operating system that includes RMS (Record Management Services). (See Chapter 12 for information on RMS.) The sorting sequence is determined by control fields, also known as key fields, within the data itself. If the user does not wish to sort the data base, SORT can still be used to extract key information, sort that information, and store the sorted information on a permanent file. Later that file can be used to access the data base in the order of the key information on the sorted file. The contents of the sorted file may be entire records, key fields, or record indices relative to the position of each record within the file (the first record on the data base is record 1, the second, 2, etc.).

SORT provides four sorting techniques:

- **Record Sort (SORTR)** produces a reordered file by using the entire contents of each record as the record key. A record sort can be used on any acceptable input device and can process any valid RMS (Record Management Services) format.

- **Tag Sort (SORTT)** produces a reordered file by sorting only the record keys. SORT then randomly reaccesses the input file to create a resequenced output file according to those record keys. The tag sort method conserves temporary storage, but can only accept input files residing on disk.

- **Address Routing Sort (SORTA)** produces an address file without reordering the input file. The address file, sorted by record keys, can later be used as an index file to read the data base in the desired sequence. Any number of address files may be created for the same data base. A customer master file, for instance, may be referenced by customer number index or sales territory index for different reports. SORTA is the fastest of the four sorting methods.

- **Index Sort (SORTI)** produces an index file containing the key field of each data record and a pointer to its location in the input file. The index file can be used for sequential or direct accessing from a random file.

The SORT utility program may be controlled by a command string and an optional specification file. There is a simple format for each. If the user's SORT application does not require that records be restructured or that only a subset of the input file be sorted, then only a command string is needed to control SORT.

SORT can handle any RMS valid file organization. Different file organizations are distinguished by the ordering of the records they contain and the way they handle the retrieval process.

The order of the records in a sequential file is determined by the order in which the records are read from the file. The first record in the file is the first record read out, regardless of whether the records are written to the file in some sorted order or not.

A relative file consists of record areas that are identified by relative record numbers. The first record area in the file is record number 1, the second is 2, etc., much the same as an apartment house where the first apartment is 1, and so forth. But, as in an apartment house, if the user wants the record that is in the twelfth record area, he must ask for record number 12, even though there may not be records in areas 1 through 11. Relative files can reside only on disk.

An indexed file contains prologue information, one or more indices, and the data records themselves. To retrieve information, the user asks for the proper record by primary or alternate key. The system looks up the key in the appropriate index and retrieves the record using the record pointer associated with the key. Indexed files can reside only on disk.

Table 11-1 shows the devices that can be used to supply data to SORT. Data may be stored in binary, ASCII, or EBCDIC form.

DATA FILES

SORT may accept a file from any one of the peripheral devices available in the system configuration: disk units, magtape units, card readers, paper tape readers, and terminals.

A record is usually divided into several logical areas called data fields. The data in each field may or may not be relevant to SORT. Each field may be interpreted as a record identifier, key data, or general data related to the logical content of the record and not relevant to the sorting process. SORT uses record identifiers to distinguish the various tapes of records in a file. SORT uses the key fields in each record to reorder an input file. Any other data field in a record may be retained in the output file or ignored by SORT.

**Table 11-1 Selecting the Sorting Process and Devices That Best
Suit the Processing Environment**

Sorting Technique	Input File	Output File	Work File
SORTR	Disk	Disk	Disk (3-8 files)
(Record Sort)	Magtape	Magtape*	
	Paper Tape	Paper Tape	
	Cards	Printer	
	Console	Console	
SORTT	Disk	Disk	Disk (3-8 files)
(Tag Sort)	Magtape	Magtape*	
		Printer	
		Console	
		Paper Tape	
SORTA	Disk	Disk	Disk (3-8 files)
(Address Routing Sort)	Magtape		
SORTI	Disk	Disk	Disk (3-8 files)
(Index Sort)	Magtape		

* Provided records are at least 18 bytes long. Magtape must be in ANSI format.

COMMAND STRING AND SPECIFICATION FILE

The user can direct the SORT program by entering a command string. The command string has three functions:
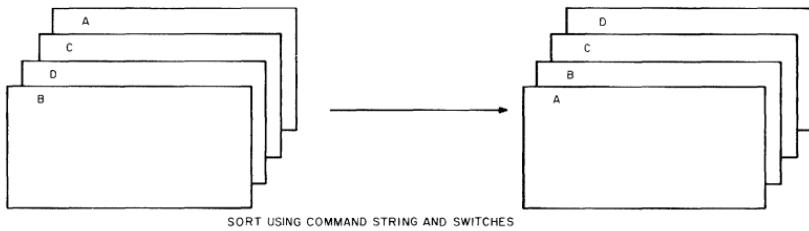
1. To reference devices in the system for each file in the current sort.

2. To specify switches that define file parameters used in the sorting process.

3. To reference a specification file or to specify other switches to control the sort.

Several command string switches define the sorting process parameters. One switch describes record formats and the maximum record

size. Another delimits the internal work files. Others provide detailed file information to RMS.

Normally, the sort must be directed with a specification file. Two additional switches may be used instead of a specification file to control a sort. One switch specifies the sorting process option; the other identifies the key fields. The use of these switches is limited to sorting an input file of uniform format:

1. The key fields must reside in the same location in every record of the input file.

2. The file must contain only the records to be included in the sort. The figure below illustrates a general sort that would require only a command string and switches.

SORT USING COMMAND STRING AND SWITCHES
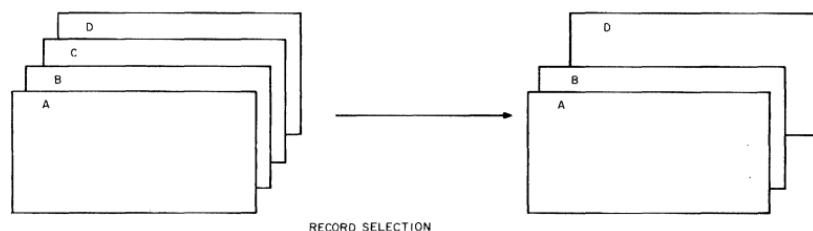
11-1521

The specification file is the supplement to the command string, which provides the basis for controlling and directing the sorting process.

The specification file provides a variety of controlling features. They are listed below:
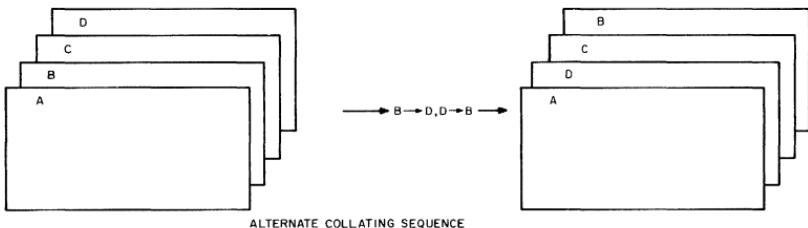
1. Record Selection

RECORD SELECTION

11-1522

The user can include or omit any records from the sorting process. The output file will contain only the specified records.

2. Alternate Collating Sequence
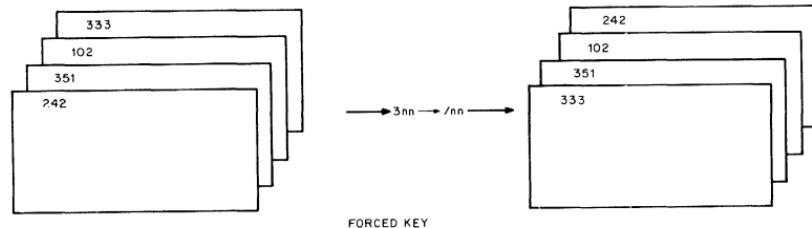
ALTERNATE COLLATING SEQUENCE

11-1523

If necessary, the user can specify an aternate collating sequence. The normal sequence is that implied in ASCII code. One alternate choice is EBCDIC values. The other is an individual alternate collating sequence (ALTSEQ). An ALTSEQ can be used to change the ASCII values of the normal sequence. It applies to all the alphanumeric key data in the records, but only during the actual sorting process. The output record remains unchanged.
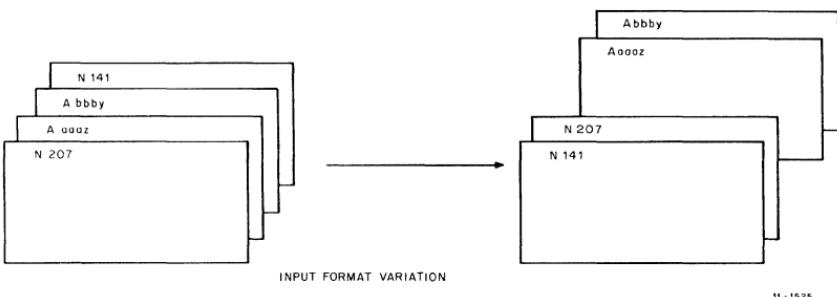
3. Forced Keys

FORCED KEY

11-1524

An ALTSEQ applies to all positions of the key. Forced keys allow the user to specify an alternate sequence for particular positions within the key. An alternate can be specified by substituting a lower-valued character, such as the slash (/) in the example above. Since the slash comes before 0, the 300-series records in the example are brought to the front of the file. Notice that the records so treated are in sequence and in front of the rest of the sorted file. The net effect is that of two sorted files, one behind the other.

4. Input Format Variation

INPUT FORMAT VARIATION
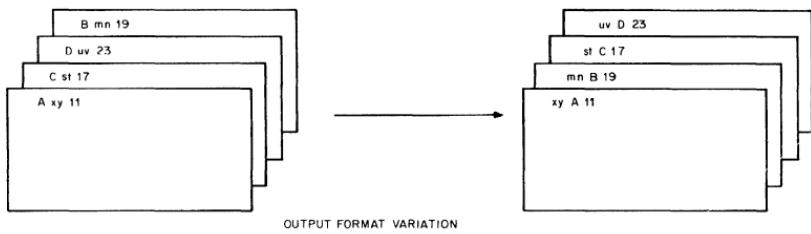
11-1525

If the input file contains records with several different formats, the user can identify those records by tape so that they may be properly handled.

In the example above, A and N are record identifiers.

5. Output Format Variation

OUTPUT FORMAT VARIATION

11-1526

The user can change the format of the data file during the sort, but cannot change the contents of any given data item.

SORT OPERATION

The SORT program consists of two basic parts: a control program and a subroutine package called SORTS. The control program directs the overall processing. SORTS serves as a collection of subroutines that the control program uses during its processing.

There are three phases of operation in the SORT control program. In the first phase, SORT reads the command string, decodes, and stores the switch values and the specification file, if present. Any errors in the command string or specification file are reported at this point.

The second phase begins the pre-sort operation. The control program is called to open and read the input file and establish the keys. The the SORTS subroutine begins the initial sorting process. At this point, the amount of available internal storage space becomes important to the efficiency of the sort. If that space is not sufficient to hold all the records, SORT builds strings of sorted records and transfers them to scratch files on bulk storage devices. In order to merge these files and complete the sort, space for at least three scratch files must be available. The SORT program normally provides for a maximum of eight scratch files. Either a switch in the command string or the amount of available internal work space can reduce the number of scratch files used.

The merge phase rebuilds the intermediate scratch files into a merged file. Another subrouting reads the records in the proper sequence. The records are then written into the output file. If there are no scratch files to merge because main memory was sufficient to hold all the records, the sorted records are written directly into the output file. After the last record is written, the control program cleans up the scratch files and returns to the first phase; SORT is then ready to accept another job.

SORT PROCESSING OPTIONS

Record Sort (SORTR)
The Record Sort (SORTR) outputs all specified record data in a sorted sequence. Each record is kept intact throughout the entire sorting process. Since it moves the whole record, SORTR is relatively slow and may require considerable main memory or external storage work space for large files.

Table 11-2 Sorting Process Options

Type of SORT	Type of File	Record Size and Format	Device	Speed
SORTR (Record Sort)	Input and Output	Any	Any appropriate device including: disk, magtape card reader, console	Slowest

Type of	Type of	Record Size	Device	Speed
SORTT (Tag Sort)	Input	Any	Disk	Slow for large files, large keys
	Output	Any	Any appropriate device (including magtape)	
SORTA (ADDROUT Sort)	Input	Any	Disk or magtape	Fastest
	Output	Fixed, six bytes	Disk	
SORTA (Index Sort)	Input	Any	Disk or magtape	Fast
	Output	Fixed, 6-byte pointer + original key	Disk	

Tag Sort (SORTT)
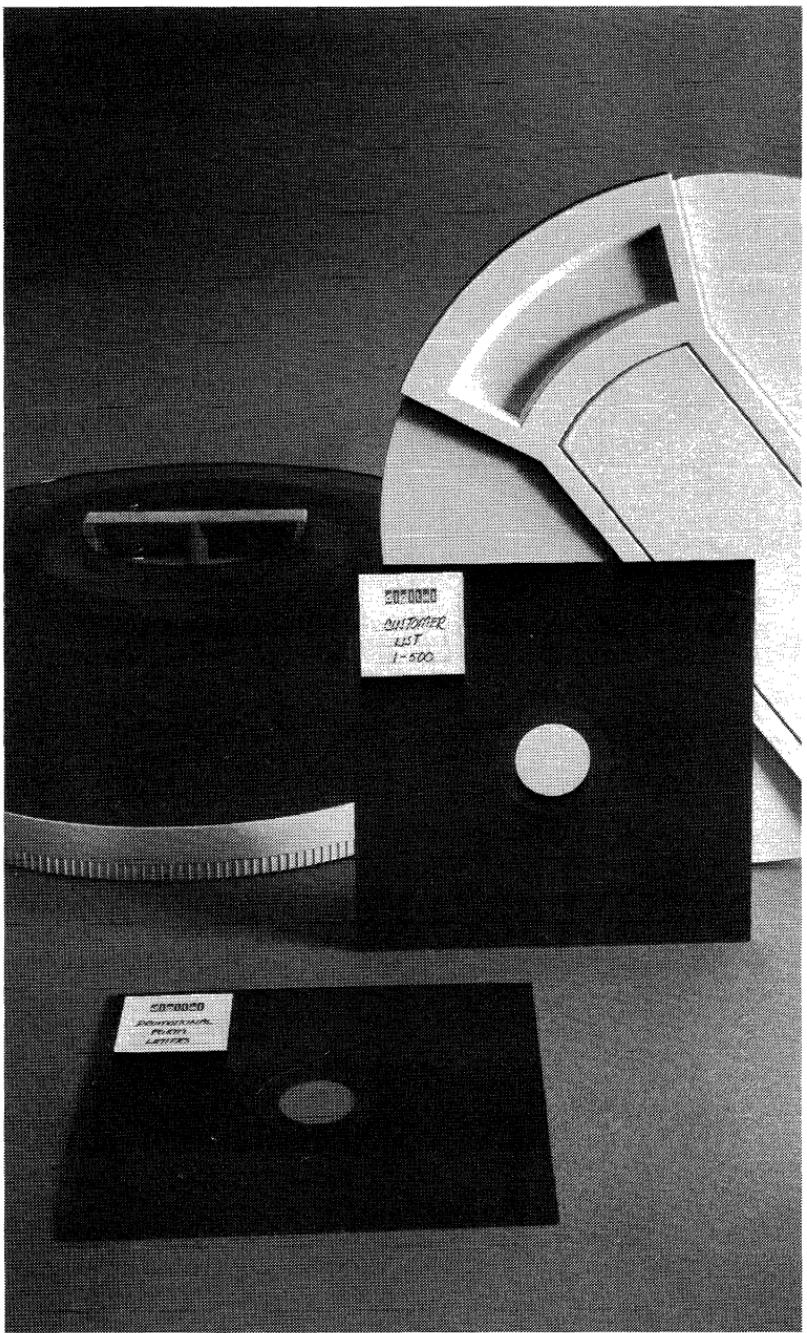
The Tag Sort (SORTT) produces the same kind of output file as SORTR, but it handles only record pointers and key fields. Since SORTT moves a smaller amount of data than SORTR, SORTT usually performs a faster sort than SORTR. The input file must be randomly re-accessed to create the entire output file, which may be lengthy process for large files.

Address Routing Sort (SORTA)

SORTA produces address routing files, which consist of relative record pointers, beginning at 1, in binary words. These files can be used as a special index file to access randomly the data in the original file. It is possible to maintain only one data file, but several different index files as needed. Like SORT, SORTA uses the minimum amount of data necessary in the sorting process. Once the input phase is completed, the input file is not read again. The output data is in a restricted mode. This means that SORTA is the fastest sorting method in the sort package.

Index Sort (SORTI)

SORTI produces an index file consisting of relative record pointers, as in SORTA, and index keys. This makes it slightly slower than SORTA. During processing SORTI handles only the relative record pointers and two forms of the key fields. One form is used for sorting and the other is left as it was in the original data.

RECORD MANAGEMENT SERVICE
RMS-11 (V.1)

OVERVIEW

RMS-11 allows user-written application programs to create, access, and maintain data files efficiently. Its variety of file organizations and access modes gives the user the ability to choose the method best suited to the application. RMS-11 supports sequential, relative, and indexed files, which users can access sequentially, randomly, or by key. The multi-keyed access option provides both generic and approximate key searches. Records can be either fixed or variable length. The system offers record handling capabilities for applications whose size and data structures do not require the full services of DBMS.

FEATURE TOPICS

● Functions and Features

● File Organizations

● RMS File Organizations
  - Sequential
  - Relative
  - Indexed

● RMS Access Modes
  - Sequential
  - Random
  - Record's File Address (RFA)
  - Dynamic Access

● File Attributes
  - Storage Media
  - File Specifications

● RMS Record Formats

● Program Operations on RMS Files

● Run Time Environment

FUNCTIONS AND FEATURES

Record Management Services for the PDP-11 is a set of general purpose file-handling capabilities. In combination with a host operatng system, it provides efficient and flexible facilities for data storage, retrieval, and modification. When writing programs, the user can select processing methods from among RMS file structuring and accessing techniques suited to specific applications. RMS's flexibility is explained in its handling of the following areas:

- File Organizations
- File Attributes
- Program Operations on RMS Files
- Runtime Environment

The manner in which RMS builds a file is called its organization. RMS provides three file organizations:

- sequential
- relative
- indexed

The organization of a file establishes the techniques one can use to retrieve and store data in the file. These techniques are known as access modes. The access modes that RMS supports are:

- Sequential
- Random
- Record's File Address (RFA)

An application program or a RMS utility can be used when creating a RMS file to specify the organization and characteristics of the file. Among the attributes specified are:

- Storage Medium
- File and Protection Specifications
- Record Format and Size
- File Allocation Information

After RMS creates a file according to the specified attributes, application programs can store, retrieve and modify data. These program operations can occur at the logical or physical level.

At the logical level, a RMS file is a collection of individual records. The record is the unit of information to which RMS provides access.

At the physical level, a file is a collection of units called virtual blocks. When bypassing the record passing processing capabilities of RMS, programs access these virtual blocks through a technique known as block I/O.

During runtime, RMS and the host operating system provide an environment for user programs that permits file sharing and reduces the number of buffers required. When a program accesses files at the logical level, RMS additionally supports:

- Multiple Access Streams

- Synchronous or Asynchronous Record Operations

- Move and Locate Record Transfer Modes

FILE ORGANIZATION

A file is a collection of related information. Application requirements establish the nature of this information. For example, a company might maintain personnel information (employee names, addresses, job titles) in one file and product information (part numbers, prices, specifications) in a second, separate, file. Within each of these files, the information is divided into records. In the personnel file, it would be logical for all the information on a single employee to constitute a single record and for the number of records in the file to equal the number of employees. Similarly, each record in the product information file would represent a description of a single product. The number of records in the file reflects the requirements of a particular application, in this case, a central registry of products sold by a company.

Each record in the personnel and product files would be subdivided into discrete pieces of information known as data fields. The user would define the number, location within the record, and logical interpretation of these data fields. Programmers at the company's data processing installation would write applications that interpret a particular data field in records of the personnel file as the name of an employee. They would interpret another data field in records of the product file as a part number. Figure 12-1 illustrates records that might occur in a personnel file and in a product file.

DATA FIELDS:	NAME	ADDRESS	BADGE NO	DEPARTMENT	TITLE
	JONES	MAIN ST, USA	1452	PAYROLL	CLERK

PERSONNEL RECORD

DATA FIELDS:	PART NO.	DESCRIPTION	PRICE	IN STOCK	SPECIFICATION
	219	WIDGET	$1.86	1430	3" × 2" × 1"

PRODUCT RECORD

Figure 12-1 Personnel and Product Records

283

Thus, the user can completely control the grouping of data fields into records and records into files. The relationship among data fields and records is known and is embedded in the logic of the programs. RMS does not require an awareness of logical relationships among information in the files. Rather, RMS processes records as single units of data. Programs either build records and pass them to RMS for storage in a file or issue requests for records while RMS performs the necessary operations to retrieve the records from a file.

The purpose of RMS, then, is to ensure that every record written into a file can subsequently be retrieved and passed to a requesting program as a single logical unit of data. The structure, or organization, of a file establishes the manner in which RMS stores and retrieves records. The way a program requests the storage or retrieval of records is known as the access mode. The access mode that can be used depends on the organization of a file.

RMS FILE ORGANIZATIONS
When creating a file, there is a choice of three file organizations:

- Sequential
- Relative
- Indexed

Sequential File Organization
In sequential file organization (see Figure 12-2), records appear in physical sequence. Each record except the first has another record preceding it, and each record except the last has another record following it. The physical order in which records appear is always identical to the order in which the records were originally written to the file by an application program.

Figure 12-2   Sequential File Organization

Relative File Organization
When relative organization is selected, RMS structures a file as a series of fixed-size record cells. Cell size is based on the size specified as the maximum permitted length for a record in the file. RMS considers these cells as successively numbered from 1 (the first) to n

(the last). A cell's number represents its location relative to the beginning of the file.

Each cell in a relative file can contain a single record. There is no requirement, however, that every cell contain a record. Empty cells can be interspersed among cells containing records.

Since cell numbers in a relative file are unique, they can be used to identify both a cell and the record (if any) occupying that cell. Thus, record number 1 occupies the first cell in the file, record number 17 occupies the seventeenth cell, and so forth. When a cell number is used to identify a record, it is also known as a relative record number. Figure 12-3 depicts the structure of a relatively organized file.
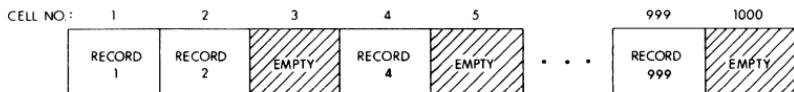
Figure 12-3 Relative File Organization

Indexed File Organization

Unlike the physical ordering of records in a sequential file or the relative positioning of records in a relative file, the location of records in indexed file organization is transparent to the program. RMS completely controls the placement of records in an indexed file. The presence of keys in the records of the file governs this placement.

A key is a character string present in every record of an indexed file. The location and length of this character string are identical in all records. When creating an indexed file, the user decides which character string in the file's records is to be a key. Selecting such a character string indicates to RMS that the contents (i.e., key value) of that string in any particular record written to the file can be used by a program to identify that record for subsequent retrieval.

At least one key, the primary key, must be defined for an indexed file. Optionally, additional keys (i.e., alternate keys) can be defined. Each alternate key represents an additional character string in records of the file. The key value in any one of these additional strings can also be used as a means of identifying the record for retrieval.

As programs write records into an indexed file, RMS locates the values contained in the primary and alternate keys. From the values in keys within records, RMS builds a tree-structured table known as an index. An index consists of a series of entries. Each entry contains a

key value copied from a record that a program wrote into the file. With each key value is a pointer to the location in the file of the record from which the value was copied. RMS builds and maintains a separate index for each key defined for the file. Each index is stored in the file. Thus, every indexed file contains at least one index, the primary key index. When alternate keys are defined, RMS builds and stores an additional index for each alternate key. Figure 12-4 shows the general structure of an indexed file that has been defined with only a single key. Figure 12-5 depicts an indexed file defined with two keys: a primary key and one alternate key.
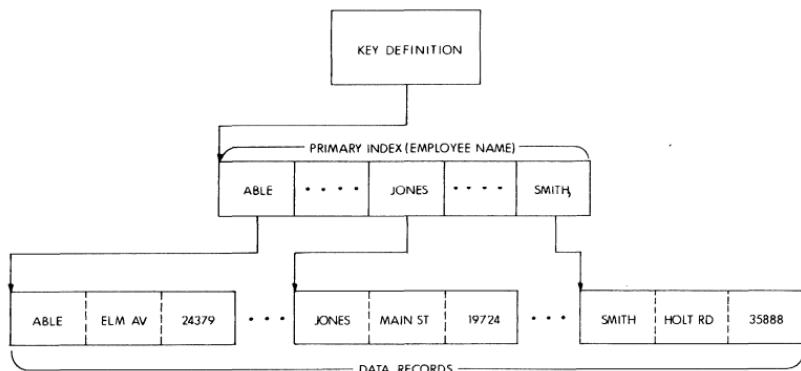
Figure 12-4   Single Key Indexed File Organization

Table 12-1
Permissable Combinations of
Access Modes and File Organizations

File Organization	Access Mode			
	Sequential	Random		RFA
		Record #	Key Value	
Sequential	Yes	No	No	Yes*
Relative	Yes	Yes	No	Yes
Indexed	Yes	No	Yes	Yes

*Disk files only.
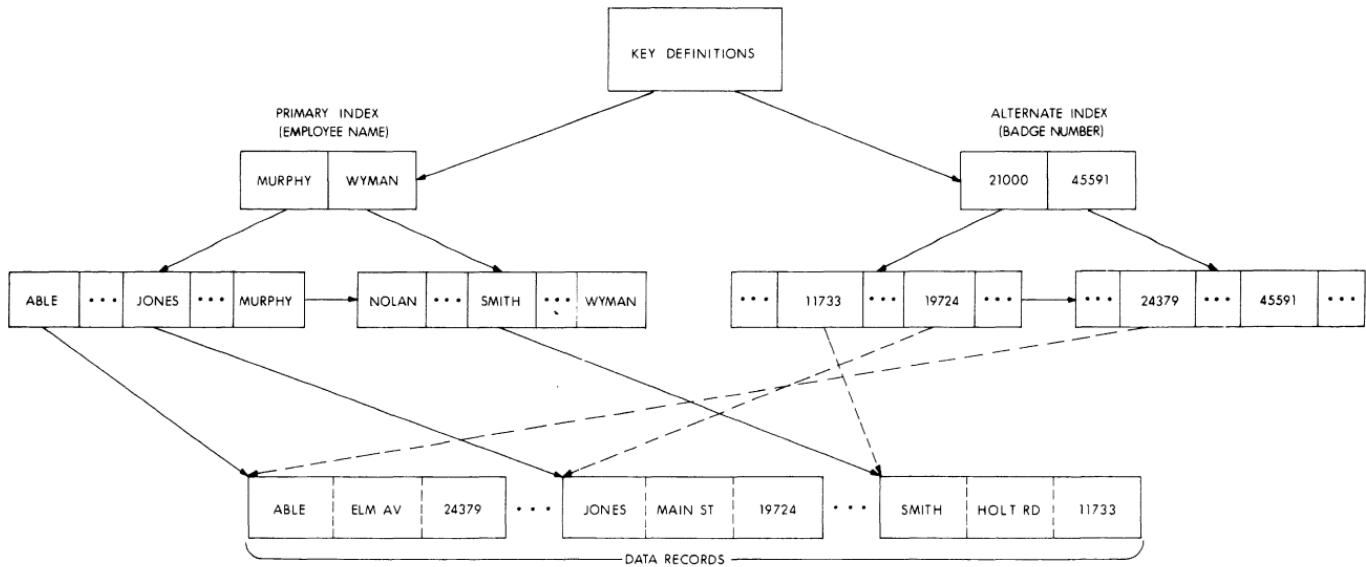
Figure 12-5 Multi-key Indexed File Organization

RMS ACCESS MODES

The various methods of retrieving and storing records in a file are called access modes. A different access mode can be used to process records within the file each time it is opened. Additionally, a program can change access mode during the processing of a file.

RMS provides three record access modes:

- Sequential
- Random
- Record's File Address (RFA)

RMS permits only certain combinations of file organization and access mode. Table 12-1 lists these combinations.

The following subsections describe RMS access modes and the capablity of changing access mode during program execution.

Sequential Access Mode

Sequential access mode can be used to access all RMS files. Sequential accesss means that records are retrieved or written in a particular sequence. The organization of the file establishes this sequence.

Sequential Access to Sequential Files — In a sequentially organized file, physical arrangement establishes the order in which records are retrieved when using sequential access mode. To read a particular record in a file, say the fifteenth record, a program must open the file and access the first fourteen records before accessing the desired record. Thus each record in a sequential file can be retrieved only by first accessing all records that physically precede it. Similarly, once a program has retrieved the fifteenth record, it can read all the remaining records (from the sixteenth on) in physical sequence. It cannot, however, read any preceding record without closing and reopening the file and beginning again with the first record.

When writing new records to a sequential file in sequential access mode, a program must first request that RMS position the file immediately following the last record. Then each sequential write operation the program issues causes a record to be written following the previous record.

Sequential Access to Relative Files — During the sequential access of records in the relative file organization, the contents of the record cells in the file establish the order in which a program processes records. RMS recognizes whether successively numbered record cells are empty or contain records.

When a program issues read requests in sequential access mode for a relative file, RMS ignores empty record cells and searches successive cells for the first one containing a record. If, for example, a relative file contains records only in cells 3, 13, and 47, successive sequential read requests cause RMS to return relative record number 3, then relative record number 13, and finally relative record number 47.

When a program adds new records in sequential access mode to a relative file, the order in which RMS writes the records depends on ascending relative cell numbers. Each write request causes RMS to place a record in the cell whose relative number is one higher than the relative number of the previous request, as long as that cell does not already contain a record. If the cell already contains a record, RMS rejects the write operation. Thus, RMS allows a program to write new records only into empty cells in the file.

Sequential Access to Indexed Files — In an indexed file, the presence of one or more indices permits RMS to determine the order in which to process records in sequential access mode. The entries in an index are arranged in ascending order by key values. Thus, an index represents a logical ordering of the records in the file. If more than one key is defined for the file, each separate index associated with a key represents a different logical ordering of the records in the file. A program, then, can use the sequential access mode to retrieve records in the order represented by any index.

When reading records in sequential access mode from an indexed file, a program initially specifies a key (e.g., primary key, first alternate key, second alternate key, etc.) to RMS. Thereafter, RMS uses the index associated with that specfied key to retrieve records in the sequence represented by the entries in the index. Each successive record RMS returns in response to a programmed read request contains a value in the specified key field that is equal to or greater than that of the previous record returned.

In contrast to a sequential read request, sequential write requests to an indexed file do not require the initial key specification. Rather, RMS uses the stored definition of the primary key field to locate the primary key value in each record to be written to the file. When a program issues a series of sequential write requests, RMS verifies that each successive record contains a key value in the primary key field that is equal to or greater than that of the preceding record.

Random Access Mode

In random access mode, the program, rather than the organization of the file, establishes the order in which records are processed. Each

289

program request for access to a record operates independently of the previous record accessed. Associated with each request in random mode is an identification of the particular record of interest. Successive requests in random mode can identify and access records anywhere in the file. Random access mode cannot be used with sequentially organized files. Both the relative and indexed file organizations, however, permit random access to records. The subsections that follow describe the use of random access with these organizations. Each organization provides a distinct way programs can identify records for access.

Random Access to Relative Files — Programs can read or write records in a relative file by specifying relative record number. RMS interprets each number as the corresponding cell in the file. A program can read records at random by successively requesting, for example, record number 47, record number 11, record number 31, and so forth. If no record exists in a specified cell, RMS returns a nonexistence indicator to the requesting program. Similarly, a program can store records in a relative file by identifying the cell in the file that a record is to occupy. If a program attempts to write a new record in a cell already containing a record, RMS returns a record-already-exists indicator to the program.

Random Access to Indexed Files — The indexed file organization also permits random access of records. However, for indexed files, a key value rather than a relative record number identifies the record.

Each program read request in random access mode specifies a key value and the index (e.g., primary index, first alternate index, second alternate index, etc.) that RMS must search. When RMS finds the key value in the specified index, it reads the record that the index entry points to and passes the record to the user program.

In contrast to read requests, which require a program-specified key value, program requests to write records randomly in an indexed file do not require the separate specification of a key value. All key values (primary and, if any, alternate key values) are in the record itself. When an indexed file is opened, RMS retrieves all definitions stored in the file. Thus, RMS knows the location and length of each key field in a record. Before writing a record into the file, RMS examines the values contained in the key fields and creates new entries in the indices. In this way RMS ensures that the record can be retrieved by any of its key values. Thus, the process by which RMS adds new records to the file is precisely the process it uses to construct the original index or indices.

Record's File Address (RFA) Access Mode

Record's file address (RFA) access mode can be used with any file organization as long as the file resides on a disk device. This access mode is further limited to retrieval operations only. Like random access mode, however, RFA access allows a specific record to be identified for retrieval.

As the term record's file address indicates, every record within a file has a unique address. The actual format of this address depends on the organization of the file. In all instances, however, only RMS can interpret this format.

The most important feature of RFA access is that the address (RFA) of any record remains constant while the record exists in the file. After every successful read or write operation, RMS returns the RFA of the subject record to the program. The program can then save this RFA to use again to retrieve the same record. It is not required that this RFA be used only during the current execution of the program. RFAs can be saved and used at any subsequent time.

Dynamic Access

Dynamic access is not strictly an access mode. Rather, it is the capability to switch from one access mode to another while processing a file. There is no limitation on the number of times such switching can occur. The only limitation is that the file organization (or, in the case of RFA access, the device containing the file) must support the access mode selected.

As an example, dynamic access can be used effectively immediately following a random or RFA access mode operation. When a program accesses a record in one of these modes, RMS establishes a new current position in the file. Programs can then switch to sequential access mode. By using the randomly accessed record (rather than the beginning of the file) as the starting point, programs can retrieve succeeding records in the sequence established by the file's organization.

FILE ATTRIBUTES

The logical and physical characteristics of a RMS file are known as its attributes. These characteristics are defined by the source language statements of an application program or by the RMS utility program DEFINE. RMS uses this information about the attributes to structure a file on the storage medium.

The most important attribute of any RMS file is its organization. A file for use in a particular application can be tailored by making the proper selection of this and other attributes. In addition to file organization, the user can choose from among the following attributes:

- Storage medium on which the file resides
- File and protection specification of the file
- Format and size of records
- Size of the file
- Size of a particular storage structure, known as the bucket, within relative and indexed files
- Definition of keys for indexed files

Storage Media
Selection of a storage medium on which RMS builds a file is related to the organization of the file. Permanent sequential files can be created on disk devices or ANSI magnetic tape volumes. Transient files can be written on devices such as line printers and terminals. Unlike sequential files, relative and indexed files can reside only on disk devices.

File Specifications
The name assigned to a new file enables RMS to find the file on the storage medium. The conventions for file specifications of the host operating system are followed when naming a file.

RMS allows for the assignment of a protection specification to a file at the time it is created. The format of this specification is the format used by the host operating system.

When a file is created, the user must provide the format and maximum size specifications for the records the file will contain. The specified format establishes how each record appears physically in the file on a storage medium. The size specification allows RMS to verify that records written into the file do not exceed the length specified when the file was created.

RMS Record Formats
- Fixed
- Variable
- Variable-with-fixed-control (VFC)
- Stream

Like the selection of a storage medium, the choice of a format for the records of a file depends on a file's organization. Table 11-? shows the allowed combinations of record format and file organization.

Fixed Length Record Format — The term fixed length record format refers to records of a file that are all equal in size. Each record occupies an identical amount of space in the file.

Variable Length Record Format — In variable length record format, records in a file can be either equal or unequal in length. To allow retrieval of variable length records from a file, RMS prefixes a count field to each record it writes. The count field describes the length (in bytes) of the record. RMS removes this count field before it passes a record to the program.

RMS produces two types of count fields, depending on the storage medium on which the file resides:

- Variable length records in files on disk devices have a 1-word (2-byte) binary count field preceding the data field portion of each record. The specified size excludes the count field.

- Variable length records on ANSI magnetic tapes have 4-character decimal count fields preceding the data portion of each record. The specified size includes the count field. In the context of ANSI tapes, this record format is known as D format.

Variable-with-Fixed-Control Record Format — Variable-with-fixed-control (VFC) records consist of two distinct parts, the fixed control area and the user data record. The size of the fixed control area is identical for all records of the file. The contents of each fixed control area are identical for all records of the file. The contents of each fixed control area are completely under the control of the program and can be used for any purpose. As an example, fixed control areas can be used to store the identifier (e.g., relative record number or RFA) of related records.

The second part of a VFC record is similar to a variable length record. It is a user data record, variable in length and composed of individual data fields.

The two parts of a VFC record correspond to the way a program writes and reads such records. Prior to an output operation, a program builds a VFC record in two locations. It builds the fixed control area in a location separate from the user data part of the record. When writing the record to the file, RMS fetches both the fixed control area and the user data part of the record from their respective program locations. RMS then prefixes the user data part of the record with the fixed control area, prefixes the result with a count field that describes the total size of both parts, and writes the record to the file.

On input operations, RMS reverses the preceding procedure. It uses the count field to locate the entire VFC record in the file. RMS removes

this count field. Then it removes the fixed control area from the record and stores it in one program location while storing the remaining part in a second location.

Stream Format Records — Records in stream format can vary in size. However, no count field precedes each record. Instead, RMS considers the entire file a stream of contiguous ASCII characters. Each record in the file is delimited by one of the following:

- Form feed (FF)
- Vertical tab (VT)
- Line feed (LF)
- Carriage return immediately followed by line feed (CR-LF)

Stream format records are supported for file interchange with non-RMS-application programs. Since this format is highly inefficient, it should be used only when such interchange is a concern.

On output operations, RMS examines the last character of the record constructed by a program. If this character is an LF, VT, or FF, RMS leaves the record unaltered and writes it to the file. If the last character is not LF, VT, or FF, RMS appends a carriage return (CR) character followed by a line feed (LF) character to the record before writing it to the file.

On input operations, RMS scans the stream of ASCII characters, removing null characters and searching for the first occurrence of an FF, VT, LF, or CR-LF combination. If the character that terminated the scan is an FF, VT, or LF (not preceded by CR), RMS passes the entire string, including the terminating character, to the program. If, however, the scan encounters a CR-LF combination, RMS removes these two characters and passes the preceding string as a record to the program. Each successive input operation causes the scan to resume at the character following the last FF, VT, LF, or CR-LF combination encountered.

Size of Records

The user must provide RMS with record size information along with the selected record format. RMS use of this information depends on the record format chosen.

When fixed format records are chosen, the actual size of each record in the file must be indicated. This size specification becomes part of the information stored and maintained by RMS for the file. Thereafter, if a program attempts to write a record whose length differs from this specified size, RMS will reject the operation.

When creating a file with variable length format records, the user can specify a maximum record size greater than zero or, for sequential and indexed files, a maximum record size equal to zero. If the specified size is greater than zero, RMS interprets the value as the size of the largest record that can be written into the file.

VFC format records require two size specifications. The first size specification identifies the length of the fixed control area of all records in the file. The second size specification represents the maximum length of the data portion of the VFC records. RMS handles this second size specification in a manner similar to its handling of the size specification for variable format records.

For stream format records, RMS permits the user to specify the same record size information as for variable format records. That is, a non-zero value represents the maximum permitted size of any record written in the file while a zero value suppresses RMS size checking.

Size of RMS Files
The size of an RMS file is expressed as an integral number of virtual blocks. Virtual blocks are physical storage structures. That is, each virtual block in a file is a unit of data whose size depends on the physical medium on which the file resides. For example, the size of virtual blocks in files on disk devices is 512 bytes.

The operating system assigns ascending numbers to a file's virtual blocks. This numbering scheme allows a file to appear as a series of adjacent virtual blocks. In reality, however, the successive numbering of virtual blocks and the physical placement of these blocks on a storage medium need not correspond.

On magnetic tapes, successively numbered virtual blocks actually occupy successive physical locations. Virtual blocks from one file are never intermixed with virtual blocks from another. On disk devices, however, the situation can be quite different. Files on disk can reside in one or more discrete areas known as extents (see Figure 12-1).

The virtual blocks of a file contain the records that programs write into the file. Depending on the size of records, a virtual block can contain one record, more than one record, or a portion of a record.

When creating an RMS file, the user can specify an initial allocation size. If no file size information is given, RMS allocates the minimum amount of storage needed to contain the defined attributes of the file.

Buckets In Relative and Indexed Files
RMS uses a storage structure known as a bucket for building and maintaining relative and indexed files. Unlike a virtual block, a bucket

can never contain a portion of a record. That is, RMS does not permit records to span bucket boundaries.

The size of buckets in a file is defined at the time the files are created. Buckets can consist of from 1 to 32 virtual blocks. The maximum bucket size on the RSTS/E operating system is 15 virtual blocks. When selecting a bucket size, considerations are: file organization, record format, record size, and the internal information RMS maintains in each bucket. Within these constraints, a large bucket size will serve to increase sequential mode processing of a file, since fewer actual I/O transfers are required to access records. Minimizing bucket size, on the other hand, means that less I/O buffer space is required to support file processing.

Key Definitions For Indexed Files

To define a key for an indexed file, the position and length of character data in the records of the file must be specified. At least one key, the primary key, must be defined for an indexed file. Additionally, up to 254 alternate keys can be defined. Each primary and alternate key represents from 1 to 255 characters in each record of the file.

When identifying the position and the length of keys to RMS, simple or segmented keys can be defined. A simple key is a single, contiguous string of characters in the record; in other words, a single data field. A segmented key, however, can consist of from two to eight data fields within records. These data fields need not be contiguous. When processing records that contain segmented keys, RMS treats the separate data fields (segments) as a logically contiguous character string.

The environment within which a program processes RMS files at runtime consists of two levels, the file processing level and record processing level.

When defining keys at file creation time, two characteristics for each key can be specified:

1. Duplicate key values are allowed.

2. Key value can change.

When it is specified that duplicate key values are allowed, the user indicates that more than one record in the file can have the same value in a given key. Such records, therefore, have the same record identifier. The capability to allow duplicate key values further distinguishes indexed files from relative files. In relative files, the record identifier, representing a relative record number, is always unique.

The personnel file can serve as an example of the use of duplicate keys. At file creation time, the creator of the file could define the

department name field as an alternate key. As programs wrote records into the file, the alternate index for the department name key field would contain multiple entries for each key·value (e.g., PAYROLL, SALES, ADMINISTRATION) since departments are composed of more than one employee. When such duplication occurs, RMS stores the records so that they can be retrieved in first-in/first-out (FIFO) order.

Using the preceding personnel file, an application could be written to list the names of employees in any particular department. A single execution of the application could list the names of all employees working, for example, in the department called SALES. By randomly accessing the file by alternate key and the key value SALES, the application would obtain the first record written into the file containing this value. Then, the application could switch to sequential access and successively obtain records with the same value, SALES, in the alternate key field. Part of the logic of the application would be to determine the point at which a sequentially accessed record no longer contained the value SALES in the alternate key field. The program could then switch back to random access mode and access the first record containing a different value (e.g., PAYROLL) in the department name key field.

The second key characteristic (key value can change) indicates that records can be read and then written back into the file with a modified value in the key. When such modification occurs, the appropriate index is automatically updated to reflect the new key value. This characteristic can be specified only for alternate keys. Further, when specifying this characteristic, the user must also specify that the duplicate key values are allowed.

If the sample personnel file was created with the department name field as an alternate key, the creator of the file would need to specify that key values can change. This specification would allow a program to access a record in the file and change the contents of a department name data field to reflect the transfer of an employee from one department to another.

The user can also declare the converse of either of these two key characteristics. That is, the user can specify for a given key that duplicate key values are not allowed or that key values cannot change. When duplicate key values are not allowed, RMS rejects any program request to write a record containing a value in the key that is already present in another record. Similarly, when the key value cannot change, RMS does not allow a program to write a record back into the file with a modified value in the key.

PROGRAM OPERATIONS ON RMS FILES

After RMS has created a file according to the user's description of file characteristics, a program can access the file and store and retrieve data. When a program accesses the file as a logical structure (i.e., a sequential, relative, or indexed file), it uses access modes to perform record operations that add, retrieve, update, and delete records. The organization of the file determines the types of record operations permitted. If the record accessing capabilities of RMS are not utilized, programs can access the file as a physical structure. As a physical structure, RMS considers the file simply as an array of virtual blocks. To process a file at the physical level, programs use a type of access known as block I/O.

Record Operations of RMS Files

The organization of a file, defined when the file is created, determines the types of operations that the program can perform on records. Depending on file organization, RMS permits a program to perform the following record operations:

- Read a record. RMS returns an existing record within the file to the program.

- Write a record. RMS adds a new record that the program constructs to the file. The new record cannot replace an already existing record.

- Find a record. RMS locates an existing record in the file. It does not return the record to the program, but establishes a new current position in the file.

- Delete a record. The program modifies the contents of a record read from the file. RMS writes the modified record into the file, replacing the old record.

Table 12-1 shows the combinations of record operations and file organizations that RMS permits. The subsections that follow discuss record operations in the context of each file organization.

Sequential File Organization Record Operations

In sequential file organization, a program can read existing records from the file using sequential or RFA access modes. New records can be added only to the end of the file and only through the use of sequential access mode. The find operation is supported in both sequential and RFA access mode. In sequential access mode the program can use a find operation to skip records. In RFA access mode, the program can use the find operation to establish a random starting point in the file for sequential read operations. The sequential file

organization does not support the delete operation, since the structure of the file requires that records be adjacent in and across virtual blocks. A program can, however, update existing records in disk files as long as the modification of a record does not alter its size.

Table 12-2
Record Formats and File Organizations

File Organization	Record Format			
	Fixed	Variable	VFC	Stream
Sequential	Yes	Yes	Yes	disk only
Relative	Yes	Yes	Yes	No
Indexed	Yes	Yes	No	No

Relative File Organization Record Operations
Relative file organization permits programs greater flexibility in performing record operations than sequential organization does. A program can read existing records from the file using sequential, random, or RFA access mode. New records can be sequentially or randomly written as long as the intended record cell does not already contain a record. Similarly, any access mode can be used to perform a find operation. After a record has been found or read, RMS permits the delete operation. Once a record has been deleted, the record cell is available for a new record. A program can also update records in the file. If the format of the records is variable, update operations can modify record length up to the maximum size specified when the file was created.

Indexed File Organization Record Operations
Indexed file organization provides the greatest flexibility in performing record operations. A program can read existing records from the file in sequential, RFA, or random access mode. When reading records in random access mode, the program can choose one of four types of matches that RMS must perform using the program-provided key value. The four types of matches are:

* Exact key match

* Approximate key match

* Generic key match

* Approximate and generic key match

299

Exact key match requires that the contents of the key in the record retrieved precisely match the key value specified in the program read operation.

The approximate match facility allows the program to select either of the following relationships between the key of the record retrieved and the key value specified by the program:

- Equal to or greater than
- Greater than

The advantage of this kind of match is that if the requested key value does not exist in any record of the file, RMS returns the record that contains the next higher key value. This allows the program to retrieve records without knowing an exact key value.

Generic key match means that the program need specify only an initial portion of the key value. RMS returns to the program the first occurrence of a record whose key contains a value beginning with those characters. This capability is useful in applications where a series of records must be retrieved according to the contents of only a part of the key field. In an indexed inventory file, for example, a company might designate its part numbers in such a way that the first three digits represent the vendor from whom the part is purchased. In order to retrieve the record associated with a particular part, the program would normally supply the entire part number. Generic selection permits the retrieval of the first record representing parts purchased from a specific vendor.

The final type of key match combines both generic and approximate facilities. The program specifies only an initial portion of the key value, as with generic match. Additionally, a program specifies that the key data field of the record retrieved must be either:

- Equal to or greater than the program-supplied value
- Greater than the program-supplied value

In addition to versatile read operations, RMS allows any number of new records to be written into an indexed file. It rejects a write operation only if the value contained in a key of the record violated a user-defined key characteristic (e.g., duplicate key values not permitted).

The find operation, similar to the read operation, can be performed in sequential, RFA, or random access mode. When finding records in random access mode, the program can specify any one of the four types of key matches provided for read operations.

In addition to read, write, and find operations, the program can delete any record in an indexed file and update any record. The only restriction RMS applies during an update operation is that the contents of the modified record must not violate any user-defined key characteristic (e.g., key values cannot change and duplicate key values are not permitted).

Table 12-3
Record Operations and File Organizations

File Organization	Record Operation				
	Read	Write	Find	Delete	Update
Sequential	Yes	Yes	Yes	No	Yes*
Relative	Yes	Yes	Yes	Yes	Yes
Indexed	Yes	Yes	Yes	Yes	Yes

*Disk files only.

Block I/O
Block I/O allows a program to bypass the record processing capabilities of RMS entirely. Rather than performing record operations through the use of supported access modes, a program can process a file as a physical structure consisting solely of virtual blocks.

Using block I/O, a program reads or writes multiple virtual blocks by identifying a starting virtual block number in the file. Regardless of the organization of the file, RMS accesses the identified block or blocks on behalf of the program.

Since RMS files, particularly relative and indexed files, contain internal information meaningful only to RMS itself, DIGITAL does not recommend that a file be modified by using block I/O. The presence of the block I/O facility, however, does permit user-created file structures. The resultant structures must be maintained using specialized programs. The structures cannot be accessed using RMS record access mode and record operations.

RMS RUNTIME ENVIRONMENT
The environment within which a program processes RMS files at runtime consists of two levels, the file processing level and record processing level.

301

At the file processing level, RMS and the host operating system provide an environment that permits concurrently executing programs to share access to the same file. RMS ascertains the amount of sharing permissible from information provided by the programs themselves. Additionally, at the file processing level, RMS provides facilities that allow programs to minimize buffer space requirements for file processing.

At the record processing level, RMS allows programs to access records in a file through one or more record access streams. Each record access stream repesents an independent and simultaneously active series of record operations directed toward the file. Within each stream, programs can perform record operations synchronously or asynchronously. That is, RMS allows programs to choose between receiving control only after a record operation request has been satisfied (synchronous operation) or receiving control before the request has been satisfied (asynchronous operation).

