Communicating with Digital's MPS microprocessor.

digital

COMMUNICATING WITH DIGITAL'S MPS MICROPROCESSOR

Prepared by

Logic Products Group
Digital Equipment Corporation

July 1975

Printed in U.S.A.

PREFACE

You may be familiar with such lofty terms as "FORTRAN," "COBOL," "foreground/background," "batch processing," etc. These terms conjure up visions of huge computer systems with row upon row of tape units, printers, video terminals, etc., requiring the use of many resident software "geniuses" to keep the system running. Such systems typified by names like PDP-10, 370, 6600, etc., do indeed exist and do require a large staff to keep them running. They represent the high end of the computer hierarchy, and because of their huge size and power, receive a great deal of publicity and are almost universally endowed with having an "intelligence" of their own. Nothing could be farther from the truth. These computers have no innate intelligence; they are absolutely passive devices, and they must be commanded to perform the simplest function, no matter how large the ultimate system turns out to be.

The publicity they receive also tends to obscure another fact--that at the low end of the computer hierarchy, computers known as <u>microprocessors</u> are revolutionizing the field of process monitor/control in the largely unsaturated industrial and professional markets where specialized computer power is required on a local basis. These processors are usually so small that they are buried in larger systems and, thus, escape much publicity. They are taken very much for granted but they do exist, and they are performing an ever-growing role in today's industrial world, especially in data communications, laboratory automation, and machine tool control.

Notwithstanding their diminutive size, they are truly computers and must be commanded to perform, just the same as their larger brothers--they must be "programmed." In their case, however, we do not have to be concerned with the "buzz-words" like FORTRAN, etc. Programming a microprocessor is a relatively simple operation--in fact, this document will strip much of the mystery from the term "programming," and focus the reader's attention on the art of communicating with and getting on thoroughly familiar terms with the microprocessor--in this case, MPS.

When the reader has assimilated the information in this book, he will be very familiar with MPS, and it will have become a powerful processing tool in his hands.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT'D.)

TABLE OF CONTENTS (CONT'D.)

CHAPTER 1

NUMBER SYSTEMS

1.1 INTRODUCTION

A basic requirement to understanding of MPS, or any digital computer system, is an understanding of various number systems. Presented below is a general overview of the decimal, binary, and octal numbering systems. These systems are extensively covered in many textbooks. We urge you to become totally well-versed in these number systems so as to make your programming exercises a lot easier.

The concept of writing numbers, counting, and performing the basic operations of addition, subtraction, multiplication, and division has been directly developed by man. Every person is introduced to these concepts during his formal education. One of the most important factors in scientific development was the invention of the decimal numbering system. The system of counting in units of tens probably developed because man has ten fingers. The use of the number 10 as the base of our number system is not of prime importance; any standard unit would do as well. The main use of a number system in early times was measuring quantities and keeping records, not performing mathematical calculations. As the sciences developed, old numbering systems became more and more outdated. The lack of an adequate numerical system greatly hampered the scientific development of early civilizations.

Two basic concepts simplified the operations needed to manipulate numbers: the concept of position and the numeral zero. The concept of position consists of assigning to a number a value which depends both on the symbol and on its position in the whole number. For example, the digit 5 has a different value in each of the three numbers 135, 152, and 504. In the first number, the digit 5 has its original value 5; in the second, it has the value of 50; and in the last number, it has the value of 500, or 5 times 10 times 10. Sometimes a position in a number does not have a value between 1 and 9. If this position were simply left out, there would be no difference in notation between 709 and 79. This is where the numeral zero fills the gap. In the number 709, there are 7 hundreds, 0 tens, and 9 units. Thus, by using the concept of position and the numeral 0, arithmetic becomes quite easy.

A few basic definitions are needed before proceeding to see how these concepts apply to digital computers.

Unit — The standard utilized in counting separate items in the unit.

Quantity — The absolute or physical amount of units.

Number System — A number system is a means of representing quantities using a set of numbers. All modern number systems use the zero to indicate no units, and other symbols to indicate quantities. The base or radix of a number system is the number of symbols it contains, including zero. For example, the decimal number system is base or radix 10, because it contains 10 different symbols (viz., 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9).

1.2 BINARY NUMBER SYSTEM

The fundamental requirement of a computer is the ability to physically represent numbers and to perform operations on the numbers thus represented. Although computers which are based on other number systems have been built, modern digital computers are all based on the binary (base 2) system. To represent ten different numbers (0, 1, 2, ..., 9) the computer must possess ten different states with which to associate a digit value. However, most physical quantities have only two states: a light bulb is on or off; switches are on or off; holes in paper tape or cards are punched or not punched; current is positive or negative; material is magnetized or demagnetized; etc. Because it can be represented by only two such physical states, the binary number system is used in computers.

To understand the binary number system upon which the digital computer operates, an analysis of the concepts underlying the decimal number system is beneficial.

1.2.1 Position Coefficient

In the decimal numbering system (base 10), the value of a numeral depends upon the numeral's position in a number, for example:

$$\begin{array}{rrcrcr} 347 = & 3 & \times & 100 & = & 300 \\ & 4 & \times & 10 & = & 40 \\ & 7 & \times & 1 & = & \underline{7} \\ & & & & & 347 \end{array}$$

The value of each position in a number is known as its position coefficient. It is also called the digit position weighting value, weighting value, or weight, for short. A sample decimal weighting table follows:

$$\ldots 10^3 \quad 10^2 \quad 10^1 \quad 10^0$$

and, as shown above,

$$347 = 3 \times 10^2 + 4 \times 10^1 + 7 \times 10^0.$$

Weighting tables appear to serve no useful purpose in our familiar decimal numbering system, but their purpose becomes apparent when we consider the binary or base 2 numbering system. In binary we have only two digits, 0 and 1. In order to represent the numbers 1 to 10, we must utilize a count-and-carry principle familiar to us from the decimal system (so familiar we are not always aware that we use it). To count from 0 to 10 in decimal, we count as follows:

```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10  with a carry to the 10^1 column
```

Continuing the counting, when we reach 0 in the units column again, we carry another 1 to the tens column. This process is continued until the tens column becomes 0 and a 1 is carried into the hundreds column, as shown below:

```
 0              10               90
 1              11               91
 2              12               92
 3              13               93
 4              14               94
 5              15               95
 6              16               96
 7              17               97
 8              18               98
 9              19               99
10  one carry   20  one carry   100  two carries
```

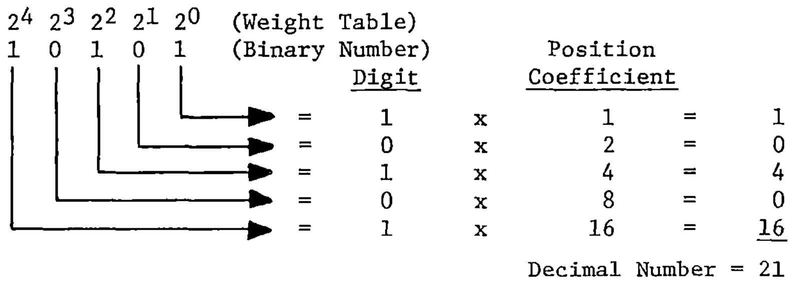
1.2.2 Counting in Binary Numbers

In the binary number system, the carry principle is used with only two digit symbols, namely 0 and 1. Thus, the numbers used in the binary number system to count up to a decimal value of 10 are the following:

Binary	Decimal	Binary	Decimal
0	(0)	110	(6)
1	(1)	111	(7)
10	(2)	1000	(8)
11	(3)	1001	(9)
100	(4)	1010	(10)
101	(5)		

When using more than one number system, it is customary to subscript numbers with the applicable base (e.g., $101_2 = 5_{10}$).

A weighting table is used to convert binary numbers to the more familiar decimal system.



It should be obvious that the binary weighting table can be extended, like the decimal table, as far as desired. In general, to find the value of a binary number, multiply each digit by its position coefficient and then add all of the products.

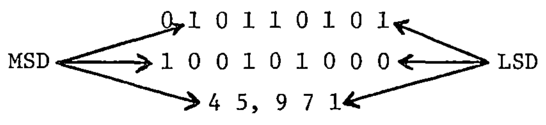
1.2.3 Arrangements of Values

By convention, weighting values are always arranged in the same manner: the highest on the extreme left and the lowest on the extreme right. Therefore, the position coefficient begins at 1 and increases from right to left. This convention has two very practical advantages. The first advantage is that it allows the elimination of the weighting table, as such. It is not necessary to label each binary number with weighting values, as the digit on the extreme right is always multiplied by 1 (2^0), the digit to its left is always multiplied by 2 (2^1), the next by 4 (2^2), etc. The second advantage is the elimination of some of the 0s. Whether a 0 is to the right or left, it will never add to the value of the binary number. Some 0s are required, however, as any 0s to the right of the highest valued 1 are utilized as spaces or place keepers, to keep the 1s in their correct positions. The 0s to the left, however, provide no information about the number and may be discarded; thus, the number 0001010111 = 1010111.

The MPS systems operate upon 8-bit (binary digit) numbers. This means that the numbers from 0 to 11111111_2 (256_{10}) can be directly represented.

1.2.4 Significant Digits

The "leftmost" 1 in a binary number is called the <u>most significant digit</u>. This is abbreviated MSD. It is called the "most significant" in that it is multiplied by the highest position coefficient. The <u>least significant digit</u>, or LSD, is the extreme right digit. It may be a 1 or 0, and has the lowest weighting value, namely 1. The terms LSD and MSD have the same meaning in the decimal system as in the binary system, as shown below:



1.3 OCTAL NUMBER SYSTEM

It is probably quite evident at this time that the binary number system, although quite nice for computers, is a little cumbersome for human usage. It is very easy for humans to make errors in reading and writing quantities of large binary numbers. The octal or base 8 numbering system helps to alleviate this problem. The base 8 or octal number system utilizes the digits 0 through 7 in forming numbers. The count-and-carry method mentioned earlier applies here also. Table 1-1 shows the octal numbers with their decimal and binary equivalents.

Table 1-1

Decimal-Octal-Binary Equivalents

Decimal	Octal	Binary	Decimal	Octal	Binary
0	0	0	7	7	111
1	1	1	8	10	1000
2	2	10	9	11	1001
3	3	11	10	12	1010
4	4	100	11	13	1011
5	5	101	12	14	1100
6	6	110	13	15	1101

The octal number system eliminates many of the problems involved in handling the binary number system used by a computer. To make the 8-bit numbers of the MPS easier to handle, they are often separated into 3-bit groups. These 3-bit groups can be represented by one octal digit, using the previous table of equivalents, as seen below.

A binary number 10111101

is separated into 3-bit groups by starting with the LSD end of the number and supplying leading zeros if necessary:

010 111 101

The binary groups are then replaced by their octal equivalents:

$$010_2 = 2_8$$
$$111_2 = 7_8$$
$$101_2 = 5_8$$

and the binary number is converted to its octal equivalent:

2 7 5

Conversely, an octal number can be expanded to a binary number using the same table of equivalents:

$$307_8 = 11\ 000\ 111_2$$

1.4 BYTES

In the MPS, each 8-bit binary grouping is called a byte. All operations within the MPS are byte-oriented; i.e., bytes are added, bytes are transferred between devices, etc.

