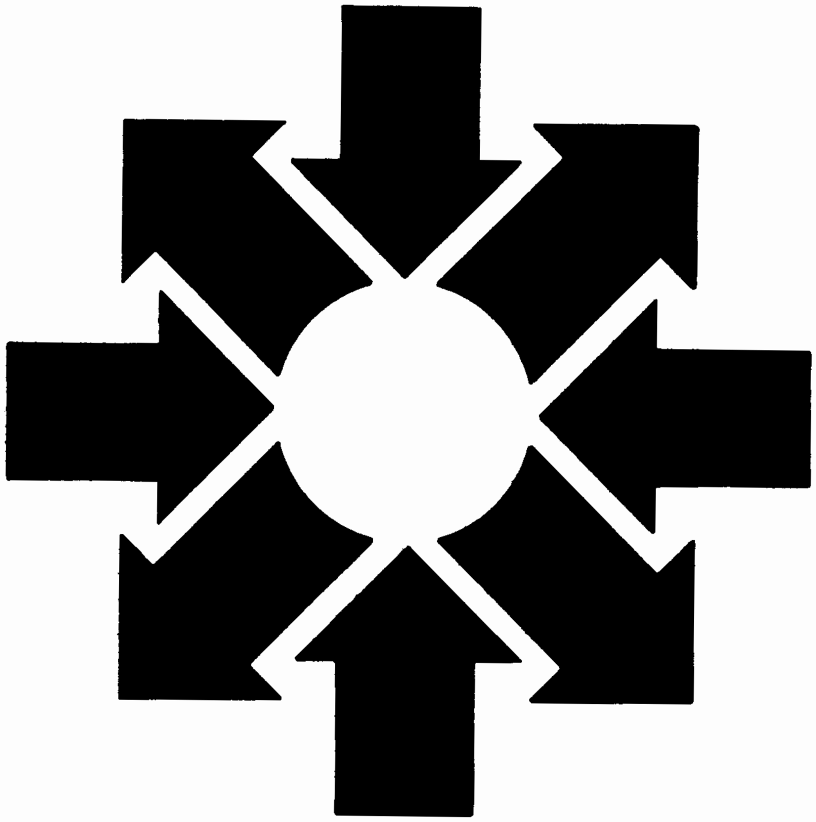
PREFACE TO PROGRAMMING



HEWLETT-PACKARD
11000 WOLFE ROAD
CUPERTINO, CALIFORNIA
95014

HP ORDER NUMBER 5951-1354
FEBRUARY, 1971

CONTENTS

INTRODUCTION

Hewlett-Packard offers a wide range of software--operating systems, programming languages, and user aids--for its 2100 family of computers. *Preface to Programming* introduces you to this software line, pointing out the similarities and differences of software products. It is designed to help you to select the most useful software for your needs.

Because applications do not all have the same requirements and no computer system can handle all needs equally well, Hewlett-Packard has many software operating environments. Each software product or system optimizes certain features and is best for a certain type of application.

Although "software" merely means "programs" and there are many types of programs, for the purposes of simplicity we will divide software into two classes:

1. Software that aids in the preparation of programs (compilers, assemblers, editors, debugging programs)

and

2. Software that controls the operation of the computer and its peripheral devices (executive monitors, program loaders and schedulers, I/O monitors and drivers).

Preface to Programming covers the following software:

Translators and User Aids

Assemblers
HP FORTRAN Compiler
HP ALGOL Compiler
HP FORTRAN IV Compiler
HP BASIC Interpreter
Educational BASIC System
Time-Shared BASIC System
Symbolic Editors
Library Subroutines

Operating Systems and Generators

SIO Dump (for SIO System)
Prepare Control System (PCS for Basic Control System)
Prepare Tape System (PTS for Magnetic Tape System)
Real-Time Generator (RTGEN for RTE System)
Disc Generator (DSGEN for Disc Operating System)
Prepare BASIC System (PBS for the BASIC Interpreter)

Since many computer applications developed by HP are based on these software products, this book is a useful introduction to software for all HP computer users. All computers of HP's 2100 family are software compatible--they have the same word size and instruction set--and vary only in memory speed, front panel, and options available. Therefore, this book treats them as equivalent. You should read the manuals mentioned within this text to determine the exact hardware requirements for each software product.

SOME GENERAL PRINCIPLES

Before you select a programming language, operating system and computer for an application, you should be familiar with some general principles. A computer system must be able to perform many activities, but some of these activities can be handled either manually or automatically. You can measure the power of a computer and its software by the extent to which the computer assumes responsibility for the total system.

But the most powerful system is not always your best choice. For every feature, convenience, or ability of a software system, you must pay a price. There are necessary trade-offs between cost and user convenience. Every increase in convenience must be balanced against its cost. Analogous situations exist in every human activity. For example, food can be packaged in large containers that are low in cost per ounce or in small containers that cost more per ounce. The difference in cost per ounce is the price you pay for the convenience of having smaller servings.

Computer memory is one resource. When the software takes over functions, the software programs must be larger. When the system software uses more memory, that leaves less memory for your programs. Thus, you will have to buy a computer with a larger memory in order to get a reasonable program area.

Speed is another computer resource. When a computer does housekeeping for the user, it must spend execution time. Thus, if the software provides more functions, it must slow down in other things. In an application where speed (such as speed of response to interrupt) is important, convenient general features of software may have to be foregone in favor of maximum efficiency. For example, if the software examines the input for special control characters or commands, it takes longer to input data than if the software just transfers data from the input device to your program buffer.

The final cost is always money -- money for faster memories to increase speed, money for more memory to allow larger user programs, and money for additional hardware for special purposes. Certain functions can only be accomplished with hardware options. For example, system integrity can only be ensured with a hardware memory protect option and I/O throughout can be increased beyond a certain point by adding the Direct Memory Access option.

Thus, it is important that you balance the value of each software system against alternative uses of the same hardware that are possibly less convenient. In order to see these principles in action, we shall trace through a logical progression of increasingly more complex HP software products in two classes: programming languages and operating systems. At each level you will be able to see what features the software product provides at what cost, and what might be desired in a more complex and expensive product.

CHOOSING A PROGRAMMING LANGUAGE

A program is a set of instructions for a computer that perform a certain task and, although it is possible to write a program directly in the computer's own "language" (binary machine codes such as "0110011010001101"), it is a difficult and time-consuming process. For this reason, various artificial programming languages have been developed that allow the programmer to write in a more natural format. The computer itself is used to translate the programmer's "source" program into a binary "object" program that the computer can execute.

TWO TYPES OF PROGRAMMING LANGUAGES

These artificial programming languages fall into two categories: assembly languages and high-level languages.

Assembly languages follow the structure of the machine language very closely. That is, each "sentence" (statement, line, or instruction) of the program consists of a machine operation and an operand address. The machine operations, however, are specified by easy-to-remember mnemonics (for example, ADD instead of 0111) and the addresses are specified by symbolic names (for example, NAME instead of 00100).

High-level languages have a structure that is farther removed from the machine and closer to the structure of human discourse. High-level languages use English phrases and words to represent common program structures and mathematical symbols to express computations and relations. For example, "IF A<B THEN A←B+1;" is a statement in the language ALGOL.

In programming languages, as in all software, there are trade-offs in design between efficiency and convenience. Hewlett-Packard recognizes this and provides many languages to allow the user flexibility. Since it is possible to separate a problem solution into parts, write each part in a different language (i.e., some in assembly language and some in high-level languages), and then link the parts together, the user can find a combination of languages perfect for his situation.