For both synchronous and asynchronous record operations, RMS provides two record transfer modes, move mode and locate mode. Move mode causes RMS to copy a record from an I/O buffer into a program-provided location. Locate mode allows programs to address records directly in an I/O buffer.

File Processing Environment
RMS provides two major facilities at the file processing level, file sharing and buffer handling.

File Sharing
Timely access to critical files requires that more than one concurrently executing program be allowed to process the same file at the same time. Therefore, RMS allows executing programs to share files rather than requiring them to process files serially. The manner in which a file can be shared depends on the organization of the file. Program-provided information further establishes the degree of sharing of a particular file. RMS coordinates the sharing of a relative or indexed file through a bucket locking mechanism. The following paragraphs describe:

- File organization and sharing
- Program sharing information
- Bucket locking

File Organization and Sharing — with the exception of magnetic tape files, which cannot be shared, every RMS file can be shared by any number of programs that are reading, but not writing, the file. Sequential files on disk can be accessed by a single writer or shared by

multiple readers. Relative and indexed files, however, can be shared by multiple readers and multiple writers. A program can read or write records in a relative or indexed file while other programs are similarly reading or writing records in the file. Thus, the information in such files can be changing while programs are accessing them.

Program Sharing — A file's organization establishes whether it can be shared for reading with a single writer or for multiple readers and writers. A program specifies whether such sharing actually occurs at runtime. The user controls the sharing of a file through information the program provides RMS when it opens the file. First, a program must declare what operations (e.g., read, write, delete, update) it intends to perform on the file. Second, a program must specify whether other programs can read the file or both read and write the file concurrently with one program.

The combination of these two types of information allows RMS to determine if multiple user programs can access a file at the same time. Whenever a program's sharing information is compatible with the corresponding information another program provides, both programs can access the file concurrently.

Bucket Locking — RMS uses a bucket locking facility to control operations to a relative or indexed file that is being accessed by one or more writers. The purpose of this facility is to ensure that a program can add, delete, or modify a record in a file without another program's simultaneously accessing the same record.

When a program opens an indexed or relative file with the declared intention of writing or updating records, RMS locks any bucket accessed by the program. This locking prevents another program from accessing any record in the bucket until the program releases it. The lock remains in effect until the program accesses another bucket. RMS then unlocks the first bucket and locks the second. The first bucket is then available for access by another concurrently executing program.

Buffer Handling

To a program, record processing under RMS appears as the movement of records directly between a file and the program itself. Transparently to the program, however, RMS reads or writes virtual blocks or buckets of a file into or from internal memory areas known as I/O buffers. Records within these buffers are then made available to the program.

The storage structures transferred between a file and I/O buffers depend on the organization of the file. When a program processes sequential files, RMS reads and writes virtual blocks. For relative and

indexed files, RMS reads and writes buckets. Thus, the storage element RMS uses to structure the file is the unit of transfer between the file and memory when RMS accesses the file in response to one program's record operation request.

In addition to buffers that contain virtual blocks or buckets, RMS requires a set of internal control structures to support file processing. The combination of these buffers and control structures is known as the space pool. RMS maintains a separate space pool for each executing program. Rather than allocating space solely on the basis of the total number of files processed, RMS provides facilities to ensure that a space pool is large enough to accommodate only the requirements of the largest number of files that can be open simultaneously. Using these facilities, a program provides information that allows RMS to calculate the minimum size requirements of the space pool.

In choosing size requirements for the I/O buffer portion of the space pool, there are two options:

- A completely centralized space pool
- Private I/O buffers for one or more files

In a completely centralized space pool, all I/O buffers, as well as the internal control structures required for file processing, are inaccessible to the program. RMS totally manages the space within the pool and allocates portions, as needed, as buffer space and control structures for open files.

Private I/O buffers allow a program some measure of control over I/O buffer space. Private I/O buffers can be allocated on a per-file basis by explicitly specifying the address and total size of the buffers to be used for a particular file. While the file is open, RMS manages this buffer space and the program must not access it. However, when the file is closed, the private I/O buffer space is available for use by the program.

The major advantage of private I/O buffers is that they avoid fragmenting a completely centralized space pool. That is, since particular files have varying buffer requirements based on their organization, a centralized space pool could have sufficient space available for the opening of an additional file, but the space could be noncontiguous. When such a situation arises, a program cannot open the desired file. Such fragmentation cannot occur in a private I/O buffer pool since there is no mixture of differing space requirements.

Record Processing Environment

After opening a file, a program can access records in the file through

the RMS record processing environment. This environment provides three facilities:

- Record access streams
- Synchronous or asynchronous record operations
- Record transfer modes

Record Access Streams

In the record processing environment, a program accesses records in a file through a record access stream. A record access stream is a serial sequence of record operation requests. For example, a program can issue a read request for a particular record, receive the record from RMS, modify the contents of the record, and then issue an update request that causes RMS to write the record back into the file. The sequence of read and update record operation requests can then be performed for a different record, or other record operations can be performed, again in a serial fashion. Thus, within a record access stream, there is at most one record being processed at any time. However, for relative and indexed files, RMS permits a program to establish multiple record access streams for record operations to the same file. The presence of such multiple record access streams allows programs to process in parallel more than one record of a file. Each stream represents an independent and concurrently active sequence of record operations. Further, when such streams update records in the file, RMS employs the same bucket locking mechanism among streams that it uses to control the sharing of a file among separate programs.

As an example of multiple record access streams, a program could open an indexed file and establish two record access streams to the file. The program could use one record access stream to access records in the file in random access mode through the primary index. At the same time, the program could use the second record access stream to access records sequentially in the order specified by an alternate index. When a program accesses a record through either stream, RMS automatically uses its bucket locking mechanism to ensure that both streams do not attempt to write the same record at the same time.

Synchronous and Asynchronous Record Operations

Within each record access stream, a program can perform any record operation either synchronously or asynchronously. The RSTS/E operating system supports synchronous record operations only. When a record operation is performed synchronously, RMS returns control to a program only after the record operation request has been satisfied (e.g., a record has been read and passed to one program). When a

record operation is performed asynchronously, RMS can return control to one program before the record operation request has been satisfied. A program, then, can utilize the time required for the physical transfer between the file and memory of the block or bucket containing the record to perform other computations. However, a program cannot issue a second record operation through the same stream until the first record operation has completed. To ascertain when a record operation has actually been performed, a program can issue a wait request and regain control when the record operation is complete.

Record Transfer Modes

In addition to specifying synchronous or asynchronous operations for each request in a record access stream, a program can utilize either of two record transfer modes to gain access to each record in memory:

- **Move Mode Record Transfers** — RMS permits move mode record operations for all file organizations and record operations. Move mode requires that an individual record be copied between the I/O buffer and a program. For read operations, RMS reads a block or bucket into an I/O buffer, finds the desired record within the buffer, and moves the record to a program-specified location.

  Before a write or update operation in move mode, the program builds or modifies a record in its own work space. Then the program issues a write or update record operation request, and RMS moves the record to an I/O buffer.

- **Locate Mode Record Transfers** — RMS supports locate mode record transfers for read operations to all file organizations. However, it permits locate mode on write operations for sequential files only.

  Locate mode reduces the amount of data movement, thereby saving processing time. This mode enables programs to access records directly in an I/O buffer. Therefore, there is normally no need for RMS to copy records from the I/O buffer to a program. To allow the program to access a record in the I/O buffer, RMS provides the program with the address and size of the record in the I/O buffer.

DATA BASE MANAGEMENT SYSTEM
DBMS (V. 1.5)

OVERVIEW

DBMS is a CODASYL-standard data base management system that uses the "set definition" as the basic building lock with which even the most complex data relationships can be defined. It separates data definitions from data references and allows definition of logical, applications-oriented relationships. DBMS suppports multi-level sequential, hierarchical (tree), and network data structures.

FEATURE TOPICS

- Features
  - Data Description Language
  - Data Manipulation Language
  - Schema Data Description Language

- Data Organization
  - DIRECT
  - CALCULATED
  - VIA

- Physical Space Management
  - Set Relationship Capabilities
  - Set Membership

- Data Base Utilities
  - Data Dictionary Facilities
  - Data Base Recovery and Journaling

- Data Manipulation Language

- COBOL/DML Compilation

- Other Language/DML Compilation

- Programming Requirements

- Execution of Object DML Programs

FEATURES

Data Base Management System (DBMS) provides a centrally managed data base that acts as a common resource for application programs.

DBMS allows each application program to access an appropriate subdivision through a simple set of commands that act as extensions to COBOL and FORTRAN programs and as calls from BASIC and MACRO programs.

DBMS is controlled through two sets of language facilities for data base administration and application programming. The Data Description Language (DDL) allows the central data base to be defined and created. The Data Manipulation Language (DML) allows individual application programmers to access portions of the data base by using simple commands that are embedded in application programs.

DBMS allows the creation of one central data base that acts as a common resource for any number of application programs. This central data base reduces redundant data, provides data consistency, and allows the data base to be maintained more easily and with more security.

The description of the central data base (called the **schema**) is done with the Schema Data Description Language (Schema DDL). The Schema DDL performs these three functions:

- defines the physical mapping of the schema to device media through the DML control language
- defines all data elements (records, groups, items) in the data base
- describes all logical relationships (structures) that are to exist between elements

Once the central data base is defined, any number of logical subdivisions can be defined using the Sub-schema DDL. Each sub-schema defines a specific combination of records and structures which apply to a given application program. This central control and controlled allocation of the data base not only provide for maintainability and security of data, but permit markedly improved application programmer efficiency. For example, the arduous data description function is removed from the scope of the application programmer. Thus, individual application programs are easier to write and debug, and have superior portability and better maintainability.

The Schema DDL, the Sub-schema DDL, and the COBOL and FORTRAN DMLs are implementations of CODASYL and approved by the COBOL Journals of Development.

DATA ORGANIZATION

The basic unit within the DBMS data base is the **record**. Individual records are differentiated from one another by their format—defined and alterable using the Schema DDL. For example, one record type might be named CUSTOMER and could include all record occurrences that have the following three data items:

● 8 characters for customer number

● 32 characters for customer name

● 32 characters for customer address

Any number of record types can be defined to meet the needs of the user community. Further, each record type can be included in any number of sub-schemas; identical record types need not be duplicated for different application programs. For example, the record type EMPLOYEE can be in the sub-schema for both the personnel and medical benefits application programs.

Since large data bases are generally stored on disk systems where the time between successive accesses is the key to optimum performance, data base performance can be improved by locating and retrieving these related records with a single access. DBMS offers three modes of record storage that provide flexible record placement control.

These record location modes are:

● DIRECT — allows the user maximum control over logical storage location. In this mode, a suggested data base key (desired location) is supplied before the record is stored. If the suggested data base key is available, it will be assigned to that record. If unavailable, the next available data base key will be assigned.

● CALCULATED (CALC) — stores the record based on the value of one or more data items within the record. This option can be used to spread records evenly over an area. Ever CALC record provides a "keyed" entry point into the data base.

● VIA — used for record occurrences that will be referenced primarily in conjunction with another record type. For example, if record type EMPLOYEE is usually accessed in conjunction with record type COMPANY, the VIA option would store record occurrences of EMPLOYEE as close to occurrences of the specific COMPANY as is possible. This clusters member records around owners, minimizing the number of physical accesses necessary.

Table 13-1 Physical Storage Allocation Under DBMS

	PAGES 1–N (Entire Physical Database)		
DBMS-11 AREAS	AREA 1 PAGES 1–1000 (1st Physical Area)	AREA 2 PAGES 1001–3000 (2nd Physical Area)	AREA "X" PAGES "b"–"N" Xth Physical Area)
POSSIBLE "FILES" MAPPINGS	FILE A	FILES B & C	FILES D–Z
		B C	

PHYSICAL SPACE MANAGEMENT

The basic unit of physical space management under DBMS is the **page**. A page is a fixed-length block transferred to and from storage (size defined by the user) which will be maintained by the DBMS input/output routines.

The total physical extent of the data base is described in pages using the Schema DDL (such as pages 1-10,000). The data base is then divided by the Schema DDL into **areas**. For example, area 1 might be pages 1-1,000; area 2, pages 1001-3000, and so on.

No special overflow storage areas are ever needed when dealing with DBMS data bases. The space management routines handle appropriate allocation and continuous optimization of the storage resources available. Through the use of special space management pages and other techniques, record retrieval and storage are optimized—even when the data base is 80% to 90% full.

Within a page, any number of different records and different record types may be stored, each with its own length. Variable length space management is employed within each page so that when a record is deleted, the associated space is made available for reuse. This eliminates efficiency-robbing gaps in the data base.

Through the use of the Device/Media Control Language (DMCL), the data base may be mapped to any combination of system **files**:

● The entire data base may be assigned to a single file.

● Related files may be assigned to areas on a one-to-one basis.

● Several files may be associated with a particular area.

The applications programmer is never aware of which files are connected to the portion of the data base associated with specific applications.

SET RELATIONSHIP RULE

While the ability to control the physical placement of records is important to optimize system speed, the most important aspect of any data base management system's usability is the number and kind of data relationships that can be supported.

Logical relationships are established by defining named collections of record types—each called a **set**. Each set must have one record type declared as the owner record and one or more record types declared as its member records. Seven basic examples of this freedom to define sets in any manner to meet exact user needs are:

Set Relationship Capabilities

DBMS includes a series of options to establish and maintain set relationships. Since each set is described independently from any other set, the record order of each set can be defined for most efficient access. Record order within a set may be defined to be:

- FIRST (New records are positioned first in a series of records.)

- LAST (New records are positioned last in the series of records.)

- PRIOR (New records go immediately before a record occurrence established by the user program.)

- NEXT (New records go immediately after a record occurrence established by the user program.)

- SORTED (Record placement is determined by a user-supplied algorithm.) In a sorted set, records will be logically positioned in ascending or descending order based on the value of one or more data items within the record.

The logical ordering of member record occurrences in a set is independent of the physical placement of the records. Additionally, the same record type can participate (be a member) in any number of different sets, each with a different ordering criterion.

Set Membership
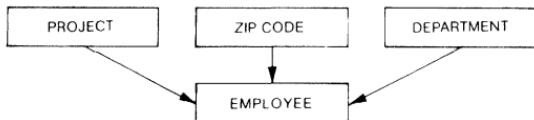
Not only can a record type participate as a member of different sets with a different relationship in each set, but the type of membership may be different in each set. Set membership can be *optional* or *mandatory*, and *manual* or *automatic*, depending on whether the user application program is permitted to CONNECT or DISCONNECT an occurrence of this record type from the set.

- **Optional** set membership allows a record to be removed from a set without the record's being deleted from the data base.

- **Mandatory** set membership defines a record as a permanent member of a set as long as the record is present in the data base.
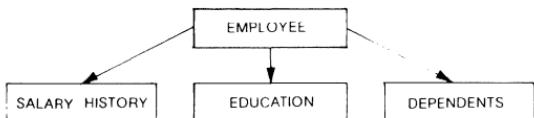
- **Automatic** set membership defines the record to be specified as a member of a set as an automatic function performed by the data base management system when the record occurrence is stored in the data base.

- **Manual** set membership allows the connection to a set of a record occurrence to be performed by the user program.

Table 13-2 Set Relationships
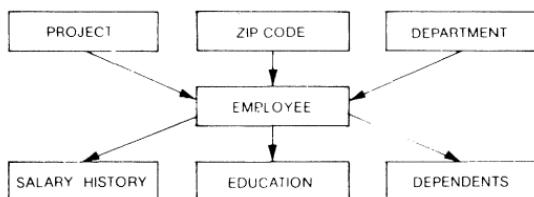
1. Any record may participate as a member in one or more SETs.

| PROJECT | ZIP CODE | DEPARTMENT |

| EMPLOYEE |
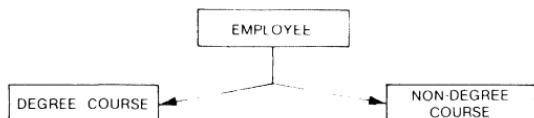
2. Any record may be specified as the owner of one or more SETs.

| EMPLOYEE |

| SALARY HISTORY | EDUCATION | DEPENDENTS |
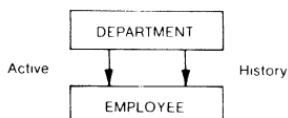
3. Any record may participate as a member of any number of SETs, and also be an owner of one or more SETs.

| PROJECT | ZIP CODE | DEPARTMENT |

| EMPLOYEE |

| SALARY HISTORY | EDUCATION | DEPENDENTS |

4. A SET may have only one record type as its owner but may have one or more record types as members.

| EMPLOYEE |

| DEGREE COURSE | | NON-DEGREE COURSE |
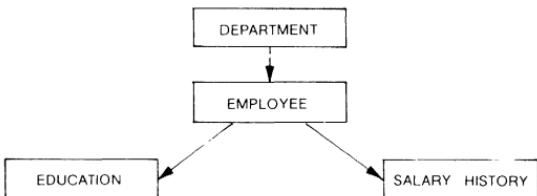
5. Any number of SET relationships may exist between two record types.

| DEPARTMENT |

Active   History

| EMPLOYEE |

6. A record type may exist without any SET participation – as neither a member nor an owner of a SET.

| GRADE/SALARY |

312

7. Any record type may be defined as an optional member of a SET. Participation of each record occurrence is established or deleted based on execution of a statement within a user program.

```
        ┌──────────────┐
        │  DEPARTMENT  │
        └──────────────┘
               │
               ▼
        ┌──────────────┐
        │   EMPLOYEE   │
        └──────────────┘
          ╱        ╲
┌──────────────┐   ┌──────────────────┐
│  EDUCATION   │   │  SALARY HISTORY  │
└──────────────┘   └──────────────────┘
```

Set Linkage

The relationship between record types within a set is maintained by the existence of pointers to other related records, known as linkage. Members of sets are always linked automatically in the forward direction (**next** pointer). Sets may optionally be defined to be linked either in the reverse direction (**prior** pointer) and/or linked to the owner (**owner** pointer). Owner pointers allow the owner record to be accessed directly from a member record without following the next or prior pointer chain to the end.

Thus, options for set linkage are: next pointers, next and prior pointers, next and owner pointers, or next, prior, and owner pointers.

DATA BASE UTILITIES

DBMS includes a comprehensive set of utilities that allows the data base administrator to monitor and audit the activity of the system, measure the performance of the system, and recover the data base after a hardware or software failure.

Data Dictionary Facilities

A group of comprehensive data dictionary reports are integral to DBMS. These listings provide tools for controlling and maintaining the data base plus documentation of non-DBMS files. Reports include a range map, record descriptions, set descriptions, a DMCL map, and user Data Dictionary.

Data Base Recovery and Journaling

DBMS has integrated journaling and recovery features. It automatically maintains a journal of all changes made to the data base. This journal includes both "before" and "after" images of modified portions of the data base. Checkpoint statistics are also included.

Security Dump and Security Restore

This facility provides for a high-speed copy of the entire data base or named area(s) to or from a serial device. It also provides statistics about the area(s) copied, including each record type found, number of occurrences, total characters occupied, and percent of copied area occupied by each record type. The security dump also indicates page loading and cluster information.

Journal Rollforward

The rollforward utility recovers the data base forward to a specified point in time by reapplying "after" images from the journal tape to the data base in chronological order. It provides information about each end-of-job checkpoint reached and determines if the data base is in a logical quiesce condition at this checkpoint. The rollforward utility also takes into account a multi-tasking environment, and at each check-point will display all other programs that were active, which areas the programs had open, and their declared usage mode in each area. It will also print "before" images in decimal and/or octal format for auditing or test evaluation. Rollforward will display page images without reapplying them to the data base, if specified.

Journal Rollback

This utility recovers the DBMS data base to a user-specified point backward in time by reapplying "before" page images from the journal tape. Journal rollback corresponds to the journal rollforward utility except that it is sensitive to "after" images and end-of-job check-points.

Initialization Utility

This utility initializes any part of the DBMS data base or portion thereof to empty pages. Each area must be formatted with this utility before any record occurrences can be stored.

Page Find/Fix

Page Find/Fix is a general debugging tool that allows display and replace functions to check the contents of any page and make modifi-cations. It will locate and display a page based on a page number or key value of a CALC record. If the key value is submitted, the utility performs the CALC transform on the value and displays the page that contains the record occurrence with that value.

CALC Routine

The mathematical transform used to store and retrieve records in the CALC location mode is available through this routine. It may be used

to test a key value set for even distribution in an area or as a mechanism to permit presenting records so that a data base can be efficiently loaded.

On-line Recovery

When certain programs end abnormally, this utility immediately recovers the data base without affecting other programs operating at the same time. Programs doing retrieval only need not be recovered. Programs in the UPDATE (shared) usage mode cannot be automatically recovered. Programs in the PROTECTED UPDATE or EXCLUSIVE UPDATE usage modes will be recovered. This utility performs as a roll-back operation replacing "before" images for the program that terminated abnormally.

Journal File Fix

This utility will copy a journal file in cases where no ending labels have been placed on that file because of operating system failure, hardware crash, etc. It will write the appropriate end-label information on the output journal file.

Data Base Query

This utility allows the user to use interactive machine DML commands to access the data base without need for read to write programs.

Data Base Utility

This utility verifies that the data contained in the data base is consistent with the Schema description.

COMMON ACCESS MONITOR PROGRAM (CAMP)

The Common Access Monitor Program (CAMP), also referred to as DBX, allows any number of programs in the central processor to use a single copy of the data base control system (DBCS) for effective multi-tasking. CAMP threads all requests from application programs to the DBCS. It also includes an automatic recovery program which will, when possible, back-out the effect of a program that ends abnormally in the on-line environment.

The buffer pool resides with the DBCS. The buffer size is defined by the user of a particular data base. The association of programs with invoked sub-schemas assures that when each program ends, its sub-schema is released immediately from main memory, keeping total overhead to a minimum.

DATA MANIPULATION LANGUAGE

The simple command structure of the data manipulation language means:

- Programmer learning time is minimized.

- Programs written by one programmer are easily understood by others.

- Application programs are more portable and more easily maintained.

- Programming productivity is increased.

Data manipulation statements may be included anywhere within the procedure coding of a COBOL, FORTRAN, or other language application program.

Control statements establish access to a portion of a data base. The READY AREA statement announces the user's intention to start processing within the specified area. The corresponding FINISH AREA statement announces the end of processing.

Locating or accessing records is done with the following statements:

- FIND locates a record occurrence in the data base that satisfies the record-selection-expression portion of the statement.

- GET causes a retrieval from the most recently located record.

- ACCEPT CURRENCY causes movement of the data base key associated with the most recently located record to a named location in the program.

Modification statements result in a change to the contents of the data base. Changes include the addition of new data, modifications to existing data-item values, or deletion of data in the data base.

- STORE uses data established by the user in working storage to create a new record occurrence in the data base.

- MODIFY changes the data content of an existing record in the data base.

- CONNECT and DISCONNECT cause a change in the set relationship of an existing record occurrence in the data base.

- ERASE causes an existing record occurrence to be removed from the data base.

The application programmer can access the data base only through a sub-schema which is predefined by the data processing manager. A sub-schema is made available to a program via a declarative statement. Records defined in the sub-schema specify the only data base data that can be manipulated by the application program. This fur-

nishes data privacy independence at the application program level. Data manipulation statements are referred to as the data manipulation language (DML). The DML processor supplies record type descriptions from the sub-schema into the working area of the user's program.
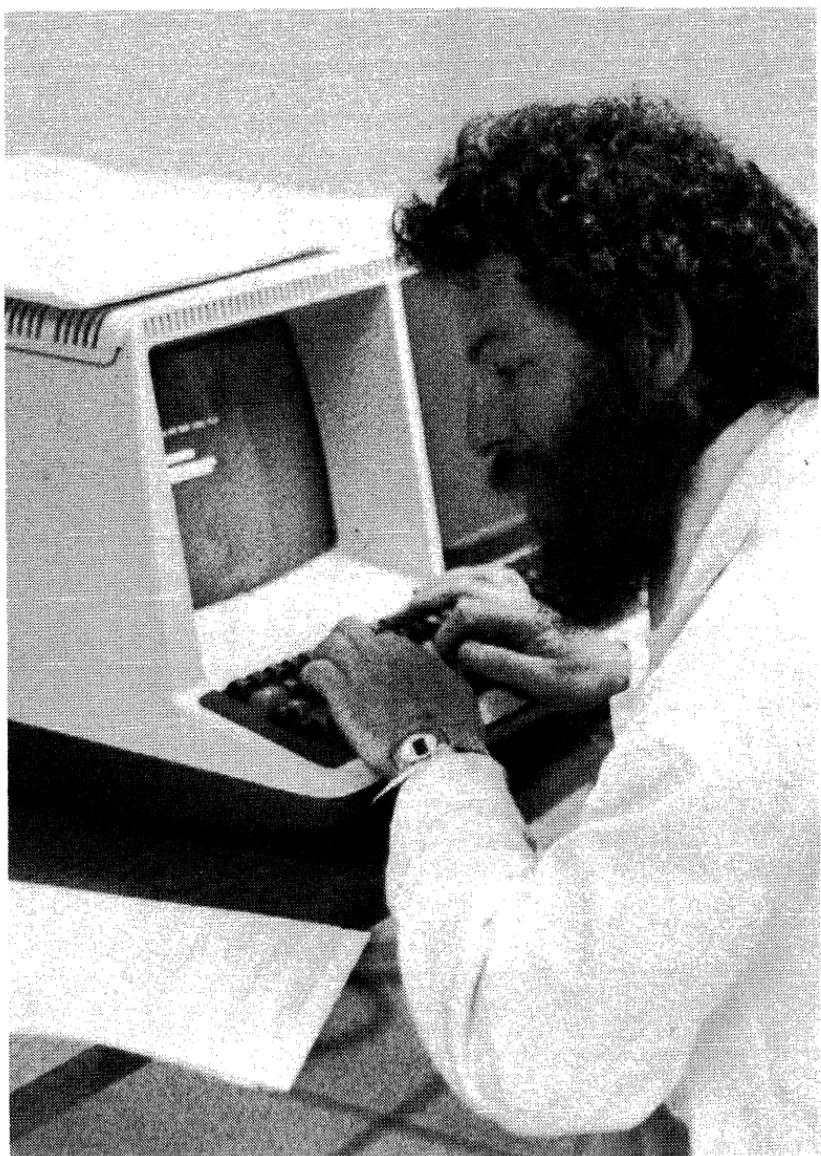
Table 13-3 DBMS Data Manipulation Language

DBMS-11 Data Manipulation Language

CONTROL STATEMENTS:

DB *sub-schema-name* **WITHIN** *schema-name.*

READY *[realm-name]* $\left[\textbf{USAGE MODE} \text{ IS } \begin{bmatrix} \textbf{PROTECTED} \\ \textbf{EXCLUSIVE} \end{bmatrix} \begin{Bmatrix} \textbf{RETRIEVAL} \\ \textbf{UPDATE} \end{Bmatrix} \right]$.

FINISH.

FIND *[record-name]* **DB-KEY** IS *identifier.*

FIND CURRENT $\left[\textbf{WITHIN} \begin{Bmatrix} set\text{-}name \\ realm\text{-}name \end{Bmatrix} \right]$.

FIND $\begin{Bmatrix} \textbf{NEXT} \\ \textbf{PRIOR} \\ \textbf{FIRST} \\ \textbf{LAST} \\ integer \\ identifier \end{Bmatrix}$ *[record-name]* **WITHIN** $\begin{Bmatrix} set\text{-}name \\ realm\text{-}name \end{Bmatrix}$.

FIND OWNER WITHIN *set-name.*

FIND $\begin{Bmatrix} \textbf{ANY} \\ \textbf{DUPLICATE} \end{Bmatrix}$ *record-name.*

FIND *record-name* **WITHIN** *set-name* **[CURRENT]** **USING** *identifier.*

IF *set-name* IS **[NOT]** **EMPTY** *imperative-statement.*

IF **[NOT]** *set-name* $\begin{Bmatrix} \textbf{MEMBER} \\ \textbf{OWNER} \end{Bmatrix}$ *imperative-statement.*

MODIFICATION STATEMENTS:

ERASE *[record-name]* $\left[\begin{Bmatrix} \textbf{PERMANENT} \\ \textbf{SELECTIVE} \\ \textbf{ALL} \end{Bmatrix} \textbf{MEMBERS} \right]$.

CONNECT *[record-name]* **TO** *set-name*

MODIFY *[record-name].*

DISCONNECT *[record-name]* **FROM** *set-name.*

STORE *record-name.*

RETRIEVAL STATEMENTS:

GET *[record-name].*

ACCEPT *identifier* **FROM** $\begin{Bmatrix} record\text{-}name \\ set\text{-}name \\ realm\text{-}name \end{Bmatrix}$ **CURRENCY.**

COBOL/DML COMPILATION

The user must identify a predefined sub-schema in the sub-schema section of the Data Division. DML statements may appear anywhere in the Procedure Division and may be considered for the programmer as an extension to the COBOL language. This means that, surrounded by other COBOL statements, they may be placed at the appropriate point in any procedure. PDP-11 COBOL DML reads the COBOL/DML source statements and searches within the Data Division for the sub-schema section to obtain the name of the sub-schema specified. Once the sub-schema name has been verified, the DML processor obtains the names of all valid records, sets, and areas. The relationships between them are also obtained from sub-schema information stored in the data base description. These are used to validate DML statements in the Procedure Division.

PDP-11 COBOL will establish an 01 level record entry followed by selected data items in working storage for each record type included in the sub-schema. System control and communication data items are included automatically.

Each DML command is validated for correct syntax and usage of record-set-area relationships. At compile time, these DML validations guard against the programmer's using the data base improperly; hence resources are not wasted later during system /program debugging. All DML errors detected by PDP-11 COBOL will be displayed with the source statement in error.

FORTRAN DML COMPILATION

The user identifies a predefined sub-schema using a DML INVOKE statement at the beginning of each main or subprogram module that contains DML statements. FORTRAN DML statements may be considered by the programmer as an extension to the FORTRAN language, and, as such, DML statements may appear anywhere in a FORTRAN program that executable statements are allowed. The FORTRAN DML preprocessor (FDML) reads the FORTRAN/DML source program and converts all DML statements to standard FORTRAN statements.

FDML includes all the functionality of the COBOL DML, plus dynamic (runtime) naming, end error phrases, USE procedures, dynamic buffer binding, and selective record copies from the sub-schema division.

OTHER LANGUAGE/DML COMPILATION

DBMS can be used with any language that supports a CALL statement, such as BASIC and MACRO-11. The user communicates with DBMS from these languages by passing arguments through the CALL statement. The CALL statement procedure is straightforward and easy to use.

PROGRAMMING REQUIREMENTS

The DBMS input/output area for the data base resides in working storage. Each record included in the sub-schema is automatically included in working storage by the DML processor as an 01 level record entry followed by the statements which describe the name, picture attribute, and use of each data item. Only those data items of the record defined in the sub-schema are transferred to the user program. Input of a record from the data base always appears in its like named area in working storage.

Storage (or output) of a record to the data base requires the movement of data from various locations in the user's program to each of the data items described for the specific record in working storage. Once this is completed by user procedure statements, the record is moved physically from working storage into the data base using the STORE statement.

The user is responsible for initializing all data items required to execute a DML statement successfully, and must ensure that the data is correct. DBMS has extensive object time error diagnostic facilities and will update error status after every DML statement. To determine the action taken by the system in response to the request, the user must examine the error status following each DML command.

In summary, three operations are required for each access to the data base:

- initialization of data items as required by the DML statement to be executed
- initiation of the data base operation by the DML statement
- error checking to determine the outcome of the preceding DML command

EXECUTION OF OBJECT DML PROGRAMS
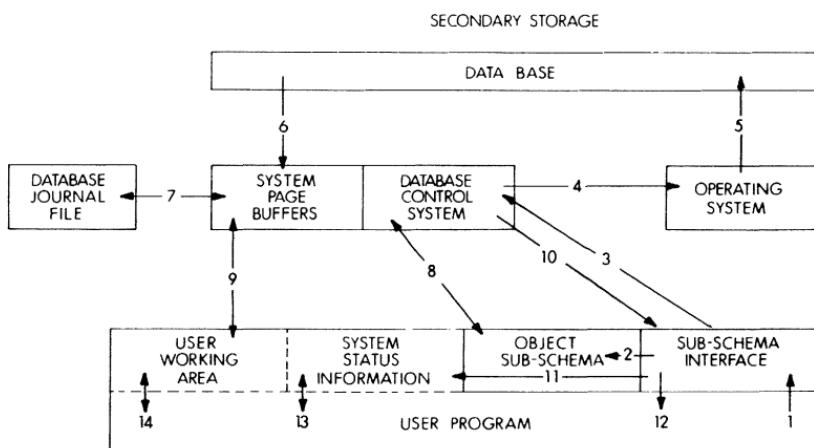
The operations that take place when a DML statement is executed are discussed this section. Numbers in parentheses refer to Table 13-4.

- A DML statement appears in the object program as a request to the sub-schema interface routine (1). The request identifies the type of data base service desired and any additional information, such as record name, set name, and area name, required to interpret the request properly.
- The sub-schema interface routine analyzes the request using information stored in the object sub-schema (2). If the request requires the use of the data base control system (DBCS), the user-supplied information is augmented by information from the object schema before control is passed the DBCS (3) itself.

- The DBCS performs the requested data base service using information supplied by the sub-schema interface routine. The operation performed depends upon the type of DML statement executed. In the event of a request to locate a record (FIND statement), the DBCS will look in the system page buffers to see if the requested record is present. If the record is not in the system page buffers, a request will be made to the operating system (4) to input a data base page from the direct access file to the system page buffers (5,6). No input is initiated if the record is already present in the system page buffers.

- Any changes to the data base are recorded on the data base journal file (7).

- The DBCS performs the requested service using additional information contained in the object sub-schema. The object sub-schema contains a representation of the data structure, record placement control, record characteristics, currency status, data base operation, statistics, and constraints on DML operations. In general, the object sub-schema controls the operations and access of the data base for each program that invokes it.

- Whenever a specified record occurrence is located by the DBCS, the data base key and other system information related to the record are moved from the system page buffer to locations within the object sub-schema (8). This information represents the currency status of the area, sets, and record type of the record occurrence which has been located.

- If the request to the DBCS specified movement of the contents of a record to the user working area (GET statement), data will be moved from a system page buffer to a specified record area in the user working area (9). Only the data portion of the requested record is delivered to the user; system controlled structure data is retained by the DBCS to ensure data base integrity. Data movement from the user working area to the system page buffers will occur in response to a STORE or MODIFY statement.

- The DBCS returns to the interface routine with an indication of the success or failure of the data base service (10).

- The sub-schema interface routine moves status information regarding the outcome of the DML statement executed to locations within the user working area of the program (11).

- Control is returned to the user's program at the statement following the DML statement just executed (12).

- The user must determine the status of the previous DML statement by examining the contents of the system status information (13). For example, if a FIND statement was just executed, the contents of

system status information would indicate whether the specified record occurrence was located or not. If the system status information condition indicates that the service requested was completed successfully, the user would access the user working area as needed (14).

Table 13-4 Execution of Object DML Programs

SECONDARY STORAGE

```
                              DATA  BASE
                                  6                        5

DATABASE        SYSTEM      DATABASE                  OPERATING
JOURNAL   — 7 — PAGE        CONTROL    — 4 —          SYSTEM
FILE            BUFFERS     SYSTEM

                               10     3
         9              8

         USER         SYSTEM      OBJECT          SUB-SCHEMA
         WORKING      STATUS      SUB-SCHEMA — 2   INTERFACE
         AREA         INFORMATION        — 11 —

         14           13      USER  PROGRAM      12           1
```

DATATRIEVE-11 (V.1)

OVERVIEW
DATATRIEVE-11 provides IAS, RSTS/E, and RSX-11M users with an inquiry language and report writing system that allows direct, easy access to data contained in RMS-11 files. This system is designed especially for unsophisticated computer users; everyday use of DATA-TRIEVE-11 requires no programming skills. And DATATRIEVE-11 data management facilities include interactive data retrieval, sort, update, and maintenance and access of data dictionary entries that define RMS-11 records.

FEATURE TOPICS
- Product Description
  - Query
  - Report Generation
  - Data Definition
- Features
- Minimum Hardware Required
- BASIC Commands
- Essential Terminology
- Processing a File as a Collection
- Special Syntactical Symbols
- Key Words
- Summary of Commands and Statements
- A Sample DATATRIEVE-11 Session

PRODUCT DESCRIPTION

DATATRIEVE-11 is an interactive query, report, and data maintenance system designed to operate in a programmerless environment.

- **Query** — allows interactive data retrieval, sort, and update.

- **Report generation** — produces summary and detailed reports, titles, headings, footnotes, group totals, and report totals.

- **Data definition** — allows creation, maintenance, and access of the data dictionary which defines RMS records.

The DATATRIEVE-11 system includes RMS-11K software and is available on the RSTS/E, RSX-11M, and IAS operating systems. RMS-11K record management services are utilized by the system to access data contained in relative, indexed, or sequential file organization. Additional facilities are provided by the system for selective data retrieval, sorting, formatting, updating, and report generation.

At the heart of the system is a Data Dictionary in which record formats and file definitions are stored. The Data Dictionary can be shared by multiple users, and can be used to save frequently accessed command sequences.

FEATURES

- Runs on RSTS/E, IAS, and RSX-11M operating systems.

- Features natural, English-like syntax that is easy to learn and easy to use.

- Notifies user of errors immediately for on-line correction.

- Maintains file security.

- Contains a Data Dictionary facility.

- Provides extensive user control of report formats.

- Handles RMS files in COBOL, BASIC-PLUS-2, FORTRAN IV, FORTRAN IV-PLUS, DIBOL, and MACRO.

- Includes interactive record updating capabilities with automatic prompts.

- Eliminates the need for many specialized applicatons programs.

- Uses COBOL-compatible record definitions.

MINIMUM HARDWARE REQUIRED

Any valid RSTS/E, RSX-11M, or IAS operating system configuration will run DATATRIEVE-11 if it includes:

- RMS-11K software

- 64K-byte user memory

- Extended Instruction Set
- A Floating Point Processor, if floating point data type is used.

BASIC COMMANDS

DATATRIEVE-11 uses a simple English-like command language for data retrieval, modification, and display. Prompting is automatic for both command and data entry. The major commands are:

- HELP — provides a summary of each DATATRIEVE-11 command.

- READY — identifies a domain for processing and controls the access mode to the appropriate file.

- FIND — establishes a collection (subset) of records contained in either a domain or a previously established collection based on a Boolean expression.

- SORT — re-orders a collection of records in either the ascending or descending sequence of the contents of one or more fields in the records.

- PRINT — prints one or more fields of one or more records. Output can optionally be directed to a line printer or disk file. Format control can be specified. A column header is generated automatically.

- SELECT — identifies a single record in a collection for subsequent individual processing.

- MODIFY — alters the values of one or more fields for either the select record or all records in collection. Replacement values are prompted for by name.

- STORE — creates a new record. The value for each field contained in the record is prompted for by name.

- ERASE — removes one or more records from the RMS-11 file corresponding to the appropriate domain.

- FOR — executes a subsequent command once for each record in record collection, providing a simple looping facility.

In addition to the simple data manipulation commands, a number of more complex commands are available for the advanced user. These commands, such as REPEAT, BEGIN-END, and IF-THEN-ELSE, may be used to combine two or more DATATRIEVE-11 commands into a single compound command. These, in turn, may be stored in the Data Dictionary as procedures for invocation by less experienced users.

DATATRIEVE-11 provides a full set of arithmetic operators (addition, subtraction, multiplication, division, and negation), a set of statistical operators (total, average, maximum, minimum, and count), and provides automatic conversion between data types used in the FORTRAN, COBOL, and BASIC-PLUS-2 languages.

The DATATRIEVE-11 report writer provides easy-to-use commands to control the following report functions:

- report name, date, and page numbering
- page width and length specification
- detail line specification
- multiple control break specification with automatic totaling at any level
- multiple report sections

A DATATRIEVE-11 report command can be freely intermixed with other DATATRIEVE-11 commands.

ESSENTIAL TERMINOLOGY
Files, **domains**, **collections**, **records**, and **fields** are five terms of fundamental importance to the file structure of DATATRIEVE-11.

Records are groups of related items of data that are treated as a unit. For example, all the pieces of data describing a model of a yacht in a marina could be grouped to constitute the record for that yacht.

Each of the individual pieces of data in a record is referred to as a **field**. The yacht's model number, length, and price are all potential fields in its record.

The term **files** refers to the logically related groups of data that are kept by RMS-11. For example, we might put all of the yacht records for a current inventory of yachts into one file.

Domains are named groups of data containing records of a single type. An RMS-11 domain consists of all the records in a particular RMS-11 file. In this case, we could say that all the yacht records for the current inventory are kept in the YACHTS domain. The number of records in any domain may change as new records are stored or old records are erased.

A **record collection** is a subset of a domain. It may consist of no records, one record, or up to all the records in the domain. Using our previous example, we could say that all the yachts manufactured by Grampian could be made to form the Grampian-collection, while those yachts manufactured by Islander could be used to form the Islander-collection. To carry this example one step further, if the inventory is currently out of stock of yachts manufactured by Seaworthy, the Seaworthy-collection will be empty, or null.

The **Data Dictionary** is a location where the definitions for procedures, records, and domains are kept in a standard fashion by DATATRIEVE-11. The Data Administrator will be concerned with the creation and maintenance of Data Dictionary information. Certain users will be able to display certain information from this dictionary, but only management will be concerned with defining it.

PROCESSING A FILE AS A COLLECTION

Perhaps the most important systems concept to master is collection processing. DATATRIEVE-11 operates on collections of records taken from the files. To get down to the level of record processing, the FIND and SELECT commands are employed to gather the collection and extract the records desired. The system provides a cursor facility to track the user's place in a collection. Figure 13-1 illustrates the cursor as a place marker, and shows how it can be manipulated through the collection, from the first, to the next, to the last record. It need not always move forward; directional movement within a collection is at the user's discretion.

In most DATATRIEVE-11 operations, the files are never changed, but a great deal of manipulation occurs on the collections. Thus, the collections can be thought of as a sort of temporary storage, kept for immediate purposes, and then released.
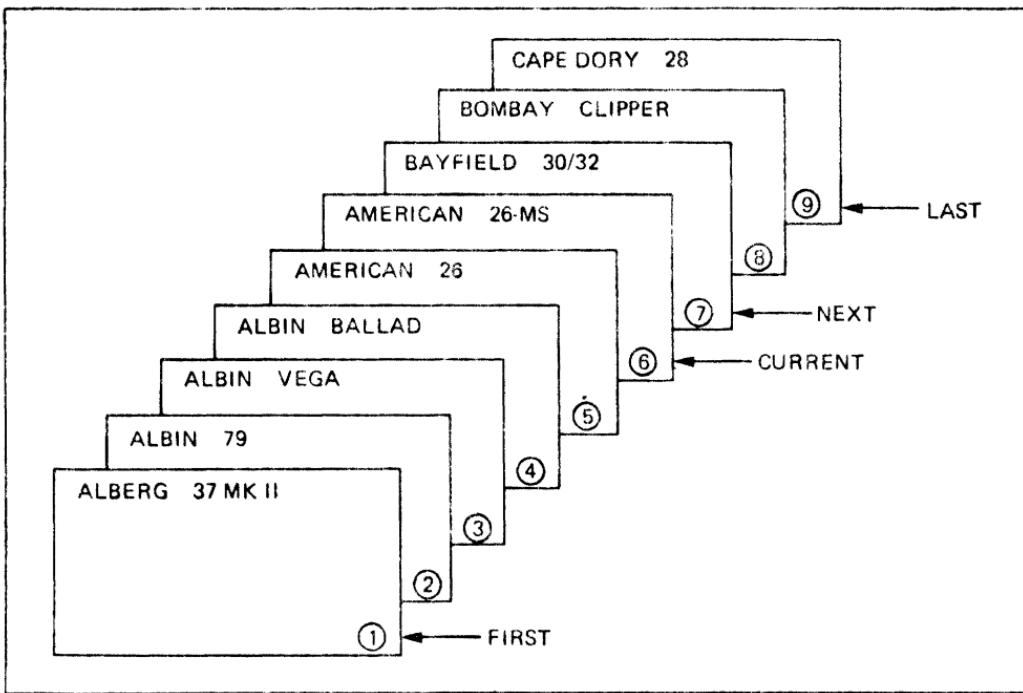
Figure 13-1 Collection Cursor

The FIRST record of the collection is the first one encountered when it was established or collected. The record numbered with (1) in Figure 13-1 identifies the FIRST record in this example.

The NEXT record is always the one immediately after the CURRENT one. In the figure, the CURRENT record is the sixth one, so the NEXT record must be the seventh.

We have been referring to records according to their numeric position in the collection, that is, sixth, seventh, and so forth. This is a proper form of reference, and in the DATATRIEVE-11 language digits will be used for the position number whenever a format specifies the nth record.

The LAST record is the one located at the very end of the collection.

If there happen to be no records at all in a collection, the collection is null, and the collection cursor will have to be null, too.

Generally, the user works on just one collection at a time, the CURRENT one. However, it is possible to name a collection and refer back to it later in the same session, if it is important enough or likely to be used again.

Thus, it is possible to be working with a number of collections, one of which is the CURRENT one, the one most recently the object of a FIND command. Each of the collections has its own collection cursor and may have a designated CURRENT record, as well. However, only one collection may be CURRENT at a time.

The lifetime of any collection is limited to a given session. Collections cannot be saved, nor can they be shared by several users. However, the user can always reproduce a collection of identical characteristics (although not necessarily identical records, since the files can change in the interim), simply by repeating the sequence of FIND commands used initially to establish and/or refine the collection. If this is a frequent requirement, the steps should be defined as a procedure.

Collections may be ordered on command. When a collection is sorted, it may well change the position a particular record holds.

Special Syntactical Symbols
Special symbols are used in syntax descriptions to clarify proper usage of DATATRIEVE-11 command elements. While these symbols are discussed individually in the following section, Figure 13-2 shows how they can combine in numerous syntactical expressions and even be nested.

Syntax Brackets ([ ]) denote items that are optional. If more than one item is enclosed in brackets, one (or none) of the items may be chosen.

Square Brackets ([ ]) denote the bracket symbols on the keyboard and are required where shown.

Braces ({ }) enclose one or more items that are required. If more than one possible entry is shown, the user must include one of the choices, but can specify no more than one.

Ellipsis (...) implies that one can choose to repeat a version of the command element immediately preceding. In DATATRIEVE-11 there are no restrictions on the number of repetitions, unless they are explicitly stated. Watch for punctuation characters needed to serve as separators for the repetitions. Note that repetitions occur only at the user's option.

Uppercase letters in boldface denote required words. In lightface, these words are optional.

Lowercase letters are used with generic terms that must be replaced by the appropriate names or values.

Boldface denotes that the uppercase word is required in every use of the command or statement, unless enclosed in other brackets.

Figure 13-2 illustrates a fairly complex syntactical structure, employing all these symbols in a English sentence rather than a command. Observe how the symbols can be nested. Whenever nesting occurs, the meaning of the outermost pair of symbols prevails. Thus, when a seemingly required item inside braces occurs inside a pair of outer brackets, the entire group is considered to be optional.

As is shown in Figure 13-2, there are a large number of legal sentences that can be constructed to conform to the syntax; only a few possiblilities are shown here.

Later, when these same symbols are used in actual DATATRIEVE-11 command definitions, keep in mind the large number of combinations of command elements they imply. The symbols are merely part of the description of the command, a sort of code to indicate a number of ways to construct each command from its parts.

Assume the following syntactical structure:

JOHN PLAYS
$$\begin{bmatrix} \text{LATE} \\ \text{sport-name} \\ \text{THE FIELD} \\ \underline{\text{THE HORSES}} \end{bmatrix} \left[\text{FORGETS HE HAS A DATE WITH} \left\{ \begin{matrix} \text{BETTY} \\ \text{SUSIE} \\ \text{MARY} \\ \text{GAIL} \end{matrix} \right\} ,... \right], \textbf{AND LOSES ALL.}$$

The following sentences are among the many possibilities the user could construct:

JOHN PLAYS, AND LOSES ALL.

None of the optional bracketed items is included; only the required underlined words in uppercase are kept.

JOHN PLAYS THE HORSES, AND LOSES ALL.

An object word for PLAYS is supplied, but the bracketed FORGETS clause is omitted.

JOHN PLAYS, FORGETS HE HAS A DATE WITH MARY, AND LOSES ALL.

The optional bracketed items after PLAYS are omitted, but the FORGETS clause is added.

JOHN PLAYS THE FIELD, FORGETS HE HAS A DATE WITH SUSIE, and LOSES ALL.

All options included except the repetition.

JOHN PLAYS BASEBALL, FORGETS HE HAS A DATE WITH BETTY, A DATE WITH GAIL, AND LOSES ALL.

Observe the substitution of BASEBALL for the lowercase word sport-name and the repetition at the site of the ellipsis.

Figure 13-2 Illustrating Syntactical Symbols

Keywords

DATATRIEVE-11 utilizes language elements called keywords which have a specific denotation and associated function. If they are used in any other context, they may serve to confuse the system about user intentions. Thus, it is good policy to avoid the use of these words as names of domains, procedures, records, fields, and collections.

The full list of keywords is reproduced in Table 13-1.