1.5 DATA

To us, data means numeric information. It can consist of physical quantities such as voltage, temperature, pressure, or velocity; or the data might be a stockroom part number or a weekly payroll statement--virtually any kind of information that must be manipulated and made useful. As mentioned earlier, the MPS is a byte-oriented processor and, thus, data takes the form of 8 bits. For the most part, data is placed in the store, or memory. This memory is conveniently organized to store bytes. Details on memory organization are discussed later.

But how does the MPS system know what it is supposed to do with data once it receives it; how does the system know what the problem is or what the process of solving the problem is? The answer is instructions.

1.6 INSTRUCTIONS

An instruction is a command that directs the MPS system to perform a particular operation on the data. An example is an "add" instruction which, as we will find out later, directs the MPS to add one byte of data to another byte of data.

A sequence of instructions arranged in a given order to perform a particular function is called a routine. The plan of action or plan of solution which the routine embodies is the program. If the routine is written directly in terms of the instructions as the MPS knows them, it is said to be in machine language; that is, the instructions are encoded in the binary numbering system or its shorthand form--octal. If the routine is written in terms of instructions foreign to the MPS, it is often said to be in pseudo-language, or symbolic language. Translation or interpretation of symbolic language into machine language must be done before the MPS does the routine or, more usually, in the MPS itself through the use of a so-called "high-level language."

In summary, we have seen that the MPS will only operate with binary numbers. Since writing routines in binary is cumbersome, the octal equivalents are more commonly used. The MPS is structured to operate on bytes (8 bits) of information at a time. This information can take the form of either data or instructions.

CHAPTER 2

COMPUTER HARDWARE

2.1 THE TYPICAL DIGITAL COMPUTER

A typical digital computer, whether it be a large-scale timesharing system, a minicomputer, or a microprocessor, needs an Arithmetic Unit to contribute the arithmetic ability; it needs a Memory to store the data bytes which the computing system will need or produce. It needs a Control section to somehow boss the overall system and make it produce useful work, and finally, it needs to communicate with the world in which it finds itself. The communications link from the world to the computer is the input, and the link from the computer back to the world, the output. Figure 2-1 shows the interrelationships of these units. In addition, as with most machinery, human intervention will often be necessary, and so there is a minor and supplementary communication link from the world into the computer: the console. The console will contain switches, displays, etc., for human monitoring of and intervention into computer affairs.

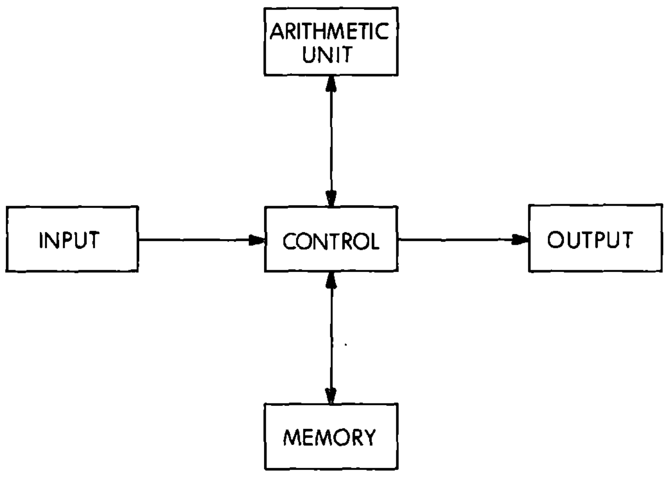


Figure 2-1. Typical Digital Computer Block Diagram

The above major functional units are common to any digital computer, including the MPS. They are the "hardware" or circuit elements consisting of interconnected flip-flops, inverters, amplifiers, etc. A detailed discussion of these functional hardware units is contained in every textbook and magazine article dealing with computer fundamentals.

2.2 THE MPS COMPUTER

In the MPS, all of the "hardware" is contained on four modules:

M7341 Central Processor Module
M7344 Read/Write Memory Module
M7345 Read-Only Memory Module
M7346 External Event Detection Module

In addition, an optional operator's control panel is available that allows the user to monitor ongoing operations. Figure 2-2 provides photographs of the MPS components. Referring to Figure 2-1, the arithmetic unit, control, input, and output circuitry elements are all contained on the 8-1/2-inch by 10-inch M7341 module.

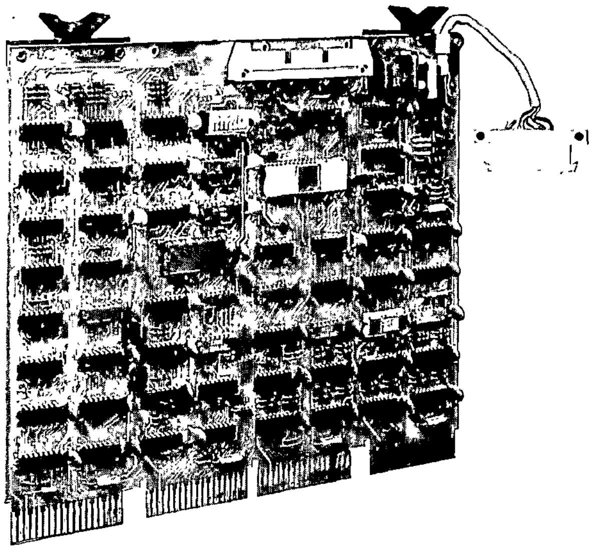
The MPS can be arranged in many different configurations, yet they all must contain one M7341 and some amount of memory. Figure 2-3 shows the basic digital computer units as they apply to MPS. Without delving into the circuitry itself, we must become familiar with the major characteristics of each unit. For purposes of this guide, we'll consider the arithmetic and the control as one unit. This arithmetic/control unit contains eight 8-bit registers and one 14-bit register of concern to us. A register can store one byte of data and can be considered analogous to scratch paper used in the intermediate calculations of an arithmetic problem. The nine registers are listed below.

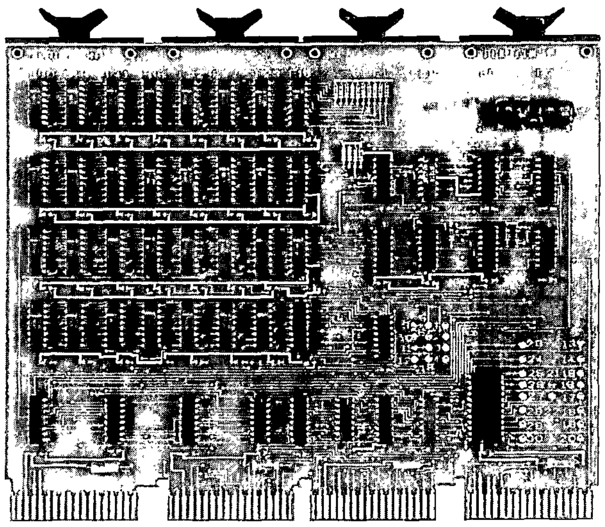
A	Accumulator
B, C, D, E	General Purpose Index Registers
H, L	General Purpose Index Registers plus Memory Address Pointer
IR	Instruction Register
PC	Program Counter (14 bits)

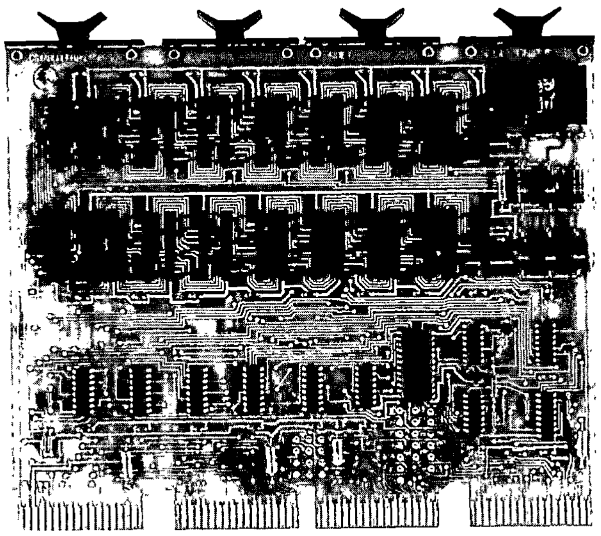
The most important, for programming purposes, is the A register. The 8-bit A register performs as the accumulator of the system; the results of all arithmetic and logical operations appear in the A register. All input/output (I/O) operations for external devices involve data transfers via the A register.

Four 8-bit general purpose Index Registers (B, C, D, and E) are used for temporary storage and for holding data for operations with the A register.

The 8-bit H and L registers perform in an identical way to the B, C, D, and E registers but have an additional feature: On operations requiring data in memory, the