ASSEMBLY LANGUAGE

Languages vary in how closely they match the structure of the machine language. Assembly Language does little more than substitute mnemonics (memory aids) for the numerical instruction codes and allow storage locations to be referenced by symbolic names. There is still a one-to-one correspondence between each assembly language instruction and the corresponding machine language instruction.

Assembly language programming requires translation before the program is accepted by the computer for execution. The program that does the translation is called an "assembler." The assembler program recognizes the mnemonics and substitutes the correct machine codes. It also keeps a table of all symbols and assigns them to binary storage locations. In this way the assembler is able to translate assembly language into machine language.

For example, the following three commands are identical:

1. English: Load the contents of memory location 605 into the A Register

2. Machine Code: 0011000110000101

3. Assembly language: LDA X (where X is the name of location 605)

Assembly Language is used to write most software provided with the computer (i.e., the assembler itself is written in assembly language). This is because assembly language produces the most efficient code (i.e., uses the least number of locations in memory to solve a problem), an important asset in small computers. It also gives the most detailed control of the hardware operation; this is necessary for handling input/output and other operations.

Compared to machine language, assembly language reduces the time needed to write programs; it is easier to learn than machine language, eliminates the need for keeping track of memory locations in such detail, makes programs easier to read and debug, and allows tasks to be divided among several programmers. Also, assembly language allows remarks and comments to be included in the program for documentation purposes. Programs are easier to modify because the assembler assigns memory addresses automatically.

Hewlett-Packard Assemblers

Hewlett-Packard provides two assemblers. The Extended Assembler has mnemonics for all legal machine codes, allows symbolic addressing, and has a series of additional instructions to make the assembler more usable (listing control and other conveniences). There are versions of the Extended Assembler for all operating environments with at least 8,192 words of memory.

For computers with only 4,096 words of memory, HP provides the Assembler. This program is smaller than the Extended Assembler and translates a subset of Extended Assembly Language. It allows all machine codes and symbolic addressing, but does not have as many conveniences as the Extended Assembler. Program assembly produces a set of binary machine instructions stored on a medium external to the computer (such as paper tape). This binary program must be loaded into the computer before it can be executed.

ASSEMBLY LANGUAGE

Assembly Language is processed by the Assembler and Extended Assembler and converted into binary object code in either absolute or relocatable form. Source listing, symbol table listing, and binary tape output are available.

Assembly Language is completely described in the *HP ASSEMBLER* manual (HP 02116-9014).

Disadvantages of Assembly Language

Assembly language is easier to write and read than binary code, but it is still removed from common methods of stating problems. Assembly language requires an intimate knowledge of the hardware because the statements of the language have a one-to-one relationship with machine instructions. Coding is detailed and laborious; requiring many instructions to accomplish anything, compared with algebraic notation. Therefore, it takes a long time to program a problem because the solution must be broken down into simple operations by the programmer. These intricate operations cause programmers to make errors. Because simple structures are repeated over and over, much unnecessary effort is exerted.

In addition, programs written in assembly language are machine-dependent; they are not transferable in any way to another computer. (Or stated another way, they are not portable.) If a user switches computer vendors, he has to rewrite all of his assembly language programs.

These problems provide the motivation for high-level languages that:

1. Develop a structure closer to normal speech and mathematical usage; and

2. Develop a standardized language structure so that programs can be transferred from one computer to another by merely retranslating the program into the new machine code.

HIGH-LEVEL LANGUAGES

Languages that allow man to communicate with computers in an English-like or mathematical manner are called high-level languages. Programs at this level consist of simple statements that define complex computer operations. Statements such as READ, WRITE, GO TO, IF, DO etc. are typical; each is translated into a series of machine instructions. Therefore, high-level languages are much easier to learn than assembly languages. High-level languages are generally machine independent, allowing the same program to be compiled and executed on different computers. The programmer does not need a detailed understanding of the computer. High-level languages reduce programming effort; a single statement generates many machine language instructions. Programs are easy to read, are somewhat self-documenting, and are easy to modify.

Since each programming structure in a high-level language handles many specific cases, each structure is somewhat inefficient in the code used to solve a specific situation. That is, any structure could, with sufficient time, usually be programmed in assembly language using fewer memory locations than the equivalent structure in a high-level language. However, the extra time required to code in assembly language is usually not worth the small savings in core or execution time.

Certain problems cannot be written in a high-level language because such a language isolates the programmer from the machine. An example is an I/O device control program; a program of this type must have access to detailed machine instructions available only in a machine-dependent assembly language.

Because the high-level language is not the cure-all for every situation, the ability to combine high-level programs with assembly language programs is an important feature of HP software. It allows the programmer to find the best compromise of high-level convenience and assembler economy and control when solving complex problems on a small computer.

Like assembly languages, high-level language programs must be processed before they can run on the computer. There are two different approaches to this problem.

One follows the same line as assemblers and translates the program into machine codes that are then loaded into the computer and run. This process is called *compilation*, and the programs that compile are called compilers.

The other approach is called *interpretation*. Programs are converted into a series of instructions in an intermediate code that is between the original program and the machine code. This intermediate code is then executed by the interpreter, not the computer directly. Although this process is inefficient in terms of execution speed and memory used, it has advantages in many applications.

Compilers

A compiler is a program that compiles a particular high-level language. Hewlett-Packard supplies compilers for several languages in several operating environments. Since the development of compilers is a complicated and expensive process, the languages should be as well-organized and easy to learn as possible; programmers should be able to write any type of program they need. In addition, the language should have an internationally accepted standard definition of its format. For small computers, two languages meet these standards:

FORTRAN	(FORmula TRANslator)
ALGOL	(ALGOrithmic Language.)

Most computer manufacturers provide compilers for at least one of these two languages. Therefore, the user has a degree of portability if he writes in these languages; he can transfer to another HP computer system with minimal program change. (Although these languages have standard definitions, every manufacturer's compiler has certain limitations and extensions that make them slightly incompatible. Therefore, small changes to programs are sometimes necessary.)

FORTRAN

The most widely used standard language in science and engineering is FORTRAN. There are probably more compilers for FORTRAN than for any other language. Because FORTRAN has been used over fifteen years, FORTRAN programs have already been written to solve many general problems in engineering, mathematics, and science. Thus, FORTRAN is a desirable language to implement on any computer. A user's FORTRAN program can be converted to a Hewlett-Packard computer with minor changes. In this way the user's program development effort may be eased.

FORTRAN is much easier to program in than assembly language because its form is closer to the language of science and mathematics. The grammer, symbols, rules, and syntax used in FORTRAN are familiar to the general public. For example, the symbols, +, -, /, and * indicate addition, subtraction, division, and multiplication respectively.

A typical statement in FORTRAN is: X = (Y + B/C)/2. This straightforward arithmetic formula would require many statements in assembly language; and in fact, the FORTRAN compiler translates this statement into a number of machine instructions.