Table 13-1 Keywords

ABORT	DESC	LESS-EQUAL	READY
ADVANCED	DESCENDING	LESS-THAN	RECORD
ALL	DISPLAY	LINES-PAGE	RECORDS
AND	DOMAIN	LT	RELEASE
ASC	DOMAINS	MAX	REPEAT
ASCENDING	EDIT-STRING	MAX-LINES	REPORT
AT	ELSE	MAX-PAGES	REPORT-HEADER
AVERAGE	END	MIN	REPORT-NAME
BEGIN	END-PROCEDURE	MODIFY	SELECT
BETWEEN	EQ	NE	SEPARATE
BOTTOM	EQUAL	NEW-PAGE	SET
BT	ERASE	NEW-SECTION	SHARED
BY	EXCLUSIVE	NEXT	SHOW
CHARACTER	EXIT	NO	SHOWP
COL	EXTEND	NO-DATE	SIGN
COLLECTIONS	FILL	NO-NUMBER	SKIP
COLUMN	FIND	NOT	SORT
COLUMN-HEADER	FINISH	NOT-EQUAL	SORTED
COLUMNS-PAGE	FIRST	NUMBER	SPACE
COMP	FOR	OF	STORE
COMP-1	GE	ON	TAB
COMP-2	GREATER-EQUAL	OR	THE
COMP-3	GREATER-THAN	PAGE	THEN
COMP-5	GT	PIC	TOP
COMP-6	HELP	PICTURE	TOTAL
COUNT	IF	PRINT	TRAILING
CURRENT	IN	PROCEDURE	UIC
DATE	INCREASING	PROCEDURES	USAGE
DECREASING	IS	PROTECTED	USING
DEFINE	JUSTIFY	PW	VERIFY
DEFINEP	LAST	QUERY-HEADER	WITH
DELETE	LE	QUERY-NAME	WRITE
DELETEP	LEADING	READ	

Summary of DATATRIEVE-11 Commands and Statements

Commands

DEFINE DOMAIN domain-name-1 **USING** record-name-1
 ON rms-file-spec-1 ;

DEFINE PROCEDURE procedure-name-1

$$\left.\begin{array}{c} \cdot \\ \cdot \\ \cdot \end{array}\right\}$$ DATATRIEVE statements and commands

 END-PROCEDURE;

DEFINE RECORD record-name-1 USING data-def-1 [data-def-2...] ;

DEFINEP resource-nme-1 [(password-str-1)] seg-number,
 lock-type-1,key-1, privilege-str-1

DELETE $\left\{\begin{array}{l} \text{domain-name-1} \\ \text{record-name-1} \\ \text{procedure-name-1} \end{array}\right\}$ [(password-str-1)];

DELETEP resource-nme-1 [(password-str-1)] seq-number

ERASE [**ALL** [**OF** rse]]

EXIT

FIND domain-name-1 [**WITH** condition]

FIND CURRENT [**WITH** condition]

FIND record-selection-expression

FINISH [domain-name-1 [,domain-name-2...]]

HELP [**ADVANCED**] [command-name-1[,command-name-2...]]

MODIFY [**ALL**] [field-name-1 [,field-name-2...]] [**OF** rse]

PRINT [**ALL**] [print-list-1] [**OF** rse] [**ON** $\left\{\begin{array}{l} \text{file-spec-1} \\ \text{*.prompt-name-1} \end{array}\right.$]

READY domain-name-1 [(password-str-1)] $\left[\begin{array}{l} \text{SHARED} \\ \text{PROTECTED} \\ \text{EXCLUSIVE} \end{array}\right]$ $\left[\begin{array}{l} \text{READ} \\ \text{MODIFY} \\ \text{WRITE} \\ \text{EXTEND} \end{array}\right]$

RELEASE collection-name-1 [,collection-name-2...]

SELECT $\begin{bmatrix} \textbf{FIRST} \\ \textbf{NEXT} \\ \textbf{LAST} \\ \text{value-exp-1} \end{bmatrix}$ [collection-name-1]

SHOW show-item-1   [,show-item-2...]

    where the show items are chosen from the following list:

$\left\{\begin{array}{l} \textbf{PROCEDURES} \\ \textbf{DOMAINS} \\ \textbf{COLLECTIONS} \\ \textbf{RECORDS} \\ \textbf{ALL} \\ \text{CURRENT} \\ \text{READY} \\ \text{procedure-name-1[(password-str-1)]} \\ \text{domain-name-1[(password-str-2)]} \\ \text{record-name-1[(password-str-3)]} \\ \text{collection-name-1} \end{array}\right\}$

SHOWP resource-1 [(password-str-1)]

SORT [collection-name-1] BY sort-key-1 [,sort-key-2...]

    where the sort-keys assume the following form:

$\begin{bmatrix} \textbf{ASC}[\text{ENDING}] \\ \textbf{DESC}[\text{ENDING}] \\ \textbf{INCREASING} \\ \textbf{DECREASING} \end{bmatrix}$ field-name-1

STORE domain-name-1   [USING statement-1][VERIFY USING statement-2]

Statements

field-name-1 = value-exp-1

field-name-1 = field-name-2

ABORT value-exp-1

BEGIN statement-1 [;statement-2...]   **END**

DISPLAY value-exp-1

FOR rse-1 statement-1

IF condition THEN statement-1 [**ELSE** statement-2]

REPEAT value-exp-1 statement 1

statement-1 **THEN** statement-1

Report Writer Statements

fld-name-1

AT $\begin{Bmatrix} \text{TOP} \\ \text{BOTTOM} \end{Bmatrix}$ OF $\begin{Bmatrix} \text{fld-name-1} \\ \text{PAGE} \\ \text{REPORT} \end{Bmatrix}$ PRINT summry-item-1 [,summry-item-2...]

PRINT detail-item-1 [,detail-item-2...]

REPORT [rse]   [**ON** file-spec-1]

REPORT END

SET parameter-1 [,parameter-1...]

   where the parameters are chosen from the following list:

      REPORT-NAME = report-name
      MAX-LINES = integer-1
      MAX-PAGES = integer-2
      NUMBER
      NO-NUMBER
      DATE = ["string-1"]
      NO-DATE
      LINES-PAGE = integer-3
      COLUMNS-PAGE = integer-4

Subexpressions

where the record selection expression (rse) assumes the following form:

$\begin{bmatrix} \textbf{ALL} \\ \textbf{FIRST } n \end{bmatrix}$ [collectn-name-2 **IN**] $\begin{Bmatrix} \text{CURRENT} \\ \text{collectn-name-3} \\ \text{domain-name-1} \end{Bmatrix}$ [**WITH** conditn]

[**SORTED** BY key-1[,key-2...]]

   where each sort-key is in the form:

$\begin{bmatrix} \textbf{ASC}[\text{ENDING}] \\ \textbf{DESC}[\text{ENDING}] \\ \textbf{INCREASING} \\ \textbf{DECREASING} \end{bmatrix}$ field-name-1

A SAMPLE DATATRIEVE-11 SESSION

Data Definition

Before DATATRIEVE-11 can access a file, a record structure and do-main name must be defined for that file. The record structure, which resembles a COBOL data definition, describes the format of the records in the file. The user can examine the record structure for the sample file by typing SHOW YACHT.

```
QL>SHOW YACHT
RECORD YACHT
 USING
01 BOAT.
   03 TYPE.
      06 MANUFACTURER PIC X(10)
         QUERY-NAME IS BUILDER.
      06 MODEL PIC X(10).
   03 SPECIFICATIONS
      QUERY-NAME SPECS.
      06 RIG PIC X(6).
      06 LENGTH-OVER-ALL PIC XXX
         QUERY-NAME IS LOA.
      06 DISPLACEMENT PIC 99999
         QUERY-HEADER IS "WEIGHT"
         EDIT-STRING IS ZZ,ZZ9
         QUERY-NAME IS DISP.
      06 BEAM PIC 99.
      06 PRICE PIC 99999
         EDIT-STRING IS $$$,$$$ .;
```

DATATRIEVE-11 always refers to a file by its domain name. The do-main name can be the same as the file name, but need not be. Often the domain name is easier to remember than the file name. For exam-ple, our sample file is named YACHT.DAT, but its domain name is BOATS.

Query Facility

When ready to use DATATRIEVE-11, the user issues the READY command to ready BOATS for access. When the prompt appears, the user can begin using SYSTEM commands. If a command error occurs, DATATRIEVE-11 responds with an error message and reprints its prompt; the user then simply types the command correctly.

QL>READY BOATS
QL>

FIND

To find records with certain characteristics, the user employs the FIND command. The FIND command creates a collection (i.e., a group of records) comprising all records found with the characteristics specified. The collection thus created becomes the basis for further commands until a different collection is established.

To create a collection of boats with price greater than $30,000, the user would proceed as in the example. Note that the greater-than sign (>) could have been used instead of GT.

QL>FIND BOATS WITH PRICE GT 30000
[17 RECORDS FOUND]

PRINT ALL

The user types PRINT ALL to print the collection of records created by the FIND command. PRINT ALL prints the current collection on the terminal. To print the collection on the line printer or on another terminal, the user specifies that device name. For example, PRINT ALL ON LPO: prints the collection on the line printer unit zero. To create a file containing the collection, use PRINT ALL and replace the device name with a file specification.

QL>PRINT ALL

MANUFACTURER	MODEL	RIG	LENGTH OVER ALL	WEIGHT	BEAM	PRICE
ALBERG	37 MK II	KETCH	37	20,000	12	$36,951
BAYFIELD	30/32	SLOOP	32	9,500	10	$32,875
CARIBBEAN	35	SLOOP	35	18,000	11	$37,850
CHALLENGER	32	SLOOP	32	12,800	11	$31,835

SORT

After establishing a collection, the user can use the SORT command to rearrange it. The SORT command can sort records by one or more fields in ascending or descending order. If more than one field is

337

specified, DATATRIEVE-11 sorts all records in the collection by the first field specified, sorts for the second field within categories of the first field, and so forth.

```
QL>SORT BY ASCENDING LOA,PRICE
QL>PRINT ALL
```

			LENGTH OVER			
MANUFACTURER	MODEL	RIG	ALL	WEIGHT	BEAM	PRICE
RYDER	S.CROSS	SLOOP	31	13,600	00	$32,500
CHALLENGER	32	SLOOP	32	12,800	11	$31,835
BAYFIELD	30/32	SLOOP	32	9,500	10	$32,857
WRIGHT	S'WND II	SLOOP	32	14,900	00	$34,480

A sort of the current collection by ascending length overall (LOA) and price rearranges the collection first into ascending order by length overall, then in order of increasing price for each length. All 32-foot models, for example, are in order of ascending price.

SELECT

A single record can be examined with the SELECT command. To get the first, last, or next record in the collection, the user simply types SELECT FIRST, SELECT LAST, or SELECT NEXT. The SELECT command also accepts an integer, as in SELECT 7, which retrieves the seventh record in the current collection.

```
QL>SELECT FIRST
```

			LENGTH OVER			
MANUFACTURER	MODEL	RIG	ALL	WEIGHT	BEAM	PRICE
RYDER	S. CROSS	SLOOP	31	13,600	00	$32,500

MODIFY

The MODIFY function is used to change information in a collection. First, the user types READY BOATS MODIFY to ready the file for modification, then uses the MODIFY command itself. DATATRIEVE-11 grants MODIFY access only to users whom the system manager or data base administrator has approved. With the MODIFY command, users name the field(s) whose values they want to change. DATA-TRIEVE-11 automatically prompts for the new value for the specified field(s). The entire collection can be modified so that the new value is in effect in every record, or it is possible to modify one record at a time (after SELECTing the record).

```
QL>READY BOATS MODIFY
QL>MODIFY BEAM
PLEASE SUPPLY VALUE FOR BEAM: 11
QL>PRINT
```

MANUFACTURER	MODEL	RIG	LENGTH OVER ALL	WEIGHT	BEAM	PRICE
RYDER	S. CROSS	SLOOP	31	13,600	11	$32,500

STORE

DATATRIEVE-11 also provides a STORE command that allows the file to be updated by adding new records. To use the STORE command, the user first opens BOATS for writing by typing READY BOATS WRITE. The STORE command can then be issued.

```
QL>READY BOATS WIRE;
QL>STORE BOATS
PLEASE SUPPLY VALUE FOR MANUFACTURER: CAPE DORY
PLEASE SUPPLY VALUE FOR MODEL: 25
PLEASE SUPPLY VALUE FOR RIG: SLOOP
PLEASE SUPPLY VALUE FOR LENGTH-OVER-ALL: 25
PLEASE SUPPLY VALUE FOR DISPLACEMENT: 4000
PLEASE SUPPLY VALUE FOR BEAM: 7
PLEASE SUPPLY VALUE FOR PRICE: 8995
QL>FIND BOATS WITH PRICE EQ 8995
[1 RECORD FOUND]
QL>PRINT ALL
```

MANUFACTURER	MODEL	RIG	LENGTH OVER ALL	WEIGHT	BEAM	PRICE
CAPE DORY	25	SLOOP	25	4,000	07	$8,995

DATATRIEVE-11 grants WRITE access only to users whom the system manager or data base administrator has approved. In response to the STORE command, the system prompts for values for each field defined in the record structure.

PRODUCING FORMATTED REPORTS

REPORT

The REPORT statement is the first statement in any report request. It invokes the report facility and specifies the collection from which the report will be extracted. To generate a report on a collection already established, a user issues the REPORT statement without a collection name. The default output device for reports is the terminal. To direct output to another device or file, the file specification is included in the REPORT statement.

SET

The SET statement names format items for the report. Using the SET statement, the user can specify

1. the name of the report

2. the number of lines and columns per page

3. the inclusion of page number and date

4. the number of lines or pages beyond which the report should be terminated automatically

All format items specified by the SET statement (except the report name) have default values.

PRINT

The PRINT statement in the report facility is an extension of the PRINT command in the query facility. Using the PRINT statement, the user can specify the content and format of each line in the report. Each PRINT statement produces one entry in the report for each record in the collection.

AT

The AT statement includes summary line information. With the AT statement, the user can control the printing of headers at the tops of pages and reports, and establish starting conditions for new pages and reports. The AT statement also can invoke the computation utility, which generates total and subtotals.

REPORT END
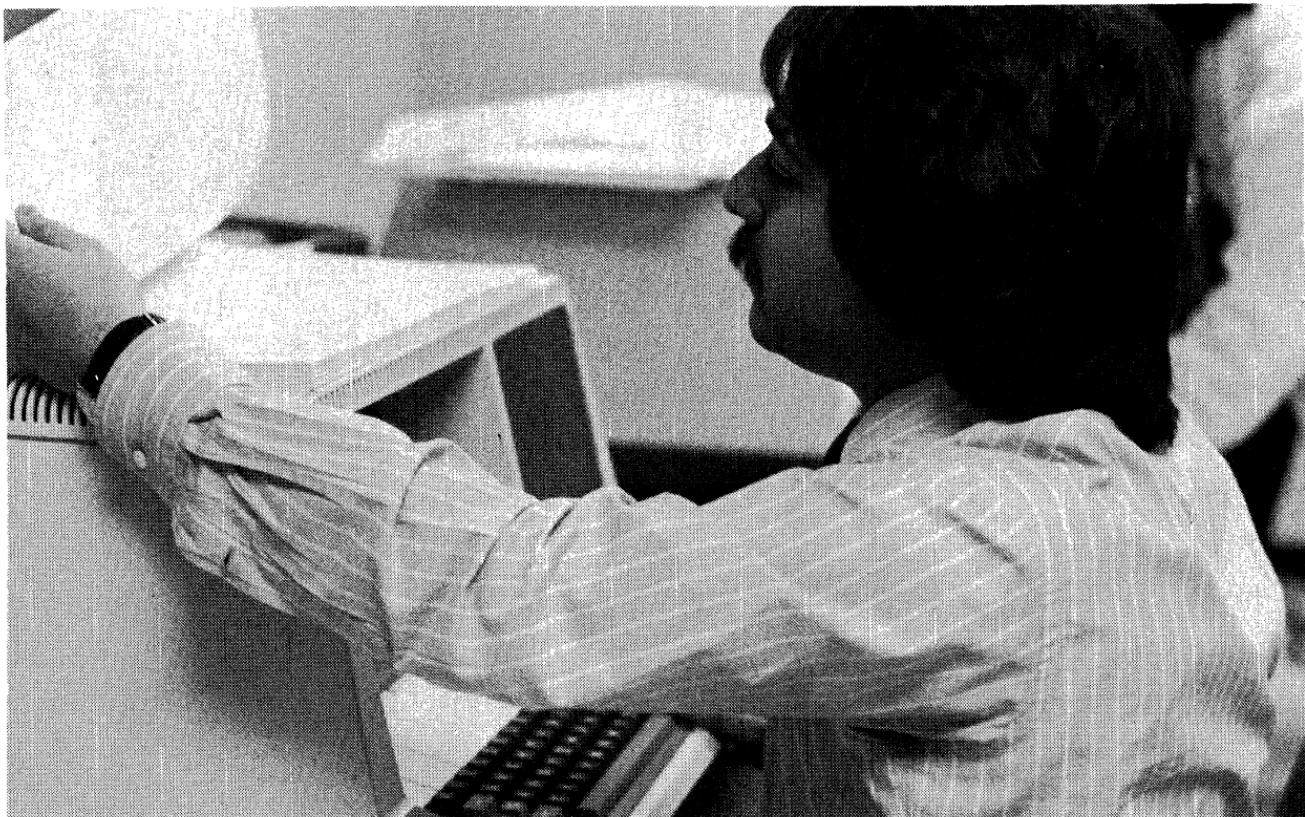
The REPORT END statement is the last statement in a report request. It ends the report specification and returns to DATATRIEVE-11 command level.

QUIT

When the user has finished using DATATRIEVE-11 on a file, he types QUIT to return to the command level of the host operating system.

languages

MACRO-11

OVERVIEW

MACRO-11 is a powerful assembly language which processes source programs and produces a relocatable object module. It has extensive macro features which allow a programmer to code directly and efficiently in machine assembly language. MACRO-11 also provides for direct access to hardware features of the system. And many PDP-11 operating systems have MACRO-11 bundled in to handle program assembly functions.

FEATURE TOPICS

- Functions and Features
- Language
- Symbols and Symbol Definitions
- Directives
  - Listing Control Directives
  - Function Directives
  - Data Storage Directives
  - Program Sectioning Directives
  - Symbol Control Directives
  - Conditional Assembly Directives
- MACRO Definitions and Repeat Blocks
- MACRO Calls and Structured MACRO Libraries
- Assembler Operation
- Assembler Operating System Environments

FUNCTIONS AND FEATURES

PDP-11 MACRO processes source programs written in the MACRO assembly language and produces a relocatable object module and optional assembly listing. MACRO is included with the RT-11, RSX-11D, RSX-11M, IAS, and RSTS/E operating systems.

MACRO provides the following features:

* relocatable object modules
* global symbols for linking separately assembled object programs
* device and file name specifications for input and output files
* user-defined macros
* comprehensive system macro library
* program sectioning directives
* conditional assembly directives
* assembly and listing control functions at program and command string levels
* alphabetized, formatted symbol table listing
* default error listing on command output device
* a Cross Reference Table (CREF) symbol listing

The MACRO assembler included in the RSX-11 and IAS systems also features:

* global arithmetic, global assignment operator, global label operator, and default global declarations
* multiple macro libraries with fast access structure
* predefined (default) register definitions
* an indirect command file facility for controlling the assembly process

LANGUAGE

A MACRO source program is composed of a sequence of source coding lines. Each source line contains a single assembly language statement followed by a statement terminator, such as a carriage return. The assembler processes source statements sequentially, generating binary machine instructions and data words or performing assembly-time operations (such as macro expansions) for each statement.

A statement can contain up to four fields, which are identified by order of appearance and by specified terminating characters. The general format of a MACRO assembly language statement is:

      label: operator   operand(s)   ;comments

The label and comment fields are optional. The operator and operand fields are interdependent; either can be be omitted depending on the contents of the other. Some statements have one operand, for example:

    CLR   R0

while others have two:

    MOV   #344,R2

A label is a unique user-defined symbol which is assigned the current location counter and entered into the user-defined symbol table. A label is a symbolic means of referring to a specific location within a program. The value of the label can be either absolute (fixed in memory independent of the position of the program) or relocatable (not fixed in memory), depending on whether the location counter value is currently absolute or relocatable.

An operator field follows the label field, if present, and can contain a macro call, a PDP-11 instruction mnemonic, or an assembler directive. When the operator is a macro call, the assembler inserts the appropriate code during assembly to expand the macro. When the operator is an instruction mnemonic, it specifies the instruction to be generated and the action to be performed on any operands which follow. When the operator is an assembler directive, it specifies a certain function or action to be performed during assembly.

An operand is that part of the statement manipulated by the operator. Operands can be expressions, numbers, symbolic arguments, or macro arguments.

The comment field can contain any ASCII text characters. Comments do not affect assembly processing or program execution, but are useful in source listings for later analysis, documentation, or debugging purposes.

Symbols and Symbol Definitions

Three types of symbols can be defined for use within MACRO source programs: permanent symbols, user-defined symbols, and macro symbols. Correspondingly, MACRO maintains three types of symbol tables: the Permanent Symbol Table (PST), the User Symbol Table (UST), and the Macro Symbol Table (MST).

Permanent symbols consist of the PDP-11 instruction mnemonics and MACRO directives. The PST contains all the permanent symbols automatically recognized by MACRO and is part of the assembler itself. Since these symbols are permanent, they do not have to be defined by the user in the source program.

User-defined symbols are those used as labels or defined by direct assignment. Macro symbols are those symbols used as macro names. The UST and MST are constructed during assembly by adding the symbols to the UST or MST as they are encountered.

The value of a symbol depends on its use in the program. A symbol in the operator field can be a macro symbol, a user-defined symbol, or a permanent symbol. To determine the value of the symbol, the assembler searches the three symbol tables in the following order: MST, UST, and PST.

A symbol used in the operand field can be either a user-defined symbol or a permanent symbol. To determine the value of the symbol, the assembler searches the User Symbol Table and the Permanent Symbol Table in that order.

These search orders allow redefinition of Permanent Symbol Table entries as user-defined or macro symbols. The same name can be assigned to both a macro and a label.

User-defined symbols are either internal or external (global) to a source program module. An internal symbol definition is limited to the module in which it appears. A global symbol can be defined in one source program module and referenced within another.

Internal symbols are temporary definitions which are resolved by the assembler. Global symbols are preserved in the object module and are not resolved until the object modules are linked into an executable program. With some exceptions, all user-defined symbols are internal unless explicitly defined as being global.

When a label is given to a program statement, a symbol table entry is made and the value of the current location counter is assigned to it. A label can be defined as a global symbol by ending the label name with two colons instead of one.

A direct assignment statement associates a symbol with a value. When a direct assignment is first used to define a symbol, that symbol is entered into the User-defined Symbol Table, and the specified value is associated with it. The general format for a direct assignment is:

symbol = expression

A symbol can be defined as a global symbol by separating the symbol from the expression with two equal signs instead of just one. That is, the statement:

symbol = = expression

automatically declares the symbol as a global symbol.

Expressions are combinations of terms that are joined together by binary operators and that reduce to a 16-bit value. Binary operators are, for example, addition, subtraction, multiplication, division, logical AND, and logical inclusive OR.

The expression in a direct assignment statement can itself be a reference to another symbol. In this way, a symbol can be redefined in a subsequent direct assignment statement if the symbol definition contains a reference to a subsequently defined symbol. Only one level of forward referencing is allowed. The following example illustrates an illegal forward reference:

X=Y (Illegal forward reference)
Y=Z (Legal forward reference)
Z=1

Although one level of forward referencing is allowed for local symbols, a global symbol defined in a direct assignment statement must not contain a forward reference. The global assignment expression (==) must not itself contain an undefined reference to another symbol.

Local symbols are specially formatted internal symbols used as labels within a given range of source code, called a local symbol block. Local symbols are of the form n$, where n is a decimal integer between 1 and 65535, inclusive. Examples of local symbols are: 1$, 27$, 59$, 104$.

A local symbol block can be delimited in one of three ways:

- The range of a local symbol block usually consists of those statements between two normally defined labels.

- The range of a local symbol block is normally terminated upon entering a new program section, as defined by a program section directive.

- The range of a local symbol block can be explicitly defined by the use of the .ENABL and .DSABL directives.

Local symbols provide a convenient means of generating labels to be referenced only within a local symbol block. The use of local symbols reduces the possibility of entry point symbols with multiple definitions appearing within a program. A local symbol, then, is not referenced from other source program modules or even from outside its local symbol block. Thus, local symbols of the same name can appear in other local symbol blocks without conflict.

347

Directives

A program statement can contain one of three different operators: a macro call, a PDP-11 instruction mnemonic, or an assembler directive. MACRO includes directives for:

- listing control
- function specification
- data storage
- radix and numeric usage declarations
- location counter control
- program termination
- program boundaries information
- program sectioning
- global symbol definition
- conditional assembly
- macro definition
- macro attributes
- macro message control
- repeat block definition
- macro libraries

Table 15-1 lists the MACRO directives.

Table 15-1 Assembly and Macro Directives

Listing Control Directives

.LIST	Controls the listing of source lines, sequence numbers.
.NLIST	Current location counter field, generated binary code, source code, comments, macro expansions, table of contents, symbol table, etc.
.TITLE	Assigns a name to the object module and provides the header of each page in the assembly listing.
.SBTTL	Identifies an element to be included in the assembly listing table of contents.
.IDENT	Provides an additional label for the object module.
.PAGE	Ejects a page in the assembly listing. Same as issuing a form feed.

Function Directives

.ENABL   Enables or disables the following function control.

.DSABL   Options: produce absolute binary output, assemble all relative addresses as absolute addresses (useful during debugging), ignore card column sequence numbers, truncate or round floating-point values, accept lower case input, permit a local symbol block to cross normal boundaries, inhibit binary output. In addition, RSX-11/IAS MACRO provides two additional function control options: inhibit the default register definitions, treat all undefined symbol references as default global references.

Data Storage Directives

.BLKB   Reserves a byte- or word-aligned block of storage in

.BLKW   the object program.

.BYTE   Stores a binary value in a byte in the object module. Used to generate successive bytes of data.

.WORD   Stores a binary value in a word in the object module. Used to generate successive words of data.

Stores the ASCII code(s) for the given character(s) following the apostrophe or quote in a byte or word. Used to generate text characters in the source code.

.ASCII   Translates a character string into its equivalent 7-bit ASCII values and stores them in the object module.

.ASCIZ   Translates a character string into its equivalent 7-bit ASCII values and stores them in the object module appending a zero byte to the string. This enables the program to identify the end of the string by searching for a null character (zero byte).

.RAD50   Allows three ASCII characters to be packed into one word (Radix-50 format); using this directive, any 6-character symbol can be stored in two consecutive words.

.FLT2   Stores a floating point number in 2-word floating point format.

.FLT4   Stores a floating point number in 4-word floating point format.

Radix and Numeric Control Operators

.RADIX	Declares any one of the following radices to apply to succeeding numbers in the source program: 2, 4, 8 or 10.
↑D, ↑O, ↑B	Declares a (temporary) decimal, octal or binary radix for the number following the control operator.
↑C	Declares that the number following the control operator is to be 1's complemented as it is evaluated during assembly.
↑F	Declares that the number following the control operator is to be interpreted as a 1-word floating point argument.
↑R	Declares that the three characters following the control operator are to be evaluated as a Radix-50 value.

Location Counter Control Directives

.EVEN	Ensures that the current location counter contains an even value by adding one if the current value is odd.
.ODD	Ensures that the current location counter contains an odd value by adding one if the current value is even.

Terminating Directives

.END	Indicates the logical end of source input and, optionally, specifies the entry point, starting or transfer address.
.EOT	End of input tape. Ignored by the assembler (included for compatibility with PAL-11 assemblers).

Program Boundaries Directive

.LIMIT	Reserves two words in the object module which, during linking, are used to store the address of the bottom of the program and the address of the first free word following the program image. This enables the program to determine its upper and lower address boundaries during execution.

Program Sectioning Directives

.PSECT   Begins or continues a program section with speci-
fied attributes. The attributes are interpreted by the
Task Builder (or Linker) to establish and control the
memory allocation of a program. The attributes are:
read-only or read/write access; contains instructions
or data; local or global program section; absolute or
relocatable program section; concatenated or over-
laid program section.

.ASECT   Begins or continues the absolute program section.
This directive is interpreted as a .PSECT directive
with the following default attributes: allow read/write
access; contains instructions; global program sec-
tion; absolute (non-relocatable); overlaid program
section.

.CSECT   Begins or continues a relocatable program section.
This directive is interpreted as a .PSECT directive
with the following default attributes if the program
section is named: allow read/write access; contains
instructions; global program section; relocatable;
overlaid program section. The following default at-
tributes are used if the .CSECT is unnamed: allow
read/write access; contains instructions; local pro-
gram section; relocatable; concatenated program
section.

Symbol Control Directive

.GLOBL   Defines (and thus provides linkage to) symbols not
otherwise defined as global symbols within a mod-
ule. (Global symbols can also be defined using the
global assignment operator (==), the global label
operator (::) or, by default reference, if a symbol is
not defined by the end of assembly pass 1.)

Conditional Assembly Directives

.IF   If the condition specified in the argument is met, in-
                cludes the following block of code in the assembly.
                Condition testing can be based on the value of an
                expression, the existence of a definition for a sym-
                bol, or the value of a macro-type argument.

.ENDC   Identifies the end of the conditional assembly block.

.IFF   The code following this subconditional directive, and
                continuing up to the next occurrence of a subcondi-
                tional directive or to the end of the conditional as-
                sembly block, is to be included in the program,
                providing that the condition tested upon entering the
                conditional assembly block is false.

.IFT   The code following this subconditional directive, and
                continuing up to the next occurrence of a subcondi-
                tional directive or to the end of the conditional as-
                sembly block, is to be included in the program, pro-
                vided that the condition tested upon entering the
                conditional assembly block is true.

.IFTF   The code following this subconditional directive, and
                continuing up to the next occurrence of a subcondi-
                tional directive or to the end of the conditional as-
                sembly block, is to be included in the program, re-
                gardless of the result of the condition tested upon
                entering the conditional assembly block.

.IIF   Assembles this line of code if the condition specified
                on the line is met.

Macro Definition Directives

.MACRO   Identifies the beginning of a macro definition.

.ENDM   Identifies the end of a macro definition.

.MEXIT   Terminates a macro expansion before the end of the
                macro is encountered.

Macro Attribute Directives

.NARG   Determines the number of arguments in the macro call currently being expanded.

.NCHR   Determines the number of characters in a specified character string. It is useful in calculating the length of macro arguments.

.NTYPE   Determines the addressing mode of a specified macro argument.

Macro Message Control Directives

.ERROR   Sends a message to the listing file during assembly pass 2. A common use of this directive is to provide a diagnostic announcement of a rejected or erroneous macro call or to alert the user to the existence of an illegal set of conditions specified in a conditional assembly.

.PRINT   Identical to the .ERROR directive, except that it is not flagged in the assembly with an error code.

Macro Repeat Block Directives

.IRP   Replaces a dummy argument with successive real arguments specified in an argument string.

.IRPC   Replaces a dummy argument with each successive character of the specified string.

.REPT   Duplicates a block of code a number of times in line with other source code.

Macro Library Directive

.MCALL   Includes in the assembly macro definitions which are taken from system or user macro library files.

LISTING CONTROL DIRECTIVES

Several listing control directives are provided in MACRO to control the content, format, and pagination of all listing output generated during assembly. Facilities also exist for creating object module names and other identification information in the listing output.

The listing control options can also be specified at assembly time through switch options included in the listing file specification in the command string issued to the MACRO assembler. The use of these switch options overrides all corresponding listing control directives in the source program.

When no listing file is specified, any errors encountered in the source program are printed on the terminal from which MACRO was initiated.

FUNCTION DIRECTIVES

Several function control options are provided by MACRO through the .ENABL and .DSABL directives. These directives are included in a source program to invoke or inhibit certain MACRO functions and operations incident to the assembly process itself. They include the ability to:

- Produce absolute binary output.

- Assemble all relative addresses as absolute addresses. This function is useful during the debugging phase of program development.

- Cause source columns 73 and greater (to the end of the line) to be treated as comment. The most common use of this feature is to permit sequence numbers in card columns 73-80.

- Truncate or round floating point literals.

- Accept lower case ASCII input instead of converting it to upper case.

- Enable a local symbol block to cross program section boundaries. A local symbol block is normally established by encountering a new symbolic label or a program section directive in the source program. By enabling a local symbol block to cross program section boundaries, a local symbol block can be established which is not terminated until another symbolic label or program section directive is encountered following a disable local symbol block function directive. Local symbols cannot, however, be defined in a program section other than that which was in effect when the block was entered. The basic function of this directive in regard to program sections is limited to those instances where it is desirable to leave a program section temporarily to store data, followed by a return to the original program section.

- Inhibit binary output.

• Inhibit the normal default register definitions.

• Treat all undefined symbol references as default global references.

CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly directives enable the programmer to include or exclude blocks of source code during the assembly process, based on the evaluation of stated condition tests within the body of the program. This capability allows several variations of a program to be generated from the same source.

The user can define a conditional assembly block of code, and within that block, issue subconditional directives. Subconditional directives within conditional assembly blocks are used to indicate:

• The assembly of an alternate body of code when the condition of the block tests false.

• The assembly of a non-contiguous body of code within the conditional assembly block, depending on the result of the conditional test on entering the block.

• The unconditional assembly of a body of code within a conditional assembly block.

Conditional assembly directives can be nested. MACRO permits a nesting depth of 16 conditional assembly levels.

MACRO DEFINITIONS AND REPEAT BLOCKS

In assembly-language programming, it is often convenient and desirable to generate a recurring coding sequence by invoking a single statement within the program. In order to do this, the desired coding sequence is first established with dummy arguments as a macro definition. Once a macro has been defined, a single statement calling the macro by name with a list of real arguments (replacing the corresponding dummy arguments in the macro definition) generates the desired coding sequence or macro expansion.

Macros can be nested; that is, the definition of one macro can include a call to another. The depth of nesting allowed is dependent on the amount of memory used by the source program being assembled.

A label is often required in an expanded macro. Normally, a label can be explicitly specified as an argument with each macro call. Care must be taken, however, in issuing subsequent calls to the same macro in order to avoid specifying a duplicate label as a real argument. This concern is eliminated through a feature of MACRO which creates a unique symbol where a label is required in an expanded macro.

MACRO can automatically create unique local symbols. This automatic facility is invoked on each call of a macro whose definition contains

a dummy argument preceded by the question mark (?) character, if a real argument of the macro call is either null or missing. If the real argument is specified in the macro call, however, MACRO does not generate a local symbol and normal argument replacement occurs.

Macro call arguments may be specified as positional or keyword. Use of the keyword feature requires that the corresponding dummy argument name be known and specified exactly. In addition, the keyword construction may be used when defining a macro to specify explicit default values for the macro arguments.

An indefinite repeat block is a structure that is very similar to a macro definition. Such a structure is essentially a macro definition that has only one dummy argument. At each expansion of the indefinite repeat range, this dummy argument is replaced with successive elements of a specified real argument list. An indefinite repeat block directive and its associated repeat range are coded in-line within the source program. This type of macro definition does not require calling the macro by name, as required in the expansion of conventional macros described above.

An indefinite repeat block can appear within or outside of another macro definition, indefinite repeat block, or repeat block.

MACRO CALLS AND STRUCTURED MACRO LIBRARIES

All macro definitions must occur prior to their references within the user program. MACRO provides a selection mechanism for the programmer to indicate in advance those system macro definitions required in the program. (System macros include the monitor programmed requests or executive directives available with each operating system.)

The .MCALL directive is used to specify the names of all the macro definitions not defined in the current program but used in the program. When this directive is encountered, MACRO searches the system macro library file to find the requested definition.

MACRO extends this macro call facility by enabling the programmer to retrieve macros from libraries of user-defined macros. The .MCALL directive provides the means to access both user-defined and system macro libraries during assembly.

The MACRO assembler assumes that the system macro library and user-defined macro libraries are constructed in a special direct access format to retrieve macro definitions quickly. These structured macro libraries are created by the Librarian utility program.

Each library file contains an index of the macro definitions it contains. When an .MCALL directive is encountered in the source program,

MACRO searches the user macro library for the named macro defini-
tions, and, if necessary, continues the search with the system macro
library. Because each macro library contains an index of all of its
entries, MACRO searches only the index in each library to find where
the macro definition is stored.

ASSEMBLER OPERATION

The MACRO Assembler assembles one or more ASCII source files
containing MACRO statements into a single relocatable binary object
program. MACRO can accept source data from any input device, such
as a disk or card reader. The sources to be included in a single assem-
bly are listed in the command string from left to right in the order in
which they are to be assembled. The last statement in the last source
specified is normally the .END statement. If the .END statement is not
provided, it is assumed.

Assembler output consists of the binary object file and an optional
assembly listing followed by the symbol table listing and a cross
reference listing.

MACRO is a two-pass assembler. During assembly pass one, MACRO
locates and reads all required macros from libraries, builds symbol
tables and program section tables for the program, and performs a
rudimentary assembly of each source statement. During assembly
pass two, MACRO completes the assembly, writes out an object file,
and generates an assembly and symbol table listing for the program.

At the end of assembly pass one, MACRO determines whether a given
global symbol is defined in the current program modules or whether it
is to be treated as an external symbol. In general, all undefined global
symbols appearing in a given program must be defined by the end of
assembly pass one. All symbols remaining undefined at the end of
assembly pass one are assumed to be default global references.

The object module MACRO produces must be processed by the oper-
ating system's linker utility program (called the Linker or Task Builder)
to create an executable program. The linker joins separately assem-
bled object modules into a single load module (or task image). The
linker fixes (makes absolute) the values of the external or relocatable
symbols in the object module.

To enable the linker to fix the value of an expression, MACRO passes it
certain directives and parameters. In the case of the relocatable
expressions in the object module, the linker adds the base of the
associated relocatable program section to the value of the relocatable
expression provided by MACRO. In the case of external expression
values, the linker determines the value of the external term in the

expression (since the external expression must be defined in at least one of the other object modules being linked together) and then adds it to the absolute portion of the external expression, as provided by MACRO.

In summary, an executable program image can be constructed from one or more source modules, which can be assembled either separately or together. The resultant object module(s) must be linked together using the linker utility. Figure 15-1 illustrates the processing steps required to produce an executable program from several sources stored as files.
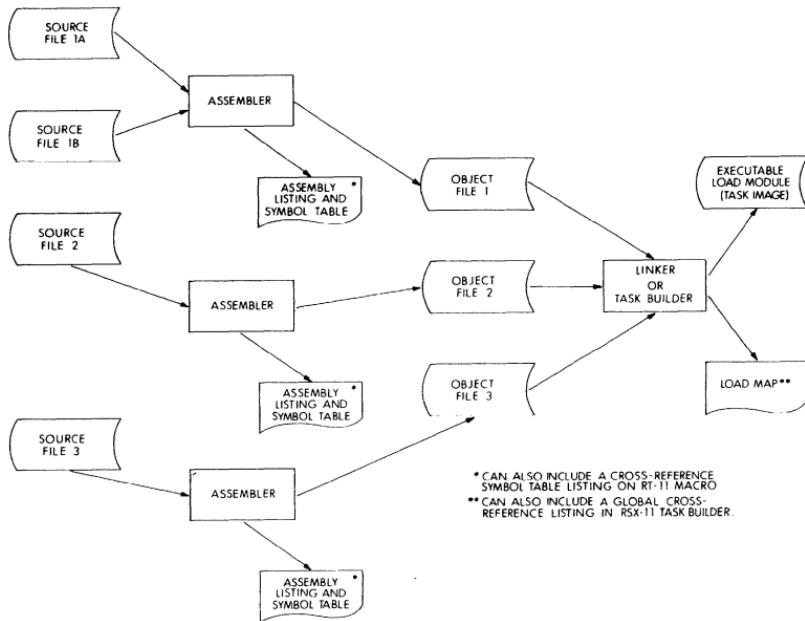
Figure 15-1 MACRO Assembly Procedure

PROGRAM SECTIONING

The MACRO program sectioning directives are used to declare names for program sections and to establish certain program section attributes. These program section attributes are used when the program is linked into an executable load module or task.

A program can consist of an absolute program section, an unnamed relocatable program section, and up to 254 named relocatable program sections. The absolute program section serves to link the program with fixed memory locations such as interrupt vectors and the peripheral device register addresses.

The relocatable program sections are also called control sections, since they normally contain instructions. The unnamed control section is internal to each object module. That is, every object module can have an unnamed control section, but the linker treats each control section independently. Each is assigned an absolute address such that it occupies an exclusive area of memory. Named control sections, on the other hand, are treated globally, in the same manner as FORTRAN COMMON.* If different object modules have control sections with the same name, they are all assigned the same absolute load address, and the size of the area reserved for loading of the section is the size of the largest. Thus, named control sections allow for the sharing of data and/or instructions among object modules.

* If declared with the .PSECT directive (see below), they must have the attributes global and overlaid.

The assembler maintains separate location counters for each section. The first occurrence of a program section directive assumes that the current location counter is set at relocatable zero. The scope of this directive then extends until a directive declaring a different program section is specified. For example:

```
        .CSECT                  ;start the unnamed relocatable
                                ;section
A:      0                       ;assembled at relocatable 0,
B:      0                       ;relocatable 2 and
C:      0                       ;relocatable 4.
ST:     CLR<HT>A                ;assemble code at
        CLR<HT>B                ;relocatable address
        CLR<HT>C                ;6 through 12
        .ASECT                  ;start the absolute section
        .=4                     ;assemble code at
        .WORD .+2,HALT          ;absolute locations 4 through 7
        .CSECT                  ;resume the unnamed relocatable
                                ;section
        INC A                   ;assemble code at
        BR ST                   ;relocatable 22 through 27
        .END
```

By maintaining separate location counters for each program section, MACRO allows the user to write statements which are not physically

contiguous within the program, but which can be loaded contiguously following assembly.

MACRO includes an additional program sectioning directive, .PSECT, because of the unique nature of the linker utility program, the Task Builder. Assembly language programs can use the .PSECT directive exclusively, since it affords all the capabilities of the .CSECT and .ASECT directives. MACRO will accept .ASECT and .CSECT directives, but assembles them as if they were .PSECT directives with implicit default attributes.

The .PSECT directive allows the user to exercise absolute control over the memory allocation of a program at task-build time, since any program attributes established through this directive are passed to the Task Builder. For example, if a programmer is writing programs for a multi-user environment, a program section containing pure code (instructions only) or a program section containing impure code (data only) can be explicitly declared through the .PSECT directive. Furthermore, these program sections can be explicitly declared as read-only code, qualifying them for use as protected, reentrant programs. In addition, program sections exhibiting the global attribute can be explicitly allocated in a task's overlay structure by the user at task-build time. The advantages gained through sectioning programs in this manner therefore relate primarily to control of memory allocation, program modularity, and more effective partitioning of memory.

The .PSECT directive allows the user to define the following program section attributes:

Access
Two types of access can be permitted to the program section: read-only or read/write. RSX-11D and IAS support read-only access by setting hardware protection for the program section.

Contents
A program section can contain either instructions or data. This attribute allows the Task Builder to differentiate global symbols that are program entry-point instructions from those that are data values.

Scope
The scope of the program section can be global or local. In building single-segment programs, the scope of the program has no meaning, because the total memory allocation for the program will go into the root segment of the task. The global or local attribute is significant only in the case of overlays. If an object module contains a local program section, then the storage allocation for that module will occur within the segment in which the module resides. Many modules can refer-

ence this same program section, and the memory allocation for each module is either concatenated or overlaid within the segment, depending on the argument of the program section defining its allocation requirements (see below). If an object module contains a global program section, the memory area allocations to this program section are collected across segment boundaries, and the allocation of memory for that section will go into the segment nearest the root in which the first memory allocation to this program section appeared.

Relocatability

A program section can be absolute or relocatable. When a program section is declared to be absolute, the program section requires no relocation. The program section is assembled and loaded, starting at absolute virtual address 0. When the program section is declared to be relocatable, the Task Builder calculates a relocation bias and adds to it all references within the program section.

Allocation Requirements

The program section can be concatenated or overlaid. When concatenated, all memory allocations to the program section are to be concatenated with other references to this same program section in order to determine the total memory allocation requirements for this program section. When overlaid, all memory allocations to the program section are to be overlaid. Thus, the total allocation requirement for the program section is equal to the largest individual allocation request for this program section.

ASSEMBLER ENVIRONMENTS

MACRO is the assembler included in the RT-11, RSX-11, and IAS operating systems. IAS MACRO is identical to the MACRO assembler used in the RSX-11 systems. In addition, MACRO is included in the FORTRAN IV package option available for the RSTS/E system.

The following sections summarize the operating environments of the RT-11 and RSX-11 MACRO products.

UNDER RT-11

MACRO requires an RT-11 system configuration (or background partition) of 12K words or more. If more than 12K words are available, MACRO will use the additional memory to increase the amount of symbol table space possible.

RT-11 MACRO provides a system macro library containing the expanded code for all the RT-11 monitor's programmed requests. Refer to the RT-11 chapter in Section II of this handbook for a list of the RT-11 programmed requests.

Under the RT-11 operating system, a smaller version of MACRO, called ASEMBL, is available for users with minimum system configurations. ASEMBL has the same features as MACRO with the following exceptions:

- Macro directives (.MACRO, .MCALL, .ENDM, .IRP, .PSECT etc.) are not recognized.
- DATE is not printed in listings.
- Wide line-printer output is not available.
- There is no lower case mode.
- There is no enable/disable punch directive.
- There are no floating point directives.
- There are no local symbols or local symbol blocks.
- CREF is not available.

Most of the macro definition features not available in ASEMBL are supported by EXPAND. EXPAND is an RT-11 system program which processes the macro references in a macro assembly language source file.

EXPAND simply copies its input files to its output file unless it encounters any of the following directives: .MCALL, .MACRO, .name, and .ENDM. The .MCALL directive instructs EXPAND to search the system macro library to find the macro names listed in the directive, and store their definition in internal tables. The .MACRO directive instructs EXPAND to copy a macro definition from the user's input file to its internal tables. The .name directive, if name is the name of a macro, instructs EXPAND to replace the macro call with the definition stored in its internal tables. The .ENDM directive terminates the macro definition when encountered while EXPAND stores a macro definition.

UNDER RSX-11
MACRO requires a minimum of 14K words of partition space to execute. The system macro library includes executive directives and file system calls. Refer to the RSX-11 chapter for a description of the executive directives and file system calls.

Under the RSX-11M system, an 8K word version is available for users who have limited memory space. The 8K version differs from the 14K version in the following ways:

- It does not search the Permanent Symbol Table for symbols appearing in the operand field of a statement.
- It does not recognize the following .ENABL/.DSABL directive function control options: produce absolute binary output, ignore card

column sequence numbers, truncate floating point values, accept lower case input, inhibit binary output.

- It does not recognize or accept the PAL-11R conditional assembly directives and .EOT directive.

- It does not flag in the assembly listing the instructions which are not common among all members of the PDP-11 family.

- It does not accept floating point directives or control operators (.FLT2, .FLT4, ↑F, etc.), or PDP-11/45 or PDP-11/70 floating point opcodes.

BASIC (V2)

OVERVIEW

BASIC is an easy-to-learn, conversational programming language that uses simple English words, abbreviations, and familiar mathematical symbols to perform operations. DIGITAL's BASIC is an immediate response, interactive language compatible with Dartmouth standard BASIC. It gives the user the capability to develop and debug a program in a minimum amount of time. In addition, relatively large data processing tasks as well as quick, one-time calculations can be performed.

FEATURE TOPICS

- Functions and Features
- Language Elements
  - Constants and Variables
  - Operators
  - Statements
- Functions
  - Arithmetic Functions
  - String Functions
  - User-defined Functions
- Programming Example
- Graphics and Laboratory Peripherals Support
- BASIC Files
- Compiler Operation
  - Immediate Mode of Execution
- BASIC Operating System Environments

FUNCTIONS AND FEATURES
BASIC is an incremental compiler which provides immediate translation and storage of a program written in the BASIC source language, while it is being entered. A single-user BASIC system is available as an option on the RT-11, RSX-11M, RSX-11S and IAS operating systems (running shared under IAS). A multi-user BASIC system is available as an option for the RT-11 operating system.

BASIC provides the following features:

* incremental compiler for immediate source translation
* immediate mode for easy debugging and use as a desk calculator
* ASCII sequential files compatible with FORTRAN
* integer, string, and floating point virtual array files for random access
* dynamic allocation of string storage
* PRINT-USING statement for output formatting
* complete set of string manipulation functions
* user-defined functions
* programs that can be chained together pass data through common CALL statement for assembly language subroutines
* graphics and laboratory peripherals support

LANGUAGE
The BASIC language is a conversational programming language which uses simple English-type statements and familiar mathematical notation to perform operations. BASIC is one of the simplest computer languages to learn, and once learned, provides advanced techniques to perform intricate data manipulations and efficient problem expression.

A BASIC program is composed of lines of statements containing instructions to the BASIC compiler. Each line of the program begins with a number that identifies that line as a statement and indicates the order of statement execution relative to other lines in the program. Each statement starts with an English word specifying the type of operation to be performed.

All BASIC statements and computations must be written on a single line. Statements cannot be continued on a following line. More than one statement, however, can be written on a single line when each statement after the first is preceded by a backslash. For example,

```
10 INPUT A,B,C
```

is a single statement line, while

```
20 LET X = 11 \ PRINT X,Y,Z \ IF X = A THEN 10
```

is a multiple statement line containing three statements: LET, PRINT, and IF.

BASIC Language Elements
CONSTANTS AND VARIABLES

BASIC treats all numbers (real and integer) as decimal numbers. Numbers used must be in the approximate range 10^{-38} to 10^{38}.

In addition to real and integer formats, BASIC accepts exponential notation. Numeric data can be input in any one or all of these formats. BASIC automatically uses the most efficient format for printing a number, according to its size. It automatically suppresses leading and trailing zeros in integer and decimal numbers and formats all exponential numbers.

BASIC also processes information in the form of strings. A string is a sequence of alphabetic, numeric, or special characters treated as a unit. A string constant is a list of characters enclosed in quotes. A string constant can be used in the PRINT, CALL, and CHAIN statements. These uses of string constants are allowed in versions of BASIC that do not support strings.

In BASIC with string support, string constants can also be used to assign a value to a string variable, for example, in the LET and INPUT statements.

BASIC recognizes six types of variables: floating point, subscripted floating point, string, subscripted string, integer, and subscripted integer. A numeric variable is an algebraic symbol representing a number and is formed by a single letter or a letter optionally followed by a single digit. For example: I, B3, or X.