M7341 Central Processor Module


M7344 Read/Write Memory Module


M7345 Read-Only Memory Module

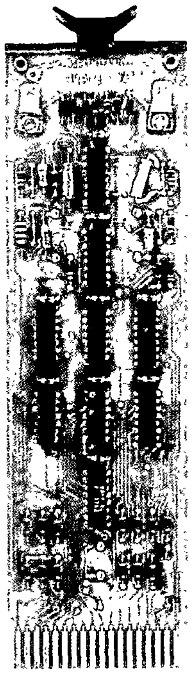

M7346 External Event Detection Module

Figure 2-2. Microprocessor Series Modules

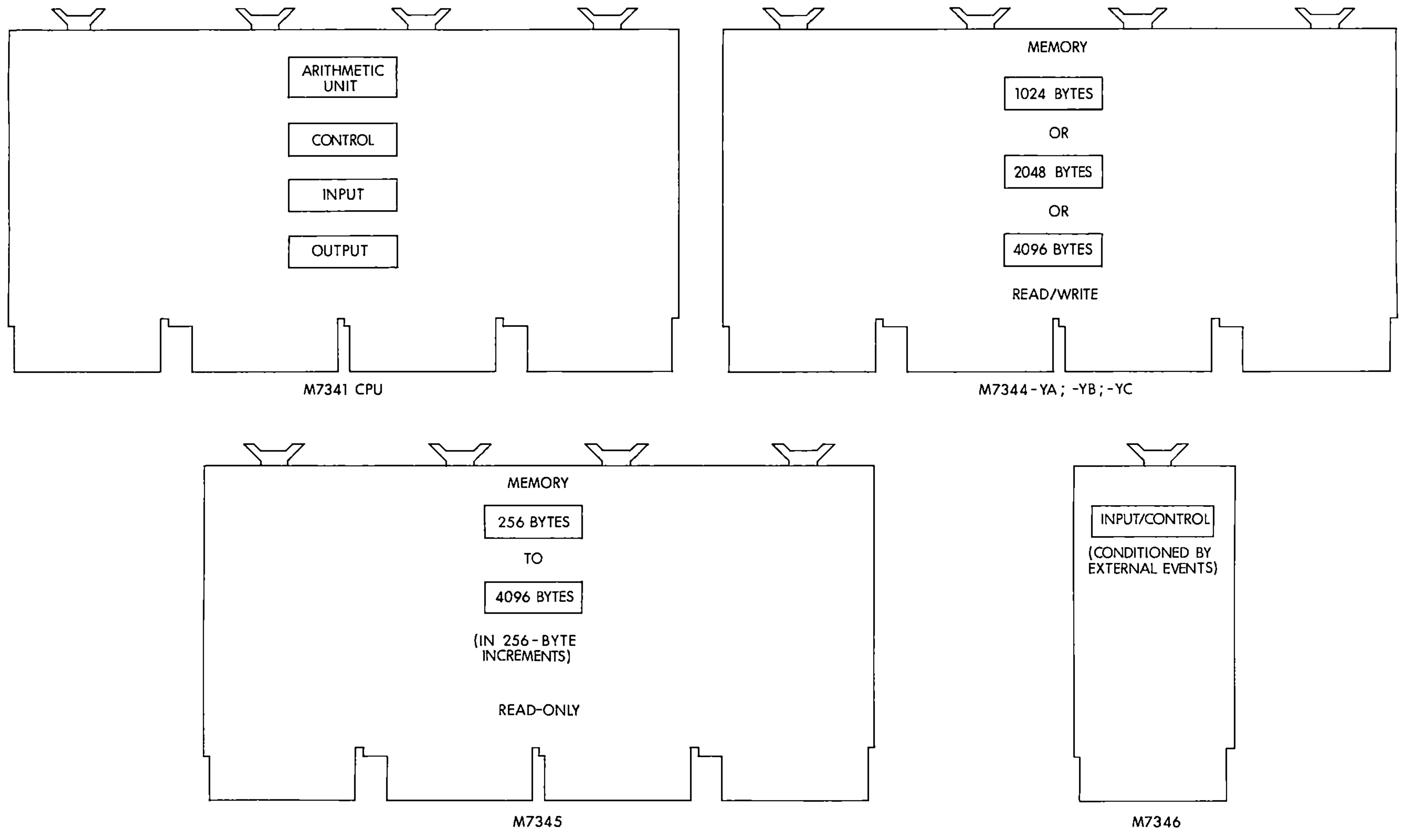


Figure 2-3. Basic Computer Units Applied to MPS

contents of the H and L registers point to the particular memory location containing the required data; i.e., they set up the actual memory locations that are to be used.

The Program Counter (PC), consisting of 14 bits, always refers to the memory location from which the next byte of instruction is to be obtained. In other words, it keeps track of the program sequence. Also associated with the PC is a functional unit called the "stack." The "stack" is actually a set of eight 14-bit registers, each of which can function as the PC at any given time during operation of MPS. The particular 14-bit register which currently functions as the PC depends on the particular MPS program being executed. The operation of the "stack" is explained later in the text.

Every instruction must pass through the 8-bit Instruction Register (IR) where it is decoded so as to be recognizable by the MPS circuits. Instructions for the IR are either fetched from the memory location, pointed to by the PC, or are externally entered through the input multiplexer (see below).

The operation of the MPS system is further enhanced by four status flip-flops (or "flags") which constantly monitor the arithmetic and logical operation of the system. Three of these flags (zero, carry, and sign) are used frequently, while the fourth flag (parity) is used primarily in a communications environment.

2.3 INPUT/OUTPUT

Data is transferred in and out of the MPS in two ways: (a) parallel, and (b) serial.

Parallel communication is done a byte (8 bits) at a time. Input data is accepted by the MPS from four separate 8-bit input channels. The choice of input channel is determined by the particular operation taking place in the MPS at that particular time. This operation of selecting a channel is known as multiplexing.

Output data is sent either to the memory modules or to a peripheral device over separate data channels.

Serial communication is controlled by a device known as a Universal Asynchronous Receiver/Transmitter (UART) which is part of the M7341 module. This mode of communication permits the MPS to interface with serial communication peripheral devices such as a Teletype or DIGITAL's line of video terminals and data acquisition devices (VT05, VT50, PDM70, RT02, etc.).

2.4 INTERRUPT

The MPS system has a full "interrupt" capability which allows I/O requests to be made on one of six priority levels. An interrupt may be defined as an event taking place external to the MPS system which is deemed to have an imperative effect on the operation of the system; e.g., loss of line power. The M7346 module handles these interrupts on a predetermined priority basis as well as offering the capability of automatic start-up and halting of the system.

2.5 MEMORY

In most digital computers, the memory (storage) section consists of either small magnetic cores or semiconductor flip-flops, each capable of representing an "ON" ("1") or "OFF" ("0") condition. A system of these cores or flip-flops arranged in a matrix can store any information which is represented in binary form.

Conceptually, memory can be in two parts: one for instructions and the other for data. It has proved more convenient and economical to provide only memory (in a functional sense), which can be allocated between the routine and the data as each user needs. If the memory is thought of as boxes (or cells) into each of which one packet of information (in MPS, an 8-bit byte) may be placed and held until needed, it is seen that locating items in the memory is simply the problem of identifying locations. This problem is analogous to that of locating houses in a town or entries in a matrix. Indeed the locations could be labeled by means of a row number and a column number or a street name and house number, but since the number of memory locations is fixed at the time of system construction, it is more convenient simply to number then sequentially. The numeric tag or <u>address</u> forevermore identifies each memory location to the MPS system.

MPS memory is divided into two separate and distinct functions: read/write and read-only. Both types of memory use semiconductor technology as opposed to core. This means, roughly speaking, that the basic storage element is the flip-flop.

In the M7344 Read/Write Memory, data is written into the storage cells by the M7341. The data will be stored intact until new data, also written by the M7341, replaces it. The MPS also reads the contents of the memory without destroying the original data, thus placing this memory in the "nondestructive read-out" category. This memory is also classed as being "volatile" because the stored data will be destroyed as soon as the system power is turned off (by virtue of the fact that the storage medium is the semiconductor flip-flop).

In the M7345 Read-Only Memory, data is written into the storage cells by some external device as opposed to the M7341. The data thus written is considered <u>permanent</u> storage and is <u>not</u> destroyed when power is turned off because of the nature of the semiconductor storage used. The MPS can only read the contents of this memory; hence, the term "read-only memory."

The M7344 Read/Write Memory modules can store a minimum of 1024 and a maximum of 4096 8-bit bytes in module increments of 1024, 2048, and 4096 bytes. The M7345 Read-Only Memory modules can store a minimum of 256 and a maximum of 4096 8-bit bytes in increments of 256 bytes.

2.6 ADDRESSING

Each MPS system can have a maximum memory storage capacity of $16{,}384_{10}$ 8-bit bytes and, thus, $16{,}384_{10}$ addresses. This 16,384 number is commonly referred to as 16K of memory. Actually, the decimal number is 16,383 since the numeral "0" is a true and valid number. Hence, the first location is assigned the address 00000 not 00001. The address, then, of the last location is always one less than the total number of bytes that can be stored. The octal equivalent of $16{,}383_{10}$ is $37{,}777_8$.

The 16K 8-bit locations in MPS memory are grouped into "blocks," each containing 377_8 (256_{10}) locations, or addresses. Thus, in a maximum memory configuration, there would be 77_8 (64_{10}) blocks (16,384 ÷ 256). Each of the 377_8 locations within a block is called an "offset." To specify a particular memory location, it is necessary to specify both the block of addresses being referenced and the offset of the address within the block. For example, location 100_8 in block 0 is referred to as "block 0, offset 100" or, the abbreviated form: "00#100." A full list of octal and decimal locations, with their block/offset equivalents, is provided in Appendix A of this guide.

As mentioned earlier, one of the functions of the 8-bit H and L registers is to "point" to the memory address being referenced. Before we can gain access to memory, then, we must load the desired address into these two registers. Two registers are required because we said the maximum MPS memory can be $37{,}777_8$ locations. Looking at this octal number in binary, we see that 14 bits are required (2^{14}):

3	7	7	7	7	octal
11	111	111	111	111	binary

These 14 bits represent two bytes: one byte containing the right-most 8 bits and the other containing the remaining 6 bits on the left. The 8-bit byte is referred to as the Low byte and is loaded into the L register. The 6-bit byte is called the High byte and is loaded into the H register. Now, look at our maximum address of $37{,}777_8$ as it resides in the H and L registers:

```
|11  111  1|11  111  111|
|   H Reg. |   L Reg.   |
```

Using the standard conventions for marking off binary numbers in octal, we see that the H register contains 77_8 and the L register contains 377_8:

```
    H Reg.          L Reg.
 (11  1)(11  1)    11  111  111
   7     7₈        3    7    7₈
```

This brings us back to block and offset. We said earlier that in the maximum memory configuration, there would be 77_8 blocks, each containing 377_8 locations. Well, we just proved how this is accomplished via the H and L registers. The above address is specified as block 77, offset 377. Another example: If we specify address "block 0, offset 100" (00#100), the H and L registers appear as:

H Reg.		L Reg.			
000	000	01	000	000	binary
0	0	1	0	0	octal

Thus, the H register contains the block number and the L register contains the offset number. One other point. We said all of the registers contain 8 bits. How, then, can the H register contain only 6 bits? Simple. The circuitry is designed to ignore the two "most significant bits" of the H register when used for addressing.

2.7 SUMMARY

All of the preceding material provides a foundation or background for understanding the MPS instruction set. Recapping, the following points were covered:

- Number systems — binary and octal familiarization
- Bytes — a grouping of 8 binary digits
- Instruction, routine, program, machine language, symbolic language
- Typical computer — arithmetic unit, control, memory, input, and output
- The MPS as it relates to the typical computer:
    - Nine registers — A, B, C, D, E, H, L, IR, PC
    - Input/Output:
        - Parallel — 8-bit input multiplexer (4 channels)
          — 8-bit output to memory
          — 8-bit output to peripheral device
        - Serial — UART
    - Memory — read/write and read-only
    - Memory addressing — block, offset

Now, let's get on to the main purpose of this guide; i.e., programming the MPS.

CHAPTER 3

PROGRAMMING BASICS

3.1 GENERAL

The MPS has 48 basic instructions. Knowledge of this instruction set is the first step in learning to program a microprocessor. The next step is to learn to use the instruction set to obtain correct results and to obtain them efficiently. The microprocessor is capable of storing information, performing calculations, making decisions based on the results, and arriving at a final solution to a given problem. The microprocessor cannot, however, perform these tasks without direction. Each step which the microprocessor is to perform must first be worked out by the user.

The user must write a routine, which is a list of instructions for the microprocessor to follow to arrive at a solution for a given problem. The list of instructions is placed in the microprocessor memory to activate the applicable circuitry so that the microprocessor can process the problem. The technique of listing the proper instructions is called coding.

3.2 CODING PHASES

In order to successfully solve a problem with a microprocessor, the user proceeds through the five phases listed below. A typical example that incorporates these phases is provided in Appendix E.

1. Definition of the problem to be solved.
2. Determination of the most feasible solution method.
3. Design and analysis of the solution--flowcharting.
4. Coding the solution in the symbolic language.
5. Program checkout.

3.2.1 Definition of Problem

The definition of the problem is not always obvious. When the problem is to sum four numbers, the defining phase is clear-cut. However, when the problem is to monitor and control the temperature in an office building, a precise definition of the problem is necessary. The question that must be answered in this phase is, "What precisely is the program to accomplish?" i.e., present the problem in language or terms understood by the person doing the job.