Languages vary in the degree of complexity and power built into them. FORTRAN allows the programmer to construct loops (repetitions), conditional jumps to another statement or subprogram, and arithmetic formulas. However, complex and intricate logical relationships, such as "if x>100 and y<100 then do exercises, else if G=10 then do recreations for K times," must still be broken down into independent FORTRAN statements. The structure of FORTRAN does not show the relationships involved, it hides them. In addition, FORTRAN has limitations on the mixing of different types of numbers in formulas.

```
FTN,L,B
C       THIS PROGRAM FINDS THE E.T.E. FOR A
C       GIVEN VELOCITY AND DISTANCE.
          PROGRAM DTET1
          WRITE (2,2ØØ)
1Ø        WRITE (2,3ØØ)
          READ (1,4ØØ)VEL
          IF (VEL)9999,9999,2Ø
2Ø        READ(1,4ØØ)DST
          ETE=DST/VEL
          ETEH=IFIX(ETE)
          ITEH=ETEH
          ITEM=IFIX((ETE-ETEH)*6Ø.)
          WRITE(6,6ØØ)VEL,DST
          WRITE(6,7ØØ)ITEH,ITEM
          GO TO 1Ø
9999      WRITE(6,8ØØ)
2ØØ       FORMAT("THIS PROGRAM DETERMINES E.T.E.")
3ØØ       FORMAT("TYPE, IN TWO LINES, VALUES FOR VELOCITY
         1, THEN DISTANCE, IN F4.2 FORMAT.")
4ØØ       FORMAT(F4.2)
6ØØ       FORMAT(2/"-VELOCITY=",F8.4," DISTANCE =",F8.4)
7ØØ       FORMAT(/"-ESTIMATED TIME ENROUTE=",13,"HOURS,",13,
         1" MINUTES."/)
          STOP
8ØØ       FORMAT(6/"-DONE"/"1")
          END
          END$
```

A Typical Small FORTRAN Program

Hewlett-Packard supports two levels of FORTRAN: a language called HP FORTRAN which is based on standard Basic FORTRAN and a more powerful version called HP FORTRAN IV. FORTRAN IV allows the user to manipulate extra-precise numbers and complex numbers and provides additional structures, such as logical operations, not found in Basic FORTRAN.

Hewlett Packard provides an HP FORTRAN compiler for all computer installations with a minimum of 4,096 words of memory and a teleprinter. The FORTRAN IV compiler requires a larger core memory and a disc storage unit.

HP FORTRAN

HP FORTRAN programs are processed by various FORTRAN compilers and converted into binary object code in a relocatable form. This means that the code must be passed through a relocating loader before it can be executed.

The Hewlett-Packard Basic FORTRAN Language is documented in the *HP FORTRAN* programmer's reference manual (HP 02116-9015).

HP FORTRAN IV

HP FORTRAN IV programs are processed by the HP FORTRAN IV Compiler, a program that operates under control of either DOS or RTE. The relocatable object code that is produced can be run under DOS, RTE, and BCS.

The Hewlett-Packard implementation of FORTRAN IV is completely specified in the programmer's reference manual, *FORTRAN IV* (HP 05951-1321)

ALGOL

The need for a more general, flexible, and powerful language than FORTRAN motivated the development of ALGOL (the ALGOrithmic Language). ALGOL defines certain basic language elements such as variables, constants, expressions, and procedures, from which the user can build more complex structures. Almost every level of structure can be nested inside another structure. The result is a general framework for expressing any logical or mathematical process similar to the way you would describe it in English.

One of the goals of ALGOL was to make programs closer to the natural way of stating problems, so that a person could see the logic of a program without having written it. This form of program can follow the logic of the problem: dependent operations can be nested within the conditions upon which they depend. Variable names can describe the information they contain. Statements can be formatted on the page in any manner the programmer finds useful. These features make ALGOL programs almost self-documenting.

This excerpt from an imaginary ALGOL program shows what you can do.

```
COMMENT THIS PROGRAM EATS ALL THE FRUIT;
IF NUMBEROFAPPLES<NUMBEROFORANGES THEN
    BEGIN
        A←NUMBEROFAPPLES + NUMBEROFORANGES;
        FOR N←1 STEP 1 UNTIL A
            DO EAT(N);
        IF NUMBEROFPEACHES<A THEN
    FOR X←1 STEP 1 UNTIL NUMBEROFPEACHES
            DO EAT(X);
    END;
```

Most algorithms (i.e., procedures for solving problems) that appear in the computer science journals today are given in ALGOL. Thus, ALGOL is a widely used language for the statement of methods underlying programs. It is easier to convert the ALGOL description of a method into FORTRAN than it is to decipher the logic behind a FORTRAN program and convert it to ALGOL.

Hewlett-Packard provides ALGOL compilers for all computer installations having at least 8,192 words of memory.

ALGOL

ALGOL compilers for both core- and disc-based operating systems accept HP ALGOL programs and produce relocatable object code.

The Hewlett-Packard version of ALGOL is explained in *HP ALGOL* (HP 02116-9072), a programmer's reference manual.

Interpreters

In FORTRAN and ALGOL, the program must be compiled and loaded before execution. If a change is made, the program must be recompiled and reloaded to see if the change is effective. During development, the program may be modified dozens of times; if the programmer has to wait ten minutes to a day everytime he recompiles, he is obviously wasting a tremendous amount of time.

An alternative to compilation and loading is interpretation. The interpreter program is a self-contained system; it always remains in core memory; it handles complete editing, translating, and executing of programs so no other systems are necessary.

The Interpreter interacts with the user conversationally to develop his program. The user types in the statements of his program in any order; if he finds he needs another step, he simply types it in. The Interpreter arranges all statements in numerical order. To change a specific statement, the user simply types in the new statement, and the Interpreter automatically effects the replacement.

BASIC

BASIC – the Beginner's All-Purpose Symbolic Instruction Code – is a simple language designed specifically for interpretation instead of compilation.

The BASIC language is defined by a very simple set of rules. It is easy for even beginners to learn, yet powerful enough for advanced users. The Interpreter program reacts instantly to each line of typing so the user knows immediately what error he has made.

Here is a simple program in BASIC that finds the average of five numbers typed in by the user:

```
100 INPUT A,B,C,D,E

200 LET X = (A+B+C+D+E)/5

300 PRINT X

400 END
```

A Sample BASIC Program

Hewlett-Packard provides several systems built around BASIC Interpreters. The stand-alone BASIC Interpreter supports one teleprinter at a time on a computer with at least 8,096 words of memory. A similar system–Educational BASIC–allows BASIC programs to be translated from marked cards. The Time-Shared BASIC Systems provide an extended version of the BASIC language to 16 or 32 users simultaneously. The extensions to BASIC allow the user to store and access large amounts of data in an external mass memory, to manipulate strings of characters, and to store and retrieve programs in mass memory.

SINGLE-TERMINAL BASIC

The Single-Terminal BASIC System allows you to prepare and run BASIC Language programs conversationally through a terminal. Programs can also be entered through tape readers and punched out on tape punches.

The BASIC Language accepted by the Interpreter is described in *HP BASIC* (HP 02116-9077), a reference and self-instruction manual.

EDUCATIONAL BASIC

The Educational BASIC System is similar to the Single-Terminal BASIC System, except that it also allows programs to be entered on marked cards.

The version of BASIC used in the Educational BASIC System is covered in *A GUIDE TO HP EDUCATIONAL BASIC* (HP 02116-91773), a reference and instructional source book.

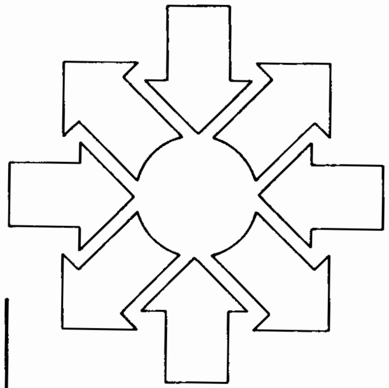
TIME-SHARED BASIC LANGUAGES

The BASIC Language used in time-sharing systems is available to 16 or 32 user terminals simultaneously.