Subscripted variables provide additional computing capabilities for dealing with lists, tables, matrices, or any set of related variables. In BASIC, variables are allowed one or two subscripts. For example, a list might be described as A(I) where I goes from 0 to 5:

$$A(0), A(1), A(2), A(3), A(4), A(5).$$

This allows reference to each of the six elements in the list, and can be considered a one-dimensional algebraic matrix. Two-dimensional matrices are also allowed.

Any variable followed by a percent sign (%) indicates an integer variable. For example: A%, C7%, C%(5).

Any variable name followed by a dollar sign ($) character indicates a string variable. For example: A$, C7$. Any list or matrix variable name followed by the dollar sign character denotes the string form of that variable. For example: V$(n), M2$(n), C$(m,n), G1$(m,n).

The user can assign values to variables by indicating the values in a LET statement, by entering the value as data in an INPUT statement, or by a READ statement. The value assigned to a variable does not change until the next time a statement that contains a new value for that variable is encountered.

OPERATORS
BASIC performs addition, subtraction, multiplication, division and exponentiation. The five operators used in writing most familiar formulas are:

+	A + B	Add B to A
−	A − B	Subtract B from A
*	A * B	Multiply A by B
/	A / B	Divide A by B
↑	A↑B	Raise A to the Bth power

In addition, BASIC allows unary plus and minus arithmetic operators. For strings, the concatenation operator (+ or &) puts one string after another without any intervening characters.

Relational operators allow comparison of two values and are used to compare arithmetic expressions or strings in an IF-THEN statement. The relational operators are:

=	Equals (alphabetically equal)
<	Less than (alphabetically precedes)
< =	Less than or equals (precedes or equals)
>	Greater than (alphabetically follows)
> =	Greater than or equals (follows or equals)
< >	Not equals (not alphabetically equal)

STATEMENTS
The following summary of BASIC statements gives a brief explanation of each statement's use.

REM	Contains explanatory comments in a BASIC program.
LET	Assigns the value of an expression to the specified variable. Variable and expression must be of the same type.
DIM	Reserves space in memory for arrays according to the subscripts specified.

DATA	Used in conjunction with READ to input listed data into an executing program. Can contain any mixture of strings and numbers.
READ	Assigns values listed in DATA statements to the specified values. Variables can be numeric or string.
OPEN FOR INPUT [OUT-PUT] AS FILE	Opens a file for input (or output) and associates the file with the specified logical unit number.
INPUT	Reads data from the file associated with the logical unit specified or from the user's terminal. Variables can be arithmetic or string.
IF END	Tests for an end-of-file condition of input sequential file associated with logical unit expression.
PRINT	Prints the values of the specified expressions on the terminal or, when specified, to the file associated with the logical unit expression. The TAB function can also be included.
PRINT USING	Prints the values of the specified expression on the terminal or, when specified, to the file associated with logical unit expression in the format determined by the given string. Both numeric and string expressions can be used.
RESTORE	Resets to the beginning either the data pointer or, when specified, the input file associated with the given logical unit number.
CLOSE	Closes the file(s) associated with the logical unit number(s) and virtual file logical unit number(s) specified. If no logical unit number is specified, closes all open files.
NAME TO	Renames the specified file.
KILL	Deletes the specified file.
RANDOMIZE	Causes the random number generator (RND function) to produce different random numbers every time the program is run.
DEF FN	Defines a user function.

CALL   Used to call assembly language subroutines from a
                BASIC program.

FOR TO   Sets up a loop to be executed the specified number
                of times.

NEXT   Placed at the end of the FOR loop to return control to
                the FOR statement.

IF   Conditionally executes the specified statement or
                transfers control to the specified line number. If the
                condition is not satisfied, execution continues at the
                next sequential line. The expressions and the rela-
                tional operator must all be string or all be numeric.

GOSUB   Unconditionally transfers control to specified line of
                subroutine.

RETURN   Terminates a subroutine and returns control to the
                statement following the last executed GOSUB
                statement.

GO TO   Unconditionally transfers control to specified line
                number.

ON GOSUB   Conditionally transfers control to the subroutine at
                one line number specified in the list. The value of the
                expression determines the line number to which
                control is transferred.

ON GO TO   Conditionally transfers control to one line number in
                the specified list. The value of the expression deter-
                mines the line number to which control is trans-
                ferred.

CHAIN   Terminates execution of the program, loads the pro-
                gram specified, and begins execution of the lowest
                line number or, when a line number is present in the
                statement, at the specified line number.

OVERLAY   Merges the current program with a program seg-
                ment stored in a file.

COMMON   Preserves values and names of specified variables
                and arrays when the CHAIN statement is executed.
                Both string and arithmetic variables and arrays can
                be passed. The statement also dimensions the spec-
                ified arrays.

END   Placed at the end of the physical end of the program
                to terminate execution (optional).

| STOP | Terminates execution of the program. Placed at the logical end of the program. |

FUNCTIONS

BASIC provides a variety of functions to perform mathematical and string operations.

Arithmetic Functions

ABS	Returns the absolute value of an expression.
ATN	Returns the arctangent as an angle in radians.
COS	Returns the cosine of an expression in radians.
EXP	Returns the value of the constant e (approx. 2.71828) raised to a given power (expression).
INT	Returns the greatest integer less than or equal to a given expression.
LOG	Returns the natural logarithm of an expression.
LOG10	Returns the base 10 logarithm of an expression.
PI	Returns the value of pi (3.141593 approx.)
RND	Returns a random number between 0 and 1.
SGN	Returns value indicating the sign of an expression.
SIN	Returns the sine of an expression in radians.
SQR	Returns the square root of an expresssion.
TAB	Causes the terminal print head to tab to column number given by an expression (valid only in PRINT).
SYS	Special system function calls; control terminal I/O and perform special functions.

String Functions

| ASC | Returns the decimal ASCII code for the one-character given expression. |
| BIN | Converts a string expression containing a binary number to a decimal value. |

CHR$	Generates a one-character string whose ASCII value is the low-order 8 bits of the integer value of the given expression.
CLK$	Returns the time as a string
DAT$	Returns the date as a string.
LEN	Returns the number of characters in the given string.
OCT	Converts a string expression containing an octal number to a decimal value.
POS	Searches for and returns the position of the first occurrence of a substring in a string.
SEG$	Returns the string of characters in the given positions in the string.
STR$	Returns the string which represents the numeric value of the given expression.
TRM$	Returns the given string without trailing blanks.
VAL	Returns the value of the decimal number contained in the given string expression.

User-defined Functions

In some programs it may be necessary to execute the same sequence of statements in several different places. BASIC allows definition of unique operations or expressions and the calling of these functions in the same way as, for example, the square root or trigonometric functions. Each function is defined once and can appear anywhere in the program.

A function definition consists of the function name, a dummy variable list (up to five), and an expression.

When the user-defined function is used in the program, the expressions in the argument list passed to the function will replace the dummy variables in the defining expression. Any variable in the defining expression that is not in the dummy variable list will have the value that the variable is currently assigned.

PROGRAMMING EXAMPLE

The POS function is used to find the position of a substring in a string. The POS function can be used to map a string of characters to a corresponding integer value which can be used for subsequent processing. This technique is called a table look-up. The table string is the

first argument of the POS function and the string to be mapped is the second argument. For example:

```
LISTNH

10 REM PROGRAM TO TRANSLATE MONTH NAMES TO NUMBERS
20 T$="JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
100 PRINT "TYPE THE FIRST 3 LETTERS OF A MONTH";
110 INPUT M$
120 IF LEN(M$)<>3 GO TO 200
130 M=(POS(T$,M$,1)+2)/3
140 REM CHECK IF MONTH IS SPELLED CORRECTLY
145 IF M=0 GO TO 200
150 IF M<>INT(M) GO TO 200
160 PRINT M$" IS MONTH NUMBER "M
170 GO TO 100
200 PRINT "INVALID ENTRY - TRY AGAIN" \ GO TO 100


READY


RUNNH


TYPE THE FIRST 3 LETTERS OF A MONTH? NOV
NOV IS MONTH NUMBER 11
TYPE THE FIRST 3 LETTERS OF A MONTH? DEC
DEC IS MONTH NUMBER 12
TYPE THE FIRST 3 LETTERS OF A MONTH? JUM
INVALID ENTRY - TRY AGAIN
TYPE THE FIRST 3 LETTERS OF A MONTH? ↑C
STOP AT LINE 110


READY
```

GRAPHICS AND LABORATORY PERIPHERALS SUPPORT
BASIC provides graphics support for the VT11, GT42, and GT44 graphics display systems. The support consists of a collection of routines accessible by the CALL statement.

Points, vector, text, and graph data can all be combined through simple CALL statements. The screen can be scaled to any coordinates. Portions of the display can be controlled independently through the use of the subpicture feature. Special graphic routines allow the display of an entire array of data by one call statement. The area of memory that is allocated to the display buffer can be dynamically controlled.

Laboratory peripheral support for BASIC allows the user to use the complete set of LPS11 hardware. LPS support enables the user to sample and display in a real-time environment a variety of data from analog to digital converters, digital I/O, and external events. Sampling is controlled by crystal clocks and/or Schmitt triggers. It is possible to specify such parameters as sampling rate and response time thus allowing maximum flexibility.

Since BASIC is a higher-level language, even the novice programmer can solve complex data acquisition problems with a minimum amount of effort. All LPS routines are issued by the BASIC CALL statement allowing the PDP-11 assembly language programmer to include or modify the routines easily to meet particular requirements.

BASIC Files

Data is stored either in sequential files or in random access, virtual array files. Sequential files are treated in the same way as terminal I/O; data is read by an INPUT statement and written by a PRINT statement. Sequential files are useful for storing data that is processed serially.

Virtual array files are similar to arrays stored in memory. An element of data in a virtual array can be part of any BASIC expression, since it is created in the same way as an element of a normal array. Virtual array files allow data to be accessed in a random, non-serial manner and are the only BASIC files in which existing data can be updated without rewriting the entire file.

There are three data types for virtual array files: integer, floating point, and string. A file can contain only one data type.

Creating, Modifying and Executing BASIC Programs

A BASIC program is entered in the system using the editing commands. Once a program has been entered, it can be saved, retrieved, listed or executed using the editing commands. When the BASIC system is running, it prints the message READY on the terminal to indicate that it is ready to accept an editing command.

The BASIC system's editor is a replacement editor. That is, an incorrect line is changed by entering a new line with the same line number as the incorrect line. The editor replaces the old line with the new line entered. A line can be deleted by typing its line number and a carriage return. Both the line number and the line are removed from the program. The following provides a summary of the BASIC editing commands.

NEW   Clears the user area in memory and assigns a speci-
               fied name to the current program. Used to create a
               new program.

LIST
LISTNH
Types on the terminal the program currently in memory. A range of line numbers can be specified. If the command does not have the "NH" suffix, the program header is printed.

RUN
RUNNH
If issued with no file specification, executes the program currently in memory. If a file specification is issued, clears the user area, reads a program in from the file, and executes the program. If the command does not have the "NH" suffix, the program header is printed.

CLEAR
Clears the contents of the user array and string buffers. This command is used when a program has been executed and then edited. Before rerunning the program, the array and string buffers are cleared to provide more memory space.

RENAME
Changes the current program name to a specified name.

SAVE
Copies the contents of the user area to a file, lists the contents on the line printer, or punches the contents on paper tape.

OLD
Clears the user area and reads a program from a specified file into the user area in memory.

REPLACE
Replaces the specified file with the program currently in memory.

APPEND
Merges the program currently in memory with a program stored in a file. All lines in the program in memory that have duplicate line numbers with the program in the file are replaced by the lines from the program in the file.

LENGTH
Displays on the terminal the amount of storage required by the BASIC program currently in memory. This information is useful in determining the minimum user area in which a specific program can run.

SCR
Erases the user area in memory.

UNSAVE
Deletes the specified file.

BYE
If the BASIC system supports multiple users, terminates the session at the terminal.

SUB   Allows a user to change portions of a line without replacing the whole line.

RESEQ   Allows a user to resequence the line numbers in a program.

In addition to the editing commands, the BASIC system recognizes the following special control characters:

CTRL/C   Interrupts program execution and prints the READY message.

CTRL/O   Enables/disables console output.

CTRL/U   Deletes the current line being entered.

RUBOUT   Deletes the last character typed.

COMPILER OPERATION

When the user enters a BASIC program, the BASIC system does not store the program exactly as it is typed or read from the input file. Instead, it translates the program into an intermediate form which can be used in two different ways. The intermediate code can be returned to its initial form by the LIST or SAVE commands to produce an ASCII program that looks like the input program, or the translated code can quickly be interpreted by the RUN command to execute a program under the operating system.

Immediate Mode of Execution

It is not necessary to write a complete program to use BASIC. Almost any BASIC statement can be executed in immediate mode. This latter facility makes BASIC an extremely powerful calculator.

BASIC distinguishes between those lines entered for immediate execution and those entered for later execution by the presence or absence of a line number. Statements which begin with line numbers are stored; those without line numbers are executed immediately.

Immediate mode operation is especially useful for program debugging and desk calculation problems.

To facilitate debugging a program, STOP statements can be placed throughout the program. When the program is run, each STOP statement causes the program to halt. The data values can then be examined and modified in immediate mode. The immediate mode statement

GO TO line number

is used to continue program execution. The values assigned to variables when the RUN command was issued remain intact until a SCRatch, CLEAR or another RUN command is issued.

If the STOP statement occurs in the middle of a FOR loop, modifications can not be made to the section of the program which precedes the FOR statement.

If CTRL/C is used to halt program exectuion, the GO TO command can be used to continue execution at the line where execution stopped.

When using immediate mode, nearly all the standard statements can be used to generate or print immediate mode results. Multiple statements can be used on a single line in immediate mode. For example:

    A = 1 \ PRINT A
    1

Program loops in immediate mode are allowed in multiple statement lines. Thus a table of square roots can be produced as follows:

FOR I = 1 TO 10 \ PRINT I,SQR(I) \ NEXT I

1	1
2	1.41421
3	1.73205
4	2
5	2.23607
6	2.44949
7	2.64575
8	2.82843
9	3
10	3.16228

Certain statements, while not illegal, make no sense when used in immediate mode, such as COMMON, DEF, DIM, DATA and RANDOMIZE. The INPUT statement is illegal in immediate mode. Also, function references in immediate mode are illegal unless the program containing the definition was previously executed.

User Area Allocation And Program Size
The BASIC system stores each user's program in memory in the following format:

    Arrays   high addresses

    Buffers

Strings

Symbol Table

User Code   low addresses

The symbol table and user code area are created when the program is entered. When the RUN command is issued, the user program is scanned and arrays are set up. The string area is created during program execution.

The SCRatch command clears all the user code, symbol table, strings, and arrays from memory. The CLEAR command clears the arrays and strings but does not affect the user code or symbol table.

A symbol table entry is created for each distinct line number or variable name referenced in the program. These entries are not deleted, however, even when all references in the program to a particular line number or variable are removed. Thus, if the program in memory is heavily modified, it may be desirable to save it with the SAVE command and then restore the program with the OLD command to obtain the largest possible user area.

The LENGTH command displays on the terminal the amount of storage required by the BASIC program in memory. This information is useful in determining the minimum user area in which a specific program can run.

LENGTH prints the number of words used and the number of words remaining free in the user's area. The LENGTH command always returns the current memory requirements; they may differ depending on when the command is issued. The number of words in use includes memory currently needed by the BASIC program itself, arrays, string variables, and file buffers in the user area. To determine the size of the program alone, issue the LENGTH command immediately after an OLD or CLEAR command. Arrays are created after the RUN command is issued and file buffers are created when the OPEN statement is executed. The memory required for string variables and string arrays varies with the current values of the strings.

BASIC ENVIRONMENTS

BASIC is available on the RT-11, RSX-11M, and IAS systems. Two versions are available for the RT-11 operating system: the standard single-user version available on IAS, and a special multi-user version.

The single-user version of BASIC available on RT-11 provides graphics and laboratory peripherals support. BASIC with Laboratory

Peripheral System support requires at least 16K words of memory. The hardware required for use of the BASIC graphics support includes a GT44 system (RT-11). In addition to the peripheral I/O device needed to support the BASIC system (disk, DECtape, cassette or paper tape), the use of the timer routines require a real-time clock. The memory required for the graphics support itself is approximately 2.5K words.

The following paragraphs discuss the features and capabilities of the BASIC versions available.

Under RT-11: Single-user Version
BASIC-11/RT-11 interfaces with the RT-11 monitor. BASIC is loaded under control of the monitor by typing the "BASIC" command. Users can access any RT-11 supported device, including disk, DECtape, cassette, magnetic tape, card reader, paper tape reader/punch and floppy disk. BASIC-11/RT-11 files can be processed by FORTRAN IV/RT-11. At least 8K words of memory are required to run BASIC. In systems with more than 8K words of memory, BASIC-11/RT-11 provides alphanumeric character string I/O and string variable support.

Under RT-11: Multi-user version
MU BASIC-11/RT-11 is a multi-user BASIC system, capable of accommodating up to eight users simultaneously. Each user independently creates and executes BASIC programs. All of the features of MU BASIC, including statements, commands, functions and immediate mode execution are available to all users.

MU BASIC runs under the RT-11 foreground/background or XM monitor. Users can access all devices supported by RT-11.

Up to eight users can be supported on single-job systems with at least 24K words of memory. Up to four users can be supported on single-job systems with at least 16K words of memory, or on Foreground/Background systems with at least 28K words of memory.

To accommodate multiple users, MU BASIC provides a scheduling supervisor. In addition, the system provides a log-on procedure and file protection as options.

The log-on procedure requires that users obtain a user ID and password from the system manager to gain access to the system. The HELLO and BYE commands are used in this case to log on and log off the system.

The file protection system provides several degrees of file access. There are three classes of files: public, group, and private. Public library files are available to all users. Group library files are accessible to all users and have the same first character in their user ID. A private file is accessible only to the user who created it.

If file protection is desired, a file can be given any of the following access characteristics:

Run   Allows access by the RUN command or CHAIN state-
               ment.

Read   Allows access by the OLD or APPEND command or
               the OPEN FOR INPUT or OVERLAY statement or use
               of the value of an element of a virtual array.

Update   Allows a virtual array file to be updated.

Complete   Allows access by all of the above and by the SAVE,
               REPLACE or UNSAVE command or the OPEN FOR
               OUTPUT, NAME TO or KILL statement.

A nonprivileged user is allowed complete access only to the user's own private files. A nonprivileged user can have run and read access to files in the public library and the user's own group library. Nonprivileged users are not allowed access to other user's private library files or other group's files. The access allowed a nonprivileged user to all files other than the user's own files can be modified by the inclusion of a protection code in the filename.

A privileged user has complete access to all files. Group library and public library files can be created only by a privileged user.

In addition to the log-on procedure and file protection, MU BASIC includes the following commands:

HELLO   Allows the user to log on to the system (optional).

ASSIGN   Assigns a specified device to a user if it is available.

DEASSIGN   Deassigns a specified device.

SET TTY   Sets the system to allow different terminal character-
               istics.

MU BASIC provides a comprehensive set of system functions. Certain system functions are available to all users. These cancel CTRL/O typed at a terminal, disable/enable echoing, enter single character input mode, scratch the user area in memory and return to the READY message, and return the current user's ID. Certain other system function calls can be executed only by a privileged user. These disable the CTRL/C interrupt, set the user ID, terminate the privileged user status, and cause BASIC to exit and return control to the RT-11 monitor.

The single character input mode system function call is useful for

special read operations. It returns the decimal ASCII value of the next character input from the terminal or a file. It is the only method for BASIC programs to process terminal input without waiting for a carriage return to be typed. This allows interactive programs to use single character response and not require a carriage return.

Single character input mode allows data in any file to be read with no need for separating commas or carriage returns. Binary files can be copied exactly.

Under IAS
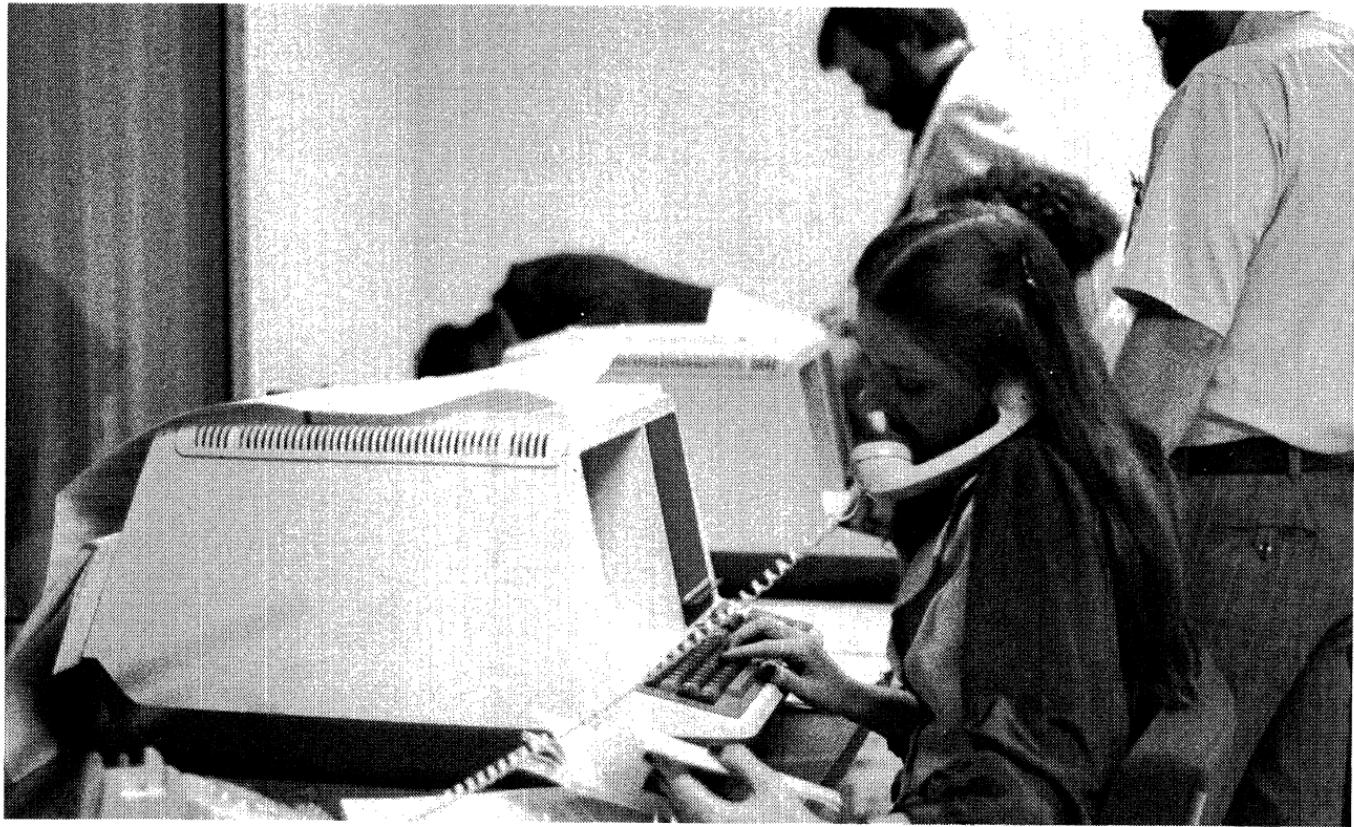
BASIC under IAS is treated as a shared single-user system. That is, a single copy of the pure area of BASIC can be shared by several users, but each user has his own copy of the impure BASIC code.

The minimum requirement for the BASIC system is a 12K-word partition. The partition can be expanded up to 28K words. BASIC/IAS can access any device supported by the IAS file system.

Under RSX-11M

BASIC-11 may run under both mapped and unmapped RSX-11M systems.

- Unmapped systems (memory management option not included in the configuration) require a 15K-word minumum partition (supports one user, BASIC-11 is overlaid).
- Mapped systems (memory management option is included in the configuration):
  - Systems in which the interpreter code is not installed as shared code require a 15K-word minimum partition for each user.
  - Systems in which the interpreter code is installed as shared code require one 15K-word minimum common partition area (shared by all users), with a 5K-word minimum partition area for each user.

BASIC-PLUS (V6C)

OVERVIEW

BASIC-PLUS has a comprehensive group of string operations that provide efficient processing of alphabetic data, including names, addresses, sentences, and paragraphs. The BASIC-PLUS user can append strings or compare them. It's possible to extract, examine, or search for a string of characters contained in a larger string. BASIC-PLUS has an immediate mode of operation that allows commands to be instantly executed instead of stored for later execution. This language has program control and storage facilities that can store both programs and data on any mass storage device and retrieve them for use during program execution.

FEATURE TOPICS

- Functions and Features
- BASIC-PLUS Language Summary (Table 18-1)
  - Variable Types
- Summary of BASIC-PLUS Statements
  - General Program Statements
  - Matrix Statements
  - Statement Modifiers
  - System Statements
  - Record I/O Statements
- Immediate Mode Operations
  - Data Formats and Operations
  - Matrix Manipulation
  - Advanced Statement and Function Features

FUNCTIONS AND FEATURES

The BASIC-PLUS programming language is DIGITAL's extension of the BASIC language originated at Dartmouth College. Its language compiler enables users to write executable programs in BASIC-PLUS as well as interact with the RSTS/E operating system. BASIC-PLUS runs exclusively (as one of many language options) on RSTS/E. BASIC-PLUS under RSTS/E offers many features not found in the original BASIC or other versions of BASIC.

BASIC-PLUS incorporates the following special features:

Immediate Mode of Operation: Commands can be executed immediately by BASIC-PLUS instead of being stored for later execution.

Program Editing Facilities: An existing program can be edited by adding or deleting lines, or renaming the program. The user can combine two programs into a single program and request the listing of a program, either in whole or in part, on a terminal or line printer.

Program Control and Storage Facilities: Facilities are included for storing both programs and data on any mass storage device and later retrieving them for use during program execution. Programs can also be entered from a terminal paper tape reader as well as from the high-speed paper tape reader available on the computer.

Documentation and Debugging Aids: The insertion of remarks and comments within a program is easy in this version of BASIC. Debugging of programs is aided by the printing of meaningful diagnostic messages that pinpoint errors detected during the program's execution.

Access to System Peripheral Equipment: The user program is able to perform input and output with various equipment, such as paper tape reader/punch, disk, DECtape, industry-compatible magnetic tape, line printer, floppy disks, and card reader.

Record I/O: Language extensions provide a means of handling records composed of fixed-length fields in a highly efficient manner.

Matrix Computations: A set of 13 commands is available for performing matrix I/O, addition, subtraction, multiplication, inversion, and transposition.

Alphanumeric String Capabilities: Alphabetic and/or alphanumeric strings can be manipulated with the same ease as numeric data. Individual characters within these strings can be accessed by the user.

Formatting of Output: The PRINT-USING statement includes facilities for tabs, spaces, and the printing of column headings, as well as precise specifications of the output line formatting and floating dollar sign, asterisk fill, and comma insertion in numeric output.

String Arithmetic: A set of functions which perform arithmetic on operands which are numeric strings instead of integer or floating point numbers. This provides a means of calculating values of higher than normal precision, or values which must not be affected by round-off error, at the expense of slower execution time.

The following sections describe some of the advanced BASIC-PLUS features, including immediate mode operations, data formats and operations, matrix manipulation, program development commands, and advanced BASIC-PLUS statements. Table 18-1 provides a summary of the BASIC-PLUS language.

Table 18-1 BASIC-PLUS Language Summary

SUMMARY OF VARIABLE TYPES

Type	Example
Floating point	A I X3
Integer	B% D7%
Character string	M$ R1$
Floating point matrix	S(4) E(5,1) N2(8) V8(3,3)
Integer matrix	A%(2) I%(3,5) E3%(4) R2%(2,1)
Character string matrix	C$(1) S$(8,5) A2$(8) V1$(4,2)

SUMMARY OF OPERATORS

Type	Operator	Operates On
Arithmetic	− unary minus	Numeric variables or constants.
	↑ exponentiation	
	* multiplication	
	/ division	

385

Type	Operator	Operates On
	+ addition	
	− subtraction	
Relational	= equals	String or numeric variables or constants.
	< less than	
	≤ less than or equal to	
	> greater than	
	≥ greater than or equal to	
	≠ not equal to	
	≈ approximately equal to	
Logical	NOT logical negation	Relational expressions composed of string or numeric elements, integer variables, or integer valued expressions.
	AND logical product	
	OR logical sum	
	XOR logical exclusive or	
	IMP logical implication	
	EQV logical equivalence	

Type	Operator	Operates On
String	+ concatenation	String variables or constants.
Matrix	± addition and sub-traction of equal dimensions	Dimensioned variables.
	* multiplication of conformable matrices	
	* scalar multiplication	

SUMMARY OF FUNCTIONS

Mathematical

ABS(X)	Absolute value
ATN(X)	Arctangent in radians
COS(X)	Cosine in radians
EXP(X)	Exponent
FIX(X)	Truncated value
INT(X)	Greatest integer
LOG(X)	Natural log
LOG10(X)	Common log
PI	Constant pi
RND	Random number between 0 and 1
SGN(X)	Sign
SIN(X)	Sine in radians
SQR(X)	Square root

Mathematical

TAN(X)   Tangent in radians

Print
POS(X%)   Current print head position

TAB(X%)   Move print head position

String
ASCII(A$)   ASCII value in decimal

CHR$(X%)   ASCII character

CVT%$(I%)   Maps integer to string

CVTF$(X)   Maps floating point to string

CVT$%(A$)   Maps string to integer

CVT$F(A$)   Maps string to floating point

CVT$$(A$,I%)   Converts string

STRING$ Creates string
(N1%,N2%)

LEFT(A$,N%)   Returns leftmost substring

RIGHT   Returns rightmost substring
(A$,N%)

MID   Returns middle substring
(A$,N1%,N2%)

LEN(A$)   Returns string length

INSTR   Search for substring
(N%,A$,B$)

SPACE$(N%)   String of spaces

NUM$(N%)   String of numerals
NUM1$(N%)

String

VAL(A$)   Computes numeric value

XLATE(A$,B$)   Translate string

System
DATE$(0%)   Current date

DATE$(N%)   Calendar date

TIME$(0%)   Current time

TIME$(N%)   Time of day

TIME(0%)   Clock time

TIME(1%)   CPU time used

TIME(2%)   Connect time

TIME(3%)   KCTs used

TIME(4%)   Device time used

STATUS   I/O status

BUFSIZ(N)   Opened device buffer size

LINE   Interrupted line number

ERR   Error code received

ERL   Line number on error receipt

SWAP%(N%)   Byte swap

RAD$(N%)   Radix-50 conversion

Matrix
TRN(X)   Transpose matrix

INV(X)   Inverse of matrix

Matrix

DET   Determinant after inverse

NUM   Number of rows on input

NUM2   Number of row elements on input

Magtape
MAGTAPE   Program control of magtape

RECOUNT   Input characters read

SUMMARY OF BASIC-PLUS STATEMENTS

General Program Statements

REM   Identifies comments in a program.

LET   Assigns a value to a variable.

DIM   Defines the maximum number of elements in a matrix.

RANDOMIZE   Causes the random number function (RND) to choose a random starting value.

IF-THEN, IF-GOTO   Transfers conditionally from the normal consecutive order of statement numbers or conditionally executes a set of statements, depending on the truth of some mathematical relation or relations.

IF-THEN-ELSE   Same as the IF-THEN statement, except that rather than executing the line following the IF statement when the condition is not met, another line number can be specified for execution.

FOR   Causes the program to cycle through a designated loop using a control variable to count the number of repetitions.

FOR-WHILE, FOR-UN-   Specifies that a loop is to be reiterated as
TIL   long as a certain condition remains true (FOR-WHILE) or false (FOR-UNTIL).

NEXT   Signals the end of the loop which began with the FOR statement.

General Program Statements

DEF, single line	Defines a single-line function.
DEF, multiple line	Defines a multiple-line function.
GOTO	Unconditionally transfers execution to some line other than the next sequential line in the program.
ON-GOTO	Transfers control to one of several lines depending on the value of an expression.
GOSUB	Transfers control to the first line of a subroutine.
ON-GOSUB	Transfers control of a program to one of several subroutines or to one of several entry points of one or more subroutines.
RETURN	Signals the end of a subroutine and transfers control to the line following the GOSUB statement line.
CHANGE	Transforms a character string into a list of numeric values or a list of numeric values into a character string.
OPEN	Associates a file on a file-structured device or some non-file structured device with an I/O channel number internal to the program.
CLOSE	Terminates I/O between the program and a peripheral device.
READ	Assigns values defined in DATA statements to specified variables.
DATA	Supplies the values to be used in READ statements.
RESTORE	Resets the data values list; a subsequent READ statement obtains the value defined by the first DATA statement in the program.
PRINT-USING	Performs formatted output.
INPUT	Enters data to a program from an external device such as the terminal, disk, DECtape, or paper tape reader.

General Program Statements

INPUT LINE — Enters an entire line of data as a single character string without any formatting.

NAME-AS — Renames and/or assigns protection codes to a disk or DECtape file.

KILL — Deletes a file from an account area.

ON ERROR GOTO — Transfers control to a user-written subroutine that handles a normally fatal program error.

RESUME — Resumes execution of a program after a user-written error routine is executed.

CHAIN — Transfers control of execution to another program.

STOP — Suspends program execution and returns terminal to immediate mode.

END — Identifies the end of a program.

Matrix Statements

MAT READ — Reads data into a matrix from DATA statements in a program.

MAT PRINT — Prints each element of a one- or two-dimensional matrix.

MAT INPUT — Enters the value of each element of a predimensioned matrix from the keyboard.

MAT — Creates initial values for the elements of a matrix (excluding row zero or column zero), which can be all zeros, all ones, or an identity matrix (all diagonal elements are ones).

Statement Modifiers

IF — The statement is executed only if the condition specified following the IF is true.

UNLESS — The statement is executed only if the condition specified following the UNLESS statement is false.

FOR — Performs reiterative execution of a single line based on a control variable.

Statement Modifiers

WHILE   Performs reiterative execution of a single
                          line as long as the condition specified fol-
                          lowing the WHILE remains true.

UNTIL   Performs reiterative execution of a single
                          line as long as the condition specified fol-
                          lowing the UNTIL remains false.

System Statements
SLEEP   Suspends a program's execution for a
                          specified interval.

WAIT   Sets a maximum period for the system to
                          wait for input from a terminal before gener-
                          ating a trappable error.

Record I/O Statements
LSET   Stores values in a string left-justified.

RSET   Stores values in a string right-justified.

FIELD   Associates a string name with an I/O buffer.

GET   Reads data from a file into an I/O buffer.

PUT   Writes data from an I/O buffer into a file.

UNLOCK   Allows another program to obtain write
                          privileges to a record currently open for
                          write operations.

BASIC-PLUS PROGRAM DEVELOPMENT COMMANDS
NEW   Allows the user to create a BASIC-PLUS program by
                          entering it on the terminal.

OLD   Retrieves the source file of a previously saved BA-
                          SIC-PLUS program and places it in the user's job
                          area.

SCALE   Allows the user to set the double precision floating
                          point numeric scale format scale factor to be used in
                          all programs subsequently compiled for the account.

CATALOG   Prints a listing of any account's device directory on
CAT   the terminal.

TAPE	Disables the terminal echo feature when reading a low speed terminal device.
KEY	Enables the terminal echo feature after reading a low speed paper tape on the terminal.
APPEND	Merges the contents of a previously saved program into a program currently in the job area.
COMPILE	Stores a compiled version of the source program currently in the user's job area on a selected device.
SAVE	Stores the current source program in the user's area on a selected device.
REPLACE	Same as SAVE, but allows the user to replace a program currently saved under the same name as the program currently in the job area.
UNSAVE	Deletes a file from a selected device.
RENAME	Changes the name of the program currently in memory to a new name.
LIST	Lists all or selected lines of a BASIC-PLUS program currently in the user's area on the terminal.
DELETE	Deletes one or more selected lines of a program currently in the user's area.
LENGTH	Prints the length, in 1K word increments, of the current program in the user's area.

IMMEDIATE MODE OPERATIONS

Most BASIC-PLUS statements can either be included in a program for later execution or be issued on-line at the terminal as commands which are immediately executed by the BASIC language processor. Immediate mode operation is especially useful in two ways: to perform simple calculations that do not justify writing a complete program, or to debug a program.

To make program debugging easier, a user can insert several STOP statements in the program. When it is run, each STOP statement causes the program to halt and identify the line in the program at which the program was interrupted. The user can then examine the current contents of variables and change them if necessary, by using immediate mode commands. The user can then type the CONTINUE command to resume program execution from the line at which it stopped.

DATA FORMATS AND OPERATIONS

BASIC-PLUS allows the user to manipulate string, integer numeric or floating point numeric data.

String data are sequences of ASCII characters treated as units. The user can define string constants and string variables (including subscripted variables). In addition, relational operators can be applied to string operands to compare and indicate alphabetic (ASCII) sequence. Using the BASIC-PLUS CHANGE statement, the user can convert individual string characters to their equivalent ASCII code (in decimal) and vice-versa. BASIC-PLUS provides a variety of string functions that allow the user to concatenate two strings, access part of a string, determine the number of characters in a string, search for substrings, and convert strings to compact storage formats. The user can also define new string functions.

Normally, all numeric values (variables and constants) specified in a BASIC program are stored internally as floating point numbers. If operations to be performed deal with integer numbers, significant economies in storage space can be achieved by the use of the integer data type, which requires only one word of storage per value. Integer arithmetic is also significantly faster than floating point arithmetic.

BASIC-PLUS permits a user program to combine integer variables or integer-valued expressions using a logical operator to give a bitwise integer result. The logical operators AND, OR, NOT, XOR, IMP, and EQV operate on integer data in a bitwise manner.

Floating point numeric operations are the default BASIC-PLUS numeric type. BASIC-PLUS users working with floating point numbers can increase accuracy of operations involving fractional numbers by using the scaled arithmetic feature or the string arithmetic functions. Furthermore, the user can perform arithmetic operations using a mix of integer and floating point numbers. If both operands of an arithmetic operation are either explicitly integer or explicitly floating point, the system automatically generates integer or floating point results. If one operand is an integer and another is floating point, the system converts the integer to a floating point representation and generates a floating point result. If one operand is an integer and the other operand is a constant that can be interpreted either as a floating point number or an integer, the system generates an integer result. The user can explicitly impose the formats and thereby control the result of the operation.

MATRIX MANIPULATION

Matrices are arrays of data which are implicitly or explicitly dimensioned by the user. Matrices can be composed of variables of any

type. A single matrix, however, is composed of a single type of data: floating point, integer, or string. Dimensioning a matrix establishes the default number of elements in each row and column and the number of elements in the matrix. Implicitly dimensioned matrices are assumed to have ten elements in each dimension referenced (size 10 for one-dimensional matrix, size 10-by-10 for two-dimensional matrix, with each dimension also having a zero row and column). Explicit dimensioning is done using the DIM dimension statement.

By using the BASIC-PLUS MAT statements, a user program can alter the number of elements in each row and the number of columns in the matrix, as long as the total number of elements does not exceed the number defined when the matrix was dimensioned. The MAT operations do not set row zero or column zero, nor do they initialize values in the space allocated to the matrix unless specific MAT functions are executed.

The operations of addition, subtraction, and multiplication (including scalar multiplication) can be performed on matrices using the common BASIC mathematical operators. The matrices indicated for any operation must be conformable to that operation. In addition, functions exist for the performance of transposition and inversion of matrices.
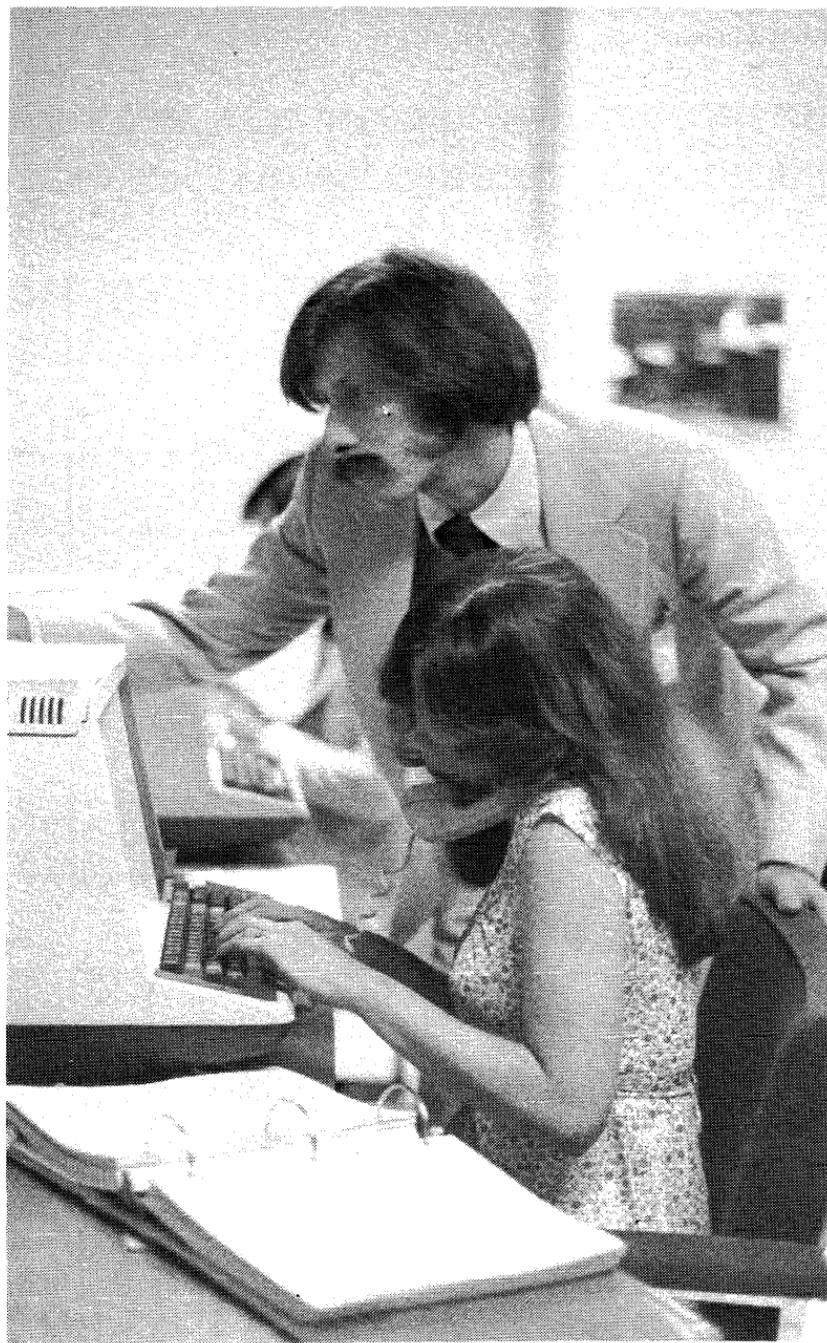
ADVANCED STATEMENT AND FUNCTION FEATURES
BASIC-PLUS extends the BASIC language by including several additional statements for easier logic flow control, function definitions, and timesharing response. The ON-GOTO, ON-GOSUB, IF-THEN-ELSE, FOR-WHILE, and FOR-UNTIL' statements provide a variety of conditional controls over looping and subroutine execution. The ON ERROR GOTO statement allows the programmer to write subroutines that handle error conditions normally considered fatal. The program can test a system variable named ERR to determine which error occurred, and can examine a system variable named ERL to determine the line number at which the error occurred. The SLEEP and WAIT statements allow program suspension, either for a specified time interval or until input from a terminal is received. The PRINT-USING statement provides special output formatting, including exponential representation, dollar signs, commas, trailing minus sign, and asterisk fill. The DEF statement allows multiple-line function definitions. Multiple-line function definitions can be nested, can be written in any data type and can contain any variety of argument types.

To increase the flexibility and ease of expression within BASIC-PLUS, five statement modifiers are available; IF, UNLESS, FOR (including FOR-WHILE and FOR-UNTIL), WHILE and UNTIL. These modifiers are

appended to program statements to indicate conditional execution of the statement or the creation of implied FOR loops.

RSTS/E also includes several system functions and statements that allow program access to system information and conversion routines. The program can obtain the current date and time, the CPU time, connect time, KCTs, and device time used for the job. The program can convert a numeric value to a string date or time or vice versa, can swap bytes, or convert an integer in Radix-50 format to a character string.

BASIC-PLUS-2   (V2)

OVERVIEW

BASIC-PLUS-2 is a superset of BASIC-PLUS with true compilation compatibility. Files produced in BASIC-PLUS-2 also interface directly to the RMS-11 record management system, enabling the user to create files, do record mapping, and access records sequentially, randomly, or by key. Another main feature of BASIC-PLUS-2 is a call statement which allows the programmer to access external subroutines. And there are a number of BASIC-PLUS-2 statements that allow for interactive observation and control of program execution.

FEATURE TOPICS

- Features
- Constants
  - Numeric Constants
  - Integer Constants
  - String Constants
- Variables
  - Numeric Variables
  - Integer Variables
  - String Variables
  - Subscripted Variables
- Forming Expressions
  - Comparing Strings Using Relational Operators
  - Logical Expressions
- Subprograms
  - Dummy and Actual Arguments
- Modifying Statements
- Matrix Operations
- Files
- Summary of Statements

FEATURES

The BASIC-PLUS-2 language is an outgrowth of Dartmouth BASIC. It encompasses both the elementary statements used to write simple programs and many new and advanced features. These new features, not found in standard Dartmouth BASIC, allow production of more complex and efficient programs.

Some of the special features of BASIC-PLUS-2 are:

- virtual arrays
- record file I/O
- extensive string support
- full matrix package
- long variable names
- IF...THEN...ELSE
- ON ERROR
- statement modifiers
- user-defined functions
- multi-statement lines
- multi-line statements

CONSTANTS

There are three types of constants in BASIC-PLUS-2:

- Numeric (real numbers, also called floating point numbers)
- Integer (whole numbers)
- String (alphanumeric and/or special characters)

Numeric Constants

A numeric constant is one or more decimal digits, either positive or negative, in which the decimal point is optional.

The following are all valid numeric constants (real numbers):

```
   5        42861
  74       −125
   6.        .95
```

Integer Constants

An integer constant is a whole number (no fractional part) written without a decimal point. An integer constant is typed as one or more decimal digits terminated by a percent sign (%). For example, the following numbers are all integer constants (whole numbers):

```
    29%      −8%
  3432%       1%
 12345%     205%
```

String Constants

A string constant (also called a literal) is one or more alphanumeric and/or special characters enclosed by double quotation marks ("text") or single quotation marks ('text').

Each character in a string constant can be a letter, a number, a space, or any ASCII character except a line terminator. The value of the string constant is determined by all its characters. For example, because of the number of spaces between the quotation marks and the characters:

"   DIGITAL   " is not the same as   "DIGITAL"

VARIABLES

Depending on the operations specified in a program, the value of a variable may change from line to line. BASIC-PLUS-2 uses the most recently assigned value of a variable when performing calculations. This value remains the same until a statement is encountered that assigns a new value to that variable.

BASIC-PLUS-2 accepts three types of variables:

* numeric

* integer

* string

Numeric Variables

A numeric variable is a named location in which a single numeric value is stored. The user names a numeric variable with a single letter followed by 29 optional characters consisting of letters, digits, or periods. Therefore, the maximum length of a numeric variable name is 30 characters:

1 letter
29 optional characters

Spaces should not be embedded between characters. The following are numeric variables:

```
C              L...5
M1             BIG47
F67T.J         Z2.
```

Integer Variables

An integer variable (like a numeric variable) is a named location in which a single value can be stored. Using an integer variable in a program indicates that space is reserved for the storage of a whole number (no fractional part).

An integer variable is named with a single letter followed by 29 optional characters consisting of letters, digits, or periods, and terminates with a percent sign (%). Therefore, the maximum length of an integer variable name is 31 characters. The following are integer variables:

    ABCDEFG%   C.8%
    B%   D6E7%

String Variables

A string variable is a named location used to store alphanumeric strings. A string variable is named with a letter followed by 29 optional characters consisting of letters, digits, or periods, and terminated with a dollar sign ($). The dollar sign ($) must be the last character in the name.

The following are examples of string variables:

    C1$ M$
    L.6$ F34G$
    ABC1$ T..$

Subscripted Variables

A subscripted variable is a numeric, integer, or string variable with one or two subscripts appended to it. The subscripts can be any positive expression type: a constant or a variable (integer or numeric), a letter or symbol, or any combination of these. BASIC-PLUS-2 converts non-integer expressions to integer by truncating the fraction. The value of the subscript can be 0 up to a maximum defined by the system.

The subscript in a subscripted variable is a pointer to a specific location in a list or table in which a value is stored. The user designates the pointer with either one or two subscripts enclosed by parentheses. If there are two subscripts, they are separated with a comma. The value stored can be numeric, integer, or string data.

To name a subscripted variable the user starts with a numeric, integer, or string variable name:

    A   A%   A$

To refer to an element in a list (one dimension), the user follows the variable name with one subscript within parentheses. For example:

    A(6)   A%(6)   A$(6)

To refer to an element in a table (two dimensions) the user follows the variable name with two subscripts. The first subscript designates the row number, and the second subscript designates the column number. The two subscripts are separated with a comma. For example:

A(7,2)   A%(4,6)   A$(17,23)

BASIC-PLUS-2 accepts the same alphanumeric characters for a simple numeric variable and a subscripted variable within the same program.

FORMING EXPRESSIONS

An expression can be numbers, strings, constants, variables, functions, array references, or any combination of these, separated by any of the following:

1. Arithmetic operators
2. Relational operators
3. String operators
4. Logical operators

Operator	Example	Meaning
+	A+B	Add B to A
−	A−B	Subtract B from A
*	A*B	Multiply A by B
/	A/B	Divide A by B
↑	A↑B	Calculate A to the power B
**	A**B	Calculate A to the power B

Operator	Example	Meaning
=	A=B	A is equal to B
<	A<B	A is less than B
>	A>B	A is greater than B
<=,=<	A<=B	A is less than or equal to B
>=,=>	A>=B	A is greater than or equal toB
<>,><	A<>B	A is not equal to B
==	A==B	A is approximately equal to B if the difference between A and B is less than 10↑(−6).