3.2.2 Choosing the Method

Determining the method to be followed is the second important phase in solving a problem with a microprocessor. There are perhaps an infinite number of methods to solve a problem, and the selection of one method over another is often influenced by the microprocessor system to be used. Having decided upon a method based on the definition of the problem and the capabilities of the microprocessor system, the user must develop the method into a workable solution. This implies a thorough knowledge of the capabilities of the microprocessor.

The user must design and analyze the solution by identifying the necessary steps to solve the problems and arrange them in a logical order, thus implementing the method. Flowcharting is a graphical means of representing the logical steps of the solution. The flowcharting technique is effective in providing an overview of the logical flow of a solution, thereby enabling further analysis and evaluation of alternative approaches.

3.2.3 Flowcharting

A single problem to add three numbers together is solved in a few easily determined steps. One could sit at his desk and write out three or four instructions for the microprocessor to solve the problem. However, he probably could have added the same three numbers with paper and pencil in much less time than it took him to write the program. Thus, the problems which one is usually asked to solve are much more complex than the addition of three numbers, because the value of the microprocessor is in the solution of problems which are inconvenient or time-consuming by human standards.

When a more complex problem is to be solved by a microprocessor, the program involves many steps, and writing it often becomes long and confusing. A method of solving a problem which is written in words and mathematical equations is extremely hard to follow, and coding the instructions from such a document would be equally difficult. A technique called flowcharting is used to simplify the writing of programs. A flowchart is a graphical representation of a given solution, indicating the logical sequence of operations that the computer is to perform. Having a diagram of the logical flow is a tremendous advantage to the coder when he is determining the method to be used for implementing the solution, as well as when he writes the coded program instructions. In addition, the flowchart is often a valuable aid when the coder checks the written program for errors.

The flowchart is basically a collection of boxes and connecting lines. The boxes indicate what is to be done and the lines indicate the sequence of the boxes. The boxes are of various shapes which represent the action to be performed in the program. A guide to the flowchart symbols and procedures which are used in this guide is contained in Appendix B.

The following are examples of flowcharts for specific problems, illustrating methods of attacking problems with a microprocessor program as well as illustrating flowcharting techniques. Example 1 adds three numbers together. Example 2 puts three numbers in increasing order.

Example 1 — Straight-Line Coding

Figure 3-1 is an illustration of straight-line coding. As the flowchart shows, there is a straight-line progression through the processing steps with no change in course. The value of X, which is equal to A + B + C is in the accumulator when the program stops.

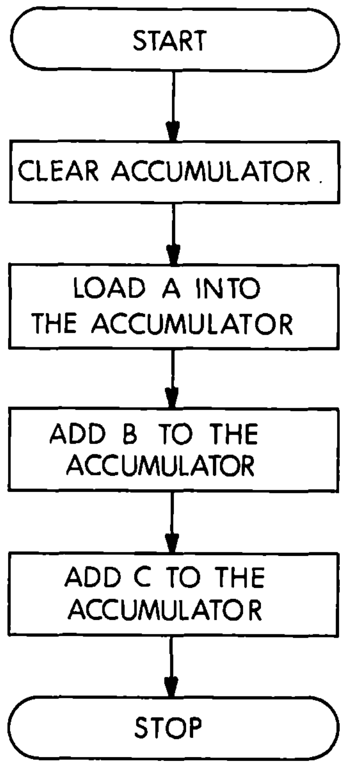


Figure 3-1. Add Three Numbers

Example 2 — Program Branching

Example 2 is designed to arrange three numbers in increasing order (Figure 3-2). The program must branch to interchange numbers that are out of order. (Branching, a common feature of coding, is described in detail later in this guide.) Note that the arithmetic operations of subtraction are done in the accumulator, which must be cleared initially.

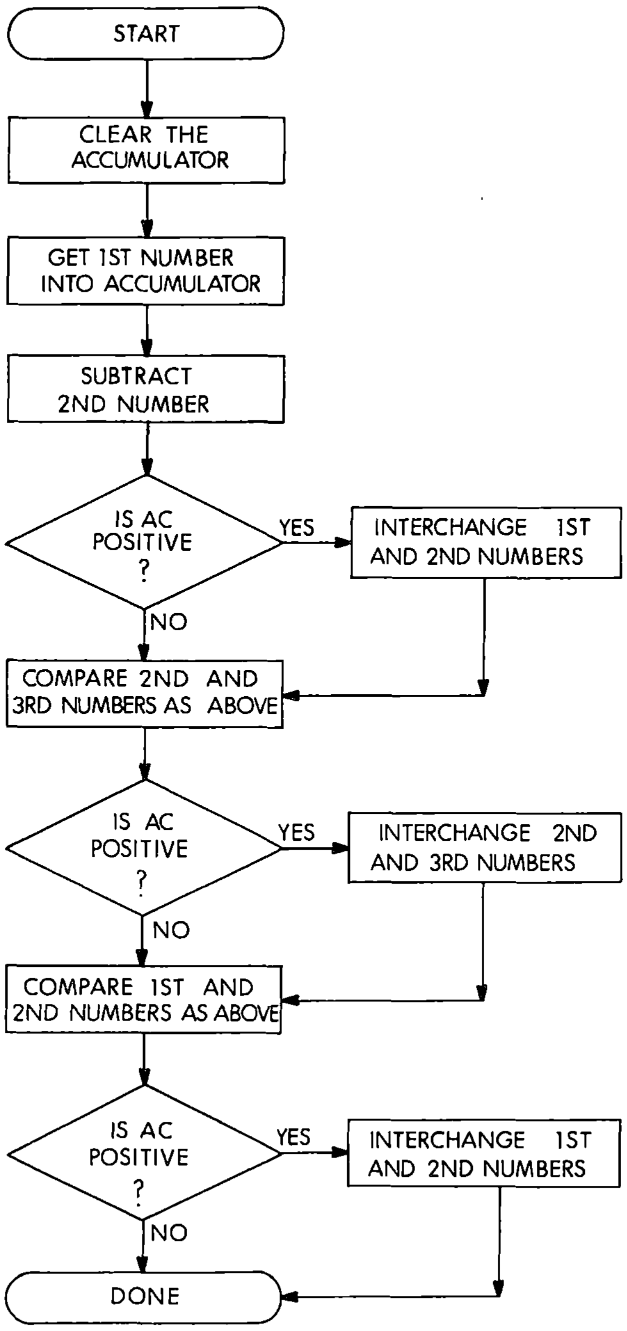

START
CLEAR THE ACCUMULATOR
GET 1ST NUMBER INTO ACCUMULATOR
SUBTRACT 2ND NUMBER
IS AC POSITIVE ?
YES
INTERCHANGE 1ST AND 2ND NUMBERS
NO
COMPARE 2ND AND 3RD NUMBERS AS ABOVE
IS AC POSITIVE ?
YES
INTERCHANGE 2ND AND 3RD NUMBERS
NO
COMPARE 1ST AND 2ND NUMBERS AS ABOVE
IS AC POSITIVE ?
YES
INTERCHANGE 1ST AND 2ND NUMBERS
NO
DONE

Figure 3-2. Arrange Three Numbers in Increasing Order

3.2.4 Coding

Having designed the problem solution, the user begins coding the solution by implementing the appropriate MPS instructions. This phase is commonly called programming but is actually coding and is only one part of the programming process. When the program has been coded and the program instructions have been stored in the microprocessor memory, the problem appears to be solved. At this point, however, the programming process is rarely complete. There are very few programs written which initially function as expected. Whenever the program does not work properly, the user is forced to begin the fifth step, that of checking out or "debugging" the program.

3.2.5 Checkout

The program checkout phase requires the user to methodically retrace the flow of the instructions step-by-step to find any program errors that may exist. One cannot tell a microprocessor: "You know what I mean!", as he might say in daily life. The microprocessor does not know what is meant until it is told, and once given a set of instructions, the microprocessor follows them precisely.

If needed instructions are left out or coding is done incorrectly, the results may be surprising. These flaws, or "bugs" as they are often called, must be found and corrected. There are many different approaches to finding bugs in a program; however, the chosen approach must be organized and painstakingly methodical if it is to be successful.

3.3 COMMUNICATING WITH MPS

Up to this point, we have only been discussing the philosophy of programming the MPS--how to write programs which are internally manipulated by the machine. We have not discussed the obvious questions of how we actually communicate physically with the MPS--how do we enter instructions and data into the machine; how do we then observe data is in the machine? In this respect, two basic methods exist for entering and observing data in MPS.

The Monitor/Control Panel (Figure 3-3) allows us to communicate directly with MPS. Data is entered in octal format (as described earlier in this text) via a set of toggle switches. Internal MPS data is observed via an array of lamps, also in octal format. The Control Panel is useful primarily for on-line system troubleshooting and the debugging of short programs. The mandatory use of octal code for data entry and retrieval makes the debugging of lengthy programs (say, in excess of 20-30 instructions) cumbersome and time-consuming. Therefore, for the purposes of this discussion, we will more or less disregard the Control Panel as a program control device, and concentrate on a more universally-accepted communication medium. In this area, a number of criteria are essential in the I/O equipment for complete program communication. These are:

- Full alphanumeric keyboard
- Hard copy read-out (or at least a video display)
- Paper tape punch
- Paper tape reader

The piece of equipment which supplies the above criteria and best suits this discussion is the DEC LT33 Teletypewriter (Figure 3-4). All references to program entry and retrieval will imply use of the LT33.

Both the Monitor/Control Panel and the LT33 Teletypewriter are described in Chapters 2 and 4, respectively, of the MPS User's Handbook.