BASIC does have certain drawbacks, however. For example, the conversational editing and running of programs is based on the interpretation process. Because code is not executed directly by the computer, execution is much slower; however, for small programs, the difference in time is not noticeable to the user. Because the Interpreter is designed around an interactive user terminal, there are no input/output capabilities other than those provided on the terminal (usually paper tape reader and punch).

In summary, if the user has many problems for which he needs solutions quickly, BASIC is the most convenient and easy language.

LANGUAGE FEATURES CHART



	ASSEMBLY LANG.	FORTRAN	FORTRAN IV	ALGOL	BASIC
Most efficient in use of memory and execution time	X				
Most convenient for writing programs					X
Most widely used			X		
Most powerful structure				X	
Machine independent		X	X	X	X
Machine dependent	X				
Readable (self-documenting)		X	X	X	X
Conversational editing					X
Mnemonic instruction codes	X				
Symbols	X	X	X	X	X
Arithmetic expressions (Math formulas)		X	X	X	X
General nesting				X	
Integer arithmetic	X	X	X	X	
Floating point arithmetic	X	X	X	X	X
Double precision and complex arithmetic	X		X		
Type mixing in expressions			X	X	
Generates absolute code	X				
Generates relocatable code	X	X	X	X	
All machine instructions possible	X				
Compilation		X	X	X	
Assembly	X				
Interpretation					X

CHOOSING AN OPERATING SYSTEM

An operating system is an organized collection of programs which increases the productivity of a computer by providing common functions for all user programs. Depending on its complexity, an operating system can be responsible for any or all of the following functions:

1. Prepare programs in one or more programming languages.
2. Load programs into the computer (including software programs such as compilers).
3. Execute programs.
4. Store and retrieve programs.
5. Store and retrieve data.
6. Control input/output devices.
7. Control or monitor program execution.
8. Perform service functions for executing programs.

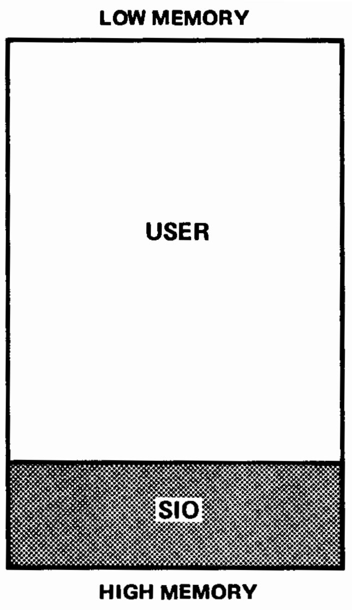
Hewlett-Packard offers a variety of software operating systems for its 2100 family of computers; these systems vary in the functions they automate and in the generality of their structure.

A general-purpose operating system provides the framework and resources that the user can tailor to meet his specific needs. The Hewlett-Packard general operating systems, listed in order of increasing automation are: Software Input/Output System, Basic Control System, Magnetic Tape System, Real-Time Executive System, and Disc Operating System.

A special-purpose operating system is a dedicated system that requires the full attention of the computer for one task. Hewlett-Packard supplies three dedicated operating systems: stand-alone BASIC, Educational BASIC, and Time-Shared BASIC.

SIO SYSTEM

Of the general-purpose operating systems, the SIO (Software Input/Output) System is by far the simplest in structure and function. Each SIO system consists of a series of programs that reside in fixed memory locations near the top of core. There is a separate program, called an SIO driver, for each type of I/O controller connected to the computer. Examples include the SIO Teleprinter Driver, the SIO Tape Reader Driver, etc. There exists a version of each driver for each core size (i.e., 4K SIO Teleprinter Driver, 8K SIO Teleprinter Driver, etc.). The drivers are integrated into an operating system by a program called SIO System Dump.



SIO CORE MAP
(Minimum - 4K)

The SIO System set of drivers occupies a relatively small area of memory; the rest of core is available to assembler-produced absolute programs. Since SIO was specifically designed for use with certain HP software programs (compilers, assemblers, etc), it uses the least amount of memory possible while providing the minimum I/O capabilities needed by the software. To see the results, we will review the basic functions of an operating system viz SIO.

Preparing Programs in SIO

Software products that operate in the SIO environment and facilitate programming include: compilers for FORTRAN and ALGOL, an Assembler and Extended Assembler, a Cross-Reference Symbol Table Generator, and a Symbolic Editor.

The FORTRAN compilers (one for computers having only 4K of memory, another for computers with 8K or more) accept programs written in a version of FORTRAN as source input and produce relocatable machine code on output tapes.

The ALGOL compiler (requiring 8K memory) compiles source programs written in ALGOL into relocatable object tapes.

The Assembler (for 4K and Extended for 8K or more) accepts assembly language source programs and produces either absolute *or* relocatable machine code output.

The Cross-Reference Symbol Table Generator program searches through assembly language source programs and creates a table that shows where each symbol is defined and referenced.

The Symbolic Editor has two inputs: a source file and a list of editing operations. It performs the requested edits on the source file and produces an updated file as output.

Loading Programs in SIO

SIO can be used to produce both relocatable and absolute object code, but the Basic Control System is used to execute relocatable programs.

Only assembler-produced absolute programs can be loaded into the computer along with an SIO System. They are loaded by a 64-word program, the Basic Binary Loader, that always resides in a protected area of core.

Loading of absolute programs with the Basic Binary Loader (BBL) is a man-machine operation, partly manual and partly automatic. The operator must place each tape in the reader and start the BBL program; the program reads each frame of the tape and loads the information into the specified memory locations. For a complicated program with several parts (such as the various passes of the FORTRAN compiler), each part of the program must be loaded separately.

Once an absolute program has been loaded along with the SIO system, it is executed by manually setting the starting address and pressing RUN.

Storing and Retrieving Programs and Data in SIO

Programs and data can be stored on paper tape, magnetic tape, or disc in SIO if the necessary hardware is available.

Controlling I/O Devices in SIO

The software I/O structure of the SIO system is quite simple. As mentioned before, an absolute subprogram, called an SIO driver, exists for each controller connected to the computer. When an I/O device is needed, the program transfers control to the subprogram for the controller of that device. The driver subprogram carries out the specified operation on the device, and returns to the user program when the operation is complete.

Notice that the hardware interrupt system is not used and there is no centralized I/O coordination. This means that only one I/O operation on one device can occur at a time in the SIO system. While this may seem to be quite a limitation, because the hardware is designed to handle multiple simultaneous I/O operations, it is all the I/O capability required by the software programs (i.e., compilers, etc.) and uses only a small amount of core memory.

Monitoring Execution in SIO

The operator is responsible for monitoring execution of user programs in the SIO system. The system contains no internal restrictions on user programs and cannot protect itself from runaway programs. For example, user programs can destroy the SIO drivers themselves or go into endless loops. The operator must intervene manually when something goes wrong during execution of a program.

Service Functions in SIO

The SIO system of drivers performs no services for the user besides the execution of I/O operations.

Summary

In summary, the SIO system is a minimum system, not meant for complex situations. SIO has the advantage of using a minimal amount of memory. In applications such as single-device, one-shot program situations, SIO is a logical choice over systems offering additional services that would not be exercised but would take up valuable core space.