Comparing Strings Using Relational Operators

When a relational operator is used to compare the value of one or more alphanumeric characters, the user creates a relational string expression. BASIC-PLUS-2 uses the ASCII character collating sequence to determine which character is greater or lesser in value

than the other. The comparison is made character by character, left to right, by ASCII value, until BASIC-PLUS-2 finds a difference in value.

When applied to strings, relational operators compare characters for alphabetic sequence.

Operator	Example	Meaning
=	A$=B$	Strings A$ and B$ are equal after removing trailing blanks and nulls.
<	A$<B$	String A$ occurs before string B$ in alphabetic sequence.
>	A$>B$	String A$ occurs after string B$ in alphabetic sequence.
<=,=<	A$<=B$	String A$ is equal to, or precedes, string B$ in alphabetic sequence.
>=,=>	A$>=B$	String A$ is equal to, or follows, string B$ in alphabetic sequence.
<>,><	A$<>B$	String A$ is not equal to string B$.
==	A$==B$	Strings A$ and B$ are identical (exactly the same length without padding and compostion of all characters).

Note that the relational operator == has a different meaning when applied to strings than when applied to numbers. When comparing strings of different lengths, BASIC-PLUS-2 treats the shorter string as if it were padded with trailing blanks to the length of the longer string. In order to perform character-to-character comparison, BASIC-PLUS-2 needs two characters to compare. This is where the trailing blanks serve their purpose.

Logical Expressions

A logical expression consists of either one operand preceded by a logical operator or two separated by a logical operator. Logical expressions are used in statements like the IF-THEN-ELSE statement where a condition is tested to determine subsequent operations within the program. The operands in this case are usually relational expressions. Logical expressions can also be used with integer data. However, logical operations on strings are illegal.

BASIC-PLUS-2 determines whether the condition is true or false by testing the bitwise result of the logical expression for non-zero and zero, respectively. (That is, a non-zero result is true, and a zero result

is false.) Notice that any non-zero value is assumed to be true. BASIC-PLUS-2 supplies the value -1 for true when it evaluates a logical or relational expression, but accepts any non-zero value when performing a test.

SUBPROGRAMS

BASIC-PLUS-2 supplies a method for writing procedures to be used several times: subprograms. A subprogram allows the user to divide a large task into smaller, more manageable units which, in turn, can be accessed individually.

Dummy and Actual Arguments

Because reference subprograms can be referenced at more than one point throughout a program, many of the values used by the subprogram may change each time it is used. Dummy arguments in subprograms represent the actual values passed to the subprogram when it is called.

These dummy arguments indicate the data type of the actual arguments they represent. The position, number, and type of each dummy argument in a subprogram list must agree with the position and type of each actual argument in the reference to the subprogram (CALL statement).

Items passed to subprograms can be any legal variable, constant, expression, array, or array element. The value of any parameter can be used as a file number in the subprogram. BASIC-PLUS-2 passes items from the main program to the subprogram either by value or by reference. When passing by value, BASIC-PLUS-2 makes a temporary copy of the value in the calling program and uses the copy for calculations in the subprogram. The value in the calling program remains unchanged. The following items are passed by value:

- constants

- expressions

- array elements

When passing by reference or address, BASIC-PLUS-2 takes the actual value from the location in the main program, uses the value in the subprogram, then replaces the value in the main program. In this case, because of calculations in the subprogram, the value passed by reference could change in the main program. The following items are passed by reference:

- variables

- entire arrays

It is not possible to pass complete arrays by value. Individual elements of a list or table, however, are always passed by value. When an individual entry in an array is passed to a subprogram, it is received as a numeric or string variable depending on its type.

MODIFYING STATEMENTS

Another useful tool for building programs is the statement modifier. In BASIC-PLUS-2, the statement modifier qualifies or restricts the execution of a statement; thus allowing the user to:

1. Indicate conditional execution of a statement.
2. Create an implied loop.

An implied loop built with a statement modifier iterates only one statement on a line. In cases where the FOR-NEXT statement loop is extremely simple, the necessity for both the FOR and next statements is eliminated.

BASIC-PLUS-2 provides five statement modifiers:

1. IF
2. WHILE
3. UNTIL
4. UNLESS
5. FOR

These statement modifiers cannot stand alone; they must be appended to a statement. Most BASIC-PLUS-2 statements can be modified.

When using statement modifiers with the various forms of the IF statement, the following rules apply:

1. Append statement modifiers to either the THEN clause or the ELSE clause of an IF STATEMENT.
2. The statement modifier applies only to the clause it is appended to and not to the statement as a whole.

If there is more than one statement on a line, the modifier applies only to the statement immediately preceding it. More than one statement modifier can be appended to a single statement. In this case, BASIC-PLUS-2 processes the modifiers from right to left. Statement modifiers are reserved words.

MATRIX OPERATIONS

With the MAT statement, the following operations can be performed with arrays:

1. Assignment
2. Addition

3. Subtraction
4. Multiplication
5. Transposition
6. Inversion

Each MAT operation statement begins with the keyword MAT followed by an expression to be evaluated. The value of one array can be assigned to another as in the following example:

    10 MAT A = B

This statement sets each entry to array A equal to the corresponding entry of array B. A is redimensioned to the size of array B.

You can also add and subtract arrays:

    10 MAT A = B + C
    20 MAT A = B − C

FILES

There are three types of files in BASIC-PLUS-2:

1. Terminal format files
2. Virtual array files
3. Record files

To distinguish one file from another, it must be labeled with a file specification. The file specification usually contains the device name, the file name, and the file type.

Terminal Format Files

A terminal format file is a collection of ASCII characters stored in lines of various lengths. The end of a line is determined by a line terminator, i.e., line feed. BASIC-PLUS-2 stores these ASCII characters, including spaces and line terminators, exactly as they would appear on the terminal; hence the name terminal format file.

Terminal format files are sequential access files. Sequential access files are files that contain information that must be read or written one item after another from the beginning of the file. This means that an item from the file cannot be retrieved without first retrieving all the items preceding it.

BASIC-PLUS-2 has a file pointer that keeps track of where the user's location is in the file. To add new items to an existing file without overwriting current information, the entire file must be read. This action places the file pointer at the end of the file where data can be added.

Virtual Array Files
A virtual array file, like a terminal-format file, is information stored on a system device (disk). Once a virtual array file is opened, the similarity with terminal format files ends. Elements in a virtual array can be accessed exactly as elements in an array in memory.

Virtual array files are random access files. The last element in a virtual array can be accessed as quickly as the first.

When BASIC-PLUS-2 stores data in a virtual array file, it does not convert them to ASCII characters but rather stores them in the internal binary representation. Consequently, there is no loss of precision caused by data conversion.

Record Files
A BASIC-PLUS-2 record file is a collection of related data stored in the form of records. The user determines the size and content of the records and the structure and access properties of the file. For more details, see Chapter 11, RMS.

Programs can write BASIC-PLUS-2 programs that deal with records and files. The following need to be defined:

1. File organization
2. Access method
3. Record format
4. Record mapping
5. File operations
6. Record operations

File Organization
The manner in which BASIC-PLUS-2 stores and retrieves records in a file is determined by the structure of the file. In BASIC-PLUS-2, the structure of a file is known as the organization. When the file is created, the user specifies its organization. The organization, in turn, determines the operations and access methods that can be used on the file. The three organizations that can be specified are:

1. Sequential
2. Relative
3. Indexed

SUMMARY OF STATEMENTS
The following list summarizes BASIC-PLUS-2 statements and functions.

SUMMARY OF BASIC-PLUS-2 STATEMENTS
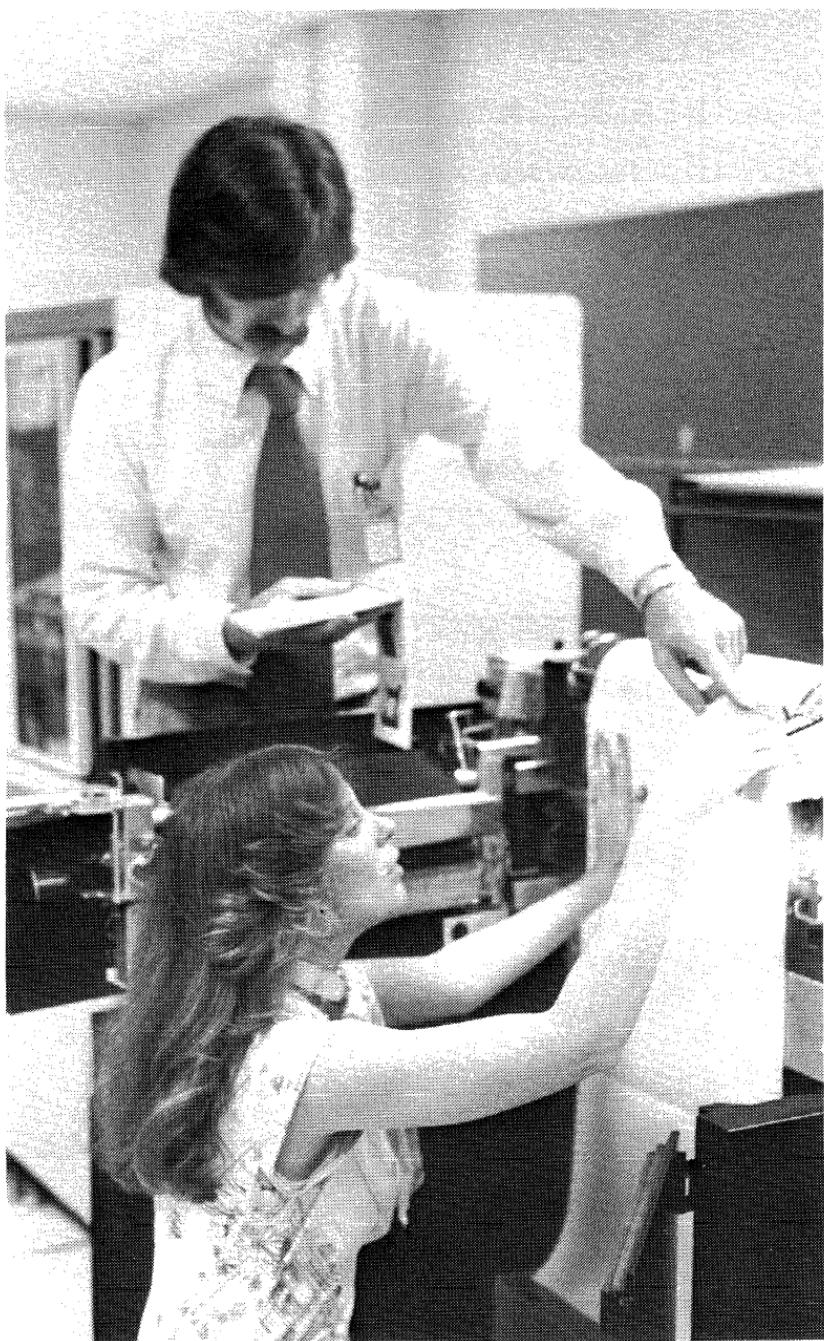
CALL   Transfers control to a specified subprogram, trans-
              fers parameters, and saves the state of the calling
              program.

CHAIN   Passes control to a specified program; if no line
              number is specified, execution starts at the begin-
              ning of the program.

CHANGE   Converts a list of integers (real numbers are truncat-
              ed) into a string of characters and vice versa.

CLOSE   Terminates I/O to a device and writes all active
              buffers.

COM   Allows the user to establish a named storage area
              that can be shared by 2 or more subprograms; the
              variables and arrays in the variable list are assigned
              to the named area and, when accessed by more than
              1 subprogram, must be of the same data type; the
              common area name must be 1 to 6 characters.

DATA   Allows the user to provide a pool of information that
              is accessible to the program by means of a READ
              statement; a DATA statement must be the last state-
              ment on the line and, when more than one is speci-
              fied, the items must be separated with commas.

DEF   Establishes a user-defined function. The function
(single line) name can be any legal variable name and must be-
              gin with FN; the variable type determines the func-
              tion type. The optional arguments represent dummy
              parameters and cannot contain array elements. The
              function definition can refer to any of the dummy
              parameters or to other program variables but the
              definition cannot be recursive. Single-line user-de-
              fined functions are local to the main program or sub-
              program in which they are contained.

DEF   Establishes user-defined functions and allows the
(multi-line) user to include other statements in the body of the
              function. The function name can be any variable
              name preceded by FN. Any statement can appear in
              a function except SUB, SUBEND, RETURN, or
              another DEF. The DATA and DIM statements are not
              local to the function definition. A GOTO, GOSUB,
              ONGOTO, or ONGOSUB transfer outside the func-
              tion is not allowed. The function definition must end
              with an FNEND statement.

DELETE   Erases the existing record from relative and indexed files.

DIM   Reserves storage for arrays.

DIM#   Allocates space for the specified arrays on the file associated with the logical number. Storage is allocated at the beginning of the file such that the rightmost subscript changes most rapidly; the default string storage length is 16 and the space is pre-allocated.

END   Terminates program execution and closes all files; optional; when used, END must be the last statement in the program.

FIND   Causes a RECORD search in the specified file.

FNEND   Causes an exit from a user-defined function and signals the function's logical and physical end.

FOR   Initiates and controls a loop. A simple numeric variable must be used after the FOR, and the same variable must appear in the required NEXT statement. The first numeric expression is the intial loop value; the second expression is the terminating loop value.

FOR (conditional)   Duplicates the previous FOR statement, except that loop termination is determined by a false expression in the WHILE clause or a true expression in the UNTIL clause.

GET   Reads a record from a specified file into a buffer. On sequential files, GET operations are performed on succeeding records starting at the beginning of the file. Relative files allow the specification of a record number, and indexed files allow the specification of a key value.

GOSUB   Transfers control to a subroutine that begins at a specified line number.

GOTO   Unconditionally transfers control to a specified line number.

IF   Allows branches in the program or conditional execution of one or more statements.

INPUT   Allows the user to type in data to the program from the terminal. The program requests data by printing a question mark on the terminal and waiting for a user response.

INPUT LINE and LINPUT   Allow a character string (ending with a line terminator) to be input to a specified variable. The line terminator is included in the string with INPUT LINE but discarded with LINPUT.

INPUTLINE # and LINPUT#   Read strings from a terminal-format file.

KILL   Deletes a file from storage.

LET   Assigns constants and expressions to variables.

MAP   Associates a named buffer with a file. Specified data in the element list are moved from the file to the buffer on a GET and from the buffer to the file on a PUT.

MAT INPUT   Allows element values to be entered in an array. Input is read from the terminal. Elements are stored in row order as they are typed.

MAT PRINT   Outputs each element of a specified array.

MAT READ   Reads the values into elements of a 1- or 2-dimensional array from a DATA statement.

MOVE   Associates the data in a record with the variables specified in the I/O list.

NAMEAS   Renames a file without changing the contents of the file.

NEXT   Terminates a FOR, WHILE, or UNTIL loop. The variable must correspond to the variable in the initial FOR statement. Nested loops cannot cross each other. No variable is allowed if the loop is started with WHILE or UNTIL.

ONERROR   Allows control to shift to an error-handling routine.

ON-GOSUB   Conditionally transfers control to one of several subroutines or to one of several entry points into one or more subroutines.

ON-GOTO   Allows the user to transfer control to another line of the program based on the value of the expression.

OPEN   Allows the user to create a new file or to access an existing file.

PRINT   Causes the specified data to be output on the terminal. The expression list can be expressions, variables, or quoted strings separated by a comma or a semicolon; commas cause output to terminal print zones; semicolons ignore the print zones.

PRINT#   Writes data into the specified terminal-format file.

PRINT USING   Causes output to be printed in a specified format. The list indicates the elements to be printed. It allows the user to format numbers so that the decimal points are aligned, making it easier to compare columns of numbers.

PUT   Writes a record from a buffer to a specified file. The RECORD clause is used for relative files; sequential files allow PUT operations only at the end of the file; the count clause allows the user to specify the size of the record.

RANDOMIZE   Changes the starting point of the RND function to a new unpredictable location.

READ   Directs BASIC to read from a list of values built into a data block by a DATA statement.

REM   Contains user-written comments and has no effect on program execution.

RESTORE   Resets the specified terminal-format file or record file to its beginning from the current position of the file. RESTORE without a file expression restores the data in a DATA statement; RESTORE with a key clause resets the key of reference.

RESUME   Indicates the last statement in an error-handling subroutine. If no line number is specified, control is shifted back to the point of error generation. If a line number is specified, control is shifted to that line.

RETURN   Indicates the last statement in a subroutine; shifts control to the statement following the last executed GOSUB statement.

SCRATCH   Allows the user to truncate the file. SCRATCH can be used only if the file was OPENed with ACCESS SCRATCH.

SLEEP   Causes a temporary halt in execution. The length of delay is determined by the value of the expression in seconds.

STOP   Causes a halt in program execution; files are not closed, and a message indicating the location of the halt is printed.

SUB   Marks the beginning of a subprogram and defines the type and number of subprogram parameters.

SUBEND   Marks the end of the subprogram and returns control to the calling program; must appear at the end of all subprograms.

UNTIL   Sets up a loop that must have a corresponding NEXT statement; the loop executes until the expression is true.

UPDATE   Changes an existing record in the file. The new record size as defined in the MAP or COUNT clause must be the same as the record it replaces; on sequential files, an UPDATE must be preceded by a successful GET or FIND.

WAIT   Specifies the maximum number of seconds allowed for input before an error is generated; a zero or null value disables the WAIT.

WHILE   Sets up a loop that must have a NEXT statement. The WHILE expression is evaluated before each loop iteration; if the expression is true, BASIC executes the statements in the loop. If the expression is false, BASIC executes the statements following the NEXT statement.

COBOL (V3)

OVERVIEW

PDP-11 COBOL provides terminal-oriented, fast data processing for commercial applications. Source programs are written in the American National Standards Institute (ANS) COBOL-74 language. DIGITAL's COBOL is a fully implemented intermediate level compiler conforming in language element, representation, symbology and coding format to ANS specification. In addition, it includes a number of features which include: DIGITAL's standard CALL sequences for external subroutines; use of the RMS-11 Record Management System; an ANS standard COBOL segmentation facility; and ANS standard string and substring manipulation.

FEATURE TOPICS

- Functions and Features
- PDP-11 COBOL Features
  - String Manipulation
  - On-Line Interactive COBOL Program Execution
  - File Organization
  - Library Facility
  - Debugging Features
- Compiler Implementation
  - Memory Segmentation
  - Flexibility
- COBOL Operating System Environments
  - Under RSTS/E
  - Under IAS and RSX-11M
- Utility Programs
  - Terminal Format
  - Convention Format
  - COBRG — COBOL Report Generator
- COBOL Language Implementation (Table 19-1)

FUNCTIONS AND FEATURES

PDP-11 COBOL provides fast direct access data processing for commercial applications. PDP-11 COBOL is available as an optional language processor for the RSTS/E, RSX-11M, IAS, and TRAX operating systems. Included in the COBOL package are the MERGE utility for merging ODL files, RFRMT source reformatting program, and COBRG report generator program utilities.

PDP-11 COBOL is a fully implemented intermediate level compiler conforming in language element, representation, symbology, and coding format to ANS-74 COBOL, specification X.3.23-1974. ANS-74 COBOL includes the following:

- full high-level nucleus module
- full high-level table handling module
- full high-level sequential I/O module
- full high-level relative I/O module
- full high-level indexed I/O module
- low-level interprogram communication module
- full high-level segmentation module
- full low-level library function, with partial high-level REPLACING facility
- conditional variables — Data Division Level 88
- nested conditionals

The hardware configuration supporting COBOL is any valid RSTS/E, RSX-11M, IAS, or TRAX operating system configuration with a line printer and enough memory to support a 24K-word COBOL task under RSTS/E and a 27K-word COBOL task for RSX-11M, IAS, and TRAX. Depending on the size of the compiler generated in a particular system, COBOL source programs can contain up to 6,000 or more statements. The recommended minimum disk storage is either two RK05 disk drives or an RP03, RP04, RP05, or RP06 disk drive. Optional hardware supported includes a card reader.

The disk-resident compiler can accept source program input from cards, console terminals, and disks—including input from source text library files stored on disks. COBOL utilizes RMS-11 to implement user file handling. In addition, COBOL programs can also create

416

and/or read ANSI standard format magnetic tape files if magnetic tape systems are included in the system's hardware configuration. Finally, COBOL programs can build files which the DATATREIVE-11 software package can process and vice versa.

LANGUAGE FEATURES

The PDP-11 COBOL processing modules are:

Nucleus	All language elements necessary for internal processing.
Table Handling	Defining and manipulating tabular data.
Sequential I/O	Defining and accessing sequential files.
Relative I/O	Defining and accessing relative files, including dynamic access.
Indexed I/O	Defining and accessing indexed sequential files, including dynamic access and multiple alternate keys.
Segmentation	Specifying overlay of the Procedure Division at object time.
Library	Copying predefined COBOL text into the source program; changing text while copying.
Inter-program Communication	Calling separately compiled subroutines and passing parameters.

PDP-11 COBOL utilizes RMS for I/O handling, and is therefore capable of handling files created under other PDP-11 languages.

The nucleus, table handling, sequential I/O, relative I/O, and indexed I/O modules of PDP-11 COBOL meet full ANS-74 high-level standards. RERUN, ENTER, and ALTERNATE are not, however, included in the PDP-11 COBOL Nucleus Level 2 code set. PDP-11 COBOL offers high-level extensions in the Segmentation and Library modules. Figure 19-1 compares the language elements implemented in PDP-11 COBOL with the ANS-74 COBOL language elements.

COBOL Data Types

PDP-11 COBOL supports a variety of standard data types. They include:

417

- Numeric DISPLAY Data
  - Trailing overpunch sign
  - Leading overpunch sign
  - Trailing separate sign
  - Leading separate sign
  - Unsigned
  - Numeric-edited
- Numeric COMPUTATIONAL Data
  - 1-word fixed binary
  - 2-word fixed binary
  - 3-word fixed binary
  - 4-word fixed binary
- Alphanumeric DISPLAY Data
  - Alphanumeric
  - Alphabetic
  - Alphanumeric-edited

These are data types which are required over a spectrum of application systems and are provided for flexibility in the specification and design of such systems.

String Manipulation

PDP-11 COBOL has the capability to manipulate data strings. It offers INSPECT, STRING, and UNSTRING—COBOL verbs for character string handling—to search for embedded character strings, with tally and replace. In addition, they have the ability to join together or break out separate strings with various delimiters.

On-line Interactive COBOL Program Execution

The Procedure Division ACCEPT and DISPLAY statements allow terminal-oriented interaction between a COBOL program and a user. Using these statements, a COBOL program can exercise interactive operation with a user running the program. This is useful in an order entry application, for example.

The ACCEPT statement allows the terminal user to enter input lines which the COBOL program can interpret. The ACCEPT statement also has a second format that allows it to retrieve the current date or time from the system.

The DISPLAY statement transfers data from a specified literal or data item to a specified device, normally the user's console. The statement can be modified by a special WITH NO ADVANCING phrase (without automatic appending of carriage return and line feed) that allows the COBOL program to control the format of the message sent. If the device handler allows it, the WITH NO ADVANCING phrase will have

the device remain positioned on the same line and the same character position following the last character displayed. This is especially useful when typing prompting messages on the console.

While the ACCEPT and DISPLAY statements are primarily intended for use with keyboard devices, PDP-11 COBOL allows the ACCEPT statement to accept cards from a card reader, and the DISPLAY statement to display data on a line printer.

File Organization
The sequential I/O, relative I/O, and indexed I/O modules meet the full ANS-74 high-level standards and include all the COBOL verbs. For indexed file organization, this means that the user can build and process indexed files with one major key and zero or more alternate keys. This multikey facility offers flexibility and power in the development of application systems and is a language feature supported by only a few COBOL implementations.

Library Facility
With PDP-11 COBOL the user has a full ANS-74 intermediate level Library facility which includes high-level features. All frequently used data descriptions and program text sections can be held in library files available to all programs. These files can then be copied into source programs to save unnecessary repetitions simultaneously during program preparation and to prevent a common source of errors.

Call Facility
PDP-11 COBOL supports the CALL statement. This language feature allows COBOL programs to invoke separately compiled subprograms, passing arguments in the process. These subprograms may be written in COBOL or other languages. This facility has the following advantages:

● provides flexibility through modular development of application systems

● permits functional specification of small, well-defined source modules

● gives the user access to operating system-dependent features via subroutines written in MACRO-11

Debugging Features
To make program debugging easier, the COBOL compiler produces source language listings with embedded diagnostics. Fully descriptive diagnostic messages are listed at the point of error. Over 400 different error conditions are checked—varying from simple warnings to major error detections.

When the compiler detects an error in the source program, the compiler attempts to recover from an error and continue to compile the program. The kind of error message, whether informational, warning, or fatal, indicates a likelihood that the assumption made to recover from the error will produce an object program that will run as the programmer intended. Normally, the COBOL compiler will not generate an object program if major fatal errors are detected. The user can, however, force the compiler to produce an object program by requesting that it accept a fatal errors command string. This facility is provided as an extra debugging option. It can be useful in shortening the compile-debug cycle, particularly if applied to large COBOL programs which take considerable compilation time, but it should be used with caution.

Debugging large source programs is made still easier by the use of the optional Data Division allocation map and modular programming techniques offered by the segmentation and inter-program communication facilities.

Another useful debugging aid is the optional cross-reference listing produced by the compiler. This is a listing of all data names, procedure names, and the source line numbers of those program lines in which the definitions and references are contained. For each name, a list of ordered source line numbers is displayed. Definitional source line numbers are distinguished from reference source line numbers.

COMPILER IMPLEMENTATION

PDP-11 COBOL is a full compiler language and can be envisioned as a three-step process. The first step is compilation, in which the compiler translates the source program into an object module and also produces an overlay description language (ODL) file. This file describes the overlay tree structure associated with the generated object module. The object module itself is not in executable format, as it needs to be processed by a linker.

In the likely event that more than one COBOL-produced object module is to be linked to produce an executable task, it is necessary to "merge" the associated ODL files before the linking operation. Thus, the second step in the process is the ODL merge operation, which is performed by the MRG utility. This utility merges the ODL files from more than one compilation into a single, composite ODL file.

The third step in the process is linking, or task building. The object files, together with the composite ODL file, are input to the system linker to produce an executable image. COBOL-produced object modules can be processed by themselves, or they can be linked with

object files produced from other language processors. In addition to linking object modules produced by the PDP-11 COBOL compiler, the linker automatically selects the required routines from the COBOL System Object Library and from the RMS-11 Object Library. After the object files are linked, the executable task is ready to run.

Memory Segmentation

Programs are overlaid according to the standard COBOL segmentation module. If the users choose, however, they can request the compiler to break up the program and overlay it on a user-specified code segment size.

Flexibility

The PDP-11 COBOL compiler and run-time system has the flexibility to allow it to be built for various operating system configurations. Specifically, the size of the COBOL task can be increased or decreased depending on the amount of memory available on a particular configuration.

The size of the COBOL task image is a general determinant of the maximum size of a COBOL program, the speed at which it can compile, and the speed at which it can execute. In general, the larger the COBOL task, the faster it will run and the greater its capacity to execute very large COBOL programs.

When the compiler code or run-time code needs to use the disk work files, it obviously takes a longer time to run. By expanding the size of the internal work areas, the user can eliminate the need to access the work file so often. This means faster compilation and execution.

The COBOL task size can vary from approximately 24K words to 28K words on RSTS/E, and 27K to 32K ON RSX-11M, IAS, and TRAX. The size variation depends on the size of the work area selected by the system manager when building the COBOL task for the particular configuration on which it will execute.

COBOL OPERATING SYSTEM ENVIRONMENTS

PDP-11 COBOL is available under four operating systems: RSTS/E, RSX-11M, IAS, and TRAX. In general, PDP-11 COBOL is completely independent of the operating systems on which it is available. It uses RMS for file handling. It does not use operating system directives or function calls. The commands used to invoke the compiler and run-time system are basically the same from system to system. The following paragraphs briefly state the implementation characteristics under each of the operating systems.

UNDER RSTS/E

Under RSTS/E, the COBOL task can run in either interactive or batch

mode. The files that COBOL programs create can be read by BASIC-PLUS-2 programs using ASCII sequential file processing techniques (INPUT, READ, and PRINT statements).

UNDER IAS AND RSX-11M
The PDP-11 COBOL compiler requires at least 27K words of memory to compile all elements of the COBOL language. The COBOL task runs, by default, in the general system-controlled GEN partition. This partition should be at least 27K words on a 48K system. This allows other tasks to reside in the partition when COBOL is running.

COBOL operates as any other language processor under IAS with these exceptions: there is no directive use and there can be no inter-task communication. COBOL does use RMS and files created using COBOL are therefore compatible with all other RSX-11 or IAS files.

UNDER TRAX
The PDP-11 COBOL compiler requires at least 27K words of memory to compile all elements of the COBOL language. Under TRAX, a COBOL task can run in either the applications environment or the support environment. COBOL user I/O is implemented with RMS-11 and, in the applications environment, the user's I/O buffer space is allocated out of system space.

UTILITY PROGRAMS
PDP-11 COBOL offers three utility programs to aid the user with data processing.

MERGE   Merges ODL files generated by COBOL compilations into a single ODL file.

RFRMT   Converts PDP-11 terminal format COBOL programs into conventional format ANS COBOL programs.

COBRG   A high-level language with efficient, commercially oriented problem solving capabilities. With COBRG, vital reports can be developed quickly by saving tedious, time-consuming format coding.

The RFRMT (Reformat) utility program reads COBOL source programs that were coded using terminal format and converts the source statements to 80-column conventional format. PDP-11 COBOL accepts source programs that are coded using either the conventional 80-column card reference format or the shorter, easy-to-enter terminal format.

- **Terminal Format** is designed for easy use with context editors controlled from an on-line terminal keyboard, and is therefore compatible for use with PDP-11 systems. It eliminates the line-number and

identification fields. It allows horizontal tab characters and short lines and therefore offers potential savings in disk space.

- **Conventional Format** produces source programs that are compatible with the reference format of other COBOL compilers thoughout the industry.

RFRMT allows the programmer to enter source programs in the simpler terminal format and then, if compatibility is ever required for those programs, provides a simple method for conversion to conventional format.

The COBRG (COBOL Report Generator) utility program provides a fast, simple mechanism for producing printed reports from data files. COBRG recognizes input specification lines which enable the user to:

- define a report's page headings
- describe the format of input and output files
- set the rules for moving information from input records to detail output lines
- set the rules for adding values in accumulators
- set the rules for monitoring the sort keys for value changes
- set the rules for constructing and printing the accumulated values

COBRG uses these specification lines to produce a tailored COBOL source program. This program, when compiled and executed, generates the actual report.

Table 19-1 COBOL Language Implementations

Language Elements	Level	Module	ANS-68	ANS-74	TOPS-10 COBOL V6	PDP-11 COBOL V3
CHARACTER SET						
Words						
0,1,...9 A,B,...Z -	1	NUC	X	X	X	X
Punctuation						
. ' ( ) space or blank	1	NUC	X	X	X	X
=	1	NUC	-	X	-	-
, ;	2	NUC	X	X	X	X
'		EXT	-	-	X	X
Arithmetic + - * / * *	2	NUC	X	X	X	-
Relational < > =	2	NUC	X	X	X	X
Editing						
B 0 + - CR DB Z * $. ,	1	NUC	X	X	X	X
/	1	NUC	-	X	-	X
Separators						
; and , not permitted	1	NUC	X	X	-	-
; and , are permitted	1	NUC	X	X	X	X
COBOL WORDS (max 30 chars)	1	NUC	X	X	X	X
User-defined Words	1	NUC	X	X	X	X
cd-name	1	COM	-	X	X	-
condition name	2	NUC	X	X	X	X
data-name (1st char alpha)	1	NUC	X	X	-	-
data-name	2	NUC	X	X	X	X
file-name	1	SEQ	X	X	X	X
index-name	1	TBL	X	X	X	X
level-number	1	NUC	X	X	X	X
library-name	2	LIB	X	X	X	-
mnemonic-name	1	NUC	X	X	X	X
paragraph-name	1	NUC	X	X	X	X
program-name	1	NUC	X	X	X	X
record-name	1	SEQ	X	X	X	X
report-name	1	RPW	X	X	X	-
routine-name (optional)	1	NUC	X	X	X	-
section-name	1	NUC	X	X	X	X
segment-number	1	SEG	X	X	X	X
text-name	1	LIB	X	X	X	X
System Names	1	NUC	X	X	X	X
computer-name			X	X	X	X
implementor-name			X	X	X	X
language-name (optional)			X	X	X	-
Reserved Words	1	NUC	X	X	X	X
key words	1	NUC	X	X	X	X
optional words	1	NUC	X	X	X	X
qualifier connectives: OF IN	2	NUC	X	X	X	X
series connectives: , ;	2	NUC	X	X	X	X
logical connectives:						
AND, OR, AND NOT, OR NOT	2	NUC	X	X	X	X
LINE-, PAGE-COUNTER registers	1	RPW	X	X	X	-

Table 19-1 (cont) **COBOL Language Implementations**

Language Elements	Level	Module	ANS-68	ANS-74	TOPS-10 COBOL V6	PDP-11 COBOL V3
LINAGE-COUNTER register	2	SEQ	-	X	-	X
DEBUG-ITEM register	1	DEB	-	X	-	-
TALLY register	1	NUC	X	-	X	X
ZERO constant	1	NUC	X	X	X	X
ZEROS, ZEROES constants	2	NUC	X	X	X	X
SPACE constants	1	NUC	X	X	X	X
SPACES contant	2	NUC	X	X	X	X
HIGH-VALUE, LOW-VALUE	1	NUC	X	X	X	X
HIGH-VALUES, LOW-VALUES	2	NUC	X	X	X	X
QUOTE constant	1	NUC	X	X	X	X
QUOTES constant	2	NUC	X	X	X	X
ALL literal	2	NUC	X	X	X	X
arithmetic special chars	2	NUC	X	X	X	X
relational special chars	2	NUC	X	X	X	X
non-numeric literals (1-120 chars)	1	NUC	X	X	X	X
quote within non-numeric literals			-	X		X
numeric literals (1-18 chars)			X	X	X	X
PICTURE strings	1	NUC	X	X	X	X
comment entries	1	NUC	-	X	X	X
No Qualification Permitted	1	NUC	X	X	-	X
Qualification Permitted	2	NUC	X	X	X	X
Subscripting to 3 Levels	1	TBL	X	X	X	X
Indexing to 3 Levels	1	TBL	X	X	X	X
IDENTIFICATION DIVISION						
PROGRAM-ID	1	NUC	X	X	X	X
AUTHOR	1	NUC	X	X	X	X
INSTALLATION	1	NUC	X	X	X	X
DATE-WRITTEN	1	NUC	X	X	X	X
DATE-COMPILED	2	NUC	X	X	X	X
SECURITY	1	NUC	X	X	X	X
REMARKS	1	NUC	X	-	X	X
ENVIRONMENT DIVISION						
Configuration Section						
Can be omitted		EXT	-	-	X	-
SOURCE-COMPUTER	1	NUC	X	X	X	X
WITH DEBUGGING MODE	1	DEB	-	X	-	-
OBJECT-COMPUTER	1	NUC	X	X	X	X
MEMORY SIZE	1	NUC	X	X	X	X
COLLATING SEQUENCE	1	NUC	-	X	-	X
SEGMENT-LIMIT	2	SEG	X	X	X	-
SPECIAL-NAMES	1	NUC	X	X	X	X
ON STATUS	1	NUC	X	X	X	X
OFF STATUS	1	NUC	X	X	X	X
SPECIAL-NAMES series	1	NUC	X	X	X	X

Table 19-1 (cont) COBOL Language Implementations

Language Elements	Level	Module	ANS-68	ANS-74	TOPS-10 COBOL V6	PDP-11 COBOL V3
STANDARD-1 alphabet-name	1	NUC	-	X	-	X
NATIVE alphabet-name	1	NUC	-	X	-	X
Implementor-name alphabet-name	1	NUC	-	X	-	-
literal alphabet-name	2	NUC	-	X	-	-
CURRENCY SIGN	1	NUC	X	X	X	X
DECIMAL-POINT	1	NUC	X	X	X	X
Input-Output Section						
FILE-CONTROL SELECT	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	-
	1	SRT	X	X	X	-
OPTIONAL	2	SEQ	X	X	X	X
ASSIGN TO implementor-name	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	-	X
	1	SRT	X	X	X	-
MULTIPLE REEL/UNIT	1	SEQ	X	-	X	-
	1	SRT	X	-	X	-
RESERVE integer AREA(S)	2	SEQ	X	X	X	X
FILE-LIMIT literal THRU literal	1	SEQ	X	X	X	X
	1	RAC	X	-	X	-
FILE-LIMIT literal series	2	SEQ	X	-	-	-
	2	RAC	X	-	X	-
data-name THRU data-name	2	SEQ	X	-	-	-
	2	RAC	X	-	X	-
FILE-LIMIT data-name series	2	SEQ	X	-	-	-
	2	RAC	X	-	X	-
SEQUENTIAL ORGANIZATION	1	SEQ	-	X	-	X
RELATIVE ORGANIZATION	1	REL	-	X	-	X
INDEXED ORGANIZATION	1	INX	-	X	-	X
SEQUENTIAL ACCESS MODE	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	-	-
RANDOM ACCESS MODE	1	REL	-	X	-	X
	1	INX	-	X	-	X
DYNAMIC ACCESS MODE	2	REL	-	X	-	X
	2	INX	-	X	-	-
PROCESSING MODE SEQ	1	SEQ	X	-	X	-
	1	RAC	X	-	X	-
ACTUAL KEY	1	RAC	X	-	X	-
RECORDING MODE		EXT	-	-	X	X
RELATIVE KEY	1	REL	-	X	-	X
RECORD KEY	1	INX	-	X	X	-

Table 19-1 (cont) **COBOL Language Implementations**

Language Elements	Level	Module	ANS-68	ANS-74	TOPS-10 COBOL V6	PDP-11 COBOL V3
SYMBOLIC KEY		EXT	-	-	X	-
ALTERNATE RECORD KEY	2	INX	-	X	-	-
FILE STATUS	1	SEQ	-	X	-	X
	1	REL	-	X	-	X
	1	INX	-	X	-	X
I-O CONTROL: RERUN	1	SEQ	X	X	X	-
	1	REL	-	X	-	-
	1	RAC	X	-	X	-
	1	INX	-	X	-	X
SAME AREA	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	X
SAME RECORD AREA	2	SEQ	X	X	X	x
	2	REL	-	X	-	X
	2	RAC	X	-	X	X
	2	INX	-	X	X	-
	2	SRT	X	X	X	-
SAME SORT-MERGE AREA	2	SRT	X	X	X	-
SAME series	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	X
MULTIPLE FILE TAPES	2	SEQ	X	X	X	X
APPLY		EXT	-	-	-	X
DATA DIVISION						
Communication Section	1	COM	-	X	X	-
File Section	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	iNX	-	X	X	-
	1	SRT	X	X	X	-
	1	RPW	X	X	X	-
linkage section	1	IPC	-	X	X	-
working storage section	1	NUC	X	X	X	X
report section	1	RPW	X	X	X	-
communications description entry	1	COM	-	X	X	-
data description entry	1	NUC	X	X	X	X
file description entry	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	-
	1	RPW	X	X	X	-
record description entry	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	-

Table 19-1 (cont) COBOL Language Implementations

Language Elements	Level	Module	ANS-68	ANS-74	TOPS-10 COBOL V6	PDP-11 COBOL V3
LINAGE clause	2	SEQ	-	X	-	X
LINE NUMBER clause	1	RPW	X	X	X	-
NEXT GROUP clause	1	RPW	X	X	X	-
OCCURS clause						
integer times	1	TBL	X	X	X	X
ASCENDING/DESCENDING	2	TBL	X	X	X	X
data-name series	2	TBL	X	X	X	X
INDEXED BY index-name	1	TBL	X	X	X	X
i1 TO i2 DEPENDING ON	2	TBL	X	X	X	-
PAGE clause	1	RPW	X	X	X	-
PICTURE clause						
Character string max 30 chars	1	NUC	X	X	X	X
Data characters: A X 9	1	NUC	X	X	X	X
Operational symbols: S V P	1	NUC	X	X	X	X
Fixed insertion characters:						
0 B , . $ + - CR DB	1	NUC	X	X	X	X
Fixed insertion character: /	1	NUC	X	X	-	X
Replacement chars: $ + - Z *	1	NUC	X	X	X	X
Currency sign substitution	1	NUC	X	X	X	X
Decimal point substitution	1	NUC	X	X	X	X
RECORD CONTAINS clause	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	-
	1	SRT	X	X	X	-
	1	RPW	X	X	X	-
REDEFINES clause (no nesting)	1	NUC	X	X	-	X
REDEFINES clause (nesting)	2	NUC	X	X	X	X
RENAMES clause	2	NUC	X	X	X	X
REPORT clause	1	RPW	X	X	X	-
SIGN clause	1	NUC	-	X	-	X
SOURCE clause	1	NUC	X	X	X	-
SUM clause	1	NUC	X	X	X	-
SYNCHRONIZED clause	1	NUC	X	X	X	X
TYPE clause	1	NUC	X	X	X	-
USAGE clause						
COMPUTATIONAL (means binary)	1	NUC	X	X	X	X
COMP-1 (floating point)		EXT	-	-	X	-
DISPLAY	1	NUC	X	X	X	X
DISPLAY-6 (SIXBIT)		EXT	-	-	X	X
DISPLAY-7 (ASCII)		EXT	-	-	X	X
INDEX	1	TBL	X	X	X	X
DATABASE-KEY		DBM	-	-	X	-
VALUE clause						
literal	1	NUC	X	X	X	X
literal series	2	NUC	X	X	X	X
literal THRU literal	2	NUC	X	X	X	X

Table 19-1 (cont) **COBOL Language Implementations**

Language Elements	Level	Module	ANS-68	ANS-74	TOPS-10 COBOL V6	PDP-11 COBOL V3
literal range series	2	NUC	X	X	X	X
VALUE OF clause						
implementor-name IS literal	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	-
	1	RPW	X	X	X	-
implementor-name IS data-name	2	SEQ	X	X	X	X
	2	REL	-	X	-	X
	2	RAC	X	-	X	-
	2	INX	-	X	X	X
	2	RPW	X	X	X	-
PROCEDURE DIVISION						
USING phrase	1	IPC	-	X	X	X
Declaratives	1	SEQ	X	X	-	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	-	X
	1	RPW	X	X	X	-
	1	DEB	-	X	-	-
Arithmetic expressions	2	NUC	X	X	X	X
Conditional expressions	1	NUC	X	X	X	X
Simple conditions	1	NUC	X	X	X	X
Relation conditions	1	NUC	X	X	X	X
[NOT] GREATER THAN	1	NUC	X	X	X	X
[NOT] >	2	NUC	X	X	X	X
[NOT] LESS THAN	1	NUC	X	X	X	X
[NOT] <	2	NUC	X	X	X	X
[NOT] EQUAL TO	1	NUC	X	X	X	X
[NOT] =	2	NUC	X	X	X	X
EQUALS		EXT	-	-	X	-
numeric operands	1	NUC	X	X	X	X
nonnumeric operands (equal size)	1	NUC	X	X	-	-
nonnumeric (may be unequal)	2	NUC	X	X	X	X
Class conditions	1	NUC	X	X	X	X
NOT option	1	NUC	X	X	X	X
Switch-status condition	1	NUC	X	X	X	X
NOT option		EXT	-	-	X	X
Condition-name condition	2	NUC	X	X	X	X
NOT option		EXT	-	-	X	X
Sign condition	2	NUC	X	X	X	X
NOT option	2	NUC	X	X	X	X
Logical AND OR and NOT	2	NUC	X	X	X	X
Negated simple conditions	2	NUC	X	X	X	X
Combined and negated combined	2	NUC	X	X	X	X
Abbreviated combined relation	2	NUC	X	X	X	X

Table 19-1 (cont) **COBOL Language Implementations**

Language Elements	Level	Module	ANS-68	ANS-74	TOPS-10 COBOL V6	PDP-11 COBOL V3
Arithmetic operands	1	NUC	X	X	X	X
Overlapping operands	1	NUC	X	X	X	X
	1	TBL	X	X	X	X
Multiple arithmetic results	2	NUC	X	X	X	X
ACCEPT statement						
Only one transfer of data	1	NUC	X	X	-	X
No restrictions on transfers	2	NUC	X	X	X	X
FROM	2	NUC	X	X	X	X
FROM DATE	2	NUC	-	X	*	X
FROM DAY	2	NUC	-	X	-	-
FROM TIME	2	NUC	-	X	*	X
MESSAGE COUNT	1	COM	-	X	X	-
ADD statement						
indentifier literal series	1	NUC	X	X	X	X
TO identifier	1	NUC	X	X	X	X
TO identifier series	2	NUC	X	X	X	X
GIVING identifier	1	NUC	X	X	X	X
GIVING identifier series	2	NUC	X	X	X	X
ROUNDED	1	NUC	X	X	X	X
SIZE ERROR	1	NUC	X	X	X	X
CORRESPONDING	2	NUC	X	X	X	X
ALTER procedure-name	1	NUC	X	X	X	X
ALTER procedure-name series	2	NUC	X	X	X	X
CALL literal	1	IPC	-	X	X	-
CALL identifier	2	IPC	-	X	X	-
CALL USING data-name	1	IPC	-	X	X	-
CALL ON OVERFLOW	2	IPC	-	X	-	-
CANCEL statement	2	IPC	-	X	-	-
CLOSE single file-name	1	SEQ	X	X	X	X
CLOSE file-name series	2	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	X
REEL	1	SEQ	X	X	X	X
UNIT	1	SEQ	X	X	X	X
NO REWIND	2	SEQ	X	X	X	X
LOCK	2	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	X
FOR REMOVAL	2	SEQ	-	X	-	X
WITH DELETE		EXT	-	-	X	-
		DBM	-	-	X	-
COMPUTE identifier series	2	NUC	X	X	-	X
DELETE statement	1	REL	-	X	-	X
	1	INX	-	X	X	X
		DBM	-	-	X	-

Table 19-1 (cont) COBOL Language Implementations

Language Elements	Level	Module	ANS-68	ANS-74	TOPS-10 COBOL V6	PDP-11 COBOL V3
DISABLE statement						
INPUT	1	COM	-	X	X	-
TERMINAL INPUT	2	COM	-	X	X	-
OUTPUT	1	COM	-	X	X	-
KEY identifier/literal	1	COM	-	X	X	-
DISPLAY statement						
only one transfer of data	1	NUC	X	X	-	-
no restriction	2	NUC	X	X	X	X
UPON	2	NUC	X	X	X	X
WITH NO ADVANCING		EXT	-	-	X	X
DIVIDE statement						
INTO identifier	1	NUC	X	X	X	X
INTO identifier series	2	NUC	X	X	-	X
BY identifier	1	NUC	X	X	X	X
GIVING identifier	1	NUC	X	X	X	X
GIVING identifier series	2	NUC	X	X	-	X
ROUNDED	1	NUC	X	X	X	X
REMAINDER	2	NUC	X	X	X	X
SIZE ERROR	1	NUC	X	X	X	X
ENABLE statement						
INPUT	1	COM	-	X	X	-
TERMINAL INPUT	2	COM	-	X	X	-
OUTPUT	1	COM	-	X	X	-
KEY identifier/literal	1	COM	-	X	X	-
ENTER statement (optional)	1	NUC	X	X	X	-
ENTRY statement		EXT	-	-	X	-
EXAMINE statement	1	NUC	X	-	X	-
EXIT statement	1	NUC	X	X	X	X
EXIT PROGRAM statement	1	IPC	-	X	X	X
FIND statement		DBM	-	-	X	-
GENERATE statement	1	RPW	X	X	X	-
GET statement		DBM	-	-	X	-
GOBACK statement		EXT	-	-	X	-
GO TO statement						
TO optional	1	NUC	-	X	-	X
procedure-name required	1	NUC	X	X	-	X
procedure-name optional	2	NUC	X	X	X	X
DEPENDING ON phrase	1	NUC	X	X	X	X
IF statement						
statements must be imperative	1	NUC	X	X	-	-
nested statements	2	NUC	X	X	X	X
ELSE	1	NUC	X	X	X	X
		DBM	-	-	X	-
INITIATE statement	1	RPW	X	X	X	-
INSERT statement	1	NUC	X	X	X	X
		DBM	-	-	X	-

Table 19-1 (cont) **COBOL Language Implementations**

Language Elements	Level	Module	ANS-68	ANS-74	TOPS-10 COBOL V6	PDP-11 COBOL V3
INSPECT statement						
single character data item	1	NUC	-	X	-	X
multi-character data item	2	NUC	-	X	-	X
INVOKE statement		DBM	-	-	X	-
MERGE statement	2	SRT	-	X	-	-
MODIFY statement		DBM	-	-	X	-
MOVE statement						
TO identifier	1	NUC	X	X	X	X
TO identifier series	1	NUC	X	X	X	X
CORRESPONDING	2	NUC	X	X	X	X
		DBM	-	-	X	-
MULTIPLY statement						
BY identifier	1	NUC	X	X	X	X
BY identifier series	2	NUC	X	X	X	X
GIVING identifier	1	NUC	X	X	X	X
GIVING identifier series	2	NUC	-	X	-	X
ROUNDED	1	NUC	X	X	X	X
SIZE ERROR	1	NUC	X	X	X	X
NOTE sentence	1	NUC	X	-	X	-
OPEN statement						
INPUT single file-name	1	SEQ	X	X	X	X
INPUT file-name series	2	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	-
	1	NUC	X	X	X	X
INPUT REVERSED	2	SEQ	X	X	-	-
INPUT NO REWIND	2	NUC	X	X	X	X
OUTPUT single file-name	1	SEQ	X	X	X	X
OUTPUT file-name series	2	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	-
OUTPUT NO REWIND	2	SEQ	X	X	X	X
I-O single file-name	1	SEQ	X	X	X	X
I-O file-name series	2	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	-
EXTEND	2	SEQ	-	X	-	X
INPUT, OUTPUT, I-O, EXTEND	2	SEQ	-	X	-	X
INPUT, OUTPUT and I-O series	1	SEQ	X	-	X	-
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	-
		DBM	-	-	X	-