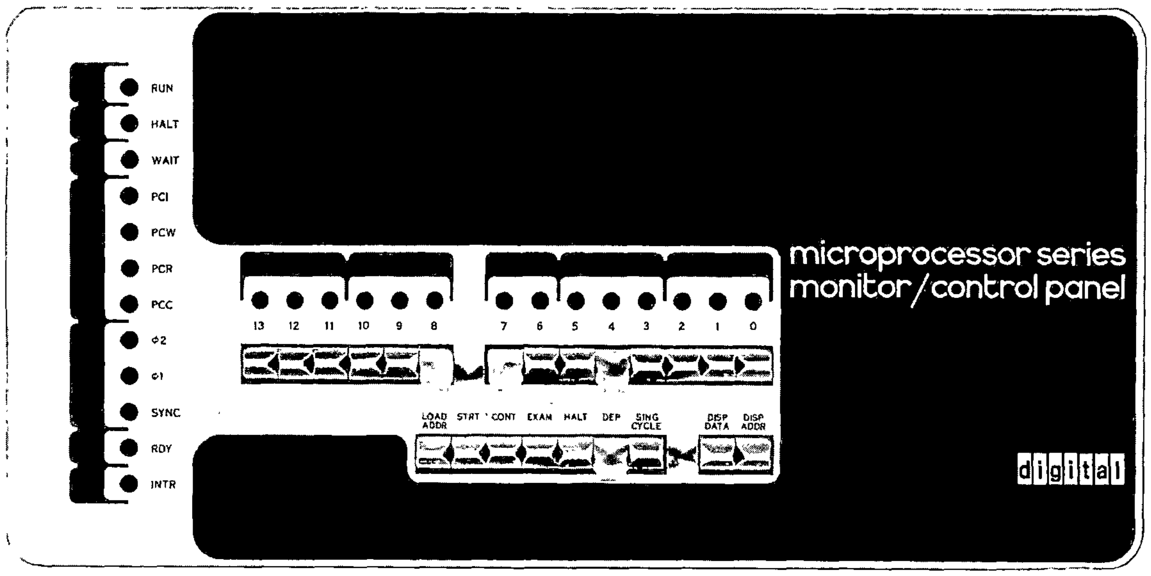


Figure 3-3. Monitor/Control Panel

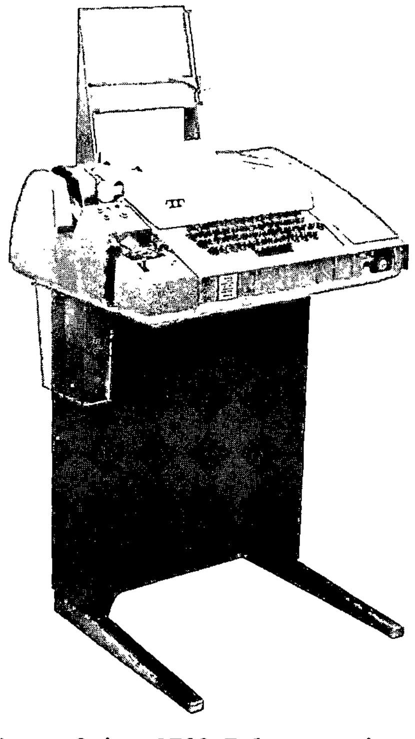


Figure 3-4. LT33 Teletypewriter

CHAPTER 4

CODING A PROGRAM

4.1 INTRODUCTION

As described earlier in this text, every microprocessor has a special code that tells it what to do. These specific codes for MPS are listed in Table 4-1. What is important to remember is that microprocessors only understand instructions and numbers that are represented in binary code. The coder tells the microprocessor whether to interpret the binary configuration as an instruction or as data by the way in which the configuration is encountered in the program. The MPS instruction to add the contents of a location in memory to the accumulator would be 10 000 111 in binary notation. If binary configurations appear cumbersome and confusing, the reader will not understand why most coders seldom use the binary number system in actual practice. Instead, they substitute the octal number system in which each digit represents 3 binary digits. Therefore, the add instruction above would be written 207 in octal. Coding a program in octal numbers, although an improvement upon binary coding, is also very inconvenient. The coder must learn a complete set of octal numbers which have no logical connection with the operations they represent. The coding is difficult for the programmer when he is writing the program, and this difficulty is compounded when he is trying to debug or correct a program. There is no easy way to remember the correspondence between an octal number and a computer operation.

To simplify the process of writing or reading a program, each instruction is often represented by a simple 3- or 4-letter <u>mnemonic</u> symbol. These mnemonic symbols are considerably easier to relate to a microprocessor operation because the letters often suggest the definition of the instruction. For example, ADM is the mnemonic for the instruction: add memory to the accumulator. The coder is now able to write a program in a language of letters and numbers which suggests the meaning of each instruction.

Table 4-1

Basic Instruction Set

Data and Instruction Formats

Data in MPS is stored in the form of 8-bit binary bytes. All data transfers are in the following format:

$\boxed{D_7\ D_6\ D_5\ D_4\ D_3\ D_2\ D_1\ D_0}$

Data Word

The program instructions may be one, two, or three bytes in length. Multiple byte instructions must be stored in successive words in program memory. The instruction formats then depend on the particular operation executed.

		Typical Instructions
One Byte Instructions		
$D_7\ D_6\ D_5\ D_4\ D_3\ D_2\ D_1\ D_0$	Op Code	Register to register, memory reference, I/O arithmetic or logical, rotate or return instructions
Two Byte Instructions		
$D_7\ D_6\ D_5\ D_4\ D_3\ D_2\ D_1\ D_0$	Op Code	Immediate mode instructions
$D_7\ D_6\ D_5\ D_4\ D_3\ D_2\ D_1\ D_0$	Operand	
Three Byte Instructions		
$D_7\ D_6\ D_5\ D_4\ D_3\ D_2\ D_1\ D_0$	Op Code	
$D_7\ D_6\ D_5\ D_4\ D_3\ D_2\ D_1\ D_0$	Low Address	JUMP or CALL instructions
X X $D_5\ D_4\ D_3\ D_2\ D_1\ D_0$	High Address*	

*For the third byte of this instruction, D_6 and D_7 are "don't care" bits.

Index Register Instructions

The load instructions do not affect the flag flip-flops. The increment and decrement instructions affect all flip-flops except the carry.

Mnemonic	Instruction Code D7 D6	D5 D4 D3	D2 D1 D0	Description of Operation
(1)Lr_1r_2	1 1	D D D	S S S	Load index register r_1 with the content of index register r_2.
(2)LrM	1 1	D D D	1 1 1	Load index register r with the content of memory register M.
LMr	1 1	1 1 1	S S S	Load memory register M with the content of index register r.
(3)LrI	0 0 B B	D D D B B B	1 1 0 B B B	Load index register r with data B...B.
LMI	0 0 B B	1 1 1 B B B	1 1 0 B B B	Load memory register M with data B...B.
INr	0 0	D D D	0 0 0	Increment the content of index register r $(r \neq A)$.
DCr	0 0	D D D	0 0 1	Decrement the contents of index register r $(r \neq A)$.

Table 4-1 (Cont'd.)

Accumulator Group Instructions

The result of the ALU instructions affect all of the flag flip-flops. The rotate instructions affect only the carry flip-flop.

Mnemonic	Instruction Code D7 D6	D5 D4 D3	D2 D1 D0	Description of Operation
ADr	1 0	0 0 0	S S S	Add the content of index register r, memory register M, or data B...B to the accumulator. An overflow (carry) sets the carry flip-flop.
ADM	1 0	0 0 0	1 1 1	
ADI	0 0	0 0 0	1 0 0	
	B B	B B B	B B B	
ACr	1 0	0 0 1	S S S	Add the content of index register r, memory register M, or data B...B from the accumulator with carry. An overflow (carry) sets the carry flip-flop.
ACM	1 0	0 0 1	1 1 1	
ACI	0 0	0 0 1	1 0 0	
	B B	B B B	B B B	
SUr	1 0	0 1 0	S S S	Subtract the content of index register r, memory register M, or data B...B from the accumulator. An underflow (borrow) sets the carry flip-flop.
SUM	1 0	0 1 0	1 1 1	
SUI	0 0	0 1 0	1 0 0	
	B B	B B B	B B B	
SBr	1 0	0 1 1	S S S	Subtract the content of index register r, memory register M, or data B...B from the accumulator with borrow. An underflow (borrow) sets the carry flip-flop.
SBM	1 0	0 1 1	1 1 1	
SBI	0 0	0 1 1	1 0 0	
	B B	B B B	B B B	
NDr	1 0	1 0 0	S S S	Compute the logical AND of the content of index register r, memory register M, or data B...B with the accumulator.
NDM	1 0	1 0 0	1 1 1	
NDI	0 0	1 0 0	1 0 0	
	B B	B B B	B B B	
XRr	1 0	1 0 1	S S S	Compute the exclusive-OR of the content of index register r, memory register M, or data B...B with the accumulator.
XRM	1 0	1 0 1	1 1 1	
XRI	0 0	1 0 1	1 0 0	
	B B	B B B	B B B	
ORr	1 0	1 1 0	S S S	Compute the inclusive-OR of the content of index register r, memory register m, or data B...B with the accumulator.
ORM	1 0	1 1 0	1 1 1	
ORI	0 0	1 1 0	1 0 0	
	B B	B B B	B B B	
CPr	1 0	1 1 1	S S S	Compare the content of index register r, memory register M, or data B...B with the accumulator. The content of the accumulator is unchanged.
CPM	1 0	1 1 1	1 1 1	
CPI	0 0	1 1 1	1 0 0	
	B B	B B B	B B B	
RLC	0 0	0 0 0	0 1 0	Rotate the content of the accumulator left.
RRC	0 0	0 0 1	0 1 0	Rotate the content of the accumulator right.
RAL	0 0	0 1 0	0 1 0	Rotate the content of the accumulator left through the carry.
RAR	0 0	0 1 1	0 1 0	Rotate the content of the accumulator right through the carry.

Input/Output Instructions

INP	0 1	0 0 M	M M 1	Read the content of the selected input port (MMM) into the accumulator.
OUT	0 1	R R M	M M 1	Write the content of the accumulator into the selected output port (RRMMM, RR ≠ 00).

Machine Instruction

HLT	0 0	0 0 0	0 0 X	Enter the STOPPED state and remain there until interrupted.
HLT	1 1	1 1 1	1 1 1	Enter the STOPPED state and remain there until interrupted.

Table 4-1 (Cont'd.)

Program Counter and Stack Control Instructions

Mnemonic	Instruction Code D7 D6	D5 D4 D3	D2 D1 D0	Description of Operation
(4)JMP	0 1 B2 B2 X X	X X X B2 B2 B2 B3 B3 B3	1 0 0 B2 B2 B2 B3 B3 B3	Unconditionally jump to memory B3...B3B2...B2.
(5)JFc	0 1 B2 B2 X X	0 C4 C3 B2 B2 B2 B3 B3 B3	0 0 0 B2 B2 B2 B3 B3 B3	Jump to memory address B3...B3B2...B2 if the condition flip-flop c is false. Otherwise, execute the next instruction sequence.
JTc	0 1 B2 B2 X X	1 C4 C3 B2 B2 B2 B3 B3 B3	0 0 0 B2 B2 B2 B3 B3 B3	Jump to memory address B3...B3B2...B2 if the condition flip-flop is true. Otherwise, execute the next instruction sequence.
CAL	0 1 B2 B2 X X	X X X B2 B2 B2 B3 B3 B3	1 1 0 B2 B2 B2 B3 B3 B3	Unconditionally call the subroutine at memory address B3...B3B2...B2. Save the current address (up one level in the stack).
CFc	0 1 B2 B2 X X	0 C4 C3 B2 B2 B2 B3 B3 B3	0 1 0 B2 B2 B2 B3 B3 B3	Call the subroutine at memory address B3...B3B2...B2 if the condition flip-flop c is false, and save the current address (up one level in the stack). Otherwise, execute the next instruction in sequence.
CTc	0 1 B2 B2 X X	1 C4 C3 B2 B2 B2 B3 B3 B3	0 1 0 B2 B2 B2 B3 B3 B3	Call the subroutine at memory address B3...B3B2...B2 if the condition flip-flop c is true, and save the current address (up one level in the stack). Otherwise, execute the next instruction in sequence.
RET	0 0	X X X	1 1 1	Unconditionally return (down one level in the stack).
RFc	0 0	0 C4 C3	0 1 1	Return (down one level in the stack) if the condition flip-flop c is false. Otherwise, execute the next instruction in sequence.
RTc	0 0	1 C4 C3	0 1 1	Return (down one level in the stack) if the condition flip-flop c is true. Otherwise, execute the next instruction in sequence.
RST	0 0	A A A	1 0 1	Call the subroutine at memory address AAA000 (up one level in the stack).

NOTES:

(1) SSS = Source Index Register DDD = Destination Index Register
These registers are designated A (accumulator—000), B(001), C(010), D(011), E(100), H(101), L(110).