SOFTWARE INPUT/OUTPUT SYSTEM

The SIO System consists of SIO Drivers for many devices and a program for linking them into a system--SIO System Dump. The resulting system is designed to run absolute assembly language programs and operate one (of many) I/O device at a time.

BASIC CONTROL SYSTEM

The Basic Control System (BCS) provides many useful features not included in the SIO System:

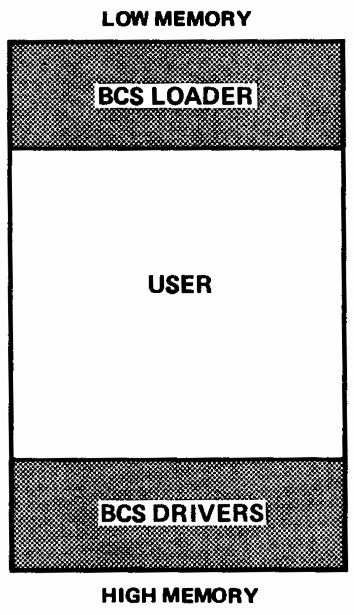
1. Relocation and linking of programs

2. I/O interrupt processing

3. Library subroutines

To perform these functions, BCS must, of necessity, be a more complex system than SIO, but it is less efficient in its use of core. BCS has a Relocating Loader that relocates and links user subprograms anywhere in core. This gives the user flexibility but loss of core (for the loader) and time (for the relocation process).

Because it provides interrupt processing, BCS has more complex I/O routines than SIO. For each type of device connected to the computer, there is a BCS driver subroutine that handles all I/O on devices of that type. A central routine, Input/Output Control (IOC), routes user requests to the appropriate driver. These various system programs are integrated into a customized operating system for a particular size computer and particular hardware by the Prepare Control System (PCS).



BCS CORE MAP
(Minimum = 4K)

Because it provides library subroutine services to user programs, BCS must be able to link main programs and subroutines during relocation. These libraries, which provide functions often required by many programs, complete the software of BCS. To see what capabilities this organization creates for the user, we will review the general functions of a computer system viz BCS.

Preparing Programs in BCS

The main reason that BCS and SIO are sister systems is that SIO is used to prepare programs for execution in the Basic Control System. BCS is oriented toward running programs; SIO is oriented toward preparing programs.

Loading Programs in BCS

The BCS Relocating Loader accepts relocatable program tapes produced by the assemblers and compilers. Instructions in a relocatable program are all relative to an arbitrary base address of 0. The Loader relocates the program relative to an actual base address (such as 2000_8) by adding the base to the address field of every memory reference instruction.

At the same time, the Loader keeps a table of all external subroutines requested by the user program; it scans the library files and loads in all the necessary subroutines. Hewlett-Packard provides several libraries for the BCS environment: a standard Relocatable Program Library which supplies mathematic functions (e.g. Sine), a Formatter, and Utility Routines; and a FORTRAN IV library which supplies arithmetic routines and a Formatter needed for double precision and complex numbers.

LIBRARY SUBROUTINES

Hewlett-Packard provides many subroutine libraries for its operating systems, both core- and disc-based. The subroutines provide common mathematic functions such as sine and cosine and extended precision arithmetic for FORTRAN IV programs.

The many subroutines provided by Hewlett-Packard for its relocatable operating systems are all documented in one manual, *RELOCATABLE SUBROUTINES* (HP 02116-91780).

Executing Programs in BCS

There are two ways to execute a program in BCS; both are manual. The first way is to simply press RUN after relocating the program into core with the Loader. This method is inefficient if the same program is run very often since it would have to be relocated every time. To avoid this, BCS provides an optional manner of relocation, whereby the program is punched onto paper tape as it is relocated. The parts of BCS required during execution (i.e., drivers, etc.) are included along with the program. This absolute tape can then be loaded into core with the Basic Binary Loader whenever the program is to be executed. Another advantage is that the program can use the space previously occupied by the loader, because the loader is not required at run time; i.e., larger programs can be loaded in this way.

Storing and Retrieving Programs and Data in BCS

Object programs can only be stored on paper tape. However, if a magnetic tape is available, it can be used to store relocatable subroutines (which will be loaded by the Loader) and data.

Controlling Input/Output in BCS

Besides relocatability, input/output is the strong point of BCS. Of all the HP operating systems, BCS gives the most efficient I/O service because it utilizes the interrupt system while minimizing supervisory overhead.

The center of the BCS I/O structure is the Input/Output Control subroutine (IOC). This routine routes all I/O requests to the appropriate driver. User programs call IOC either directly (assembly language only) or indirectly through the Formatter. IOC either passes the request on to the driver or, if the driver is busy, rejects the request and returns to the program.

When requests reach the driver, the driver initiates the operation and returns to the program. Whenever an interrupt occurs on the device, control is transferred automatically to the driver and back to the program when the interrupt has been processed. As a result, BCS can handle operations on many different type devices concurrently without delaying program execution.

All I/O operations in BCS are program request-oriented; that is, I/O operations are always initiated by user programs, not by external interrupts. BCS drivers handle only one operation at a time; during that time, they are not available to start a new operation. They will, however, handle operations on more than one device of the same type (though not at the same time) by configuring themselves to a particular I/O channel each time they are initiated.

BCS drivers handle all housekeeping associated with an I/O operation themselves. IOC does very little. Because many systems functions are thus deferred to the driver level, BCS drivers tend to use more core than SIO drivers.

Since the interrupt system is almost always enabled, interrupts receive almost instantaneous response from the computer. Drivers for synchronous devices (such as magnetic tape), however, turn off the interrupt system because they cannot afford to be interrupted or they may lose data. If two synchronous devices are running concurrently, one of them should circumvent the interrupt system by using Direct Memory Access.

Buffered IOC

One of the problems with BCS is that the user cannot start another operation on the same device until the previous operation is complete. Programs have to keep checking with IOC to find out when the driver is available. An option is available in BCS called "Buffered IOC" that provides buffering of all low/medium speed output devices.

The effect of buffering is that the user's output requests are never rejected. If the requested driver currently is busy, IOC stores the user's buffer in available memory until the driver is free. The user program does not have to keep track of output. Once he has made an output request, he can assume it will be done and can continue with his processing. As a result, all output devices are driven at close to their maximum speeds. The advantage of buffering is most remarked in low-speed devices, where the overhead cost of setting up the buffer is small compared to the actual time required for the transfer.

The effect of using buffered IOC is similar to dumping buckets of water into a sink. The water escapes through the drain at a fixed rate, lowering the level in the sink. When water is dumped in, the level in the sink rises. The sink is analagous to the buffer space available to buffered IOC, the drain is analagous to the output devices. Requests are processed on a first-in, first-out basis.

This scheme makes very efficient use of output resources. (It has no effect on input usage.) With buffered IOC, BCS also takes over responsibility of memory management. A library routine keeps track of all memory and allocates it to IOC upon request for buffer space. However, the cost of buffering is a larger IOC routine, and a memory management subroutine.

Controlling Program Execution in BCS

BCS monitors and controls program execution only to a very small extent. Unused memory is filled with halt instructions so that programs will halt if they try to execute areas that are not part of the program. Various library routines detect certain program errors: ALGOL programs referencing beyond the bounds of arrays and user programs attempting impossible computations such as the square root of a negative number. Also, the drivers will abort execution for certain I/O error conditions.

Performing Service Functions for Programs in BCS

BCS performs several functions, such as the previously mentioned optional output buffering and memory management. Other functions provided by library subroutines include performing utility functions on I/O devices, positioning magnetic tape, pausing the program, and mathematical operations.