Table 19-1 (cont) **COBOL Language Implementations**

Language Elements	Level	Module	ANS-68	ANS-74	TOPS-10 COBOL V6	PDP-11 COBOL V3
PERFORM statement						
procedure name	1	NUC	X	X	X	X
THRU	1	NUC	X	X	X	X
TIMES	1	NUC	X	X	X	X
UNTIL	2	NUC	X	X	X	X
VARYING	2	NUC	X	X	X	X
READ statement						
file-name	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	-
INVALID KEY	1	REL	-	X	-	X
	1	RAC	X	-	X	-
INTO identifier	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	-	-
NEXT	2	REL	-	X	-	X
	2	INX	-	X	*	-
AT END	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	-
KEY IS	2	INX	-	X	-	-
RECEIVE statement						
MESSAGE	1	COM	-	X	X	-
SEGMENT	2	COM	-	X	X	-
INTO identifier	1	COM	-	X	X	-
NO DATA phrase	1	COM	-	X	X	-
RELEASE statement						
record name	1	SRT	X	X	X	-
FROM	1	SRT	X	X	X	-
REMOVE statement		DBM	-	-	X	-
RETURN statement						
file-name	1	SRT	X	X	X	-
INTO	1	SRT	X	X	X	-
AT END	1	SRT	X	X	X	-
REWRITE statement						
FROM identifier	1	SEQ	-	X	-	X
	1	REL	-	X	-	X
	1	INX	-	X	X	X
INVALID KEY phrase	1	REL	-	X	-	X
	1	INX	-	X	X	X
SEARCH statement	2	TBL	X	X	X	-
SEEK statement	1	RAC	X	-	X	-

Table 19-1 (cont) **COBOL Language Implementations**

Language Elements	Level	Module	ANS-68	ANS-74	TOPS-10 COBOL V6	PDP-11 COBOL V3
SEND statement						
FROM identifier	2	COM	-	X	X	-
FROM identifier WITH	1	COM	-	X	X	-
WITH identifier	2	COM	-	X	X	-
WITH EGI	1	COM	-	X	X	-
WITH EMI	1	COM	-	X	X	-
BEFORE/AFTER ADVANCING	1	COM	-	X	X	-
SET statement	1	TBL	X	X	X	X
SORT statement						
max 1 SORT STOP, I-O procedure	2	SRT	X	X	-	-
not limited to 1 SORT	2	SRT	X	X	X	-
COLLATING SEQUENCE phrase	2	SRT	-	X	-	-
START statement	2	REL	-	X	-	X
	1	INX	-	X	-	X
STOP statement	1	NUC	X	X	X	X
STORE statement		DBM	-	-	X	-
STRING statement	2	NUC	-	X	X	X
SUBTRACT statement						
indentifier/literal series	1	NUC	X	X	X	X
FROM	1	NUC	X	X	X	X
FROM series	2	NUC	X	X	X	X
GIVING identifier	1	NUC	X	X	X	X
GIVING identifier series	2	NUC	X	X	X	X
ROUNDED	1	NUC	X	X	X	X
SIZE ERROR	1	NUC	X	X	X	X
CORRESPONDING	2	NUC	X	X	X	X
SUPPRESS statement	1	RPW	-	X	-	-
TERMINATE statement	1	RPW	-	X	-	-
TRACE statement		EXT	-	-	X	-
UNSTRING statement	2	NUC	-	X	X	X
USE statement		DBM	-	-	X	-
EXCEPTION/ERROR PROCEDURE						
ON fil-nam/INPUT/OUTPUT/I-O	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	X
On file-name series	2	SEQ	X	X	-	X
	2	REL	-	X	-	X
	2	RAC	X	-	X	-
	2	INX	-	X	-	X
ON EXTEND	2	SEQ	-	X	-	X
LABEL PROCEDURE	2	SEQ	X	-	X	-
BEFORE REPORTING	1	RPW	X	X	X	-
USE FOR DEBUGGING statement						
procedure-name	1	DEB	-	X	-	-
procedure-name series	1	DEB	-	X	-	-
ALL PROCEDURES	1	DEB	-	X	-	-

Table 19-1 (cont) **COBOL Language Implementations**

Language Elements	Level	Module	ANS-68	ANS-74	TOPS-10 COBOL V6	PDP-11 COBOL V3
ALL REFERENCES OF identifier	2	DEB	-	X	-	-
file-name series	2	DEB	-	X	-	-
cd-name series	2	DEB	-	X	-	-
WRITE statement						
record-name	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	X
FROM identifier	1	SEQ	X	X	X	X
	1	REL	-	X	-	X
	1	RAC	X	-	X	-
	1	INX	-	X	X	X
BEFORE/AFTER ADVANCING						
integer LINES	1	SEQ	X	X	X	X
identifier LINES	2	SEQ	X	X	X	X
mnemonic-name	2	SEQ	X	X	X	-
PAGE	1	SEQ	-	X	-	X
AT END-OF-PAGE	2	SEQ	-	X	-	X
INVALID KEY	1	REL	-	X	-	X
Positioning Clause	1	RAC	X	-	X	-
	1	INX	-	X	X	X
SEGMENTATION						
segment-number (priority-number)	1	SEG	X	X	X	X
Fixed Memory Range 0-49	1	SEG	X	X	X	X
Non-fixed Memory Range 50-99	1	SEG	X	X	X	X
SEGMENT-LIMIT	2	SEG	X	X	X	X
LIBRARY						
COPY	1	LIB	X	X	X	X
text-name	1	LIB	X	X	X	X
literal		EXT	-	-	-	X
literal OF/IN LIBRARY	2	LIB	-	X	-	-
literal REPLACING	2	LIB	X	X	X	X
used like a COBOL word	1	LIB	-	X	-	X
pseudo-text may be replaced	2	LIB	-	X	-	-
REFERENCE FORMAT						
Sequence numbers	1	NUC	X	X	X	X
may be omitted		EXT	-	-	X	X
Area A	1	NUC	X	X	X	X
Division header	1	NUC	X	X	X	X
Section header	1	NUC	X	X	X	X
Paragraph header	1	NUC	X	X	X	X
Data Division entries	1	NUC	X	X	X	X

Table 19-1 (cont) **COBOL Language Implementations**

Language Elements	Level	Module	ANS-68	ANS-74	TOPS-10 COBOL V6	PDP-11 COBOL V3
Area B	1	NUC	X	X	X	X
Paragraphs	1	NUC	X	X	X	X
Data Division Entries	1	NUC	X	X	X	X
Continuation Lines						
Nonnumeric literals	1	NUC	X	X	X	X
Words and numeric literals	2	NUC	X	X	X	X
Comments with *	1	NUC	X	X	X	X
Comments with /	1	NUC	X	X	-	X

Abbreviations:

X = feature implemented according to standard
- = feature not implemented
* = feature available through a non-standard method

NUC	= Nucleus	RPW	=	Report Writer
TBL	= Table Handling	SEG	=	Segmenation
SEQ	= Sequential I/O	LIB	=	Library
REL	= Relative I/O	DEB	=	Debugging
INX	= Indexed I/O	IPC	=	Inter-program comm
SRT	= Sort/Merge	COM	=	Communications

DIBOL-11/DECFORM (V1)

OVERVIEW
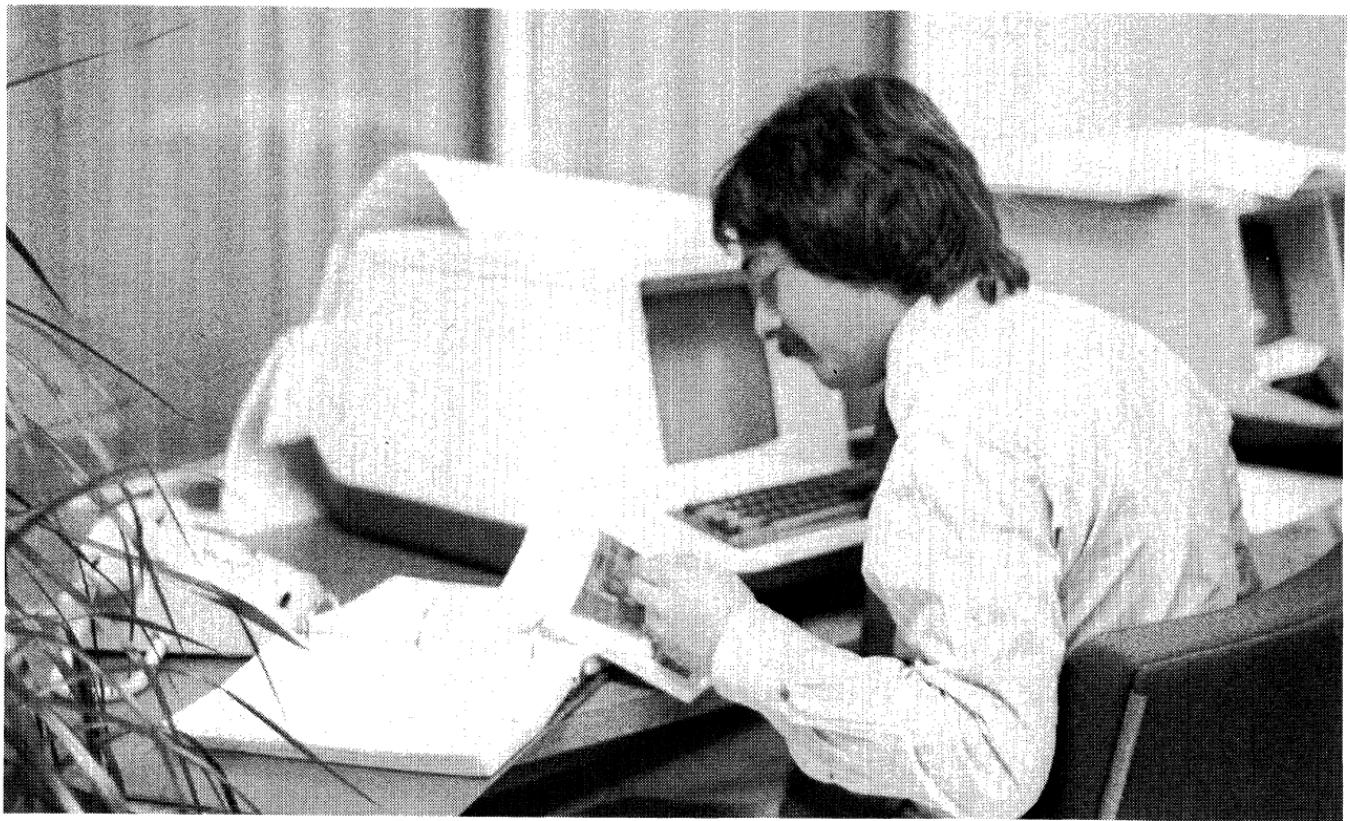
DIBOL-11/DECFORM is a software configuration option for DIGITAL's commercial data systems.The package includes the DIBOL-11 language processor and the DECFORM screen formatting and file review utility. DECFORM facilitates additions, reviews, changes, and verifications to files. DIBOL provides the ability to do data manipulation, arithmetic expression evaluation, subscripting of tables, redefinition of records, external calls to other programs, and both sequential and random access to files.

FEATURE TOPICS

- DIBOL Features
- Program Structure
- DIBOL Statements
    - Compiler Directive Statements
    - Data Specification Statements
    - Data Manipulation Statements
    - Control Statements
    - Input/Output Statements
- Subroutine Library
- DECFORM

Program Structure

A DIBOL-11 program is separated into two major parts: a Data Division and a Procedure Division, both of which are linked to a compiler. The Data Division identifies the data elements and the record structure of all the data files used within the program. The Procedure Division contains the logic and data manipulation statements that implement a particular application program.

DIGITAL BUSINESS-ORIENTED LANGUAGE IMPLEMENTATION
SOURCE LANGUAGE INPUT MEDIA
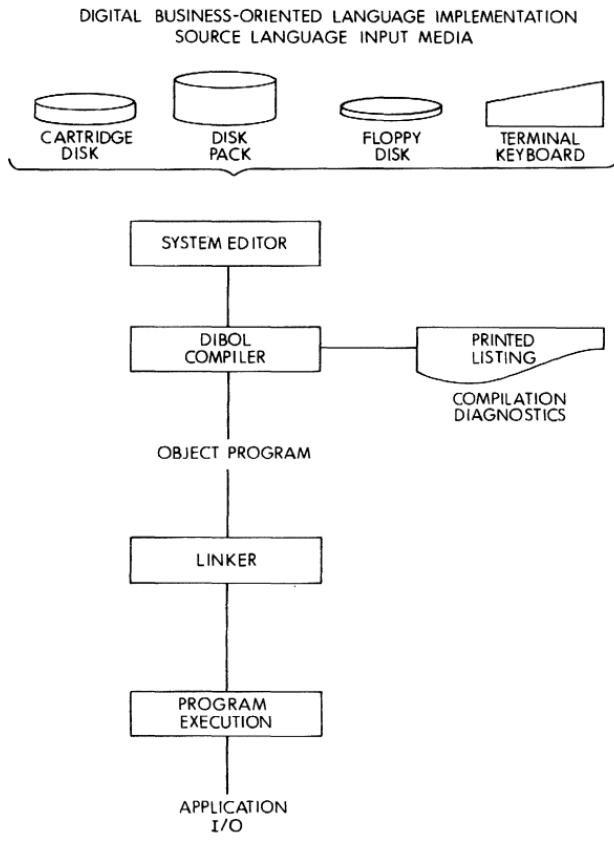
Figure 20-1

DIBOL-11 STATEMENTS

DIBOL-11 statements fall into five functional groups:

- Compiler (linked to both divisions) = Compiler directives
- Data Division = Data specifications
- Procedure Division = Data manipulation
  - Control statements
  - Input/Output statements

Compiler Directive Statements

These statements are not executable at run time and do not affect program operation. They are instructions to the compiler. The compiler directive statements include:

START   Causes the DIBOL-11 program listing to skip to a new page.

SUBROUTINE   Identifies this program as an external subroutine.

PROC   Separates Data Division statements from Procedure Division statements.

END   Ends the last statement in a program.

Data Division Data Specification Statements

Data specification statements (also referred to as field definition statements) define and identify the characteristics of the data processed by a DIBOL-11 program; for example, data can be either numeric or alphanumeric, have certain size requirements and initial values. Fields of data that are grouped together are preceded by a RECORD or COMMON statement, and may be redefined at that level. The data specification statements are:

RECORD   Defines the beginning of one or more grouped fields.

COMMON   Defines the beginning of one or more grouped fields and allows external subroutine to use/share a field without specifying it as an argument in an XCALL.

Procedure Division Data Manipulation Statements

These statements are used to perform calculations as well as data modification, conversion, and movement. They include both arithmetic and logical expressions, as stated below:

INCR   Increments a variable by one.

LOCASE   Converts uppercase letters to lowercase.

UPCASE   Converts lowercase letters to uppercase.

=   Moves_the results of the expression on the right of the equal sign to the left of the equal sign.

The operators in an expression represent various arithmetic and manipulative functions of the DIBOL-11 language. Operators are classified either as unary or binary operators. DIBOL-11 expression operators include:

Type	Symbol	Description
Arithmetic (unary)	+	Plus value
	−	Minus value or negation
Arithmetic (binary)	+	Addition
	−	Subtraction
	*	Multiplication
	/	Division
	#	Rounding
Relational (binary)	.EQ.	Equal
	.NE.	Not equal
	.GT.	Greater than
	.LT.	Less than
	.GE.	Greater than or equal to
	.LE.	Less than or equal to
Boolean (binary)	.AND.	Boolean conditional argument AND
	.OR.	Boolean conditional argument OR
Format (binary)		Formats converted decimal data

Control Statements

Control statements govern the order of a program's instructions by modifying the normal order of statement execution. The DIBOL-11 control statements are:

CALL   Calls a subroutine within the program.

XCALL   Calls an external subroutine.

GOTO   Transfers control to another statement.

IF   Executes a statement based on the results of a logical condition. IF uses relational and Boolean operators.

OFFERROR   Disables trapping of run-time errors.

ONERROR   Enables trapping of run-time errors.

RETURN   Returns from subroutine.

SLEEP   Suspends program operation for a specified time
                interval when operating in a timesharing environ-
                ment.

STOP   Terminates program execution and optionally chains
                to another program.

Input/Output Statements
Input/output statements control the transmission and reception of
data between memory and PDP-11 input/output devices such as the
disk, the line printer, and the terminal. The input/output statements
are:

ACCEPT   Receives a character from a device.

CLOSE   Terminates use of an input/output channel and
                closes the associated file.

DELETE   Deletes a record from an ISAM file.

DETACH   Disconnects the program from its terminal when op-
                erating in a time-sharing environment.

DISPLAY   Sends a character string to a device.

FORMS   Sends special form control characters used by line
                printers.

LPQUE   Requests printing of a file.

OPEN   Initializes a file in preparation for input/output oper-
                ations.

READ   Reads a record from a file (direct access).

READS   Reads the next record in sequence from a file.

RECV   Receives a message from another program.

SEND   Transmits a message to another program.

STORE   Stores a record into an ISAM file.

UNLOCK   Releases a record for use by another program when
                operating in a time-sharing environment.

WRITE   Writes a record to a file (direct access).

WRITES   Writes the next record in sequence to a file.

SUBROUTINE LIBRARY

The DIBOL-11 external subroutine capability allows the user to develop subroutines to perform special purpose functions. A program can include any of the following routines as well as any user may develop.

ASCII   ASCII equivalent of decimal character.

DATE   Return system date.

DECML   Decimal equivalent of ASCII character.

DELETE   Delete specified file on specified device.

ERROR   Return error code and line number of last error.

FLAGS   Set special run-time options.

MONEY   Define currency character for field editing.

RENAM   Rename a file.

RSTAT   Return size and terminating characters of last record read.

RUNJB   Start-up one job from another.

SLICE   Set interpreter cycle count for job priorities.

TIME   Return system time.

TNMBR   Return terminal number identification.

TTSTS   Return terminal status.

VERSN   Return run-time monitor version number.

DECFORM

DECFORM is DIGITAL's interactive, screen formatting, data entry, edit, file-review and maintenance system. It is a high-level code translator or "forms compiler" that operates with DIBOL-11 on both CTS-300 and CTS-500 operating systems. DECFORM brings to the small business environment many stand-alone, key-to-disk functions and file-review capabilities. In addition, it has its own forms-creation language and terminal.

Features

- Facilitates systems integration to permit processing of large quantities of data.

- Supports the special function keys of the VT50H and VT52 CRT terminals to provide cursor control and efficient operator interface.

- Offers custom programming options in the high-level DIBOL-11 language to aid in system flexibility.

- Offers an organized and structured approach to the implementation of data entry and file-review functions to provide easy applications implementation.

- Offers automatic edit checking of keyed input data at the source to reduce error rate and assure data integrity.

- Supports direct access to system files with both linear and ISAM search capability for maintenance and review to shorten application development time.

- Offers an easy-to-use forms-creation language. The format developer lays out each screen by line and column position, with associated edit checks and labels, to produce desired formats quickly.

TECHNICAL OVERVIEW

Format Control File

The code serving as input to DECFORM is called the Format Control File, which the format developer generates to specify the exact layout of data entry and review formats. A separate Format Control File is required for each format. DECFORM takes the code composing the Format Control File and translates it into DIBOL-11 program source statements with annotated comments. It is then compiled into intermediary code, linked, and stored. Each terminal using a DECFORM format operates independently of any other terminal and therefore requires its own copy of the linked format code. The amount of memory space necessary to handle each format (and thus each terminal) varies, depending on the size and complexity of the format. Generally, the space requirements are 4-6K words of memory per format.

Specially Designed Terminal

DECFORM utilizes its uniquely designed terminal keyboard located to the right of the VT52 and VT50H DECscope terminals. The VT52 is the recommended terminal because its 24-line by 80-character screen layout allows more room for screen formatting by offering 23 formatting lines.

Memory Requirements

The DECFORM compiler will execute on any D350 system running a single-job monitor or in the background partition of an 11/34-based D350 system supporting memory management. Compiled DECFORM programs require a minimum of 8-9K bytes per program. Maximum program size is primarily a function of the number of fields in a program, the number of files opened, and the number of edit checks per field. It is also greatly influenced by the size of supporting data tables and the size of any user-written subroutines. These factors determine the number of DECFORM programs that can be run simultaneously, sharing available memory with the monitor. For example, a standard D350 system with 56K bytes of memory could support more than one DECFORM program, sharing available memory with the CTS-300 monitor and ISAM.

System Supervisor Functions

Common to most key-to-storage data recording systems is a set of facilities generally referred to as system supervisor functions. The following section will outline those supervisory functions which are either standard or easily instituted under DECFORM.

1. Field resequencing on output of records to files:

    Under CTS-300 — Accomplished through DECFORM. Any ordering of fields may be created, independent of the displaying sequence, by so specifying in the DECFORM Format Control File.

    Under CTS-500 — Same as CTS-300.

2. Sequence numbering (that is, a method of identifying which terminal was responsible for sending a record, via communications):

    Under CTS-300 — Implementable under DECFORM via the programmable option, through a DIBOL-11 routine.

    Under CTS-500 — Using the send/receive function, the information of the sender terminal must be embedded in the message. This must be programmed.

3. Ability of the system supervisor to define constants (values) for specific fields:

    Under CTS-300 — Accomplished through DECFORM. At the time a format is created, the designer has the option of defining fields to constants and entering those constants, or having the operator enter the constants at the time the format is used. The system supervisor can also relate the constant directly to the operator via the display screen.

4. Ability to define the system parameters, that is, number of terminals and facilities available for use:

    Under CTS-300 — These are defined initially at the time of SYSGEN.

    Under CTS-500 — Falls out of the system definition functions which are specified at the time timesharing is started up.

5. Ability to initialize the system controlled by the system supervisor:

    Under CTS-300 — A normal function; performed by the system operator whenever the system is started up.

    Under CTS-500 — Most functions can be reset any time needed (for example, TTY SET, UTILITY). Some basic ones, however, (e.g., DEFAULT, REFRESH, DISKINIT, etc.) must be booted in specifying the start of time-sharing.

6. Loading (entering) of the date and time of day:

    Under CTS-300 — A normal function, performed by the system operator, whenever the system is up.

    Under CTS-500 — Using UTILITY, this can be accomplished at any time required.

7. Allow system supervisor to perform a DUMP (of data files entered):

    Under CTS-300 — The operating system provides a DUMP and PIP utility.

    Under CTS-500 — Using PIP, this is a standard function.

8. Interactive format definition and creation, where there is a question-and-answer dialog between the system and the format developer:

    Under CTS-300 — Through DECFORM, the forms creator generates a control file which specifies all parameters required. It is not interactive; however, the creator tells the system what is needed.

    Under CTS-500 — Same as CTS-300.

9. Data Substitution — substitution of data values in fields after the data entry operator has finished with them:

    Under CTS-300 — The data record can be passed to a (DIBOL-11) program running in the system, which will manipulate the fields before storing the record in a file.

    Under CTS-500 — Same as CTS-300, except BASIC-PLUS, BASIC-PLUS-2, and COBOL could be used as well as DIBOL-11.

10. Printing data (hard copy):

    Under CTS-300 — System provides a line printer spooler function called LPTSPL.

    Under CTS-500 — Same as CTS-300 facility but is called QUE.

11. Production (audit trail and operator) statistics:

    Under CTS-300 — Not presently implemented.

    Under CTS-500 — A utility called MONEY specifies much of this information by project/programmer number: connect time, CPU time, K-core ticks.

12. Remote terminals and data transmission:

    Under CTS-300 — Communications through the use of DL11s.

    Under CTS-500 — Same as CTS-300.

13. Supervisor's station (CRT) has control over other stations:

    Under CTS-300 — Through forced job start-up, one terminal can force the start of DECFORM jobs on other terminals.

    Under CTS-500 — Privileged users can force the start of DECFORM jobs on other terminals.

14. Supervisor can obtain a listing of jobs (or formats) each terminal is using:

    Under CTS-300 — Use of STATUS utility gives this information.

    Under CTS-500 — Use of VIDEO DISPLAY utility gives this information.

File Review and Update Modes

There are four file-review and update modes of importance to a commercial system utilizing DECFORM.

1. The **Inquiry** mode. This mode inquires into a record within a file. This mode allows the operator to review data previously entered as a record within the system files. (No changes to any information within the file are allowed.) Both sequential and indexed sequential (ISAM) files may be used under this review function.

2. The **Change** mode. This mode allows for both review and update of information in a record within a file. Once the record is displayed on the screen in the appropriate format, the operator may examine the data and make changes where necessary. This function is one of the primary methods by which files within the system are maintained. Both sequential and ISAM files are supported.

3. The **Delete** mode. This mode allows for selective deletion of records from a file. Again, the record is displayed on the screen, using the appropriate display format. Once the operator is sure that this is the record which is to be deleted, a message is sent to the system to delete it. The delete function is supported only under ISAM file structures.

4. The **Verification** mode. This mode is analogous to the verify mode found on many stand-alone, key-to-disk data entry systems. This function increases the reliability of the data. With the verify function, an operator reviews a record displayed on the screen and selectively re-keys data fields. If any of the re-keyed fields differ from the original contents of those fields, the operator is informed of this difference and asked which data is desired. As with the data which was originally entered, any changes must also pass any edit checks associated with the fields in question.

This function improves the overall accuracy of data entered into the system through the DECFORM formats.

Without key verification of input data, statistics have shown error rates in the neighborhood of 2-3 percent for data such as payroll input, accounts payable, and other financial data. **With keyed verification, error rates have been shown to be reduced to as little as one error in 100,000 key strokes.** In areas where a single wrong character or a missed field might cause serious problems, key verification of data maximizes the assurance of data integrity.

Application Example

A typical application using DECFORM capability is commercial loan processing (automobile loans, home improvement loans, etc.) Application for a loan involves filling out a complicated form with interrelated fields that must be completed correctly to avoid delay or, worse, rejection of the loan because of improperly formatted data. The easy construction of screen masks that DECFORM provides allows the loan department to construct different forms for different loans. The ability to verify the correctness of account numbers by doing immediate inquiries to local data bases is extremely important when a payment is due. The many editing features that DECFORM provides allow for entry of data with maximum efficiency and accuracy.
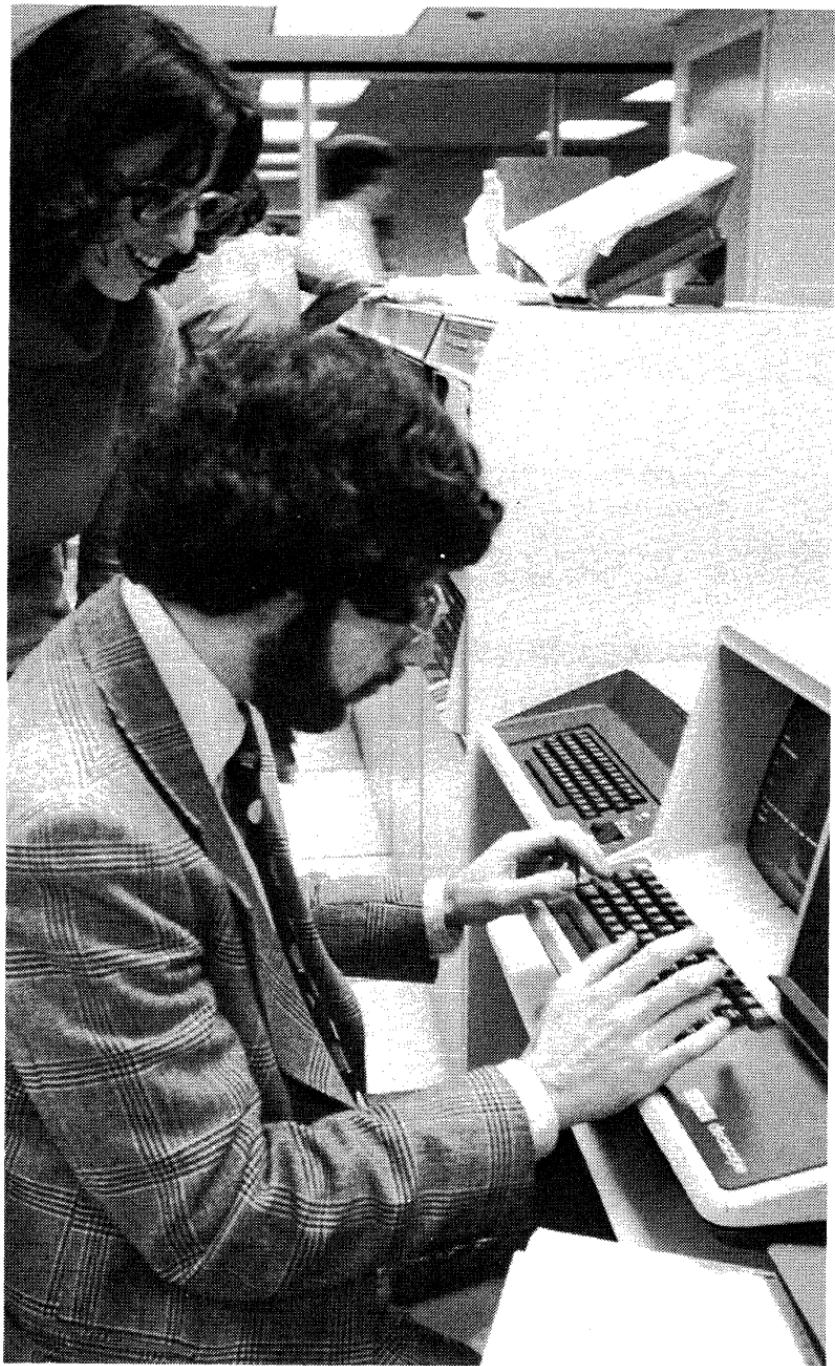
DIBOL-11

SAMPLE PROGRAM (DIBOL-11)

```
START      ;DATA SECTION
           RECORD IOBUF                          ;AREA USED FOR INPUT & OUTPUT
STOCKN,    D4                                    ;STOCK NUMBER, 4 DIGITS
DESC,      A25                                   ;DESCRIPTION, 25 ALPHA CHARS.
UCOST,     D5                                    ;UNIT COST, 5 DIGITS WITH ACCURACY
                                                 ;TO THREE (IMPLIED) DECIMAL PLACES
QORDER,    D4                                    ;QUANTITY ORDERED, 4 DIGITS
ECOST,     D8                                    ;EXTENDED COST 8 DIGITS WITH ACCURACY
                                                 ;TO TWO (IMPLIED) DECIMAL PLACES
           RECORD ,X                             ;REDEFINE THE ORIGINAL RECORD
MAXREC,    D6
           RECORD                                ;WORKING STORAGE AREA
COUNT,     D6, 000001                            ;RECORD COUNT
LIMIT,     D6

PROC(1)    ;PROCEDURE SECTION
           OPEN (1, U, 'INVENT')                 ;INITIALIZE CHANNEL 1 FOR INPUT/OUTPUT
                                                 ;DEVICE WITH FILENAME 'INVENT'
           READ (1, IOBUF, COUNT)                ;READ THE FIRST RECORD IN THIS FILE
           LIMIT = MAXREC                        ;SAVE MAXREC WHICH IS THE NUMBER OF
                                                 ;RECORDS IN THIS FILE
LOOP,      INCR COUNT                            ;ADD 1 TO COUNT
           READ (1, IOBUF, COUNT)                ;READ A RECORD FROM CHANNEL 1
           IF (STOCKN.LT.1000)GO TO LOOP         ;IF STOCK # IS UNDER 1000 GO TO THE
                                                 ;STATEMENT CALLED LOOP
           ECOST =(UCOST*QORDER)#1               ;MULTIPLY UNIT COST BY ORDER QUANTITY
                                                 ;STORE AS EXTENDED COST, ROUND &
                                                 ;TRUNCATE ONE DECIMAL PLACE BEFORE
                                                 ;STORING RESULT
           WRITE (1, IOBUF, COUNT)               ;WRITE THE MODIFIED RECORD OVER THE
                                                 ;OLD ONE
           IF (COUNT.NE.LIMIT)GO TO LOOP         ;IF NOT END OF FILE, GO BACK AND READ
                                                 ;ANOTHER RECORD
           CLOSE(1)                              ;INPUT/OUTPUTFILE IS CLOSED
           STOP 'REPRT8'                         ;END THIS PROGRAM AND START A PROGRAM
                                                 ;CALLED REPRT8
END                                              ;OPTIONAL STATEMENT INDICATING END OF
                                                 ;PROGRAM
```

FORTRAN

OVERVIEW
FORTRAN IV is a substantially improved form of the standard scientific programming language that can be used on any PDP-11 configuration. FORTRAN IV-PLUS is a superset of FORTRAN IV supporting the same enhancements to the ANS standard, but also including numerous extensions. The primary diferences between the two FORTRANs are that the FORTRAN IV-PLUS compiler produces highly optimized PDP-11 machine language code; creates hard code that uses the floating point processor option in high-end PDP-11s; and can produce sharable code.

Emphasis in this chapter is placed on the common features of the two languages.

FEATURE TOPICS
- FORTRAN Specifications and National Standards
- PDP-11 FORTRAN Language Description
  - Statements and Expression Operators
  - Fortran Library Functions
- Fortan IV Functions and Features
- Fortran IV Compiler Structure and Operation
- Fortran IV Operating System Environments
- Fortran IV-PLUS Functions and Features
  - Language Extensions and Statements
  - Library Functions
- Fortran IV-PLUS Compiler Structure and Operation
- Fortran IV-PLUS Operating System Environments

SPECIFICATIONS AND STANDARDS

The PDP-11 FORTRAN language is based on the specifications for the ANS FORTRAN X3.9-1966. The following are enhancements to the American National Standard:

- Array Subscripts — Any arithmetic expression can be used as an array subscript. If the value of the expression is not an integer, it is converted to integer type.

- Array Dimensions — Arrays can have up to seven dimensions.

- Alphanumeric Literals — Strings of characters bounded by apostrophes can be used in place of Hollerith constants.

- Mixed-mode Expressions — Mixed-mode expressions can contain any data type, including complex and byte.

- End of line comments — Any FORTRAN statement can be followed, in the same line, by a comment that begins with an exclamation point.

- Debugging Statements — Statements that are included in a program for debugging purposes can be so designated by the letter D in column 1. Those statements are compiled only when the associated compiler command string option switch is set. They are treated as comments otherwise.

- Read/Write End-of-file or Error Condition Transfer — The specifications END=n and ERR=n (where n is a statement number) can be included in any READ or WRITE statement to transfer control to the specified statement upon detection of an end-of-file or error condition. The ERR=n option is also permitted in the ENCODE and DECODE statements, allowing program control of data format errors.

- General Expressions in I/O lists — General expressions are permitted in I/O lists of WRITE, TYPE, and PRINT statements.

- General Expression DO and GO TO Parameters — General expressions are permitted for the initial value, increment, and limit parameters in the DO statement, and as the control parameter in the computed GO TO statement.

- DO Increment Parameter — The value of the DO statement increment parameter can be negative.

- Optional Statement Label List — The statement label list in an assigned GO TO is optional.

- Override Field Width Specifications — Undersized input data fields can contain external field separators to override the FORMAT field width specifications for those fields (called "short field termination"), permitting free-format input from terminals.

- Default FORMAT Widths — The FORTRAN IV programmer may specify input or output formatting by type and default width and precision values will be supplied.

- Additional I/O Statements:
  File Control and Attribute Definition
     OPEN
     CLOSE
  List-directed (free format) u = logical unit number
     READ (u,*)
     WRITE (u,*)
     TYPE*
     ACCEPT*
     PRINT*
  Device-oriented I/O
     ACCEPT
     TYPE
     PRINT
  Memory-to-memory formatting
     ENCODE
     DECODE
  Unformatted direct access I/O
     DEFINE FILE
     READ (u'r)   u = logical unit number
     WRITE (u'r)   r = record number
     FIND (u'r)
  The unformatted direct access I/O facility allows the FORTRAN programmer to read and write files written in any format.

- Logical Operations on INTEGER Data: The logical operators .AND., .OR., .NOT., .XOR., and .EQV. may be applied to integer data to perform bit masking and manipulation.

- Additional Data Type: The byte data type (keyword LOGICAL*1 or BYTE) is useful for storing small integer values as well as for storing and manipulating character information.

- IMPLICIT Declaration — The IMPLICIT redefines the implied data type of symbolic names.

PDP-11 FORTRAN LANGUAGE

A FORTRAN program consists of FORTRAN statements and optional comments. There are two kinds of statements; executable, and non-executable. Executable statements describe the action of the program. Non-executable statements describe the data arrangement and characteristics, and provide editing and data conversion information.

There are assignment statements, control statements, I/O statements, FORMAT statements and specification statements. FORMAT and specification statements are non-executable. Table 12-1 summarizes the PDP-11 FORTRAN language components.

Table 12-1 PDP-11 FORTRAN Language Summary

Expression Operators

TYPE	OPERATOR		OPERATES ON
Arithmetic	**	exponentiation	arithmetic or logical con-
	*	multiplication	stants, variables and ex-
	/	division	pressions
	+,-	addition, subtraction, unary plus and minus	
Relational	.GT.	greater than	arithmetic or logical
	.GE.	greater than or equal to	constants, variables and expressions (all
	.LT.	less than	relational operators
	.LE.	less than or equal to	have equal priority)
	.EQ.	equal to	
	.NQ.	not equal to	
Logical	.NOT.	.NOT.A is true if and only if A is false	logical or integer constants, variables and expressions
	.AND.	A.AND.B is true if and only if A and B are both true	
	.OR.	A.OR.B is true if and only if A or B or both are true	
	.EQV.	A.EQV.B is true if and only if either A and B are both true or A and B are both false	
	.XOR.	A.XOR.B is true if and only if A is true and B is false or B is true and A is false	

Assignment Statements

variable=expression	Arithmetic/Logical Assignment: The value of the arithmetic or logical expression is assigned to the variable.
ASSIGN-TO	The ASSIGN statement is used to associate a statement label with an integer variable. The variable can then be used as a transfer destination in a subsequent assigned GO-TO statement in the same program unit.

Control Statements

GO TO	Unconditional	Transfers control to the same statement every time it is executed.
	Computed	Permits a choice of transfer destinations, based on a value of an expression within the statement.
	Assigned	Transfers control to a statement label that is represented by a variable. Because the relationship between the variable and a specific statement label must be established by an AS-SIGN statement, the transfer destination can be changed, depending upon which ASSIGN statement was most recently executed.
IF	Arithmetic	Transfers control to a statement depending on the value of an arithmetic expression. Used for conditional control transfers.
	Logical	Executes a statement if the test of a logical expression is true.
DO		Causes the statements in its range to be repeatedly executed a specified number of times. The range of the DO begins with the statement following the DO and ends with a specified terminal statement. The number of iterations is determined by the values for the initial, terminal, and increment parameters.

CONTINUE

Causes no processing. Passes control to the next executable statement. Used primarily as the terminal statement of a DO loop when that loop would otherwise end with a GO TO, arithmetic IF, or other prohibited control statement.

CALL

Calls a SUBROUTINE subprogram and passes it actual arguments to replace the dummy arguments in the subprogram.

RETURN

Returns control from a subprogram to the calling program unit.

PAUSE

Prints a message (if specified) on the terminal and suspends execution until the user responds.

STOP

Terminates program execution and prints a message (if specified) on the terminal.

END

Marks the end of a program unit. In a main program, if control reaches the END statement, a CALL EXIT is implicitly executed. In a subprogram, a RETURN statement is implicitly executed.

OPEN

Associates an existing file with a logical unit, or creates a new file and associates it with a logical unit. In addition, the statement can contain specifications for file attributes that direct the creation or subsequent processing. The attributes include specifying: the file name, the method of access (direct, sequential or append), protection (read only or read/write), form (formatted, unformatted), record size, block allocation or extension, whether the file can be shared, and whether the file is to be deleted or saved when closed (disposition). In addition, the OPEN statement can be modified by an ERR keyword which specifies the statement to which control is transferred if an error is detected.

CLOSE Disassociates a file from a logical unit. Disposition attributes specified in the OPEN statement can be modified. For example, a file opened as a file to be deleted can be saved, or a file opened to be saved can be deleted.

Input/Output Statements

READ Formatted Reads at least one logical record from the specified unit according to the given format specifications, and assigns values to the elements in a list.

Unformatted Reads one logical record from the specified unit, assigning the input values to the variables in a list.

Direct Access Reads the specified logical record form the specified unit and assigns the input values to the variables in a list.

List-directed Reads data from the specified unit, converts it into internal format, and assigns the input values to the elements of the I/O list, converting the value to the data type of the element if necessary.

Error Control Optional elements in the READ statement allow control transfer on error conditions. If an end-of-file condition is detected and the END option is specified, execution continues at a given statement. If a recoverable I/O error occurs and the ERR option is specified, execution continues at a given statement.

WRITE Formatted Writes one or more logical records containing the values of the variables in a list onto the specified unit in the given format.

Unformatted Writes one logical record containing the values of the variables in the list onto the specified unit.

457

Direct Access	Writes one logical record containing the values of the variables in the list into the specified record of the given unit.
List-directed	Writes the elements of the I/O list to the specified unit, translating and editing each value according to the data type of the value.
Error Control	Optional elements in the WRITE statement allow control transfer on error conditions. If an I/O error occurs and the ERR option is specified, execution continues at the given statement.
ACCEPT	Identical to a formatted or list-directed READ statement, except that input comes from a logical unit normally connected to the terminal keyboard.
TYPE	Identical to a formatted or list-directed WRITE except that output is directed to a logical unit normally connected to the terminal printer.
PRINT	Same as a TYPE statement, except that output is directed to a logical unit normally connected to the line printer.
DEFINE FILE	Defines the record structure of a direct access file: the logical unit number, the number of fixed-length records in the file, the length of a single record, and the pointer to the next record.
REWIND	The given logical unit is repositioned to the beginning of the currently open file.
BACKSPACE	The currently open file on the given logical unit is backspaced one record.
END FILE	An end-of-file record is written on the file open on the given logical unit.
FIND	Positions the direct access file on the given logical unit to the specified record and sets the associated variable.

ENCODE

Writes the elements in the I/O list into a memory buffer, translating the data into AS-CII format. The ERR option allows control transfer to a given statement if an error condition is detected.

DECODE

Reads the elements in the I/O list from a memory buffer, translating the data from AS-CII format into internal binary format. The ERR option allows control transfer to a given statement if an error is detected.

Format Statements

FORMAT

Describes the format in which one or more records are to be transmitted. The format descriptors include integer and octal, logical, real, double precision, complex, literal and editing. Real, double precision and complex formats can be scaled.

Specification Statements

IMPLICIT

Overrides the implied data type of symbolic names, in which all names that begin with the letters I, J, K, L, M, or N are presumed to be INTEGER values, and all names beginning with any another letter are assumed to be REAL values, unless otherwise specified. IMPLICIT allows the programmer to define the initial letters for implied data types. If a variable is not given an explicit type, and its name begins with a letter defined in an IM-PLICIT statement, its default type is that defined by the IMPLICIT statement.

type var1,var2,...,varn

Type Declaration:
The given variable names are assigned the specified data type in the program unit. Type is one of INTEGER*2, INTEGER*4, RE-AL*4, REAL*8, DOUBLE PRECISION, COM-PLEX*8, LOGICAL*4, LOGICAL*1 or BYTE.

DIMENSION

Reserves storage space for the specified array(s).

COMMON	Reserves one or more blocks of storage space under the specified name to contain the variables associated with the block name.
EQUIVALENCE	Declares two or more variable names in the same program unit to be associated with the same storage location.
EXTERNAL	Permits the use of external procedures (functions, subroutines and FORTRAN library functions) as arguments to other subprograms.
DATA	Assigns initial values to variables and array elements prior to program execution.
PROGRAM	Assigns a name to a main program unit. If present, it is the first statement in the main program.

User-Written Subprograms

name (var1, var2, ...) = expression	Arithmetic Statement Function: Creates a user-defined function having the variables as dummy arguments. When referenced, the expression is evaluated using the actual arguments in the function call.
FUNCTION	Begins a FUNCTION subprogram, indicating the program name and any dummy variable names. An optional type specification can be included.
SUBROUTINE	Begins a SUBROUTINE subprogram, indicating the program name and any dummy variable names.
BLOCK DATA	Specifies the subprogram which follows as a BLOCK DATA subprogram. An optional name for the program unit may be given.

FORTRAN Library Functions

ABS(X)	Real absolute value
IABS(X)	Integer absolute value
DABS(X)	Double Precision absolute value
CABS(Z)	Complex to Real, absolute value
FLOAT(I)	Integer to Real conversion

IFIX(X)	Real to Integer conversion
SNGL(X)	Double to Real conversion
DBLE(X)	Real to Double conversion
REAL(Z)	Complex to Real conversion
AIMAG(Z)	Complex to Real conversion
CMPLX(X,Y)	Real to Complex conversion
AINT(X)	Real to Real truncation
INT(X)	Real to Integer conversion
IDINT(X)	Double to Integer conversion
AMOD(X,Y)	Real remainder
MOD(I,J)	Integer remainder
DMOD(I,J)	Double Precision remainder
AMAX0(I,J,...)	Real maximum from Integer list
AMAX1(I,J,...)	Real maximum from Real list
MAX0(I,J,...)	Integer maximum from Integer list
MAX1(X,Y,...)	Integer maximum from Real list
DMAX1(X,Y,...)	Double maximum from Double list
AMIN0(I,J,...)	Real minimum of Integer list
AMIN1(X,Y,...)	Real minimum of Real list
MIN0(I,J,...)	Integer minimum of Integer list
MIN1(X,Y,...)	Integer minimum of Real list
DMIN1(X,Y,...)	Double minimum from Double list
SIGN(X,Y)	Real transfer of sign
ISIGN(I,J)	Integer transfer of sign
DSIGN(X,Y)	Double Precision transfer of sign
DIM(X,Y)	Real positive difference
IDIM(I,J)	Integer positive difference
EXP(X)	e raised to the X power (X is Real)
DEXP(X)	e raised to the X power (X is Double)
CEXP(Z)	e raised to the Z power (Z is Complex)
ALOG(X)	Returns the natural log of X (X is Real)
ALOG10(X)	Returns the log base 10 of X (X is Real)
DLOG(X)	Returns the natural log of X (X is Double)
DLOG10(X)	Returns the log base 10 of X (X is Double)
CLOG(Z)	Returns the natural log of Z (Z is Complex)
SQRT(X)	Square root of Real argument
DSQRT(X)	Square root of Double Precision argument
CSQRT(Z)	Square root of Complex argument
SIN(X)	Real sine
DSIN(X)	Double Precision sine
CSIN(Z)	Complex sine
COS(X)	Real cosine
DCOS(X)	Double Precision cosine
CCOS(Z)	Complex cosine

TANH(X)	Hyperbolic tangent
ATAN(X)	Real arctangent
DATAN(X)	Double Precision arctangent
ATAN2(X,Y)	Real arctangent of (X/Y)
DATAN2(X,Y)	Double Precision arctangent of (X/Y)
CONJG(Z)	Complex conjugate
RAN(I,J)	Returns a random number between 0 and 1

FORTRAN IV FUNCTIONS AND FEATURES

The FORTRAN IV compiler and Object Time System is available as an optional language processing system for the RT-11, RSTS/E, RSX-11M, and IAS operating systems. The FORTRAN compiler accepts source programs written in the FORTRAN IV language and produces an object file which must be linked prior to execution. FORTRAN IV supports all PDP-11 hardware options: EAE, EIS, FIS, and FPB-11.

The FORTRAN IV compiler is characterized by extremely rapid compilation rates. The FORTRAN IV compiler also performs well in small environments. On an RT-11 system with as little as 8K words of memory, FORTRAN IV can compile programs containing as many as 450 lines. On an RT-11 system with 28K words, FORTRAN IV can compile programs containing as many as 2200 lines.

Despite its small size requirements and high compilation rate, FORTRAN IV provides a high level of automatic object program optimization. The compiler performs redundant expression elimination, constant expression folding, branch structure optimization, and several types of subscripting optimizations.

FORTRAN IV has no statement ordering requirements, allowing declarations to appear anywhere within the source program. Terminal format input (using the tab character to delimit field) makes program preparation easier.

In order to allow larger FORTRAN programs, FORTRAN IV can allocate array storage outside a program's logical address space. These arrays are called virtual arrays and can be of any data type; they may also require operating system support of memory management directives.

FORTRAN IV COMPILER OPERATION

The FORTRAN IV compiler accepts a source written in the FORTRAN language as input and produces an object file and a listing file as output. The object file must subsequently be processed by the operating system's linker program, for example, the Linker or Task Builder, to produce an executable program.

COMMAND STRING SPECIFICATION OPTIONS

In the input/output file specification command string issued to the FORTRAN IV compiler to request program compilation, the user can specify a number of switch parameter options. Some of the parameters are:

Specify Listing Options

The user can request a number of listing options. By default, the user is supplied with diagnostics (if any), a source program listing, and the storage map. In addition, the user can request a generated code listing, or can combine any of the listing options in a single listing. The generated code listing contains a symbolic representation of the object code generated by the compiler, including a location offset from the base of the program unit, the symbolic Object Time System (OTS) routine names, and routine arguments. The code generated for each statement is labeled with the same internal sequence number that appears in the source program listing, for easy cross reference.

Selectively Compile Debugging Statement Lines

The user can request the compiler to include in the compilation those lines with a D in column one. These statements allow the inclusion of programmer-selected debugging aids (see below).

Code Generation Options

The compiler can generate in-line code which directly supports FIS, EIS, EAE or threaded code for machines without the additional arithmetic hardware.

Enable/Disable the Common Subexpression Optimizer

In general, the optimizer will make the program run faster. Disabling the optimizer can reduce program storage requirements, but will increase execution time.