(2) Memory registers are addressed by the contents of registers H and L.

(3) Additional bytes of instruction are designated by BBBBBBBB.

(4) X = "Don't Care."

(5) Flag flip-flops are defined by C4C3: carry (00--overflow or underflow), zero (01--result is zero), sign (10--MSB of result is "1"), parity (11--parity is even).

The microprocessor still does not understand any language except binary numbers. Now, however, a program can be written in a symbolic language and translated into the binary code of MPS because of the one-to-one correspondence between the binary instructions and the mnemonics. This translation could be done by hand, defeating the purpose of mnemonic instructions, or the microprocessor could be used to do the translating for us. By instructing the microprocessor to perform a translation, substituting binary numbers for the alphanumeric characters, a program is generated in the binary code of the microprocessor. This process of translation is called "assembling" a program. The program that performs the translation is called an assembler. A specific mnemonic language for MPS, called MLA (Microprocessor Language Assembler), is described in the MPS User's Handbook.

To illustrate how a program can be written, assembled, and loaded in this manner, consider the following example.

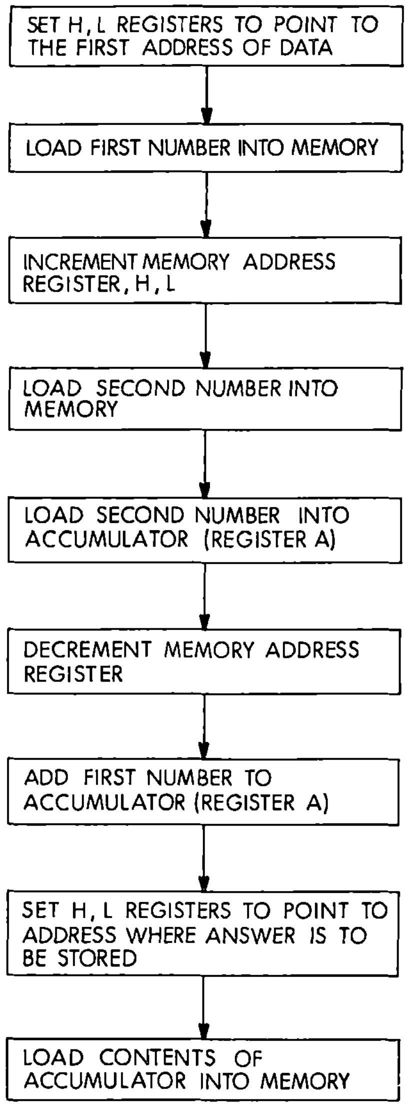
Program Example 1

This is an elementary program which involves adding two numbers from specified memory locations and storing the result in another specified memory location. The sequence of machine operations to do this is shown in Figure 4-1. As illustrated, the memory address registers, designated H and L, must first be set to point to the address where our initial piece of data is to be stored. After the data is loaded into the specified memory address, we must next increment that memory address so that the next piece of data will be stored in the next location. Once the second number is placed into memory, it is then brought out to the accumulator (register A), where it can be operated upon. By decrementing the L register, we point to the address of the first number and bring it out to the accumulator where it is added to the second number. A third memory location is selected to store the result of the addition. A program to accomplish this task, using the MPS instruction set in mnemonic form, is presented.

In this example, the two data numbers will be arbitrarily chosen as 63_{10} and 59_{10}. in binary notation, they would be represented as 00 111 111 and 00 111 011. Since this binary notation is quite confusing, we will henceforth use octal notation to represent the binary numbers which the computer uses (refer back to Chapter 1). Our data numbers may now be written as 077_8 and 073_8, respectively. We have chosen block 21 offset 102 as the memory location where our result will reside.

Mnemonic Code	Description
LLI 100	Load low byte address (offset 100).
LHI 021	Load high byte address (block 21).
LMI 077	Load data into memory (77_8 or 63_{10}).
INL	Increment low byte address register.
LMI 073	Load data into memory (73_8 or 59_{10}).
LAM	Load contents of memory into accumulator.

Mnemonic Code	Description
DCL	Decrement low byte address register.
ADM	Add the contents of memory address 21 100 to the accumulator.
LLI 102	Load low byte address (offset 102).
LMA	Load memory address 21 102 with the contents of accumulator.
HLT	Halt.



SET H, L REGISTERS TO POINT TO THE FIRST ADDRESS OF DATA
LOAD FIRST NUMBER INTO MEMORY
INCREMENT MEMORY ADDRESS REGISTER, H, L
LOAD SECOND NUMBER INTO MEMORY
LOAD SECOND NUMBER INTO ACCUMULATOR (REGISTER A)
DECREMENT MEMORY ADDRESS REGISTER
ADD FIRST NUMBER TO ACCUMULATOR (REGISTER A)
SET H, L REGISTERS TO POINT TO ADDRESS WHERE ANSWER IS TO BE STORED
LOAD CONTENTS OF ACCUMULATOR INTO MEMORY

Figure 4-1. Sequence of Operations

Writing the above program was greatly simplified because mnemonic codes were used for the octal instructions. However, writing down the address of each instruction is clearly an inconvenience. If the coder later adds or deletes instructions, thus altering the memory location assignments of his program, he has to rewrite those instructions whose operands refer to the altered assignments. If the coder wishes to move the program to a different section of memory, he must rewrite the program. Since such changes must be made often, especially in large programs, a better means of assigning locations is needed. The assembler provides this better means.

4.2 THE ASSEMBLER

Program assembly can be achieved by two methods: (a) hand assembly, and (b) machine assembly. Let's discuss both ways of doing the job.

4.2.1 Hand Assembly

Hand assembly begins in the form of mnemonic instructions used by the programmer. Because these instructions will be assembled manually (actually written), an accurate track of the memory locations involved must be maintained by the programmer. Figure 4-2 illustrates a typical form on which a user program might be written. This form is visualized as a "copy" of memory; therefore, if an instruction requires more than one memory location or byte, lines must be skipped to allow for the entry of code in these locations.

For example, Figure 4-3 illustrates how our previous addition program would be written. By referring to the MPS Programmer's Reference Card or to Table 4-1, the programmer can determine the number of bytes required for each instruction and can skip the proper number of lines to allow for these extra memory locations. Since INL, LAM, DCL, ADM, LMA, and HLT require only one byte of memory, no extra locations need be allowed to assemble these instructions. Multiple byte instructions, however, such as LLI, LHI, LMI, and JMP require two or three bytes and must have extra locations set aside for them.

In the case of LLI 100, byte one of the instruction contains the machine code for LLI, while byte two contains the binary code which represents the octal 100 or the low byte memory address.

After constructing a form such as that shown in Figure 4-2, the programmer should write his program in the usual way, allowing extra memory locations for instructions which require more than one byte for assembly, as illustrated in Figure 4-3. Using the card, the user can then determine the octal code for each instruction and write it in the corresponding location in the table. For example, let us assume we have decided to locate our program in block 21 of memory, starting at location 21 000. As illustrated in Figure 4-3, we would write our program allowing extra locations for those instructions which require them. The next step is to look up the octal code for

Block	Offset	Mnemonic	Assembly Code
21	000		
21	001		
21	002		
21	003		
21	004		
21	005		
21	006		
21	007		
21	010		
21	011		
21	012		
21	013		
21	014		
21	015		
21	016		
21	017		
21	020		
21	021		
21	022		
21	023		
21	024		
21	025		
21	026		
21	027		
21	030		

Figure 4-2. Blank Programming Form

Block	Offset	Mnemonic	Assembly Code
21	000	LLI 100	
21	001		
21	002	LHI 021	
21	003		
21	004	LMI 077	
21	005		
21	006	INL	
21	007	LML 073	
21	010		
21	011	LAM	
21	012	DCL	
21	013	ADM	
21	014	LLI 102	
21	015		
21	016	LMA	
21	017	JMP 0#100	
21	020		
21	021		
21	022		
21	023		
21	024		
21	025		
21	026		
21	027		
21	030		

Figure 4-3. Typical Addition Program Illustrating How Locations Are Skipped for Multiple Byte Instructions

each instruction and enter it on the proper line. Since LLI 100 loads the L register with the contents of byte two of the instruction (location 21 001 in this case), that byte should contain a 100. Upon completion, we have "assembled" our program by hand. The completed programming form is shown in Figure 4-4.

Although this method appears to be "the way to go" because of its relatively direct and simple approach, it has certain inherent disadvantages, not the least of which is maintaining the instruction listing. An even easier--indeed the ultimate, accepted solution--exists in the machine assembler.

4.2.2 Machine Assembler

We are now entering into the "meat" of programming a processor, and at this point, the reader should thoroughly understand the foregoing discussions and pay more than just cursory attention to the information about to come. The keys to the operation of machine assembly are <u>location assignment</u> and <u>symbolic addressing</u>.

4.2.2.1 Location Assignment

As in the previous program example, most programs are written in successive memory locations. If the coder assigned an absolute location to the first location, the assembler could be told to assign the next instructions to the following locations in order. In programming MPS, the initial location is denoted by a precedent asterisk (*). The assembler maintains a current location counter by which it assigns successive locations to instructions. The asterisk causes the current location counter to be set to the value following the asterisk.

Each location in memory of the microprocessor contains a single byte (8 bits of binary information). Some instructions require several bytes of binary data to fully define the operation. As mentioned previously, the largest address may be 14 bits long; therefore, we need 2 bytes (16 bits) to define a memory reference. The assembler knows which instructions require one, two, or three bytes of binary data and updates the current location counter to accommodate the extra memory locations needed.

4.2.2.2 Symbolic Addresses

The coder does not at the outset know which locations he will use for temporary storage and results of arithmetic operations. Nor does he know the exact locations of instructions he may wish to branch to until all his coding is complete. Let us consider, for example, the following program which multiplies 18_{10} by 36_{10} using successive addition:

```
400     XRA          /clear the accumulator
401     LBI -22      /set up a tally equal
                     /to -18₁₀ to count the
                     /additions of 36
```

Block	Offset	Mnemonic	Assembly Code
21	000	LLI 100	066
21	001		100
21	002	LHI 021	056
21	003		021
21	004	LMI 077	076
21	005		077
21	006	INL	060
21	007	LMI 073	076
21	010		073
21	011	LAM	307
21	012	DCL	061
21	013	ADM	207
21	014	LLI 102	066
21	015		102
21	016	LMA	370
21	017	JMP	104
21	020		100
21	021		000
21	022		
21	023		
21	024		
21	025		
21	026		
21	027		
21	030		

Figure 4-4. Typical Program with Corresponding Assembled Octal Code

```
     403       LCI 44          /set up a register with
                               /the value of 36₁₀
     405       ADC             /add 36 to the accumulator
     406       INB             /increment tally
     407       JFZ 405         /add another 36 if tally is not
                               /zero
     412       HLT             /stop after 18 times
```

In this program, the coder must count the number of locations used after the initial instruction in order to assign the correct value to the JFZ instruction. Actually, this is not necessary, because he may assign symbolic names (a symbol followed by a comma is a symbolic address) to the locations to which he must refer, and the assembler will assign address values for him. The assembler maintains a symbol table in which it records the octal values of all symbolic addresses. With symbolic address name tags, the program is as shown below:

```
               *400
               XRA
               LBI -22
               LCI 44
     MULT,     ADC
               INB
               JFZ MULT
               HLT
               $
```

NOTES: (1) The dollar sign is the terminal character for the assembler.