Summary

BCS has certain limitations which would require more complex and expensive hardware to provide: memory protection, batch processing, program scheduling, load and go, segmentation, or disc storage of files.

Where these features are not required, BCS adds flexibility to a system through relocation, convenience through library subroutines, and efficiency through interrupt processing. BCS is ideal for real-time applications where quick response to interrupts is required because it makes full use of the priority interrupt system and has low overhead.

BASIC CONTROL SYSTEM

The Basic Control System is designed to relocate and execute FORTRAN, ALGOL, FORTRAN IV, and Assembly Language programs. It provides these programs with interrupt-directed control of several I/O devices concurrently. BCS consists of a relocating loader, a central I/O control routine, I/O drivers for many devices, and a program that links all of these together into a system.

BCS is described in the *BASIC CONTROL SYSTEM* manual (HP 02116-9017).

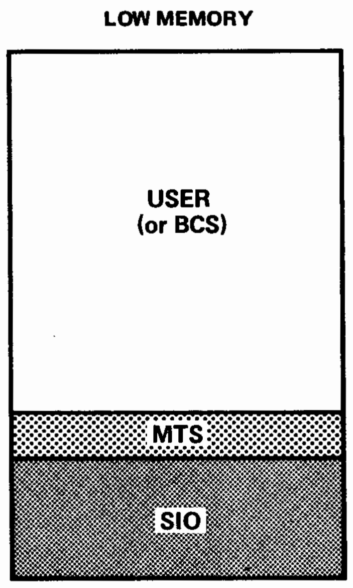
MAGNETIC TAPE SYSTEM

The purpose of MTS is to make the previously described software (SIO and BCS) more usable, to minimize the number of paper tapes that must be read and the amount of operator intervention required, and to speed up the transition between different computer operations.

The Magnetic Tape System (MTS) combines the BCS and SIO environments into an operating system based on magnetic tape storage of programs and data. The magnetic tape provides a skeleton framework in which all of the independent software modules (compilers, loader, etc.) are stored so that they can be loaded into core memory more quickly than from paper tape.

MTS consists of three parts:

1. A magnetic tape SIO driver,
2. A core-resident control program called IPL, and
3. A magnetic tape containing absolute and relocatable programs and a scratch area for temporary data storage.



HIGH MEMORY

MTS CORE MAP
(Minimum = 8K)

The magnetic tape driver is responsible for all final read or write operations on the magnetic tape unit. The control program (IPL) is responsible for loading absolute programs from the magnetic tape into core (using the driver) in response to user requests and transferring execution control to those programs. The magnetic tape containing programs is prepared using the Prepare Tape System (PTS). In order to see what this structure gains the user, we will review the functions of a computer system with respect to MTS.

Preparing Programs in MTS

MTS includes all the software from SIO used to prepare programs: Symbolic Editor, Assemblers, FORTRAN compiler, ALGOL compiler, and the Cross-Reference Symbol Table Generator. In addition, the stand-alone BASIC Interpreter can be used in the MTS-environment.

There are two advantages to using this software in MTS instead of SIO:

1. The software programs themselves (i.e., the compilers, etc.) need not be loaded from paper tape each time they are stored permanently on the magnetic tape. The user simply requests the program from the control program (IPL) by typing in a directive on the teleprinter or reading it in through the card reader. For example, :PROG,FTN causes the FORTRAN compiler to be loaded into core and initiated.

2. The source programs only have to pass through the low-speed input device (paper tape or card reader) once. After that, they are stored on magnetic tape if necessary and read automatically from there. This significantly cuts down the time needed to edit and translate programs.

NOTE: The ALGOL compiler and BASIC Interpreter do not store source programs on the magnetic tape.

Loading Programs in MTS

Both absolute and relocatable programs can be loaded into the computer under MTS, but the procedure varies for the two types of programs.

Absolute assembly language programs can be added to the program file on the magnetic tape. Once stored there, they can be loaded into core by means of a :PROG directive. In this respect, user absolute programs are equivalent to systems software that resides on the same file of the magnetic tape.

Since the BCS relocating loader can be stored on the magnetic tape and loaded into core, relocatable programs can be loaded using BCS in the MTS environment. Since the library of subroutines is stored on the magnetic tape instead of on paper tape, the loader can locate and link necessary subroutines much more quickly in MTS than in stand-alone BCS.

Other than this, BCS operates exactly the same in MTS as it does alone. The optional absolute dumps can be added to the magnetic tape and called by a :PROG directive if desired.

NOTE: The cost of having BCS operate under MTS is the loss of the core-memory needed to contain the SIO drivers and IPL. This means that programs must be smaller than would be true in stand-alone BCS.

Executing Programs in MTS

When programs are stored on the magnetic tape, they are assigned starting addresses so that when they are loaded in response to a :PROG directive, they can start executing automatically. Thus, the execution of programs in MTS is an automatic result of loading and requires no operator intervention.

Storing and Retrieving Data in MTS

A magnetic tape unit increases the data storage flexibility of programs immensely. Programs can store large amounts of information on the scratch area of the tape quickly and easily. This data is then read and scanned by the program as necessary. Although this method of data storage is not suitable for editing or updating of data files (multiple tape units or a random-access storage unit such as a disc are required for efficient file manipulating), it is still a significant improvement over paper tape storage.

Storing and Retrieving Programs in MTS

We have already mentioned the use of magnetic tape to store programs and the use of :PROG to retrieve them, but it is necessary to point out certain limitations of this feature. Programs can only be added to the magnetic tape by shutting down MTS and running a special system generation program called Prepare Tape System. There is no facility to add and delete programs on the magnetic tape during system operation.

Controlling I/O in MTS

MTS uses both the SIO and the BCS methods of I/O control. Absolute assembly language programs and the Magnetic Tape System itself must use the SIO drivers for input and output. Relocatable programs are run under BCS control and thus have the interrupt capability of BCS available to them.

Controlling or Monitoring Program Execution in MTS

The magnetic tape drivers in BCS can optionally protect the two library files on the magnetic tape.

Service Functions for Executing Programs in MTS

There is one important service function available in the Magnetic Tape System. When programs are through executing in MTS, the control is returned to IPL to process the next user directive. On this return, however, the program can request that IPL load in another program from the magnetic tape immediately instead of returning to the user. This option provides a means of linking or chaning programs so that complex problems can be solved by more than one program chained together.

MAGNETIC TAPE SYSTEM

The Magnetic Tape System uses a magnetic tape to store the software from SIO and BCS: editors, assemblers, compilers, loaders, etc. These programs can then be called into core by the user. The program that actually loads the programs from magnetic tape to core is called the Inter-Pass Loader (IPL).

MTS itself is well documented by the *MAGNETIC TAPE SYSTEM* manual (HP 02116-91752), while the system generation program, PTS, is covered in the *PREPARE TAPE SYSTEM* manual (HP 02116-91751).

Summary

The Magnetic Tape System provides important gains for the user over BCS and SIO. All the features of these two systems are available, with the addition of automatic program loading, a form of batch processing (i.e., job decks or cards consisting of directives, source programs, and data can be processed by MTS with little operator intervention), large-scale data storage, and program chaining.

The cost of these features is twofold: additional hardware (i.e., the magnetic tape unit and 8K memory are required) and loss of core memory used by IPL. The only compiler feature that is not available on MTS is" load and go".