Include or Suppress Internal Sequence Numbers

Suppressing internal sequence number accounting reduces program storage requirements for generated code and slightly increases execution time, but disables line number information during traceback.

Allocate Two Words for Default Length of Integer Variables

Normally, single storage words will be the default allocation for integer variables not given an explicit length specification (i.e., INTEGER*2 or INTEGER*4). Only one word is used for computation. The user can request that the default allocation be two storage words.

Enable/Disable Vectoring of Arrays

Array vectoring is a process which decreases the time necessary to reference elements of a multidimensional array by using some addi-

tional memory to store array accessing information. If array vectoring is enabled, the compiler decides whether to vector a multidimensional array based on the ratio of the amount of space required to vector the array over the total space required by the array. If this ratio is greater than 25%, the array is not vectored, and standard mapping is used instead. If size is a more critical factor than speed, the user can disable the vectoring of all arrays. If arrays are vectored, it is so noted in the storage map listing.

Enable/Disable Compiler Warning Diagnostics
Warning diagnostics report conditions which are not fatal error conditions, but which can be potentially dangerous at execution time, or which may present compatibility problems with other FORTRAN compilers running on PDP-11 operating systems. For example, a warning message is generated if a variable name exceeds six characters in length. This is potentially dangerous if another variable name has the same first six characters. The warning diagnostics are normally enabled, but the user can suppress their inclusion in the diagnostics listing.

Internal Operation and Structure
Instead of using temporary files to process source programs, the FORTRAN IV compiler performs all its activities in main memory. It reads the entire source program once, stores it in memory in a compacted format, and processes the compacted code in memory. Since a disk device is not used for temporary file operations, compilation speed is significantly increased.

To reduce the memory requirements of such a compilation system, the FORTRAN IV compiler employs a multi-phase overlaid structure. The compiler consists of a large number of overlays. Most of the space allocated to the compiler is occupied by the compressed source code. Figure 12-1 illustrates the compile-time memory map.

```
┌──────────────────────────┐   HIGH ADDRESSES
│      SYMBOL TABLE         │
│       (DYNAMIC)           │
│                          │
│    (GROWS DOWNWARD)       │
│            │              │
│ ─ ─ ─ ─ ─ ─▼─ ─ ─ ─ ─ ─   │
│                          │
│    CURRENT INTERNAL       │
│    FORM OF SOURCE         │
│       PROGRAM             │
│       (DYNAMIC)           │
│                          │
├──────────────────────────┤
│     I/O  BUFFER AREA      │   .25K
├──────────────────────────┤
│                          │   MINIMUM
│    ACTIVE OVERLAY AREA     │   1.25K WORDS
│                          │
├──────────────────────────┤
│    ROOT SEGMENT  AND      │   .25K WORDS
│     OVERLAY LOADER        │
├──────────────────────────┤
│   VECTORS  AND SYSTEM     │   .25K WORDS
│   COMMUNICATION AREA      │   LOW ADDRESSES
└──────────────────────────┘
```
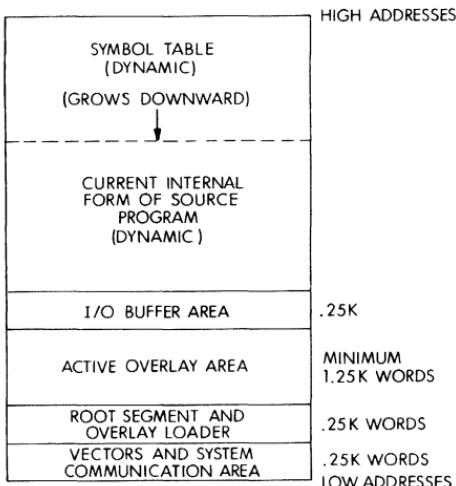
Figure 12-1 Compile-Time Memory Map

The compiler goes through a series of processing phases, one for each of its overlays. Each program segment is processed separately. The basic processing phases are:

1. Source program compaction and listing

2. Syntax analysis and error reporting

3. Statement processing

4. Code generation

5. Code optimization

The compiler begins by reading in as much of the source program as can fit in memory. It then compresses the source code in memory by removing blanks and other unnecessary data. It continues to read in more source code, compressing it as it goes, until the entire program segment fits in memory.

Once the source code is compacted into memory, the compiler begins processing the internal form of the source code as a whole. Because the entire program segment is available to the compiler, FORTRAN IV does not require statement ordering restrictions.

During the first stage of code generation, the compiler immediately writes as much information as possible to the object file. This step is necessary to further compress the internal source code to enable the symbol table to grow in the later stages of processing.

The non-executable statements are eligible for immediate processing, since the information they provide is not needed until run-time. Therefore, the compiler searches for all the occurrences of non-executable statements, such as FORMAT and DATA statements, produces the beginning of the object module, and compacts the internal source code further.

The compiler enters all variables and constants not yet processed into the symbol table, and performs the syntax scan of the executable statements. The program is translated into an internal format in preparation for final code generation.

Object Code Generation
A few executable FORTRAN statements can be translated directly into machine instructions. Typical FORTRAN operations, however, require long sequences of PDP-11 machine instructions. For example, standard sequences are needed to locate an element of a multidimensional array, initialize an I/O operation, or simulate a floating point operation not supported by the hardware configuration.

The common sequences of PDP-11 machine instructions are contained in a library known as the FORTRAN Object Time System (OTS). The FORTRAN IV compiler does not always generate pure machine instructions for the FORTRAN source code statements. It simply determines which combination of appropriate OTS routines is needed to implement a FORTRAN program. During the linking process for an object program, the linker utility includes the needed OTS routines into the load module. During program execution, these routines are chained together to effect the desired result. However, in-line code is used for improved execution speed for some operations where appropriate.

The compiler references a library instruction sequence by generating a word containing the address of the first instruction in the OTS routine, followed by the information upon which the routine is to operate (the operands). For example, an OTS routine used to perform the end-of-DO-loop sequence must be passed to the location of the index variable, the limit value, and the address of the beginning of the loop.

The compiler and OTS make use of the PDP-11 general register and indirect addressing facility to have the OTS routines executed at run-time. Register 4 (R4) is used to chain together the selected OTS routines. The last instruction executed in each library routine is a JMP (R4)+, which transfers control to the next library instruction sequence.

Optimizations

The FORTRAN IV compiler performs the following optimizations during compilation:

1. Compiled FORMAT Statements

The compiler interprets the FORMAT statements at compile-time, translating the format into an internal form. This not only increases the execution speed of the program, it decreases its size, because less run-time code is needed.

2. Array Vectoring

Array vectoring decreases the time necessary to reference elements of a multidimensional array by using additional memory to store the array. If an array is vectored, a particular element in the array can be located by a simplified mapping function, without the need for multiplication operations.

3. Constant Folding

Integer constant expressions are evaluated at compile-time.

4. Compile-time Evaluation of Constant Subscript Expressions

Constant subscript expressions in array calculations are evaluated at compile-time.

5. Elimination of Unreachable Code

Statements that are never reached by flow of control are eliminated from the object code.

6. Common Subexpression Elimination

Redundant subexpressions whose operands are not changed between computations are replaced by a temporary value calculated only once.

7. Peephole Optimizations

The compiler examines the internal form of the object code on an operation-by-operation basis to replace sequences of operations with shorter and faster equivalent operations. For example, the compiler replaces a divide-by-two operation with a multiply-by-one-half operation. There is a large set of these kinds of operations.

8. Branch Optimizations for Arithmetic and Logical IF

Branch structure optimizations improve program speed and decrease its size. For example, an arithmetic IF statement can often be improved:

```
IF(A-7.0)100,200,100 !goto 200 if A is equal to 7.0
100 CONTINUE
```

The compiler will optimize this statement to:

```
IF(A .EQ. 7.0) GOTO 200
```

9. Register Allocation

Register allocation is optimized over a series of statements or loops to minimize direct memory references for variables.

10. Loop Optimization

The optimizer locates expressions dependent on the loop index variable, and reduces them to less complex arithmetic operations. For example multiplies are replaced with adds and subscripts are replaced with direct memory references.

Libraries

The FORTRAN programmer can create a library of commonly used assembly language and FORTRAN functions and subroutines. The operating system's librarian utility provides a library creation and modification capability. Library files may be included in the command string to the linker utility. The linker recognizes the file as a library file and links only those routines in the library that are required in the executable program. By default, the linker also automatically searches the FORTRAN system library for any other required routines.

Debugging a FORTRAN Program

Two debugging facilities are available to the FORTRAN programmer. The FORTRAN Object Time System provides the traceback feature for fatal run-time errors. This feature locates the actual program unit and line number of a run-time error. Immediately following the error message, the error handler will list the line number and program unit name in which the error occurred. If the program unit is a subroutine or function subprogram, the error handler will trace back to the calling program unit and display the name of that program unit and the line number where the call occurred. This process will continue until the calling sequence has been traced back to a specific line number in the main program. This allows the exact determination of the location of an error even if the error occurs in a deeply nested subroutine.

In addition to the FORTRAN OTS error diagnostics which include the traceback feature, there is another debugging tool available. A "D" in column one of a FORTRAN statement allows that statement to be conditionally compiled. These statements are considered comment lines by the compiler unless the appropriate debugging lines switch is issued in the compiler command string. In this case, the lines are compiled as regular FORTRAN statements. Liberal use of the PAUSE statement and selective variable printing can provide the programmer with a method of monitoring program execution. This feature allows the inclusion of debugging aids that can be compiled in the early program testing stages and later eliminated without source program modification.

FORTRAN IV OPERATING ENVIRONMENTS

The FORTRAN IV compiler and OTS is available as an optional language processor for the RT-11, RSTS/E, RSX-11M and IAS operating systems. The compiler operation and facilities under each of these systems are essentially identical.

Each operating system provides additional features particular to the environment. For example, the monitor programmed requests or executive directives are usually available as a library of FORTRAN-callable routines.

Under RT-11

The entire FORTRAN IV system is operational in 8K words under the RT-11 SJ, FB, or XM monitors. The RT-11 System Subroutine Library (SYSLIB) is a collection of FORTRAN-callable routines which allow a FORTRAN user to utilize various features of the RT-11 Foreground/Background (F/B) and Single-Job monitors. SYSLIB also provides various utility functions, a complete character string manipulation package, and 2-word integer support. SYSLIB is provided as a library of object modules to be combined with FORTRAN programs at link-time. SYSLIB allows the RT-11 FORTRAN user to write almost all application programs in FORTRAN with no assembly language coding.

Also available under RT-11 are:

- A library of FORTRAN-callable graphics routines supporting the VT11, GT40, GT42, and GT44 graphics hardware systems.
- Plotting support for the LV11 electrostatic printer/plotter.
- Laboratory data acquisition and manipulation routines used in conjunction with the LPS-11 and AR11 laboratory peripheral hardware.
- The Scientific Subroutine Library, providing FORTRAN-language routines for mathematical and statistical applications.
- Stand-alone program execution

Under RSTS/E

RSTS/E FORTRAN IV operates in interactive or batch mode under the RSTS/E monitor. The FORTRAN IV system includes the FORTRAN IV compiler, the Object Time System (OTS), and several utility programs.

The entire system (including compiler and optimization components) is completely functional in an 8K-word user area. A system interface occupying 4K words of memory is sharable among all FORTRAN IV users on the system. In addition, the FORTRAN IV system provides overlay support for programs and data, allowing extremely large programs to be run in a small region of memory.

RSTS/E FORTRAN IV provides assembly language subprogram support, using the MACRO assembler. Although the assembly language subprogram can not issue any monitor calls, MACRO provides the experienced user with a tool to further enhance computational performance.

Under RSX-11 and IAS

In RSX-11M, the FORTRAN IV compiler runs in a minimum partition of 7K words. If run in a larger partition, it uses the extra space for program and symbol table storage. In RSX-11D and IAS, the compiler task requires 8K words minimally and can be extended when it is installed. As with RSX-11M systems, the additional space allows the processing of larger FORTRAN programs.

An RSX-11/IAS library consists of object modules. Two types of libraries exist, shared and relocatable.

Relocatable libraries are stored in files. Object modules from relocatable libraries are built into the task image of each task referencing the module. The Task Builder is used to include modules from relocatable libraries in a task image. When a library specification is encountered in the command string, those modules in the library which contain definitions of any currently undefined global symbols are included in the task image. The user can construct relocatable libraries of assembly language and FORTRAN routines using the Librarian utility.

Shared libraries are located in main memory and a single copy of each library is used by all referencing tasks. Access to a shared library is gained by specifying the name of the library in an option at task build time. Shared libraries are built using the task builder. They must contain sharable (reentrant) code.

Each RSX-11/IAS system has a system relocatable library. The system relocatable library is automatically searched by the Task Builder if any undefined global references are left after processing all user-specified input files. The FORTRAN OTS may be included in the system library and hence is loaded automatically with FORTRAN programs.

The RSX-11/IAS system library provides FORTRAN-callable forms of most executive directives. The FORTRAN programmer can schedule the execution of tasks, communicate with concurrently executing tasks, and manipulate system resources through these calls.

Industrial Society of America (ISA) extensions for process I/O control are available in FORTRAN-callable format under RSX-11M. Support for laboratory and process control peripherals is also included.

FORTRAN IV-PLUS FUNCTIONS AND FEATURES

The FORTRAN IV-PLUS Compiler and Object Time System is an optional language processing system for the RSX-11D, RSX-11M, and IAS operating systems. The FORTRAN IV-PLUS compiler accepts source programs written in the PDP-11 FORTRAN language and produces an object file which must be linked prior to execution. The FORTRAN IV-PLUS language is a superset of PDP-11 FORTRAN and is based on the specifications for the American National Standard FORTRAN X3.9-1966.

Both the FORTRAN IV and the FORTRAN IV-PLUS compilers can be used in the same RSX family environments. If both FORTRAN compilers are to be used on the same system, two separate FORTRAN library files are maintained. One compiler must be selected as the "default" compiler. The one selected as the default is the one that can be used in batch processing.

The primary differences between the FORTRAN IV compiler and the FORTRAN IV-PLUS compiler are that the FORTRAN IV-PLUS compiler:

● supports extended language features

● produces highly optimized PDP-11 machine language code

● requires the FPP Floating Point Processor option

The FORTRAN IV-PLUS compiler generates optimized code resulting in fast user program execution. The FORTRAN IV compiler is designed to provide high-speed program compilation and to operate on minimum core configurations.

With the exception of virtual array support in FORTRAN IV, the FORTRAN IV-PLUS language is upward compatible with the PDP-11 FORTRAN IV language. The FORTRAN IV-PLUS system supports the same enhancements to the language standard as FORTRAN IV. In addition, FORTRAN IV-PLUS also includes the following extensions:

● ENTRY statements can be used in SUBROUTINE and FUNCTION subprograms to define multiple entry points in a single program unit.

● PARAMETER statements can be used to give symbolic names to constants.

● Lower bounds as well as upper bounds of the array dimension can be specified in array declarators. The value of the lower bound dimension declarator can be negative, zero, or positive.

● The data type INTEGER*4 provides a sign plus 31 bits of precision. INTEGER*4 allows a greater range of values to be represented than INTEGER*2. Both data types can be used in the same program.

- A compiler command line specification allows all INTEGER and LOGICAL declarations without explicit length specifications to be considered as INTEGER*2 and LOGICAL*2, or INTEGER*4 and LOGICAL*4, respectively.

- The following I/O statements have been added:
                    READ (u'r,fmt)
    Direct access using formatted records
                    WRITE (u'r,fmt)

    These I/O statements provide formatted direct access I/O operations, since the READ and WRITE statements contain references to FORMAT statements or format specifications in arrays.

- Generic function selection by argument data type is provided for many FORTRAN library functions.

- The control variable of a DO statement can be a double precision data type as well as an INTEGER*2, INTEGER*4, or REAL data type. The initial, terminal, and increment parameters can be of any data type and are converted before use to the type of the control variable if necessary.

- The INCLUDE statement incorporates FORTRAN source text from a separate file into a FORTRAN program.

- The number of times a DO loop is executed (called the iteration count) is determined at the initialization of the DO statement and is not re-evaluated during successive executions of the loop. Consequently, the number of times the loop is executed will not be affected by changing the variables used in the DO statement. That is, the terminal and increment parameters can be modified within the loop without affecting the iteration count.

LANGUAGE EXTENSIONS

The following paragraphs discuss some of the additional language components that FORTRAN IV-PLUS provides to the FORTRAN IV language. Table 13-1 at the end of this chapter compares the implementation of the FORTRAN IV and FORTRAN IV-PLUS languages.

I/O Statements

FORMATTED DIRECT ACCESS INPUT/OUTPUT

Formatted direct access READ and WRITE statements are used to perform direct access I/O of character data with a file on a direct access device. The OPEN statement is used to establish the attributes of the file. Each READ or WRITE contains an expression that specifies the number of the record to be accessed.

The formatted direct access READ statement causes the specified record to be read from the direct access file currently associated with the given logical unit. The characters in the record are scanned and converted as indicated by the given format specification. The resulting values are assigned to the elements in a list.

The formatted direct access WRITE statement writes the specified record in the direct access file currently associated with the given logical unit. A list specifies a sequence of values which are converted to characters and positioned as specified by a format specification.

Specification Statements
INCLUDE
Specifies that the contents of a designated file are to be incorporated in the FORTRAN compilation directly following the INCLUDE statement. An INCLUDE statement can appear anywhere a comment line can appear. When the compiler encounters an INCLUDE statement, it stops reading statements from the current file and starts reading statements from the included file. When the end of the included file is reached, compilation resumes with the statement following the IN-CLUDE statement. An INCLUDE statement can be contained in an included file.

The INCLUDE statement provides a mechanism for writing modular reliable and maintainable programs by eliminating duplication of source code. A section of program text that is used by several program units, such as a COMMON block specification, can be created and maintained as a separate source file. All program units which referenced the COMMON block then merely INCLUDE this common file. Any changes to the COMMON block will be reflected automatically in all program units after compilation.

EXTERNAL*name
Specifies that a name refers to a user-defined external FUNCTION or SUBROUTINE subprogram, in order to differentiate it from a FORTRAN library processor-defined function.

PARAMETER
Allows a constant to be given a symbolic name. The symbolic name of a constant assumes the type implied in the form of its corresponding constant. The initial letter of the name has no effect on its type.

User-Written Subprograms
ENTRY
Provides multiple entry points within a subprogram. It is not executable and can appear within a function or subroutine program after the FUNCTION or SUBROUTINE statement. Execution begins with the first executable statement following the ENTRY statement.

Library Functions

The following additional FORTRAN library functions are provided:

ASIN(X)	Real arcsine
DASIN(X)	Double precision arcsine
ACOS(X)	Real arccosine
DACOS(X)	Double precision arccosine
SINH(X)	Real hyperbolic sine
DSINH(X)	Double precision hyperbolic sine
COSH(X)	Real hyperbolic cosine
DCOSH(X)	Double precision hyperbolic cosine
TANH(X)	Real hyperbolic tangent
DTANH(X)	Double precision hyperbolic tangent
TAN(X)	Real tangent
DTAN(X)	Double precision tangent
NINT(X)	Real to integer nearest integer
ANINT(X)	Real nearest integer
IDNINT(X)	Double to integer nearest integer
IAND(I,J)	Integer bitwise AND
IOR(I,J)	Integer bitwise OR
IEOR(I,J)	Integer bitwise Exclusive OR
NOT(I)	Integer NOT
ISHFT(I,J)	Integer bitwise shift

Generic Function References

Generic function names provide a means by which some of the FORTRAN mathematical functions can be called with selection of the actual library routine used, based on the type of the argument that occurs in the function reference. For example, if X is a real variable, then SIN(X) will reference the real-valued sine function. If D is a double precision variable, then SIN(D) will reference the double precision sine function. It is not necessary to write DSIN(D).

Generic function selection is performed independently for each function reference. Given the above example, both SIN(X) and SIN(D) can be used in the same program unit.

The set of functions for which generic name selection is performed is shown below. Generic function selection can only be used with the argument types shown.

SYMBOLIC NAME	TYPE OF ARGUMENT	TYPE OF RESULT
ABS	Integer Real Double Complex	Integer Real Double Complex
AINT, ANINT	Real Double	Real Double
INT, NINT	Real Double	Integer Integer
SNGL	Integer Double	Real Real
DBLE	Integer Real	Double Double
MOD, MAX, MIN, SIGN, and DIM	Integer Real Double	Integer Real Double
EXP, LOG, SIN, COS, and SQRT	Real Double Complex	Real Double Complex
LOG10, TAN, ATAN, ATAN2, ASIN, ACOS, SINH, COSH, and TANH	Real Double	Real Double

COMPILER OPERATION AND OPTIMIZATIONS

The FORTRAN IV-PLUS compiler accepts a source written in the FORTRAN language and produces an object file which must be linked by the IAS/RSX-11 Task Builder prior to execution. The compiler uses a work file system to produce the object file. This work file system allows very large FORTRAN programs to be compiled in a limited amount of memory.

The compiler generates PDP-11 machine language code, including FP-11 instructions, for the object program. During compilation, the FORTRAN IV-PLUS compiler performs many code optimizations.

The FORTRAN IV-PLUS optimizations are designed to produce an object program that executes in less time than an equivalent non-optimized program. The optimizations are also designed to reduce the size of the object program.

The FORTRAN IV-PLUS compiler performs the following optimizations:

- Constant folding. Integer constant expressions are evaluated at compile-time.

- Compile-time evaluation of constant subscript expressions in array calculations.

- Elimination of unreachable code. An optional warning message is issued to mark unreachable statements in the source program listing.

- Recognition and replacement of common subexpressions.

- Peephole optimization. The code is examined on an operation-by-operation basis to replace sequences of operations with shorter and faster equivalent operations.

- Branch instruction optimizations for arithmetic or logical IF statements.

- Compile-time constant conversion.

- Argument-list merging. If two function or subroutine references have the same arguments, a single copy of the argument list is generated.

- Removal of invariant computations from DO loops.

- Local register assignment. Frequently referenced variables are retained (if possible) in registers to reduce the number of load and store instructions.

- Assignment of frequently used variables and expressions to registers across DO loops.

- JMP/BRANCH instruction resolution. The BRANCH instruction is used wherever possible to eliminate unnecessary JMP instructions.

A FORTRAN IV-PLUS program is computationally equivalent to a program according to the definition of the FORTRAN language. Thus identical numerical results are obtained and equivalent run-time diagnostics are produced. Messages may not, however, occur at exactly the same statements in the source programs.

Compile-Time Operations on Constants
The FORTRAN IV-PLUS compiler performs the following compile-time computations on expressions involving constants, including PARAMETER constants.

- Negation of Constants. For example,
    $X = -10.0$

- Type Conversion of Constants. For example,
    $X = 10*Y$
  is compiled as
    $X = 10.0*Y$

- Integer Arithmetic on Constants. For example,
  >   PARAMETER NN=27
  >   I = 2*NN+J
  > is compiled as
  >   I = 54+J

In addition, array subscripts involving constants are simplified at compile-time where possible. For example,

>   DIMENSION I(10,10)
>   I(1,2) = I(4,5)

is compiled as a single MOV instruction:

>   MOV I+130,I+26

This not only significantly increases the speed of the program, it reduces its size.

Elimination of Common Subexpressions

Often the same subexpression appears in more than one computation. If the values of the operands of a common subexpression are not changed between computations, the value of the subexpression can be computed once and its result can be substituted where the subexpression appears. For example, the sequence:

>   A =   B*C+E*F
>   .
>   .
>   .
>   H =   A+G-B*C
>   .
>   .
>   .
>   IF((B*C)-H)10,20,30

contains the common subexpression B*C. The sequence is compiled as:

>   t =   B*C
>   A =   t+E*F
>   .
>   .
>   .
>   H =   A+G-t
>   .
>   .
>   .
> IF((t)-H)10,20,30
>   .
>   .

where t is a temporary variable created by the compiler. Two computations of the subexpression B*C are eliminated from the sequence.

A more subtle application of common subexpresson elimination occurs in the following example. The statements:

DIMENSION A(25,25), B(25,25)
A(I,J) = B(I,J)

are compiled, without optimization, as the sequence of instructions in the following form:

t1 =   J*25+I
t2 =   J*25+I
MOVE B(t2) TO A(t1)

The variables t1 and t2 represent equivalent expressions. The redundancy is recognized and the following shorter, faster sequence is generated:

t =   J*25+I
MOVE B(t) TO A(t)

Removal of Invariant Computations from DO Loops

The speed with which a given algorithm can be executed is increased if computations are moved from frequently executed program sequences to less frequently executed program sequences. In particular, computations within a loop involving only constants and variables whose values are not changed within the loop can be moved outside the loop.

For example, in the sequence:

DO 10, I=1,100
10   F = 2.0*Q*A(I)+F

the value of the subexpressions 2.0*Q is the same during each iteration of the loop. Transformation of the sequence to:

t = 2.0*Q
DO 10, I=1,100
10   F = t*A(I)+F

moves the calculation 2.0*Q outside the loop and eliminates 99 multiply operations.

Generated Code Example

The FORTRAN routine:

0001   DIMENSION A1(25)
                         .
                         .
                         .
0007   AMIN= A1(1)
0008   AMAX= A1(1)
0009   DO 40 I=2,N

478

```
0010              IF (A1(I) .LT. AMIN) AMIN= A1(I)
0011              IF (A1(I) .GT. AMAX) AMAX= A1(I)
0012    40        CONTINUE
```

is compiled into the following code:

```
                          ;Statement 0007.
        SETF
        LDF     A1,F0     ;AMAX is bound to FPP register
                          ;F0 and initialized to A1(1).
        LDF     F0,F1     ;AMIN is bound to FPP register
                          ;F1 and initialized to A1(1).
                          ;Statement 0009.
        MOV     #2,R0     ;The DO loop control variable I
                          ;is bound to register R0 and
                          ;initialized to 2.
L$GACD:
                          ;Statement 0010.
        MOV     R0,R1
        ASL     R1
        ASL     R1
        LDF     A1-4(R1),F2
;For each iteration of the
        CMPF    F1,F2     ;loop, A1(I) is bound to FPP
        CFCC              ;register F2.
        BLE     L$GAPE
        LDF     F2,F1
L$GAPE:
                          ;Statement 0011.
        CMPF    F0,F2
        CFCC
        BGE     L$GAFF
        LDF     F2,F0
L$GAFF:
                          ;Statement 0012.
        INC     R0
        CMP     R0,N
        BLE     L$GACD
        STF     F1,AMIN   ;At the end of the loop, the
        STF     F0,AMAX   ;values of AMIN, AMAX and I
        MOV     R0,I      ;are stored in memory.
```

ENVIRONMENTS

The FORTRAN IV-PLUS compiler is available as an optional language
processor for the RSX-11M, RSX-11D and IAS operating systems. The

compiler's operation and facilities under each of these operating systems are essentially identical.

In all operating systems, the hardware configuration must include the FP-11 Floating Point Processor. The FORTRAN IV-PLUS compiler requires a minimum partition size of 17K words to execute in an RSX-11D or IAS system. Under RSX-11M, the compiler requires a minimum partition size of 18K words.

APL (V1)

OVERVIEW
APL-11 uses one of the most concise, consistent, and powerful character sets ever devised. It is flexible enough to solve problems in text-handling and commercial data processing as easily as it can solve problems in mathematics and statistics. The RSTS/E and RT-11 operating systems support APL-11 as a language option.

FEATURE TOPICS
- Features and Functions
  - Applications
  - APL on the PDP-11 Systems
  - APL Equipment and Character Set

- Language Elements
  - Data Structures
  - Statements
  - Monadic and Dyadic Primitive Functions
  - Primitive Scalar Functions
  - Extension of Scalar Functions to Arrays
  - Primitive Mixed Functions
  - Relational Functions

- Input/Output Operations
  - Quad Input Mode
  - Quad-Del Input Mode
  - Normal Output Mode
  - Heterogeneous Output Mode

- Communication with the System
  - System Commands
  - I-beams

- APL Statement Execution

- APL Operating System Environment
  - Workspaces
  - File System

FEATURES AND FUNCTIONS

APL is a concise programming language suited for handling array-structured alphanumeric data. APL is used as a general data processing language as well as a mathematical tool. The language is flexible enough to solve problems in text handling and commercial data processing as concisely and easily as it can solve problems in numerical mathematics and statistics.

APL allows user-defined functions to be expressed with the same syntax used to express primitive language functions. Thus the user can expand the capabilities of the language to handle the requirements of any application.

Applications

APL is used in engineering, commercial, and educational applications. Current applications include: data reduction and analysis, simulation and forecasting, financial modeling, design engineering, electric circuit analysis, engineering analysis, inventory and payroll management, data base manipulation, reservation systems, computer-assisted instruction (CAI), and education (high school and college level) in programming.

APL on the PDP-11

The APL system is implemented as a language interpreter on the PDP-11. APL can operate on a wide range of hardware processors and has been implemented to run under either of two operating systems, RT-11 or RSTS/E.

The APL run-time system is preconfigured by DIGITAL to match such installation-dependent characteristics as:

- the PDP-11 processor used
- the operating system (RT-11 or RSTS/E) under which APL will run
- the availability of hardware floating point processor
- the selection of single precision or double precision arithmetic

These characteristics are supplied as different modules in the distribution kit.

APL Equipment and Character Set

The user interacts with APL using a terminal or CRT. Two types of terminal are supported by the PDP-11 for use with the APL system.

DIGITAL APL Terminals

Description	**Character Set**
Any terminal without the APL character set	ASCII
DECwriter II model LA37 with APL option	APL/ASCII

The full APL character set can be represented using a keyboard illustrated in Figure 22-1. All characters on this keyboard are received and interpreted by APL. Note that letters, numbers, and some of the special characters appear in the conventional keyboard positions.

Terminals Without the APL Character Set

ASCII terminals do not support the use of the special APL characters illustrated in Figure 22-1. If the user has an ASCII terminal or is operating in console terminal mode on an APL terminal, special APL characters can be represented by keyboard mnemonics. To represent the APL rho symbol (ρ), for example, the user enters the .RO mnemonic. The .GO mnemonic is used to express an APL right arrow ($\rightarrow$).
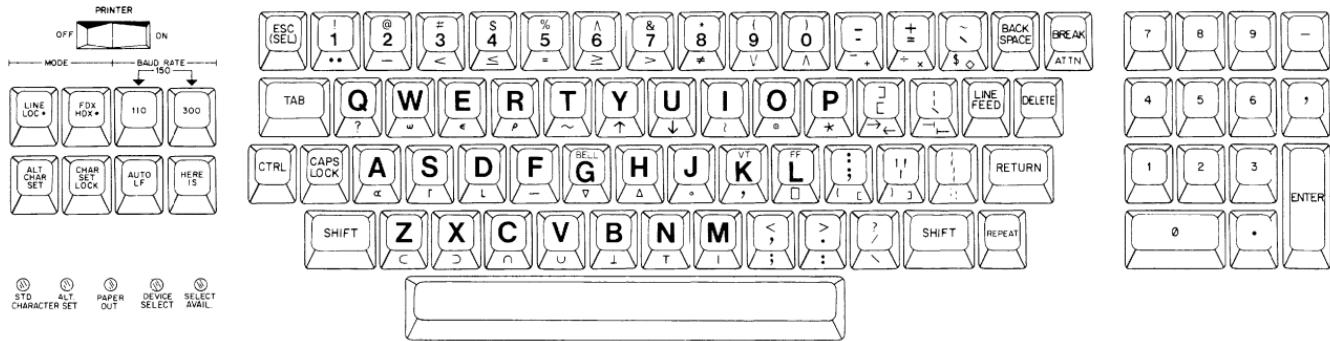
Figure 22-1 The APL Keyboard (LA37 Terminal)

LANGUAGE ELEMENTS

The APL language system provides functions and operators to manipulate data, and system commands to control the program environment.

Data Structures

Numeric and character data can be structured in a variety of ways. The following data structures are supported by APL:

- scalars
- vectors
- matrices
- arrays with three or more dimensions

A scalar is a single numeric or character value with no dimensions. A character scalar is enclosed in single quotes; for example:

enter:	A ← 'C'
	A
returned:	C

A vector is a one-dimensional array or character string consisting of any number of values. A numeric vector is entered as a list of values separated by at least one space; for example:

enter:	A ← 1 2 3 4
	A
returned:	1 2 3 4

A matrix is a two-dimensional array consisting of rows and columns. The user must enter values corresponding to each element of an array, and must also specify the shape of the array. The shape of an array is the number of dimensions which it has and the length of each of these dimensions. For example, a matrix can have six elements arranged as two rows and three columns or three rows and two columns, as illustrated by arrays A and B below.

```
A
1       2       3
4       5       6
B
1       2
3       4
5       6
```

The APL primitive function rho (ρ) is used to specify the shape of a new array or to reshape an existing array. It can also be used to create a null vector, which is useful in certain APL operations. Following is an example of creating a simple matrix with the rho function:

enter:　　　　A ← 4 2 ρ 0 1 2 3 4 5 6 7
　　　　　　　A

returned:　　0　　　1
　　　　　　　2　　　3
　　　　　　　4　　　5
　　　　　　　6　　　7

Arrays of three or more dimensions are also supported by APL. There is a limit of 16 dimensions on arrays in the current implementation of APL; however, a more significant restriction is that the size of the array must not exceed the size of the user's workspace.

Statements

A program consists of one or more lines called statements. There are two types of APL statements:

- assignment statements
- branch statements

Assignment statements include both calculation and input/output operations. Branch statements are used to restart a function or to handle the transfer of control from one part of a program to another. Branch statements are relevant only to user defined functions.

An APL statement can contain the following components:

- identifiers
- constants
- APL primitive functions
- user-defined functions

Monadic and Dyadic Primitive Functions

APL primitive functions are implemented in two forms: monadic and dyadic. Monadic functions take a right argument and are of the type $\div A$ (reciprocal), $!B$ (factorial) or ~ 1 (logical NOT). Dyadic functions take both left and right arguments and are of the type $3+2$ (addition), and $X = Y$ (equal). The syntax of the function (i.e., the presence of one or two arguments) determines whether the function is monadic or dyadic. For example, $|A$ is a monadic function used to determine the magnitude or absolute value of the argument A. $A|B$ is a dyadic function used to obtain the residue or remainder available after dividing B by A. The particular operation specified by the $|$ symbol is dependent on the context of the statement.

Primitive Scalar Functions

APL primitive functions are of two types: scalar and mixed. Scalar functions generally take single-number (scalar) arguments and yield scalar results. They are used primarily for basic arithmetic and logical operations, such as addition, exponentiation, maximum value, and logical OR. With a few exceptions, the primitive scalar functions take numeric scalar arguments. The relational functions ($<$, $\leq$, $=$, $>$, $\geq$, $\neq$) can take either character or numeric arguments, but only the equal ($=$) and not equal ($\neq$) primitives may have one character argument and one numeric argument. The logical functions (Λ, V, $\sim$, etc.) must have arguments equal to 0 or 1.

Table 22-1 Primitive Dyadic Circle Functions

X ° Y	X	(−X) ° Y
(1−Y*2)*0.5	0	(1−Y*2)*0.5
SIN Y	1	ARCSIN Y
COS Y	2	ARCCOS Y
TANGENT Y	3	ARCTAN Y
(1+Y*2)*0.5	4	(⁻+Y*2)*0.5
SINH Y	5	ARCSINH Y
COSH Y	6	ARCCOSH Y
TANH Y	7	ARCTANH Y

Tables 22-2 and 22-3 summarize the primitive scalar functions available in APL.

Table 22-2 Primitive Scalar Monadic Functions

FUNCTION	MEANING
+Y	Y
−Y	negative of Y
×Y	sign of Y (⁻1,0,1)
÷Y	reciprocal of Y
*Y	e to the Yth power
| Y	magnitude of Y
⌈ Y	ceiling of Y
LY	floor of Y
● Y	natural logarithm of y
!Y	factorial Y (for integral Y gamma function of Y+1 for non-integral Y)
?Y	a random integer from Y
°Y	π times Y

APL

Table 22-3 Primitive Scalar Dyadic Functions

FUNCTION   MEANING

X + Y   add X to Y
X − Y   subtract Y from X
X × Y   multiply X and Y
X ÷ Y   divide X by Y
X*Y   X to the Yth power
X|Y   residue of Y
X Γ Y   maximum of X and Y
X L Y   minimum of X and y
X ● Y   log of Y to the base X
X ! Y   binomial coefficient (number of combinations
             of Y things taken X at a time)
X ° Y   trigonometric function (Y is in radians)

Extension of Scalar Functions to Arrays

The primitive functions are considered scalar functions because they generally take scalar arguments and yield scalar results. The operations performed by these functions can, however, be extended to arrays. A primitive scalar function is applied to an array on an element-by-element basis. Thus, if the user specifies an addition operation in which both arguments are vectors, the corresponding elements of the vectors are added, for example:

        enter: 5 8 9 + 6 7 2
        returned: 11 15 11

The arrays on which the primitive scalar functions operate can be of any dimension. If a dyadic function is being executed, the arrays specified as the arguments of the function must generally have the same number of elements and be the same shape (e.g., a 2-by-3 array is not equivalent to a 3-by-2 array). There is one exception to this rule. If one argument is an array and the other is a scalar or a single-element array, the single value is applied to every element of the array. The following two examples are therefore equivalent.

        enter: 5 5 5 + 6 7 2
        returned: 11 12 7

        enter: 5 + 6 7 2
        returned: 11 12 7

Table 22-4 Primitive Mixed Functions

FUNCTION	MEANING
ρY	RETURN SHAPE OF Y
XρY	RESHAPE Y TO MAKE DIMENSION X
ιY	GENERATE THE FIRST Y CONSECUTIVE INTEGERS FROM CURRENT ORIGIN
XιY	FIND THE FIRST OCCURRENCE OF Y WITHIN VECTOR X
,Y	RETURN THE RAVEL OF Y (MAKE Y A VECTOR
X,Y	CATENATE X TO Y ALONG THE LAST DIMENSION OF X
X,[N]Y	LAMINATE X TO Y ALONG THE Nth DIMENSION OF X
X/Y	X (LOGICAL) COMPRESSION ALONG THE LAST DIMENSION OF Y
X/[N]Y	X(LOGICAL) COMPRESSION ALONG THE Nth DIMENSION OF Y
X⌿Y	X (LOGICAL) COMPRESSION ALONG THE LAST DIMENSION OF Y
X\Y	X (LOGICAL) EXPANSION ALONG THE LAST DIMENSION OF Y
X\[N]Y	X (LOGICAL) EXPANSION ALONG THE Nth DIMENSION OF Y
X⍀Y	X(LOGICAL) EXPANSION ALONG THE FIRST DIMENSION OF Y
X↑Y	FOR X>0, TAKE FIRST X ELEMENTS OF Y FOR X<0, TAKE LAST $\vert$X ELEMENTS OF Y
X↓Y	FOR X>0, DROP FIRST X ELEMENTS OF Y FOR X<0, DROP FIRST $\vert$X ELEMENTS OF Y
⍉Y	TRANSPOSE THE DIMENSIONS OF Y (FOR A MATRIX, EXCHANGE THE ROWS AND COLUMNS)
X⍉Y	TRANSPOSE ARRAY Y ACCORDING TO X
⌽Y	REVERSE ALONG THE LAST DIMENSION OF Y
⌽[N]Y	REVERSE ALONG THE Nth DIMENSION OF Y
θY	REVERSE ALONG THE FIRST DIMENSION OF Y
X⌽Y	ROTATE BY X ALONG THE LAST DIMENSION OF Y
X⌽[N]Y	ROTATE BY X ALONG THE Nth DIMENSION OF X
XθY	ROTATE BY X ALONG THE FIRST DIMENSION OF Y
X⍋Y	GENERATE AN INDEX VECTOR SUCH THAT X[⍋Y] IS IN ASCENDING ORDER
X⍒Y	GENERATE AN INDEX VECTOR SUCH THAT X[⍒Y] IS IN DESCENDING ORDER
X⊥Y	DECODE THE REPRESENTATION OF Y IN NUMBER SYSTEM X
X⊤Y	ENCODE Y IN NUMBER SYSTEM X
?Y	ROLL AN INTEGER SELECTED RANDOMLY IN RANGE 1 THROUGH Y (SCALAR FUNCTION)
X?Y	DEAL X INTEGERS SELECTED RANDOMLY IN RANGE 1 THROUGH Y WITHOUT DUPLICATION
εY	EXECUTE THE CHARACTER STRING Y
XεY	DETERMINE THE MEMBERSHIP OF X IN ARRAY Y
⌹Y	INVERT THE MATRIX Y
X⌹Y	PERFORM MATRIX DIVISION, SOLVE LINEAR EQUATIONS, FIND A LEAST SQUARE SOLUTION

Primitive Mixed Functions

Scalar functions take scalar arguments, yield scalar results, and are extended to arrays on an element-by-element basis. Mixed functions, on the other hand, may take vector arguments and yield scalar or vector results, or may take scalar arguments and yield vector results. In expressing mixed functions for arrays of greater dimensions, it may be necessary to specify the particular coordinate of the array to which the function applies. Table 22-4 summarizes these functions.

Table 22-5 Composite Operators

OPERATOR	MEANING
α/Y	The α reduction along the last dimension of Y
α/[N]Y	The α reduction along the Nth dimension of Y
$\alpha \neq$ Y	The α reduction along the first dimension of Y
X$\alpha.\alpha$Y	Generalized inner product of X and Y
X °.αY	Generalized outer product of X and Y

Relational Functions

In APL, the relational functions ($<$, $\leq$, $=$, $>$, $\geq$, $\neq$) return results; they are not simply comparison operators. An expression of the form A$<$B yields a result of 1 true if A is less than B and 0 false if A is greater than or equal to B. For example:

enter:	9$>$6
returned:	1
enter:	4$>$6
returned:	0
enter:	'C'$>$'A'
returned:	1

Table 22-6 summarizes these functions.

Table 22-6 Logical Functions

FUNCTION	MEANING
X < Y	X less than Y
X ≤ Y	X less than or equal to Y
X = Y	X equal to Y
X ≥ Y	X greater to or equal to Y
X > Y	X greater than Y
X ≠ Y	X not equal to Y
X Λ Y	X AND Y
X V Y	X OR Y
X Λ Y	X NAND Y (not both X and Y)
X ⩝ Y	neither X nor Y
~Y	NOT Y

INPUT/OUTPUT OPERATIONS

Input/output statements are a special variety of assignment statements. In APL, input and output operations are generally expressed by the special quad operator, □.

Quad Input Mode

The most basic form of input to APL is called evaluated input. In this mode, the value entered by the user is assigned to the variable to the left of the specification arrow. In the following example:

K←□
18

The K variable takes on the value 18 entered by the user at the terminal.

If a quad symbol appears anywhere in an APL statement except immediately to the left of a left-arrow, input will be accepted from the terminal as in the following:

A←3□+5
7

Here the value of A becomes 36=3×(7+5).

Quote-Quad Input Mode

A version of the quad operator called the quote-quad operator (▢) is used especially for the input of character data. An example of quote-quad mode is shown below.

X←□
THAT'S AMAZING
X
THAT'S AMAZING

Unlike evaluated input, quote-quad input allows character strings to be entered without explicit quote characters. When APL encounters a □ symbol, it positions the carriage at the left margin and accepts the data entered by the user up to the next carriage return as a character string. If a single character is entered, APL treats it as a literal scalar; a string is stored as a literal vector. If the user enters only a carriage return, APL treats this input as a vector of length zero. This is significantly different from the handling of empty input in evaluated input mode, in which APL rejects the input and waits for the user to reenter it.

Quad-Del Input Mode
A special version of the quad operator, the quad-del operator (▱) is used to input characters exactly as typed by the user. No special editing of APL characters is performed. The backspace, for example, is treated as a special character, and an overstrike symbol is not created. The following statements illustrate this difference between quad-del and quote-quad modes.

    X←□
▱   φA
     ρ**X**
     4
    X←▱
▱   φA
     ρ**X**
     2
     ρ'φ**A**'
     2

Normal Output Mode
If a quad symbol appears immediately to the left of a left arrow, the value of the expression to the right of that specification arrow is output. Terminal output can also be accomplished simply by entering the name of the variable whose value is to be displayed. For example, the APL statement:

    □←B

using the quad symbol is equivalent to the statement:

    B

since both have the effect of displaying the value of B.

The quad output mode is especially helpful when an APL statement consists of multiple specifications. For example:

    X←15−☐←3+4
    7

Here the quantity 7 is assigned to the quad operator and displayed. The value of X is computed (its value is 8) but not displayed.

Table 22-7   Keyboard I/O Operators

OPERATOR	MEANING
X←☐	quad (evaluated) input from keyboard
X←☐	quote-quad (character) input from keyboard, up to but not including carriage return
X←☑	quad-del (unedited) input from keyboard
☐←X	quad output (display value of X)

Heterogeneous Output Mode

It is often desirable to mix character and numeric data on the same output line. Mixed output lines of this kind are called heterogeneous output. The APL user requests heterogeneous output simply by entering a series of vaues or expression, separated by semicolons, in the order in which they are to appear; the values can be parenthesized. The output displayed as a result of this specification contains no carriage returns, except where required by the data.

COMMUNICATION WITH THE SYSTEM

There are several ways in which the user can communicate with the APL system to change system parameters, determine hardware or operational characteristics, and modify workspace parameters. The system commands facilitate many of these system operations, and the I-beam functions allow APL users to communicate with the system from within the APL language itself.

System Commands

System commands provide a means of communicating with the APL system and controlling the operational environment in which an APL session is conducted. System commands allow the user to examine or change the state of the system in the following ways:

- Clear, identify, or save the active workspace
- Load or delete a workspace from a secondary storage device
- List variable and function names
- Display the status of functions and variables in the workspace
- Set the index origin, maximum number of significant digits, output line width, and comparison tolerance

I-beams

I-beams are APL functions used to communicate with the APL system to change user workspace characteristics and to report statistics about the workspace and the APL system. Unlike system commands, these functions are subject to the APL language syntax and rules of function definition. They can be included in user functions and defined in conjunction with other language operations.

There are two types of I-beam functions. The first category consists of functions used to return workspace and system information. Examples of information returned by the I-beams in this category are:

- symbol table size and remaining available space
- date and time of day
- system job number, user's project-programmer number, and terminal character set
- line numbers in the state indicator
- precision of APL version
- values of system assembly parameters

Some of these I-beams report general system characteristics (e.g., date) and others return information relevant only to the particular user's workspace and session.

The second I-beam category consists of functions used to perform system actions and change workspace parameters. Examples of actions performed by the I-beams in this category are:

- turning error displays for the execute operator on and off
- terminating the APL session
- selecting the terminal character set
- changing the random number sequence

APL STATEMENT EXECUTION

APL statements can be executed in either of two modes:

- Immediate mode, in which functions, statements, and expressions entered by the user are evaluated and executed immediately

- Function-definition mode, in which the user can contruct a program consisting of APL statements, and name and save the program

The APL user can shift from one mode to the other by typing a mode-transfer "del" (∇) symbol. The syntax of the APL language itself is identical in both modes. Some special symbols are defined for ease of editing in function-definition mode.

Defining the Function

In the APL language, a program is implemented as a user-defined function. A user-defined function can include both APL primitive functions and other user-defined functions. The user develops a program in APL function-definition mode. Once developed, that program may be used with the convenience of a primitive function.

A function is constructed in two parts: a function header and a function body. The function header defines the name of the function, the syntax of the function call, and any variables defined to be local to the function. The function body consists of a number of program statements that define the actions to be performed by the function when it is executed.

Editing the Function

A function definition can be altered by the user in a variety of ways. Definition lines can be added, deleted, displayed, and changed; and the function header can be altered. The APL statements that make up a function definition are neither executed nor checked for syntactic validity when entered by the user. In function-definition mode, the user simply enters statements, edits them to correct obvious mistypings and inconsistencies, and saves them for future use.

Executing the Function

The process of defining a function associates the function header provided by the user with the statements entered as the function body. When the user executes the defined function, the function is used as if it were a primitive APL function. The information provided in the function header specifies the number of arguments to be supplied in the function call and determines whether or not a value will be returned.

Debugging the Function

Function execution is suspended before normal completion if an error occurs or if a stop vector is set. When execution is suspended, the name of the suspended function and the line number of the statement that would have been executed next are displayed. APL then awaits input in immediate mode. The user can perform any other APL operations at this time. The user can resume execution after fixing the problem and can observe function nesting.

The Trace Vector

For debugging purposes, the user may find it helpful to obtain an automatic display of the intermediate results of function exection. As a program tracing aid, the values computed by one or more function statements can be output each time those statements are executed. To establish a trace for lines 4, 6, and 7 of function F, the user includes the following statement in the function definition:

T△F←4 6 7

For execution of the specified line numbers this command causes the following information to be displayed, in the order shown:

* function name
* line number
* final value returned by the statement

The Stop Vector

APL allows the user to suspend execution of a function from within the function itself. A stop control vector is available, with a syntax similar to that of the trace vector. The stop can be used to suspend function execution just before execution of one or more specified statements. To cause function F to be suspended before executing line 12 and line 19, the user includes the following statement in the function definition:

S△F←12 19

For each suspension, this command causes the function name and line number that was about to be executed to be displayed. To disable the stop vector for function F, the following specification is supplied:

S△F←ι0

After function execution has been suspended by the stop control vector, the system is in the normal suspended state. Execution can be resumed by specifying a branch to the desired line number.

The stop control vector can be set from within a function to cause suspension only under certain circumstances. Editing a line for which a stop vector has been defined causes the stop vector to be disabled for that line.

ENVIRONMENTS

The RT-11 and RSTS/E operating systems provide APL users with many of the standard features of the PDP-11 real-time and timesharing environments. When configured for single-user access under RT-11, the APL interpreter uses four overlay segments and requires about 14K words of memory. When configured for use with RSTS/E, several users can simultaneously access the APL system. The APL interpreter can be configured as a RSTS/E run-time system. Each user shares the reentrant interpreter code, and only the user's workspaces are swapped. The APL run-time system uses approximately 16K words of memory.