(2) The comma after a symbol (e.g., MULT,) indicates to the assembler that the symbol is a symbolic address.

4.2.2.3 Symbolic Coding Conventions

Any sequence of letters (A, B, C, ..., Z) and digits (0, 1, ..., 9) beginning with a letter and terminated by a delimiting character (see Table 4-2) is a symbol. For example, the mnemonic codes for the MPS instructions are symbols for which the assembler retains octal equivalents in a permanent symbol table.

User-defined symbols may be of any length; however, only the first four characters are retained by the assembler and any additional characters are ignored. (Symbols which are identical in their first four characters are considered identical.)

Any sequence of digits followed by a delimiting character forms a number. The assembler will accept numbers which are octal, decimal, or hexadecimal. The "radix" or number base--e.g., n_8 (octal); n_{10} (decimal); n_{16} (hexadecimal)--is initially set to octal and remains octal unless otherwise specified. The pseudo-instruction DEC or HEX may be inserted in the coding to instruct the assembler to interpret all numbers as decimal or hexadecimal until the next occurrence of the pseudo-instruction OCT, DEC, or HEX in the coding. Pseudo-instructions are so-called because they are not part of the MPS regular instruction set. They are special instructions designed to interpret (in this case) different numbering schemes within the MPS system. These pseudo-instructions affect all numbers included in the symbolic program, including those preceded by an asterisk to denote change of origin.

The special characters in Table 4-2 are used to specify operations to be performed by the assembler upon symbols or numbers in MPS symbolic programs. Each symbol or number written in an MPS program must represent an 8-bit binary value or a 14-bit binary address in order to be interpreted by the assembler.

Table 4-2

Special Characters for the MPS Symbolic Language

Character		
Keyboard	Name	Use
Space Bar	space (non-printing)	Combine symbols or numbers (delimiting)
CTRL/I	tab (non-printing)	Combine symbols or numbers or format the symbolic tape (delimiting)
Return	carriage return (non-printing)	Terminate line (delimiting)
+	plus	Combine symbols or numbers
-	minus	Combine symbols or numbers
,	comma	Assign symbolic address
=	equals	Define parameters
*	asterisk	Set current location counter
;	semi-colon	Data separator (delimiting)
$	dollar sign	Terminate pass (delimiting)
.	point	Has value equal to current location counter
/	slash	Indicates start of a comment
&	ampersand	Logical AND
!	exclamation point	Logical OR
↑	up-arrow	High byte selection
<>	angle brackets	Used to enclose numeric representation of an ASCII character
#	pound sign	Block-offset separator

• The comma after a symbol in a line of coding (e.g., MULT, ADC) indicates to the assembler that the value of symbol MULT is the address of the location in which the instruction ADC is stored. When an instruction that references MULT (e.g., JFZ MULT) is encountered, the assembler supplies the correct address value for MULT. (Care must be taken that a symbolic address is never used twice in the same program and that all locations referenced by memory reference instructions are identified somewhere in the program.)

- The space and tab are used to delimit or indicate the end of a complete symbol or number. The space and tab similarly delimit the mnemonic from the symbolic address.

```
JFZ MULT   or   JFZ MULT
   ↑               ↑
   └space          └CTRL/I
```

- The carriage return is used to terminate a line of coding.

- The assembler will recognize the arithmetic symbols + and - in conjunction with numbers or symbols, thereby enabling "address arithmetic." For example, the instruction JMP BEG+1 will cause the microprocessor to execute the instruction in the next location after BEG. The numbers specified in such instructions are subject to the pseudo-instructions DEC, HEX, and OCT; therefore, the number is interpreted as an octal number unless otherwise directed. If you never use a number larger than 7, then it will be the same no matter what radix you're in at the time.

- The decimal point, or period, is a character which is interpreted by the assembler as the value of the current location counter. This special symbol can be used as the operand of an instruction; for example, the instruction JMP .-1 causes the microprocessor to execute the preceding instruction. Relative addressing with "." should be used only if available symbol table space is very tight. When instructions are added or deleted during the "debugging" phase, one must be very careful to check all relative addressing for possible errors.

- The equal sign is used to define symbols. This character is used to replace an undefined symbol with the value of a known quantity. When used, the equal sign must delimit the symbol it is defining. For example, MDP = 100; i.e., MDP, previously not assigned a value, is now assigned the value of 100.

- The slash is used to insert comments and headings in the code; e.g., JMP MDP/JUMP TO 100. The slash is used here as the comment that MDP = 100.

- The dollar sign, as previously noted, is a terminal character for the assembler itself. When this character is encountered, the assembler stops accepting input and terminates the assembly pass.

- Two logical operators or special symbols have been implemented for use with the assembler. These are "&," indicating a logical AND operation, and "!," indicating a logical OR operation. Logical and arithmetic operators can be mixed in the same expression with evaluation proceeding from left to right without precedence. If initially RST has the value 005, then RST!10 would have the value of 015. Similarly, RST&10 would have the value 000.

```
Example:   Logical OR:   RST = 00 000 101
                          10 = 00 001 010
                               ----------
                               00 001 111 = binary 015

           Logical AND:  RST = 00 000 101
                          10 = 00 001 010
                               ----------
                               00 000 000 = binary 000
```

- The high byte-selection operator ↑ (up-arrow) is a post-operator used to indicate selection of the high byte of the entire expression (from the beginning) which the ↑ follows. As explained earlier, addresses are expressed by 14-bit binary numbers and address arithmetic is carried out in 15-bit signed numbers. Therefore, two bytes are used to define a memory address. The most significant bits of the address are found in the high byte. By selecting the high byte, this operator performs an effective signed divide by 256. For example, if the following assignments have been made

$$A = 400_8$$
$$B = 377_8$$

the expression

A + B + 10↑+1 results in 003.

As shown below, A + B would be equal to 01,377 in high and low 8-bit bytes. When we add 10, the expression becomes 02,007. The high-byte select operator (↑) replaces the low byte with the high byte and zeros the high byte, and the result is 00,002. Adding 1 yields the final result.

		High Byte		Low Byte			
$A = 400_8$	=	000	001	00	000	000	
+ $B = 377_8$	=	000	000	11	111	111	
		000	001	11	111	111	= $01,377_8$
+ 10_8	=	000	000	00	001	000	
		000	010	00	000	111	= $02,007_8$

- The block-offset operator # is used to indicate an address in terms of its block number (high byte) and an offset (low byte) within that block. For example, an origin (*) pseudo-instruction of the form

*2#200

sets the location counter to location 200 within block 2. This would be equivalent to octal location 1200.

The example

JMP 35#377

sets the location counter to block 35 offset 377 (octal 16777). As covered earlier in this guide, the conversion from block-offset to octal notation proceeds as in the following. The bit pattern of 35#377 can be represented by:

```
011 101 11 111 111
 3   5  3   7   7
```

The effect of the # operator is to shift the block number 35 to the right, causing the following displacements and conversion:

```
01 110 111 111 111
 1  6   7   7   7
```

- The <u>angle brackets</u> (<>) are used to enclose a numeric representation of an ASCII character in a TEXT pseudo-instruction expression. For example,

  TEXT /A/<15>

  would be assembled into 101 (the numeric value of the ASCII character A) and 15 (the octal code for a carriage return) would be assembled without conversion. The slashes in this case are delimiters. The TEXT pseudo-instruction is explained fully on pages 6-10 of the <u>MPS User's Handbook</u>.

These characters and conventions will be used throughout the remainder of this guide to code programs in MLA, the symbolic language of MPS. Thus, all examples given may be directly punched on paper tape as described in Chapter 5 and assembled by the procedure described in Chapter 6 of the <u>MPS User's Handbook</u>.

CHAPTER 5

ARITHMETIC OPERATIONS

5.1 INTRODUCTION

The instructions for MPS may be used to perform the basic arithmetic operations within the limits of the machine to represent the necessary numbers. That is, numbers may be added unless the sum exceeds 255_{10} or 377_8. When a sum exceeds the size of the accumulator, "overflow" occurs and incorrect answers result. This condition can usually be detected by checking the value of the carry flip-flop by means of the JTC or JFC instruction, for example.

The following instructions will add numbers and check for overflow, halting the program if the carry is 1.

```
ADDN,     XRA        Clear accumulator and carry flip-flop.
          ADB        Add contents of B register to accumulator.
          ADC        Add contents of C register to accumulator.
          JFC .+4    If carry = 0, go to LDA.
          HLT        Halt if carry ≠ 0.
          LDA        Store accumulator (sum) in D register.
           .                     ↓
           .         Resume rest of program.
           .
```

Since the carry is initially cleared in the above example by the XRA instruction, a carry value equal to 1 (true) is an indication that the sum of the contents of registers B and C is too large to be represented by the 8-bit accumulator alone. The microprocessor will halt if the overflow is detected with the actual sum in the combined 9 bits of the accumulator and the carry flip-flop.

5.2 ARITHMETIC OVERFLOW

Since MPS regards the numbers 0 through 177 as positive numbers and the numbers 200_8 through 377_8 as negative numbers, the addition of two positive numbers could result in either a positive or negative number, depending upon the size of the numbers added. Arithmetic overflow is said to occur whenever two positive numbers add to form a negative number, as shown in the following example:

```
      071₈   (a positive number)
    + 143₈   (a positive number)
      ----
      234₈   (considered a negative number by MPS)
```

Likewise, two negative numbers could be added to yield a positive number, as in the following example:

```
            275₈   (-103₈)
          + 261₈   (-117₈)
            ----
  Carry → 1 156    (considered a positive number by MPS)
```

5.3 CONDITION FLIP-FLOPS

Because of situations like those illustrated in the two preceding examples, the coder must consider the size of the numbers used in programmed arithmetic operations. If the coder suspects that overflow may occur in the result of an arithmetic operation, he should follow such an operation by a set of instructions to correct the error or at least to indicate that such an overflow occurred. To assist the MPS user, there are four condition flip-flops which may be affected by execution of an instruction. These condition flip-flops and their truth status are as follows:

Meaning	Truth Status
Carry (C)	Overflow or underflow.
Zero (Z)	Result is zero.
Sign (S)	Most significant bit of result is set.
Parity (P)	Number of bits set in the result is even.

All the condition codes (flip-flops) are tested by the conditional instructions JT_n, JF_n, CT_n, CF_n, RT_n, RF_n, where n represents the condition flip-flops C, Z, S, and P. The possible outcomes outlined below may be used to test for arithmetic overflow.

Sign of Numbers Added	Overflow and Carry Value
Positive + Positive	May result in negative sum; no change in carry.
Positive + Negative	No overflow possible; carry value is ignored.
Negative + Negative	May result in positive sum; carry is always set regardless of the sign of the result.

The following program coding uses these above facts, assuming an initially cleared carry to quickly determine the sign of the sum of two unknown quantities, B and C.