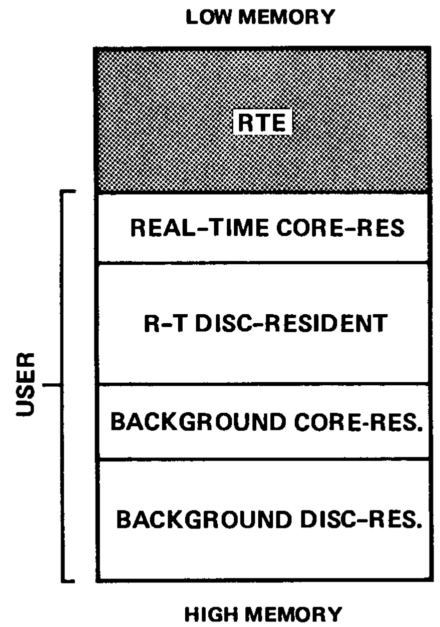
DISC-BASED SYSTEMS

To increase the power of a computer system much beyond the Magnetic Tape System, it is necessary to add a disc or drum memory. A high-speed disc allows the operating system to store and retrieve a multitude of programs and data files quickly and efficiently. Adding a disc provides the resources to remove from the user responsibility for the mundane details of operating his system,

This higher level of sophistication requires new concepts and a more precise determination of the purpose of each system. Is the system going to be used to process many large user jobs, to control a process, to run many programs at a time, or to maintain a complex data base? The answers to these questions will effect the design of the system.

REAL-TIME EXECUTIVE SYSTEM

One approach to the disc-based environment is the Hewlett-Packard Real-Time Executive (RTE) System. With the RTE System, the user is able to have many programs running concurrently in the same computer system. This situation,



RTE CORE MAP
(Minimum = 16K)

called multiprogramming, is controlled by an Executive program which schedules the many programs for execution according to their user-defined priority. In addition, since the RTE System is often used in constantly changing situations -- such as laboratories -- it is possible to compile and debug new user programs while existing programs are still being executed.

The RTE System operates on a computer with at least 16K core memory and certain additional hardware such as memory protect; and a fixed-head disc or drum memory. The system is designed to control many external processes by allowing many programs to execute. The user and system programs are combined into an integrated system for a particular hardware configuration by the Real-Time Generator (RTGEN). To see what this organization allows the user to accomplish, we will review the functions of a computer with regard to the RTE System.

Preparing Programs in RTE

The process of preparing and running programs in RTE is even simpler than in MTS. The use of disc to store source programs, intermediate code, relocatable object programs, and executable absolute programs allows programs to be edited, computed, loaded and executed after reading the original source only once.

The Editor, assembler, and compilers (ALGOL, FORTRAN, and FORTRAN IV) reside on the disc and can be initiated by the operator to run in a background mode (i.e., giving way to higher priority programs when necessary).

Note: *The ALGOL compiler requires a 24K computer. The BASIC Interpreter cannot run in the RTE environment.*

Loading and Executing Programs in RTE

Because the compilers and assemblers generate relocatable code (which can be run under BCS), a relocation process is necessary before programs can be executed in RTE. There are two places where this process occurs: during system generation, when all the modules of system and user software are combined into a tailored operating system; or by the Relocating Loader during operation of RTE. The Relocating Loader normally accepts relocatable object programs from the compilers (via the disc or paper tape if desired) and generates an executable core image on the disc. Programs which have been tested in the background mode can be added to the permanent set of programs either by generating a new system or by using the Relocating Loader in a special mode.

Program execution is the most complex part of RTE. Four classes of programs (real-time core-resident, real-time disc-resident, background core-resident or background disc-resident) are allowed. Each type is assigned its own area of core. Core-resident programs remain in their area permanently; disc-resident programs are loaded into their area one at a time when needed.

The core-resident programs can be incorporated into the system only at system generation time, but disc-resident programs can be added during operation by the Relocating Loader as well.

The Executive schedules these programs for execution in response to interrupts from external devices, operator or program requests, or completion of time intervals. Since many programs can be scheduled simultaneously, the Executive looks at user-defined priorities for each program to resolve conflicts.

The real-time area is usually assigned to programs that control external events. The core-resident programs provide the fastest response, because they do not have to be loaded in from the disc. Only one real-time disc-resident program can execute at a time, but an optional swapping feature allows these programs to be swapped out of core while a higher priority program is executed.

Programs run in the background disc-resident area can be segmented, that is, broken into parts so that the effective size of the program is not limited to the core available.

Storing and Retrieving Programs and Data in RTE

The existence of the disc increases the storage capabilities of an operating system greatly: source programs can be stored on the disc and edited; object programs can be stored on the disc by the compilers and retrieved by the Loader; executable programs can be stored on the disc where they can be loaded into core by the Executive in milliseconds. Programs can use the disc to store and retrieve data during execution.

Controlling I/O in RTE

All interrupt and I/O processing in RTE is controlled by the Executive. For each type of controller which is connected to the computer, RTE contains a driver subroutine. Each driver can handle simultaneous operations on many different devices of the same type. The Executive channels all interrupts and requests for I/O to the appropriate driver. Programs are suspended during I/O operations (so that other programs can use the computer).

Because all interrupts are channeled through a central routine, a certain amount of overhead time is incurred. For processes which cannot tolerate any delay, a privileged interrupt option is provided which allows the device to contact its driver directly without going through the Executive.

Controlling Program Execution in RTE

The existence of memory protect allows the Executive to confine user programs to their allotted area of core. The user programs cannot modify the system programs or other user programs and must call the Executive for all communication.

Memory protect is an important feature in systems where it is imperative that the system not stop operating for even a moment. For example, it might be disastrous if a computer monitoring a chemical reaction were to fail because a user program destroyed the system software.

Services for Executing Programs in RTE

The RTE System provides a wealth of services for an executing program. Any program can call the Executive to:

1. Perform input and output and determine status.
2. Access the disc.
3. Terminate or suspend itself.
4. Load its segments.
5. Schedule other programs for execution.
6. Obtain the time of day.
7. Set execution time intervals.

Summary

When more than one program must be executed concurrently with high system integrity, the Real-Time Executive System is a logical choice. It provides everything that an operator might need to control several operations simultaneously and prepare new programs.

Because it provides significantly more power than any previous system, RTE requires additional hardware and the system software uses more core space. Also, since the the Executive is monitoring many processes and examining all interrupts, response to external events is somewhat slower than in BCS.

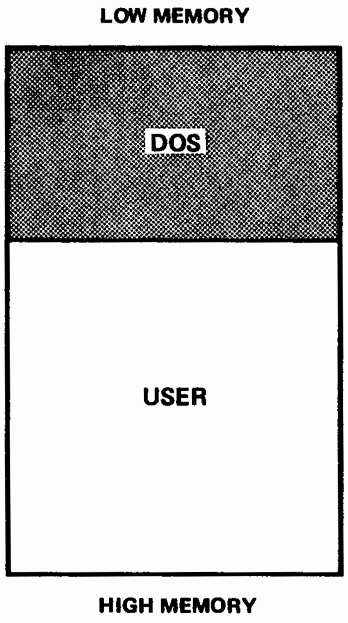
REAL-TIME EXECUTIVE SYSTEM

The Real-Time Executive (RTE) uses a fixed-head disc, multiprogramming, and priorities to schedule both real-time and background programs that can be core- or disc-resident. RTE controls all input/output and interrupt processing, except for special privileged interrupts, which can circumvent RTE for extra-quick response.