APL can be used on a variety of PDP-11 processors. It has been optimized to make efficient use of systems that offer hardware floating point processors, for example, the PDP-11/34, 11/45, 11/55, and 11/70. However, APL can also be configured for use with processors that do not have floating point processors, or, in the case of RT-11, those processors that have the FIS instruction set. If a hardware processor is not available, a software floating point package will be included with the APL interpreter to simulate the floating-point hardware.

APL can be generated to perform either single precision or double precision arithmetic. Single precision provides an accuracy of approximately 7 digits, and double precision offers an accuracy of about 16 digits. I-beam 37 can be used to determine the precision of a particular APL system.

Workspaces

An APL workspace is a buffer in the user's memory area which stores the functions, variables, values, and temporary results obtained while executing APL statements. Using the APL system commands, workspaces can be saved, retrieved, and erased in the same manner as any other file. They can be stored on a variety of PDP-11 devices, including disk, magnetic tape, DECtape, and floppy disk.

A workspace can be saved in either memory-image or ASCII format. Workspaces saved in ASCII form can be created and edited with any DIGITAL editor. This is an important feature not found on many APL systems.

There may be several workspaces stored in the user's disk area. The workspace currently available to the user is known as the active workspace. The maximum APL workspace depends upon the operating system and the amount of memory in the system. In an RT-11 system with 28K words of memory, the workspace may be approximately 24,000 bytes.

File System

The APL file system allows the APL user to access data and program files on a variety of system devices, including disk, DECtape, magnetic tape, and floppy disk. The file system is implemented as an integral part of the APL language and provides an interface to the RSTS/E and RT-11 operating systems.

The APL file system support is provided by:

• System commands for use in assigning, creating, closing, reading, writing, and renaming files

• File operators for byte pointer, input, and output functions

Table 22-8 File I/O Operators

OPERATOR		MEANING
Channel-Number	[type] N	set file pointer
Channel-Number	[type] N	file input
Channel-Number	[type] data	file output

In the APL system, input and output functions can be specified for files associated with any of 13 channels, one of which is reserved of use by the user's terminal. The ASSIGN file system command is used to associate an existing file with a channel. CREATE is used to create new files on specified channels by allocating space for them.

Two types of files are supported by the APL system:

• ASCII sequential

• Random access

APL ASCII sequential data files can be read and written sequentially by any other RT-11 or RSTS/E language processor (e.g., BASIC, FORTRAN, or MACRO). In addition, an APL program can create and read any standard ASCII files. Because APL workspaces can be read and stored in ASCII format, the file system can be used to save, retrieve, and manipulate these workspaces.

The APL system also supports the use of random access files. The system treats a file as random access memory, and the user can access any byte in the file directly by specifying the individual byte or value to be read or written. APL can access random files in any format, created with almost any language processor or system. For example, random access mode can be used to read and write FORTRAN direct access data files.

RPG II (V8)

OVERVIEW

RPG II is a complete program generating system that provides users with a ready means of developing applications programs. It supports almost all functions offered by industry versions of RPG II, and provides significant improvements over RPG I. The RSTS/E operating system supports RPG II as a language option.

FEATURE TOPICS

- Description
- Language Specifications
- Features
  - File Support for Peripherals
  - File Organizations
  - Record Access Methods

DESCRIPTION

DIGITAL's Report Program Generator (RPG II) is a high level computer language system with a one-pass multi-phase compiler and an object time library system.

RPG II requires as input an ordered set of RPG II source specifications and generates as output a machine language object program. In addition, RPG II optionally produces a source program listing with error diagnostic messages, if any.

LANGUAGE SPECIFICATIONS

Effective use of RPG II requires following these basic steps: analyzing the problem to be solved, including design of the input and output data formats; encoding the data formats and requisite calculations for interpretation by RPG II; entering the coded specifications into the system; compiling the specifications into an object program by RPG II; preparing an executable program from the object program; and executing the resulting RPG II program to perform the desired processing.

The coding specifications have seven possible formats. They are:

- the Control Specifications (H Format), which supply information pertaining to the compilation as a whole

- the File Description Specifications (F Format), which describe files to be used by the program

- the Extension Specifications (E Format), which describe the tables and arrays to be used by the program and provide for additional file information

- the Line Counter Specifications (L Format), which give special information about print output

- the Input Specifications (I Format), which describe input records and fields

- the Calculation Specifications (C Format), which describe the operations to be performed on previously specified data and define the data fields which are not previously defined

- the Output Specifications (O Format), which describe the format of output records and the types of data fields.

FEATURES

DIGITAL's RPG II supports almost all functions offered by industry versions of RPG II and provides significant improvements over the first level of RPG. These features make the language easier to use and more flexible. Among the features of RPG II are:

- An easy-to-use programming language with standard application-oriented specification statements.

- A full set of 31 instructions.

- A complete set of compiler diagnostics available to aid in program debugging; also, a DEBUG facility that allows tracing of user RPG II programs. DEBUG allows source-level debugging by printing the setting of indicators and the contents of any field.

- Full support for a wide range of peripheral devices including:
    Card Readers
    Magnetic Tape
    Disk
    Printers
    Terminals

- File organizations:
    Sequential
    Direct
    Indexed

- Record access methods:
    Consecutive
    Sequential by key
    Sequential within limits
    Random by key
    Random by relative record number
    Random utilizing ADDress ROUTing files

- A DSPLY operation code that provides the ability to display messages on a user's terminal during program execution and to accept data in reply.

- Support of console devices as normal files. This differs from the DSPLY function which supports the console as a field-entering device.

- Support of ASCII, binary, packed decimal, over punched, and zoned decimal numeric data.

- Control of page length and overflow via line counter specifications.

- Repetitive printing of the initial first page output line to assist in the proper alignment of printer forms.

- Up to nine matching fields to control multi-file processing.

- Access to the system data previously entered via the system monitor. This allows the entire date, or any of its three parts, to be referenced by the reserved names UDATE, UMONTH, UDAY, or UYEAR.

- Access to the contents of fields in records awaiting processing through the Look Ahead feature. Calculations, testing, comparisons, or output may be made using the contents of these fields.

- Control over which file and record are to be processed in the next RPG II program cycle, through the FORCE operation code.

- Use of switches (U1-U8) which are set at execution time to control calculations, input files, output files, or specific output records.

- Access to records from a demand file during calculations cycle through the READ operation code.

- Control of random record access by key or record number through the CHAIN operation.

- Ability to load tables and arrays at compile time or from input files.

- Ability to punctuate numeric output fields by specifying a single character edit code.

- Ability to write records during total or detailed calculations through the EXCPT operation.

- Executions of internal subroutines from any point within calculations.

- Ability to compute the square root of the value in a given field through the SORT operation.

- Page-overflow processing control provided by the fetch overflow feature.

- Ability to set and test bits of a one-byte binary field.

- Table and array processing features including maintenance, argument searching, and calculations operations on individual elements or entire arrays.

- Ability to do multiple updating functions within the same program cycle.

FOCAL (V1)

OVERVIEW

FOCAL, the FOrmula CALculator programming language has been designed to function effectively in a real-time scientific environment where extensive statistical analysis and numerical calculations are required. FOCAL allows calculations and operations to be performed immediately in response to a user command. The user can also string together FOCAL commands to form a program. In addition, a complete set of statements to perform arithmetic operations, program control, and input/output functions is provided.

FEATURE TOPICS

- Features
  - Real-Time Support
  - Time Queuing
  - Interrupt Handling
  - Analog Input
  - Digital Input and Output
  - Time Measurements
- Graphics Support
  - Video Terminal Graphics
  - VT11 Graphics
  - Point-Plot Graphics
- Minimum Focal System Requirements
  - Support Options
- Language
  - The Command Interpreter
  - The Program Storage Area
  - The Variable Storage Area
- FOCAL Program Structure
- FOCAL Commands
- FOCAL Function Calls

FEATURES

FOCAL (FOrmula CALculator) is an interactive high-level programming language designed for scientists who require a real-time language that is easy to learn and use. FOCAL provides both data acquisition and experiment control, as well as data analysis capabilities.

FOCAL allows calculations and operations to be performed immediately in response to a user command (calculator mode), or the creation of programs from these commands. Because FOCAL is an interactive language, users can rapidly develop, debug, and modify their problem-solving programs without separate edit, compile, link, and load phases. All programs may be saved or loaded from mass storage files, and very long programs may be implemented by dividing them into segments that are chained together; i.e., each segment can be pulled from mass storage as needed.

Since FOCAL operates under the RT-11 operating system, a program may read or write data in a totally device-independent manner. Device handlers and buffers are resident only when actually being utilized. FOCAL may define single or double subscripted variables to be virtual variables. In mass storage, such variables may hold vectors or arrays composed of hundreds or thousands of elements, and yet the user may use these variables in computations as if they were held entirely in memory. Virtual variable files may also be read or written by BASIC or FORTRAN IV programs. They are also compatible with LA-11 (Laboratory Applications) data files. Also, user-written MACRO routines may be easily incorporated into FOCAL to provide additional functions.

Two versions of FOCAL are provided. One version supports single precision floating point arithmetic, which allows approximately 7 significant digits. The other version supports double precision arithmetic, which allows approximately 17 significant digits.

Real-Time Support

FOCAL has real-time features which enable the user to control complex processes while acquiring data.

Time Queuing

Mini-tasks written in FOCAL may be scheduled to occur at certain time intervals. The time intervals may be specified either in one-second units, or at line frequency resolution.

Interrupt Handling

Mini-tasks in FOCAL may be linked to interrupt vectors so that external events can bring about a series of FOCAL commands. Each mini-task may have its software priority set to one of seven levels.

Analog Input

Multi-channel analog data may be acquired and placed into a ring buffer at rates up to 5 kHz. Moreover, by using special functions, analog data can be moved continously to a mass storage file at rates over 3 kHz. Data can be read from the ring buffer by the FOCAL program while sampling proceeds.

Digital Input and Output

A function is provided for handling multiple digital input/output options, as well as multiple digital-to-analog channels. This function will also control user-implemented devices connected to the system. A function has been implemented for the setting, clearing, and logical testing of specific bits in data words.

Time Measurements

Extensive real-time clock functions have been implemented to allow a wide variety of time-related operations. Timing resolutions of 1 millisecond may be effected both for generating timed events and for measuring the time intervals between external (or internal) events.

Graphics Support

Graphics displays have an important role in both data collection and data analysis. FOCAL is strongly graphics oriented and supports a range of graphic display systems.

Video Terminal Graphics

The VT55 video graphics terminal is fully supported by FOCAL/RT-11. This support is provided by a set of functions which may be used with the real-time laboratory functions. Thus, a researcher may use the VT55 to display the statistical analysis of data. The VT55 support module may be used without the lab functions to provide a computational terminal with graphic display utilizing double precision data and virtual subscripted variables.

VT11 Graphics

FOCAL provides control of the VT11 graphics processor. Both points and vectors may be drawn at any one of eight intensities on the CRT, and hardware-generated characters may also be displayed. FOCAL provides control over the hardware features of the VT11, including the light pen.

Point-Plot Graphics

FOCAL supports graphics for point-plot, refresh CRTs. Relative or absolute points may be specified by loading coordinates into the addressable graphics buffer. It is possible to specify alphanumeric characters in the same manner. Characters may be positioned anywhere on the screen and may be generated in one of five different sizes. The

user may expand or alter the character set, as well as create a new dictionary which generates characters composed in other than the standard 5 x 7 matrix format. Elements of the display, including characters, may be moved dynamically on the CRT under program control. Moreover, entire pictures may be created and saved in a mass storage file, and later retrieved and displayed on the CRT at rates of up to 24 pictures per second.

General Features

A variety of general features has been implemented in FOCAL. For example, error conditions may cause a specific group of statements to be executed; constants may be either decimal or octal; data may be printed in integer, decimal, scientific notation, octal, or binary formats; and a user may respond to a request for information by entering it in scientific notation, as a decimal or octal number, a series of alphabetic characters, or even an arithmetic expression.

Minimum FOCAL System Requirements

1. Any system which supports RT-11
2. 8K of memory (16K recommended)

Supported Options

- Memory: up to 28K words
- Line frequency clock
- Extended Arithmetic Element (EAE)
- Extended Instrument Set (EIS)
- Floating Point Instruction Set (FIS)
- Floating Point Unit (FPU)
- Line printer
- UNIBUS programmable real-time clock
- UNIBUS multi-channel A/D converter
- UNIBUS multi-channel D/A converter (with or without CRT control)
- Multiple digital I/O options
- VT11 graphics processor system with CRT, light pen
- VT55 video graphics terminal

LANGUAGE

The FOCAL system consists of three parts:

The **command interpreter** that reads the FOCAL command and performs the indicated operation.

The **program storage area**, used by FOCAL to remember the program written by the user.

The **variable storage area**, the area sharing the same space in the computer's memory as the program storage area. When FOCAL is loaded, all memory not occupied by the command interpreter is set aside for programmed variable storage.

FOCAL PROGRAM STRUCTURE

A FOCAL **program** is a collection of commands that are organized to perform a given task. The method by which these commands are organized is the key to FOCAL programming.

A FOCAL **statement** is a set of one or more FOCAL **commands** placed on line. The FOCAL statement is the smallest section of a program which may be referenced by FOCAL. An example of a FOCAL statement is:

SET A = 15;TYPE A;QUIT

The FOCAL commands SET, TYPE, and QUIT would be executed in that order. When more than one FOCAL command is placed on single line, they are separated by semi-colons (;).

When a user wishes to store a FOCAL statement for later use, it is assigned a statement number as a reference. The statement number may be any value from 1.01 to 99.99 in .01 increments, with the exception of those numbers ending in .00. Line numbers do not require that two digits be specified after the decimal point. The line number 2.1 is equivalent to 2.10. When a user wishes to store a FOCAL statement, the line need only be prefixed by the appropriate statement number. For example:

1.1 SET A = 1.5;TYPE A;QUIT

This FOCAL statement would be entered into the FOCAL program storage area, and could be referenced by FOCAL commands using the number 1.1. If another FOCAL statement with this line number currently exists in the program storage area, it will be replaced when the new line is entered.

All FOCAL statements that have the same integer as a portion of their line number are collectively referred to as a FOCAL **group**. This allows certain FOCAL commands to refer to these FOCAL statements as if they were all a single long FOCAL statement. FOCAL groups are referenced using the integer portion of the line number. For example:

1.1 SET A = 1.5;TYPE A
1.2 SET B = 2.4;TYPE B
1.3 SET C = A*B;TYPE C

507

2.1 SET D=A+B;TYPE D
2.2 TYPE THE END
2.3 QUIT

The FOCAL command ERASE 1.2 would remove just statement 1.2, while ERASE 1 would delete lines 1.1, 1.2, and 1.3.

FOCAL COMMANDS

Each FOCAL statement consists of one or more FOCAL commands with their associated arguments (if any). FOCAL commands may be abbreviated by a single character. They require at least one space between the command and its associated arguments. This conserves memory, as one word of memory is required for each two characters in the user's program. The following table lists the FOCAL commands.

Command	Abbrev	Example	Function
ASK	A	ASK X	Assigns values to variables from the keyboard.
COMMENT	C	C EXAMPLE	Allows the user to document his program with comments or non-executable program steps.
DO	D	DO 2	Directs program execution to a statement or group of statements which will be executed until either a "RETURN" command is executed or the end of a statement or group is reached. When this occurs, the next statement executed is the statement following the last DO command encountered.
ERASE	E	ERASE 4.1	Erases part of a program or an entire program.

Command	Abbrev	Example	Function
FOR	F	FOR I=1, 5; SET X=X+1	Executes the remaining command on a line while incrementing the value of the variable specified in the FOR command over the defined limits. This is used for loop control.
GO	G	GO 2.4	Begins executing commands at the statement specified. Used to direct program control to the lowest line number, or to a specific group or statement.
IF	I	IF(A)1.1, 1.2,1.3	Conditionally directs program execution using the results of testing the sign of A. (−,0,or +)
KILL	K	KILL	Stops the program and I/O activity.
LIBRARY	L	LIBRARY RUN TEST	Allows the program to access file-structured devices. (RT-11 version only)
MODIFY	M	MODIFY 1.1	Used to alter words or characters in a stored program statement.
OPERATE	O	OPERATE TK	Used to select non-file structured I/O devices for I/O.
QUIT	Q	QUIT	Used to terminate program execution and return control to user.

Command	Abbrev	Example	Function
RETURN	R	RETURN	Used to return program execution to the statement following the last "DO" command executed.
SET	S	SET A = 1.531	Assigns the value of the result of an expression to the variable.
TYPE	T	TYPE "HI"!, A	Sends output to the currently selected output device. Used to print text, results of calculations, and values of variables.
WRITE	W	WRITE 1	Used to list part or all of a program.
XECUTE	X	XECUTE FPRM (3,1)	Used to call and execute functions.

Several commands may be put together in the same FOCAL statement as long as the total statement does not exceed 79 characters in length. Semi-colons are used to separate FOCAL commands within a statement.

All FOCAL statements must be terminated by the carriage-return character. A FOCAL command can be terminated by either a semi-colon (;) or a carriage-return. For example:

1.10 FOR I = 1,5;DO 2;TYPE "I",I,!

There are a few exceptions to this rule: the COMMENT, ERASE, and certain LIBRARY commands can be terminated only by a carriage-return.

FOCAL FUNCTIONS

The FOCAL functions (subprograms internal to FOCAL) improve and simplify arithmetic capabilities, and give the potential for expansion to additional input/output devices.

In general, the FOCAL functions may be used anywhere a number or a variable is legal in a mathematical expression. A standard function call consists of two or more letters beginning with the letter F and followed by an argument expression in parenthesis, Fxxx(expression).

The following standard functions are available:

FOCAL Function Call	Function
FSIN(R)	Sine function (radians)
FCOS(R)	Cosine function (radians)
FEXP(ARG)	Exponential function
FLOG(ARG)	Logarithm to the base 10.
FLN(arg)	Natural logarithm
FX(func,addr,data)	Access to UNIBUS or Q-bus devices
FCHR(arg)	Print and accept ASCII codes
FRAN()	Random number function
FADC(channel)	Analog to digital converter function
FCLK()	Clock function
FABS(arg)	Absolute value function
FSGN(arg)	Sign function
FITR(arg)	Integer part function
FSQT(arg)	Square root function
FSBR(group, arg)	User programmed function
FPRM (parameter, value)	Alter FOCAL internal parameters
FERR(group/line)	Define error handling routine
FINT(vector, group, pri, CSRaddr, mask)	Establish a routine to be executed on the detection of a specific hardware interrupt.
FQUE(count, group, interval, delay, priority)	Schedule a group or line number to be run count times, once every interval seconds, starting delay seconds from now. The routine will have a priority of priority.

GLOSSARY

absolute address   A binary number that is assigned as the address of a physical memory storage location.

absolute loader   A stand-alone program which, when in memory, enables the user to load into memory data in absolute binary format.

account number   A discrete code used to identify a system user. It normally consists of two numbers, separated by a comma, called the project number and programmer number or the group number and member number. See also **user identification code**.

active task list   A priority-ordered list of active tasks used normally in an event-driven multiprogrammed system to determine the order in which tasks receive control of the CPU.

address   A label, name or number that designates a location where information is stored.

alphanumeric   Referring either to the entire set of 128 ASCII characters or the subset of ASCII characters which include the 26 alphabetic characters and the ten numeric characters.

ancillary peripherals   In the MUMPS-11 system, peripherals not under control of the data base supervisor.

append   To add information to the end of an existing file.

application program   A program that performs a task specific to a particular end-user's needs. Generally, an application program is any program written on a program development operating system that is not part of the basic operating system.

array   An ordered arrangement of subscripted variables.

ASCII   The American Standard Code for Information Interchange, consisting of 128 7-bit binary codes for upper and lower case letters, numbers, punctuation and special communication control characters.

assembler   A program that translates symbolic source code into machine instructions by replacing symbolic operation codes with binary operation codes and symbolic addresses with absolute or relocatable addresses.

assembler directives   The mnemonics used in an assembly language source program that are recognized by the assembler as commands to control and direct the assembly process.

assembly language   A symbolic programming language that can normally be translated directly into machine language instructions and is, therefore, specific to a given computing system.

assembly listing   A listing produced by an assembler that shows the symbolic code written by a programmer next to a representation of the actual machine instructions generated.

assigning a device   Putting an I/O device under control of a particular user's job either for the duration of the job or until the user relinquishes control. See also **attach**.

asynchronous   A mode of operation in which an operation is started by a signal that the operation on which it depends is completed. When referring to hardware devices, it is the method in which each character is sent with its own synchronizing information. The hardware operations are scheduled by ready and done signals rather than by time intervals. In addition, it implies that a second operation can begin before the first operation is completed.

asynchronous system trap   A system condition which occurs as the result of an external event such as completion of an I/O request. On occurrence of the significant event, control passes to an AST service routine.

attach   To dedicate a physical device unit for exclusive use by the task requesting attachment. See also assigning a device.

backup file   A copy of a file created for protection in case the primary file is unintentionally destroyed.

bad block   A defective block on a storage medium that produces a hardware error when attempting to read or write data in that block.

base address   An address used as the basis for computing the value of some other relative address.

base segment   The portion of a program using overlays that is always memory-resident. See also **root segment**.

batch processing   A processing method in which programs are run consecutively without operator intervention.

batch stream   The collection of commands and data interpreted by a batch processor that directs batch processing.

binary   The number system with a radix of two.

binary code   A code that uses two distinct characters, usually the numbers 0 and 1.

binary loader   See **absolute loader**.

bit   A binary digit.

bit map   A table describing the state of each member of a related set. A bit map is most often used to describe the allocation of storage space. Each bit in the table indicates whether a particular block in the storage medium is occupied or free.

block   A group of physically adjacent words or bytes of a specified size which is particular to a device. The smallest system-addressable segment on a mass-storage device in reference to I/O.

Boolean valued expression   An expression which, when evaluated, produces either a true or false result.

bootstrap   A technique or device designed to bring itself into a desired state by its own action.

bootstrap loader   A routine whose first instructions are sufficient to load the remainder of itself into memory from an input device and normally start a complex system of programs.

bottom address   The lowest memory address in which a program is loaded.

breakpoint   A location at which program operation is suspended in order to examine partial results. A preset point in a program where control passes to a debugging routine.

buffer   A storage area used to temporarily hold information being transferred between two devices or between a device and memory. A buffer is often a special register or a designated area of memory.

bug   An instruction or sequence of instructions in a program that causes unexpected and undesired results.

byte   The smallest memory-addressable unit of information in a PDP-11 system. A byte is equivalent to eight bits.

carriage return key   The key on a terminal keyboard most often used in PDP-11 systems to terminate input lines.

Central Processing Unit or Central Processor   That part of a computing system containing the arithmetic and logical units, instruction control unit, timing generators and memory and I/O interfaces.

character   A single letter, numeral, or symbol used to represent information.

checksum   A number used for checking the validity of data transfers.

clock   A time-keeping or frequency-measuring device within a computing system.

code   A system of symbols and rules used for representing information.

coding   Writing instructions for a computer using symbols meaningful to the computer itself, or to an assember, compiler or other language processor.

collate   To combine items from two or more ordered sets into one set having an order not necessarily the same as any of the original sets.

command or command name   A word, mnemonic, or character which, by virtue of its syntax in a line of input, causes a predefined operation to be performed by a computer system.

command language   The vocabulary used by a program or set of programs that directs the computer system to perform predefined operations.

Command Language Interpreter   The program that translates a predefined set of commands into instructions that a computer system can interpret.

command string   A line of input to a computer system that generally includes a command, one or more file specifications, and optional qualifiers.

Command String Interpreter   A special program or routine that accepts a line of ASCII string input and interprets the string as input and output file specifications with recognized qualifiers.

common   A section in memory which is set aside for common use by many separate programs or modules.

compile   To produce binary code from symbolic instructions written in a high-level source language.

compiler   A program which translates a high-level source language into a language suitable for a particular machine.

completion routine   A routine that is called at the completion of an operation.

compute bound   A state of program execution in which all operations are dependent on the activity of the central processor; for example, when a large number of calculations are being performed. See also **I/O bound**.

computer operator   A person who performs standard system operations such as adjusting system operation parameters at the system console, loading a tape transport, placing cards in a card reader, and removing listings from the line printer.

concatenate   To combine several files into one file, or several strings of characters into one string, by appending each file or string one after the other.

conditional assembly   The assembly of certain parts of a symbolic program only when certain conditions are met.

configuration   A particular selection of hardware devices or software routines or programs that function together.

consecutive access   The method of data access characterized by the sequential nature of the I/O device involved. For example, a card reader is an example of a consecutive access device; each card must be read one after another, and no distinction is made between logical sets of data in or among the cards in the input hopper.

console   The console of a central processor is the set of switches and display lights used by an operator or programmer to determine the status and control the operation of the computer.

console terminal   A keyboard terminal which acts as the primary interface between the computer operator and the computer system and is used to initiate and direct overall system operation through software running on the computer.

constant   A value which remains the same throughout a distinct operation. Compare with **variable**.

context switching   The switching between one mode of execution and other, involving the saving of key registers and other memory areas prior to switching between jobs, and restoring them when switching back. A common example of context switching is the temporary suspension of a user program so that the monitor or executive can execute an operation.

contiguous file   A file consisting of physically adjacent blocks on a mass-storage device.

control character   A character whose purpose is to control an action rather than to pass data to a program. An ASCII control character has an octal code between 0 and 37. It is typed by holding down the CTRL key on a terminal keyboard while striking a character key.

control section   A named, contiguous unit of code (instructions or data) that is considered an entity and that can be relocated separately without destroying the logic of the program.

core memory   The most common form of main memory storage used by the central processing unit, in which binary data is represented by the switching polarity of magnetic cores.

core common   See **common**.

crash   A hardware crash is the complete failure of a particular device, sometimes affecting the operation of an entire computer system. A software crash is the complete failure of an operating system characterized by some failure in the system's protection mechanisms. In actual occurrence, it is a system-level trap, e.g., trap to location 4 or 10 (attempt to execute an illegal instruction, parity error, etc.) when the system's trap routines have been destroyed.

create   To open, write data to, and close a file for the first time.

cross reference listing or table   A printed listing that identifies all references in a program to each specific label in a program. A list of all or a subset of symbols used in a source program and statements where they are defined or used.

CTRL/C   The control character issued from a terminal which is most commonly used to return the operator to communication with the system-level program. In most PDP-11 systems, it is typed on the terminal keyboard to gain the attention of the operating system before commencing the login procedure, or to terminate the currently-executing program and return to communication with the monitor. In some cases, it simply issues a call to the console listener or console service routine without interrupting current program execution.

CTRL/U   The control character issued from a terminal that tells the program currently accepting input to ignore the characters entered on the line up to the point where CTRL/U was typed.

CTRL/Z   Used in RSX-11 systems to terminate the system program currently waiting for input from the terminal. It is essentially an end-of-file character.

data base   A collection of interrelated data items organized by a consistent scheme that allows one or more applications to process the items without regard to physical storage locations.

data base management system   A scheme used to create, maintain and reference a data base.

debug   To detect, locate, and correct coding or logic errors in a computer program.

DECnet   A family of hardware/software products that create distributed networks from DIGITAL computers and their interconnecting data links.

DECtape   A convenient pocket-sized reel of magnetic tape developed by DIGITAL for extremely reliable data storage and random access.

default   The value of an argument, operand or field assumed by a program if a specific assignment is not supplied by the user.

delimiter   A character that separates, terminates or organizes elements of a character string, statement or program.

detach a device   Free an attached physical device unit for use by tasks other than the one that attached it.

device   A hardware unit such as an I/O peripheral, e.g., magnetic tape drive, card reader, etc. Also often used synonymously with volume.

device controller   A hardware unit which electronically supervises one or more of the same type of devices. It acts as the link between the CPU and the I/O devices.

device driver   A program that controls the physical hardware activities on a peripheral device. The device driver is generally the device-dependent interface between a device and the common, device-independent I/O code in an operating system.

device handler   A program that drives or services an I/O device. A device handler is similar to a device driver, but provides more control and interfacing functions than a device driver.

device name   A unique name that identifies each device unit on a system. It usually consists of a 2-character device mnemonic followed by an optional device unit number and a colon. For example, the common device name for DECtape drive unit one is "DT1:".

device unit   One of a set of similar peripheral devices; e.g., disk unit 0, DECtape unit 1, etc. Also used synonymously with volume.

DIGITAL Network Architecture (DNA)   The common network architecture of DECnet.

direct access   See **random access**.

direct mode   The mode of MUMPS-11 system operation which enables the programmer to: enter commands and or functions for immediate execution, and create or modify steps of a user's program.

directive   A type of executive request issued by a program that provides a facility inherent in the hardware which is controlled and organized by the operating system. See also **programmed request**.

directory   A table that contains the names of and pointers to files on a mass-storage device.

directory device   A mass-storage retrieval device, such as disk or DECtape, that contains a directory of the files stored on the device.

double-buffered I/O   An input or output operation which uses two buffers to transfer data. While one buffer is being used by the program, the other buffer is being read from or written to by an I/O device.

executive   The controlling program or set of routines in an operating system. The executive coordinates all activities in the system including I/O supervision, resource allocation, program execution, and operator communication. See also **monitor**.

executive mode   A central processor mode characterized by the lack of memory protection and relocation by the normal execution of all defined instruction codes.

exponentiation   A mathematical operation denoting increases in the base number by a factor previously selected.

expression   A combination of operands and operators which can be evaluated to a distinct result by a computing system.

external storage   A storage medium other than main memory.

file   A logical collection of data treated as a unit which occupies one or more blocks on a mass-storage device such as disk, DECtape, or magtape. A file can be referenced by a logical name.

file gap   A fixed length of blank tape separating files on a magnetic tape volume.

file name   The alphanumeric character string assigned by a user to identify a file, and which can be read by both an operating system and a user. A file name identifies a unique member of a group of files which: 1) has the same file name extension and version number (if any), 2) is located on the same volume, and 3) belongs in the same User File Directory (if any). A file name has a fixed maximum length which is system dependent (generally six or nine characters).

filename extension   The alphanumeric character string assigned to a file either by an operating system or a user, and which can be read by both the operating system and the user. System-recognizable filename extensions are used to identify files having the same format or type. If present in a file specification, a filename extension follows the file name in a file specification, separated from the file name by a period. A file name extension has a fixed maximum length which is system dependent (generally three characters, excluding the preceding period).

file specification   A name that uniquely identifies a file maintained in any operating system. A file specification generally consists of at least three components: a device name identifying the volume on which the

file is stored, a file name, and a filename extension. In addition, depending on the system, a file specification can include a User File Directory name or UIC, and a version number.

file structure   A method of recording and cataloging files on mass-storage media.

file-structured device   A device on which data is organized into files. The device usually contains a directory of the files stored on the device.

file type   See **filename extension**.

floating point numeric   A floating point number which, if stored in four words, is approximately in the range 10^{-38} to 10^{38}.

foreground   The area in memory designated for use by a high-priority program. The program, set of programs, or functions that gain the use of machine facilities immediately upon request.

format   The arrangement of the elements comprising any field, record, file or volume.

formatted ASCII   Refers to a mode in which data is transferred. A file containing formatted ASCII data is generally transferred as strings of 7-bit ASCII characters (bit eight is zero) terminated by a line feed, form feed or vertical tab. Special characters, such as NULL, RUBOUT and TAB may be interpreted specially.

formatted binary   Refers to a mode in which data is transferred. Formatted binary is used to transfer checksummed binary data (8-bit characters) in blocks. Formatting characters are start of block indicators, byte count and checksum values.

formatted device   A volume which has been prepared for use on a system under program control.

function   An algorithm accessible by name and contained in the system software which performs commonly-used operations. For example, the square root calculation function.

generation number   See **version number**.

global   A value defined in one program module and used in others. Globals are often referred to as entry points in the module in which they are defined, and externals in the other modules which use them. Also, in the MUMPS-11 system, a global array.

global array   A data file stored in the common MUMPS-11 data base. Global arrays constitute an external system of symbolically referenced arrays.

global variable   A global variable in the MUMPS-11 system is a subscripted variable which forms a part (or node) of a global array.

handler   See **device handler**.

hardware   The physical equipment components of a computer system.

high-level language   A programming language whose statements are translated into more than one machine language instruction. Examples are BASIC, FORTRAN, and COBOL.

I/O page   That portion of memory in which specific storage locations are associated directly with I/O devices.

I/O rundown   A process which delays the availability of a partition until all transfers to and from that partition have been stopped or have been allowed to complete. I/O rundown is invoked when a task is terminated and has outstanding transfers pending to or from its partition.

idle time   That part of uptime in which no job could run because all jobs are halted or waiting for some external action such as I/O.

image mode   Refers to a mode of data transfer in which each byte of data is transferred without any interpretation or data changes.

impure code   The code which is modified during the course of a program's execution; e.g., data tables.

incremental compiler   A compiler that immediately translates each source statement into an internal format, ready for execution.

indirect file   A file containing commands that are processed sequentially, yet which could have been entered interactively at a terminal.

indirect mode   The mode of MUMPS-11 system operation in which steps of a stored program can be executed. In this mode, neither commands nor functions can be entered at the terminal, nor can programs be created or modified.

indirect reference   A feature of the MUMPS language which permits the symbolic representation of an argument or argument list in a command by a string variable. In operation, the string value of the variable is taken as the argument or argument list for the command. The indirection symbol, a back-arrow or underscore must precede the variable reference.

initialize   To set counters, switches, or addresses to starting values at prescribed points in the execution of a program, particularly in preparation for re-execution of a sequence of code. To format a volume in a particular file-structured format in preparation for use by an operating system.

interactive   A technique of user/system communication in which the operating system immediately acknowledges and acts upon requests entered by the user at a terminal. Compare with **batch**.

interpreter   A computer program that translates and executes each source language statement before translating and executing the next statement.

interrupt   A signal which, when activated, causes a transfer of control to a specific location in memory, thereby breaking the normal flow of control of the routine being executed. An interrupt is normally caused by an external event such as a done condition in a peripheral. It is distinguished from a trap which is caused by the execution of a processor instruction.

interrupt service routine   The routine entered when an external interrupt occurs.

interrupt vector address   A unique address which points to two consecutive memory locations containing the start address of the interrupt service routine and priority at which the interrupt is to be serviced.

I/O bound   A state of program execution in which all operations are dependent on the activity of an I/O device. For example, when a program is waiting for input from a terminal. See also **compute bound**.

job   A group of data and control statements which does a unit of work; e.g., a program and all its related subroutines, data and control statements; also, a batch control file.

journaling   The parallel writing of updated records to a second medium in addition to the original file.

keyboard monitor   A program that provides and supervises communication between the user at the system console and an operating system.

latency   The time from initiation of a transfer operation to the beginning of actual transfer; i.e., verification plus search time. The delay while waiting for a rotating memory to reach a given location.

leader   A blank section of tape at the beginning of a reel of magnetic tape or at the beginning of paper tape.

library   A file containing one or more relocatable binary modules which are routines that can be incorporated into other programs.

library   A class of MUMPS programs listed in the system program directory and available to all users of the system.

line   A string of characters terminated with a vertical tab, form feed or line feed.

linked file   A file whose blocks are joined together by references (a link word or pointer imbedded in the block) rather than consecutive location.

linker   A program that combines many relocatable object modules into an executable program module. It satisfies global references and combines control sections.

linking loader   A program that provides automatic loading, relocation and linking of compiler and assembler generated object modules.

listing   The hard copy generated by a line printer.

literal   An element of a programming language which permits the explicit representation of character strings in expressions and command and function elements. In most languages, a literal is enclosed in either single or double quotes to denote that the enclosed string is to be taken "literally" and not evaluated.

load   To store a program or data into memory. To mount a tape on a device such that the read point is at the beginning of the tape. To place a removable disk in a disk drive and start the drive.

load image file   A program that can be executed in a stand-alone environment without the aid of relocation.

load map   A table produced by a linker that provides information about a load module's characteristics; e.g., the transfer address and the low and high limits of the relocatable code.

load module   A program in a format ready for loading and executing.

local variable   In the MUMPS-11 system, a local variable is a variable which is stored only in the partition in which a program is executed (as opposed to a global variable).

location   An address in storage or memory where a unit of data or an instruction can be stored.

log in   To identify oneself to an operating system as a legitimate user of the system and gain access to its services.

log out or log off   To sign off a system.

logical block   An arbitrarily-defined fixed number of contiguous bytes which is used as the standard I/O transfer unit throughout an operating system. For example, the commonly-used logical block in PDP-11 systems is 512 bytes long. An I/O device is treated as if its block length is 512 bytes, although the device may have an actual (physical) block length which is not 512 bytes. Logical blocks on a

device are numbered from block 0 consecutively up to the last block on the volume. A logical block is synonymous with a physical block on any device that has 512-byte physical blocks. See also **virtual block**, **physical block**, **logical record**, and **physical record**.

logical device name   An alphanumeric name assigned by the user to represent a physical device. The name can then be used synonymously with the physical device name in all references to the device. Logical device names are used in device independent systems to enable a program to refer to a logical device name which can be assigned to a physical device at run-time.

logical record   A logical unit of data within a file whose length is defined by the user and whose contents have significance to the user. A group of related fields treated as a unit.

logical unit number   A number associated with a physical device unit during a task's I/O operations. Each task in the system can establish its own correspondence between logical unit numbers and physical device units.

macro   Directions for expanding abbreviated text. A boilerplate that generates a known set of instructions, data or symbols. A macro is used to eliminate the need to write a set of instructions which are used repeatedly. For example, an assembly language macro instruction enables the programmer to request the assembler to generate a predefined set of machine instructions.

main memory   The set of storage locations connected directly to the Central Processing Unit. Also called (generically) core memory.

main program   The module of a program that contains the instructions at which program execution begins. Normally, the main program exercises primary control over the operations performed and calls subroutines or subprograms to perform specific functions.

mapped system   A system which uses the hardware memory management unit to relocate virtual memory addresses.

mass storage   Pertaining to a device which can store large amounts of data readily accessible to the Central Processing Unit; for example, disk, DECtape, magnetic tape, etc.

master file directory   The system-maintained file on a volume that contains the names and addresses of all the files stored on the volume.

memory   Any form of data storage, including main memory and mass storage, in which data can be read and written. In the strict sense, memory refers to main memory.

memory image   A replication of the contents of a portion of memory.

memory mapping   A mode of computer operation in which the high-order bits of a virtual address are replaced by an alternate value, providing dynamic relocatability of programs.

memory protection   A scheme for preventing read and/or write access to certain areas of memory.

modulo   A mathematical operation that yields the remainder function of division. Thus 39 modulo 6 equals 3.

monitor   The master control program that observes, supervises, controls or verifies the operation of a computer system. The collection of routines that controls the operation of user and system programs, schedules operations, allocates resources, performs I/O, etc.

monitor command   An instruction issued directly to a monitor from a user.

monitor console   The system control terminal.

Monitor Console Routine (MCR)   The executive routine that allows the user to communicate with the system using an on-line terminal device. MCR accepts and interprets commands typed on the terminal keyboard and calls appropriate routines to execute the specified requests.

mount a device or volume   To logically associate a physical mass storage media with a physical device unit. To place a volume on a physical mass storage drive unit; for example, place a DECtape on a DECtape drive and put the drive on-line.

multiprocessing   Simultaneous execution of two or more programs by two or more processors.

multiprogramming   A processing method in which more than one task is in an executable state at any one time.

naked syntax   A feature of the MUMPS language, providing an abbreviated method for accessing global variables, which controls the disk access time. The node reference includes only subscript(s) for the element; the global variable name is assumed from the last global reference in which a name was explicitly stated.

node   A dynamically allocated set of bytes from a node pool used for system communication and control in an RSX-11/IAS system. An element of a global array in a MUMPS-11 system (also called a global variable).

non-contiguous file   A file whose blocks are not physically contiguous on the volume.

non-file structured device   A device, such as paper tape, line printer or terminal, in which data is not referenced as a file.

object code   Relocatable machine language code.

object module   The primary output of an assembler or compiler, which can be linked with other object modules and loaded into memory as a runnable program. The object module is composed of the relocatable machine language code, relocation information, and the corresponding symbol table defining the use of symbols within the module.

object program   The relocatable binary program which is the output of a compiler or assembler.

Object Time System   The collection of modules that is called by compiled code in order to perform various utility or supervisory operations. For example, an Object Time System usually includes I/O and trap handling routines.

off-line   Pertaining to equipment or devices not under direct control of the Central Processing Unit.

offset   The difference between a base location and the location of an element related to the base location. The number of locations relative to the base of an array, string or block.

on-line   Pertaining to equipment or devices directly connected and under control of the Central Processing Unit.

operating system   The collection of programs, including a monitor or executive and system programs, that organizes a central processor and peripheral devices into a working unit for the development and execution of application programs.

overlay description language   The set of instructions interpreted by a linker that defines the overlay structure of a task.

overlay segment   A section of code treated as a unit which can overlay code already in memory and be overlaid by other overlay segments.

overlay structure   A task overlay system consisting of a root segment and optionally one or more overlay segments.

p-section   A section of memory that is a unit of the total task allocation. A source program is translated into object modules that consist of p-sections (program sections) with attributes describing access, allocation, relocatability, etc.

pack   To compress data in storage by using an algorithm for its storage and retrieval. A removable disk.

parity bit   A binary digit appended to a group of bits to make the sum of all the bits always odd (odd parity) or always even (even parity). Used to verify data storage.

parse   To break a command string into its elemental components for the purpose of interpretation.

part number   In the MUMPS language, the integer portion of a program step which is used to refer collectively to all steps having a common integer base.

partition   A contiguous area of memory within which tasks are loaded and executed.

patch   To modify a program by changing the binary code rather than the source code.

peripheral   Any device distinct from the central processor which can provide input or accept output from the computer.

physical address space   The set of memory locations where information can actually be stored for program execution. Virtual memory addresses can be mapped, relocated or translated to produce a final memory address which is sent to hardware memory units. The final memory address is the physical address.

physical block   A physical record on a mass storage device.

physical device   An I/O or peripheral storage device connected to or associated with a central processor.

physical record   The largest unit of data that the read/write hardware of an I/O device can transmit or receive in a single I/O operation. The length of a physical record is device dependent. For example, a punched card can be considered the physical record for a card reader; it is 80 bytes long. The physical record for an RK11 disk is a block; it is 512 bytes long.

position independent code   Code which can execute properly wherever it is loaded in memory, without modification or relinking. Generally, this code uses addressing modes which form an effective memory address relative to the central processor's Program Counter (PC).

priority   A number associated with a task that determines the preference its requests for service receive from the executive, relative to other tasks requesting service.

privilege   A characteristic of a user or program that determines what kinds of operations that user or program can perform. In general, a privileged user or program is allowed to perform operations normally

considered to be the domain of the monitor or executive, or which can affect system operation as a whole.

program development　The process of writing, entering, translating, and debugging source programs.

programmed requests　An instruction available only to programs that is used to invoke a monitor service.

programmer access code　The system identification code that enables a user to gain access to a MUMPS-11 system in direct mode to create, modify and execute programs.

project-programmer number　See **account number**.

pseudo device　A logical entity treated as an I/O device by the user or the system, but which is not actually any particular physical device.

public disk structure　The disk volume or set of volumes which are used as a general storage pool available to any users having quotas on the public structure.

pure code　Code that is never modified during execution. It is possible to let many users share the same copy of a program that is written as pure code.

qualifier　A parameter specified in a command string that modifies some other parameter. See **switch**.

queue　Any list of items; for example, items waiting to be scheduled or processed according to system or user assigned priorities.

Radix-50　A storage format in which three ASCII characters are packed into a 16-bit word.

random access　Access to data in which the next location from which data is to be obtained is not dependent on the location of the previously obtained data.

real-time processing　Computation performed while a related or controlled physical activity is occurring so that the results of the computation can be used in guiding the process.

record　A collection of adjacent data items treated as a unit. See **logical record** and **physical record**.

record gap　An area between two consecutive records.

recursive　Pertaining to a process that is inherently repetitive. The result of each repetition of the process is usually dependent on the result of the previous repetition.

reentrant　The property of a program that enables it to be interrupted at any point by another program, and then resumed from the point where it was interrupted.

resident   Pertaining to data or instructions that are normally permanently located in main memory.

restart address   The address at which a program can be restarted. It is normally the address of the code required to initialize variables, counters, etc.

root segment   The segment of an overlay tree that, once loaded, remains resident in memory during the execution of a task.

RUNOFF   A program that is used to prepare printed documents by performing formatting, case conversion, line justification, page numbering, titling, and indexing.

secondary storage   Mass storage other than main memory.

sentinel file   The last file on a cassette tape which represents the logical end-of-tape.

sequential access   A data access method in which records or files are read one after another in the order in which they appear in the file or volume.

sharable program   A (reentrant) program that can be used by several users at the same time.

significant event   An event or condition which indicates a change in system status in an event-driven system. A significant event is declared, for example, when an I/O operation completes. A declaration of a significant event indicates that the executive should review the eligibility of task execution, since the event might unblock the execution of a higher priority task. The following are considered to be significant events: I/O queuing, I/O request completion, a task request, a scheduled task execution, a mark time expiration, a task exit.

single user access   The status of a volume that allows only one user to access the file structure of a volume.

single-stream batch   A method of batch processing in which only one stream of batch commands is processed.

sliver   A 32-word section of memory.

source language   The system of symbols and syntax, easily understood by people, which is used to describe a procedure that a computer can execute.

sparse array   Refers to the method of storage allocation used in MUMPS-11 for local and global arrays in which space is allocated only as variables are explicitly defined (unlike other languages which require dimension or size statements for preallocation of storage).

spooling   The technique by which output to low-speed devices is placed into queues on faster devices to await transmission to the slower devices.

step number   A number in the range 0.01 to 327.67 used to identify each line of a MUMPS program.

subscript   A numeric valued expression or expression element which is appended to a variable name to uniquely identify specific elements of an array. Subscripts are enclosed in parentheses. Multiple subscripts must be separated by commas. For example, a two-level subscript might be (2,5).

swapping   The process of copying areas of memory to mass storage and back in order to use the memory for two or more purposes. Data is swapped out when a copy of the data in memory is placed on a mass storage device; data is swapped in when a copy on a mass storage device is loaded in memory.

swapping device   A mass storage device especially suited for swapping because of its fast transfer rate.

switch   An element of a command or command string that enables the user to choose among several options associated with the command. In PDP-11 software systems, a switch element consists of a slash character (/) followed by the switch name and, optionally, a colon and a parameter. For example, a command used to print three copies of a file on the line printer could be: "PRINT filename/COPIES: 3".

synchronous   The performance of a sequence of operations controlled by an external clocking device. Implies that no operation can take place until the previous operation is complete.

synchronous system trap   A system condition which occurs as a result of an error or fault within the executing task.

system device   The device on which the operating system is stored.

system generation   The process of building an operating system on or for a particular hardware configuration with software configuration modifications.

system manager   The person at a computer installation responsible for the overall nature of its operation.

system operator   See **operator**.

system program   A program that performs system-level functions. Any program that is part of the basic operating system. A system utility program.

system programmer   A person who designs and codes the programs that control the basic operations of a computer system, as opposed to an application program.

system UCI   The User Class Identifier (UCI) code in a MUMPS-11 system which is assigned to the first entry in the system's UCI table. The program and global directories associated with the System UCI are used to contain both system and library programs and globals.

task   In RSX-11 terminology, a load module with special characteristics. In general, any discrete operation performed by a program.

terminal   An I/O device, such as an LA36 terminal, which includes a keyboard and a display mechanism. In PDP-11 systems, a terminal is used as the primary communication device between a computer system and a person.

time slice   The period of time allocated by the operating system to process a particular program.

TRAX   A dedicated high-volume transaction processing operating system.

transaction   A single pre-defined data processing operation within an application.

transaction processor   A collection of data tables and software capable of processing an application's transactions.

staging   The delay of each update to a file until the end of the transaction instance requesting the update.

unformatted ASCII   A mode of data transfer in which the low-order seven bits of each byte are transferred. No special formatting of the data occurs or is recognized.

unformatted binary   A mode of data transfer in which all bits of a byte are transferred without regard to their contents.

unmapped system   An RSX-11M or RSX-11S system that does not have a hardware memory management unit available for virtual address relocation.

user class identifier   An identification code that enables a user to gain access to a MUMPS-11 system to execute programs.

user identification code   The number or set of numbers that serves to distinguish a particular user or collection of files in a multi-user system. The common format for a user identification code is two numbers separated by a comma, enclosed in brackets.

user program   An application program.

utility   Any general-purpose program included in an operating system to perform common functions.

variable   The symbolic representation of a logical storage location which can contain a value that changes during a discrete processing operation.

virtual address space   A set of memory addresses that is mapped into physical memory addresses by the paging or relocation hardware when a program is executed.

virtual array   A RSTS/E file structure that is logically organized as a dimensioned array.

virtual block   One of a collection of blocks comprising a file (or the memory image of that file). The block is virtual only in that its block number refers to its position relative to other blocks within the file, instead of to its position relative to other blocks on the volume. That is, the virtual blocks of a file are numbered sequentially beginning with one, while their corresponding logical block numbers can be any random list of valid volume-relative block numbers.

volume   A mass storage media that can be treated as file-structured data storage.

word   Sixteen binary digits treated as a unit in PDP-11 processor memory.

zero a device   To erase all the data stored on a volume and re-initialize the format of the volume.

INDEX

MUMPS special, 223,224

Variable-with-fixed-control (VCF) records   293,294

VERIFY program   42

Version number
file specification, 25

VFY (File Verification)   164

VIA record locaton mode   309

Video terminal graphics
FOCAL, 505

Virtual array files   408

Virtual arrays   111,112

Virtual blocks   29,154,155

Volume
description, 15
logical, 16

VT11 graphics display system   373,505

VT52 DECscope terminal
use with DECFORM, 444

VT62 application terminal   233 to 237

VT50H DECscope terminal
use with DECFORM, 444

Warning diagnostics
FORTRAN IV, 464

Wildcard   26

Workspaces
APL, 497

Write protection
see Protection

XDT (Executive Debugging Tool)   163

ZAP (Task Patch) utility   163

Z commands   218 to 221

Z function   222,223

Z special variables   224,225