There are other ways to organize a disc-based system, depending on your objectives. Although it is disc-based, the next system to be considered is quite different from RTE.

DISC OPERATING SYSTEM

Another approach to the use of disc storage is exemplified in the Hewlett-Packard Disc Operating System (DOS). This system emphasizes batch processing of user jobs to combine editing, compilation, and execution of FORTRAN, FORTRAN IV, ALGOL, and Assembly Language programs.



DOS CORE MAP
(Minimum = 8K)

Operating systems have been created (for both fixed-head and moving-head disc units) that operate on a computer with a minimum of 8,192 words of memory and appropriate I/O devices. The DOS software is customized for a specific hardware system by the DOS Generator (DSGEN). In a typical DOS installation, the users submit their card decks to a computer operator who feeds them into a card reader. The DOS supervisory software automatically loads the assembler, compilers, loader, and other software from the disc into core as needed. The usual handling of paper tapes at each stage is therefore eliminated.

Batch Processing

DOS batch processing is organized around the concept of a job: a self-contained set of directives to the system, data, and source programs. Once the operator places a job in the card reader, the job is usually run without operator intervention. A job allows the user to employ all the features of DOS quickly, efficiently, and automatically. DOS can also be put into a keyboard mode where the operator enters all directives through the system terminal.

Directives

A directive is a command to DOS, issued by the system operator through the terminal or by the user through punched cards or tape included in a job. Directives call upon system routines (compilers, etc.), manipulate files on disc, and perform many service functions such as dumping core or disc.

To highlight the features of DOS more clearly, we will reivew the functions of a computer system with regard to DOS.

Preparing, Loading, and Executing Programs in DOS

In one self-contained card deck, a user can set up a job that will automatically edit, translate, load, and execute user programs. Since, as in RTE, all software programs, source programs, intermediate code, and object programs are stored on the disc, the entire processing is only limited by the speed of the disc. DOS is even more convenient than RTE, however, because it provides batch processing of these tasks; the operator need not (but can) type in every command manually as in RTE.

Software for DOS includes compilers (FORTRAN, FORTRAN IV, ALGOL) assembler, Relocating Loader, and Relocatable Library.

Storing and Retrieving Programs and Data in DOS

DOS has a particularly powerful file system. User files can contain source statements, relocatable and loader-generated object programs, and ASCII or binary data. All files have symbolic names and are referenced by them, not their disc addresses. Programs and users need not know exactly where files are located to use them.

Files can be edited, purged, listed, searched and dumped by user directives, and data files can be accessed by user programs. With a moving-head disc, users can have private disc cartridges or packs which they can take with them when their job is completed.

Controlling I/O in DOS

As in RTE, all I/O operations are performed by the Executive software. User programs call the system when they want to access I/O devices or the disc. In DOS, I/O operations are not buffered by the system, although user programs need not be suspended during I/O unless the user so requests. There is no multiprogramming facility in DOS.

Unlike RTE, drivers in DOS need not be core-resident. In order to cut down on core requirements, the user may store all of his drivers (except system teleprinter and disc drivers) on the disc. The Executive will load the drivers into core one at a time as they are needed.

Controlling Program Execution in DOS

The extent to which DOS can control program execution depends on whether certain hardware options are available. If the computer contains memory protect, user programs cannot modify the executive area of core or do their own

I/O. If a time-base generator is present, programs can be set to run for a limited time and the system can keep an account of how much computer time each job uses.

Services for Executing Programs in DOS

In addition to the I/O and file capabilities already mentioned, DOS provides many other services for programs. They can load segments, perform I/O control operations (e.g. backspace magnetic tape), determine I/O device status, signal completion, suspend themselves, determine the time, and be dumped (memory contents printed) upon completion. As in other systems, there is an extensive library of mathematical and utility subroutines also.

Summary

The resources involved in DOS are simple: starting with a properly equipped computer one need only add a disc unit and whatever I/O devices one desires. The choice of a core size is a simple trade-off: the larger the core, the more system modules can be core-resident and the faster the system will operate.

DOS is quite a flexible and powerful system. Because of the number of hardware options (core sizes from 8K to 32K, fixed or moving-head disc, memory protect, time base, and I/O devices) available and the ability to make modules of the system either core- or disc- resident at system generation, customized versions of DOS can be easily developed to meet almost any batch processing need.

DISC OPERATING SYSTEM

The Disc Operating System uses a fixed- or moving-head disc unit to store software such as compilers, loaders, etc. It then accepts jobs from users and carries out the tasks specified in them by loading software from the disc. The main advantage of DOS (fixed-head) and DOS-M (moving-head) is the batch processing that this provides.

DOS is described in the manuals, *DISC OPERATING SYSTEM* (HP 02116-91748) and *MOVING-HEAD DISC OPERATING SYSTEM* (HP-02116-91779).

BASIC LANGUAGE SYSTEMS

Hewlett-Packard has three types of systems dedicated to the preparation and execution of programs written in BASIC. They are the stand-alone (or single-terminal) BASIC Interpreter, Educational BASIC, and Time-Shared BASIC. All three are complete operating systems that allow the user to enter, edit, and run programs conversationally.

Conversational Programming

The user sits at a terminal and converses with the computer. As the user types in each line, the computer examines it. If there is an error or something the computer cannot decipher, the computer reports back immediately. The line can then be corrected. The computer's instant-response greatly speeds up the programmer's learning and debugging process.

A program is an ordered sequence of instructions on how to perform a specific task. In BASIC, a program is a set of numbered "statements" terminated by an END statement. Each statement has a unique statement number that determines its order in the program. For example:

```
25Ø LET A = 265.3
```

The user types the statements into the computer in any order. Whenever a statement is typed in, the computer inserts it in its proper place (replacing a previous statement with the same number if necessary). This flexible editing feature allows the user to change his program quickly and easily.

SINGLE-TERMINAL BASIC

Single-Terminal BASIC is a system consisting of a computer (8K minimum), user terminal, paper tape reader (optional), and paper tape punch (optional). The user can enter BASIC programs through the tape reader or terminal and can output programs through the tape punch or terminal. The BASIC System is configured for specific hardware using the Prepare BASIC System (PBS).

EDUCATIONAL BASIC

The Educational BASIC System is quite similar to single-terminal BASIC with the addition that programs can be entered through a Mark Sense Card Reader at up to 200 cards per minute. The ability to mark program cards with a pencil eliminates keypunching equipment. The ability to process programs continuously (like batch processing) eliminates the common line of students waiting while each student slowly types his program in through the terminal.

An important feature of single-terminal and Educational BASIC (which is not found in the more complex Time-Shared BASIC), is the ability to include assembly language subroutines in the system and call them from BASIC programs. This allows each installation to provide customized capabilities for its users.

BASIC SYSTEMS

The BASIC Systems interpret programs written in the BASIC Language that has already been described. The main system advantage of these systems is their self-contained nature and the conversational development of programs.

The BASIC Systems are described in *HP BASIC* (HP 02116-9077), and *A GUIDE TO EDUCATIONAL BASIC* (HP 02116-91173).

HEWLETT hp PACKARD

digital computers

11000 WOLFE ROAD, CUPERTINO, CALIFORNIA 95014, TELEPHONE 408 257-7000, TWX 910-338-022
EUROPE: 1217 MEYRIN-GENEVA, SWITZERLAND • CABLE "HEWPACKSA" TEL. (022) 41

FEBRUARY, 1971

5951 - 1