		Result of Adding Only Bit 7 of B to All of C	
Sign of B	Sign of C	Carry Value	Bit 7 of AC
Positive	Negative	0	1
Negative	Positive	0	1
Positive	Positive	0	0
Negative	Negative	1	0

```
/Coding to Add Two Numbers
/Testing for Arithmetic Overflow
STRT,   XRA          /clear AC and carry
        ADB          /add B to accumulator
        NDI 200      /mask out all but bit 7
        ADC          /add C to bit 7 of B
        JTC BNEG     /carry true implies both
                     /are negative
        RAL          /rotate bit 7 into carry
        JTC OPSN     /bit 7 true implies opposite signs
BPOS,   LAB          /bit 7 false, both positive
        ADC          /if two positive numbers add
        JTS PERR     /to form a negative number
                     /JMP to error routine
        LDA          /otherwise, store sum in D
        HLT
PERR,   .            /routine to signal arithmetic
        .            /overflow of positive numbers
NERR,   .            /routine to signal arithmetic
        .            /overflow of negative numbers
OPSN,   LAB          /if B and C are of opposite
        ADC          /signs, the addition cannot
        LDA          /result in overflow
        HLT
BNEG,   XRA          /if two negative numbers
        ADB          /add to form a positive
        ADC          /number, JMP to error
        JFS NERR     /routine, otherwise, store
        LDA          /in register D
        HLT
```

5.4 SUBTRACTION

Subtraction in MPS is accomplished by negating the subtrahend (replacing it by its two's complement) and then adding it to the minuend, ignoring the overflow, if any.

The following example shows the contents of the accumulator for each step of the subtraction process. Assume register B contains 46_8 and register C contains 37_8 before execution of the program.

Subtraction Program	Resulting Contents	
	Carry	Accumulator
XRA	0	00 000 000 (000)
ADB	0	00 100 110 (046)
SUC	0	00 000 111 (007)

Note that the number to be subtracted is complemented and incremented by MPS before it is added to the accumulator, i.e.,

```
       37₈             =  00 011 111
       2's complement  =  11 100 001
       Add 46₈         =  00 100 110
                          ----------
         Sum           = [1]00 000 111
                          ↑  0   0   7
 ┌────────────────────────┘
 └Carry bit is ignored by the hardware.
```

5.5 MULTIPLICATION AND DIVISION

A previous example illustrated the method of performing multiplication with the basic MPS instructions, namely by repeated addition. Obviously, multiplication by this method is also subject to the limitation of overflow. The largest positive number which can be directly represented is 127_{10} or 177_8.

Multiplication by repeated addition will properly handle positive and negative numbers within the limits of positive and negative arithmetic overflow. For example, 377_8 is the MPS representation for -1. If it is multiplied by itself, the answer should be +1. In other words, adding 377_8 to itself 377_8 times should leave (after carries from the most significant bit) the accumulator equal to 1.

```
                          377    1st    ⎫
                        + 377    2nd    ⎪
                    ↗1    376           ⎪
                        + 377    3rd    ⎪
                    ↗1    375           ⎪
                           .            ⎪
Disregarded                .            ⎬  If the reader wishes, he
  Carries                  .            ⎪  can verify result by doing
                           .            ⎪  this exercise in binary
                    ↘1    003           ⎪  notation.
                        + 377   376th   ⎪
                    ↘1    002           ⎪
                        + 377   377th   ⎪
                    ↘1    001           ⎭
```

Thus, successive addition will work properly as a method of multiplying negative as well as positive numbers in the MPS.

Similarly, division could be performed by repeated subtraction. This method of division could be used to obtain a quotient and remainder, because only whole numbers are directly represented in MPS. Multiplication and division can also be performed through use of the floating point packages (available through DECUS, The Digital Equipment Corporation User's Society).

5.6 DOUBLE-PRECISION ARITHMETIC

Two memory locations (16 bits) are used to express double-precision numbers. Using these 16 bits allows the representation of numbers in the range $-32,767_{10}$ to $+32,767_{10}$. The following program adds two double-precision numbers, obtaining a double-precision result.

```
          *400
DBLA,     XRA            /clear AC and CARRY
          ADC            /low byte of 1st number
          ADE            /low byte of 2nd number
          LLA            /save low byte of result in L
          LAB            /get high byte of 1st number
          ACD            /add high byte of 2nd number and
                         /carry
          LHA            /save high byte of result in H
          HLT
          $
```

DBLA is the symbol representing the operation taking place.
It serves as an entry point to the routine.

Note that if the addition of register C and register E produces a carry, it will appear in the carry flip-flop. The LLA and LAB instructions do not affect the condition codes; therefore, the carry flip-flop would still reflect its setting as a result of the addition of C and E. The ACD instruction is a special MPS instruction which says add the contents of register D and contents of the carry flip-flop to the contents of the accumulator and store the result in the accumulator. This add with carry instruction simplifies the coding of multiple-precision arithmetic operations.

A similar procedure is followed to subtract two double-precision numbers. The following program illustrates the technique.

```
          *400
DBLS,     XRA            /clear AC and CARRY
          ADC            /low byte of 1st number
          SUE            /subtract low byte of 2nd number
          LLA            /save low byte of result
          LAB            /get high byte of 1st number
```

```
                SBD             /subtract high byte of 2nd number
                                /and the contents of the carry flip-flop
                LHA             /save the high byte of result
                HLT             /stop
                $
```

DBLS is the symbol representing the operation taking place. It would be used in a JMP or a CAL statement to gain entry to the routine.

As in the double-precision add routine, SBD is a special MPS instruction which facilitates multiple-precision arithmetic. SBD says subtract the contents of register D and the contents of the carry flip-flop from the contents of the accumulator and store the result in the accumulator.

5.7 POWERS OF TWO

In the decimal number system, moving the decimal point right (or left) multiplies (or divides) a number by powers of ten. In a similar way, shifting a binary number one bit left or one bit right multiplies (or divides) by powers of two, respectively. However, because of the logical connection between the accumulator and the carry flip-flop, care must be taken to use the appropriate shift instructions so that unwanted digits do not get shifted into the accumulator. Multiplication by powers of two is performed by shifting the accumulator left; division is performed by shifting the accumulator right. Multiplication and division by this method are subject to the limitation of 8-bit numbers (unless double precision is used). That is, significant bits shifted out of the accumulator by multiplication or division are lost and incorrect results are therefore obtained. For example, the following program multiplies a number by 8 (2^3).

```
                *400
        CUBE,   LAB             /get number into AC
                NDI 340         /test high 3 bits for non-zeroes
                JFZ ERR         /if not zero, jump to error routine
                LAB             /restore number
                RLC             /rotate left
                RLC             /rotate left
                RLC             /rotate left
                LBA             /store result in register B
                HLT
                $
```

If B originally contained 011_8, it would contain 110_8 after execution of this routine.

CHAPTER 6

SUBROUTINES, LOOPING, BRANCHING

6.1 WRITING SUBROUTINES

We have referred to the program counter previously as a 14-bit register containing the address of the next instruction to be executed. Actually, MPS contains a ring buffer (referred to as a pushdown stack) made up of eight 14-bit registers. Another 3-bit register acts as a pointer to one of the 14-bit registers, and this register is called the current program counter. When a JMP instruction is encountered in the program execution, the address you wish to jump to replaces the current program counter and the pointer is unchanged. Therefore, there is no link between the new program counter and the old program counter after a JMP occurs.

Included in the MPS instruction set, given in Table 4-1 of this guide and in Chapter 3 of the MPS User's Handbook, is the instruction CAL (call to subroutine). This instruction is a modified JMP command which makes return to the point of departure from the main program possible. The CAL instruction automatically increments the 3-bit register that points to the current program counter before modifying the 14-bit registers. Now, when the jump is taken to the new program counter address, the old program counter is not destroyed. In this manner, subroutines could be nested up to seven levels. To return to the previous program counter before the last executed CAL instruction, the coder need only terminate the subroutine with a RET instruction. The RET instruction decrements the program counter pointer, and control returns to the location after the CAL instruction. The JMP, CAL, and RET instructions can be unconditional or conditional based on the status of one of the four condition flip-flops. The following simple program illustrates the use of a subroutine to increment a double-precision number contained in registers H and L (the H and L registers are used as address pointers for memory reference instructions).

```
                        (Main Program)
PMSG,     XRA               /clear AC
          ADM               /add next character in
                            /memory to the accumulator
          JTZ EXIT          /jump out; if zero
          OUTO              /display it
          CAL INHL          /jump to subroutine
          JMP PMSG          /return here from subroutine
EXIT,     .
          .
          .
          .
                        (Subroutine)
INHL,     INL               /increment register L
          RFZ               /return to main program if not
                            /zero (no overflow)
          INH               /otherwise, increment register H
          RET               /and then return
          $
```

Notice that there are two return statements in the subroutine, RFZ and RET. The first one is a conditional return based on the status of the zero flip-flop. If the contents of register L goes to zero after incrementing, the conditional return will not be taken and the next instruction, INH, will be executed. Then the unconditional RETurn will be taken to get back to the main program.

Another way to call a subroutine in MPS is with the RST instruction. The restart instruction can be used as a one-byte unconditional call to one of eight specified locations in the first 64 bytes of memory. Each of these eight locations can be used as the starting address of a subroutine. The following program increments the contents of memory using one of the general purpose registers and the RST instruction.

```
          *0
          LBM               /load B with contents of memory
          INB               /increment B
          LMB               /load memory with new contents
          RET               /return to main program

          *4#0
          LMI -10           /set counter in memory to -10₈
MAIN,     ADC               /add register C to accumulator
```

```
RST               /call subroutine at location 0
                  /to increment counter in memory
JFZ MAIN          /loop if counter is not zero
 .
 .
 .
 .
 .
```

The preceding example illustrates an important point. A conditional flip-flop set by the INB instruction within the subroutine is tested by the JFZ instruction after returning to the main program.

An interesting modification to the previous program is achieved by defining a "new operation" INCM by including in the coding the following statement:

```
OPDEF INCM; RST; 0
```

NOTE: All OPDEF statements must appear in the coding before any other user-defined symbols. (See Chapter 6 of the MPS User's Handbook and Appendix D of this guide for more information on the use of OPDEF.)

6.2 INSERTING COMMENTS AND HEADINGS

Because programs very seldom are written to be used only once, the coder should strive to document his procedure and coding as much as is reasonably possible. There are many instances where changes or corrections must be made by people unfamiliar with a program; or, more commonly, the original coder is asked to modify a program months after his original effort. In both cases, the success of the attempt to change the program depends largely upon the documentation provided by the original coder. A complete and accurate flowchart is the first form of documentation. It is extremely important to document modifications made in the program by incorporating these changes in the flowchart as well.

Many times it is desired to include headings and dates to identify a program within the actual coding of the symbolic program. It is often helpful to add comments to simplify the reading of a symbolic program and to indicate the purpose of any less than obvious instruction. MPS coding allows comments and headings to be inserted simply by preceding any commentary with a slash (/).

The following example illustrates the method used to insert commentary in an MPS program. It also illustrates the use of a rotate or shift instruction. The program takes a binary number stored in memory and counts the number of non-zero bits. Although the program may have no useful application, it does serve to familiarize the reader with the structure of the accumulator and the carry flip-flop and the action of a rotate instruction. The flowchart (Figure 6-1) and comments will aid the reader in understanding the program.

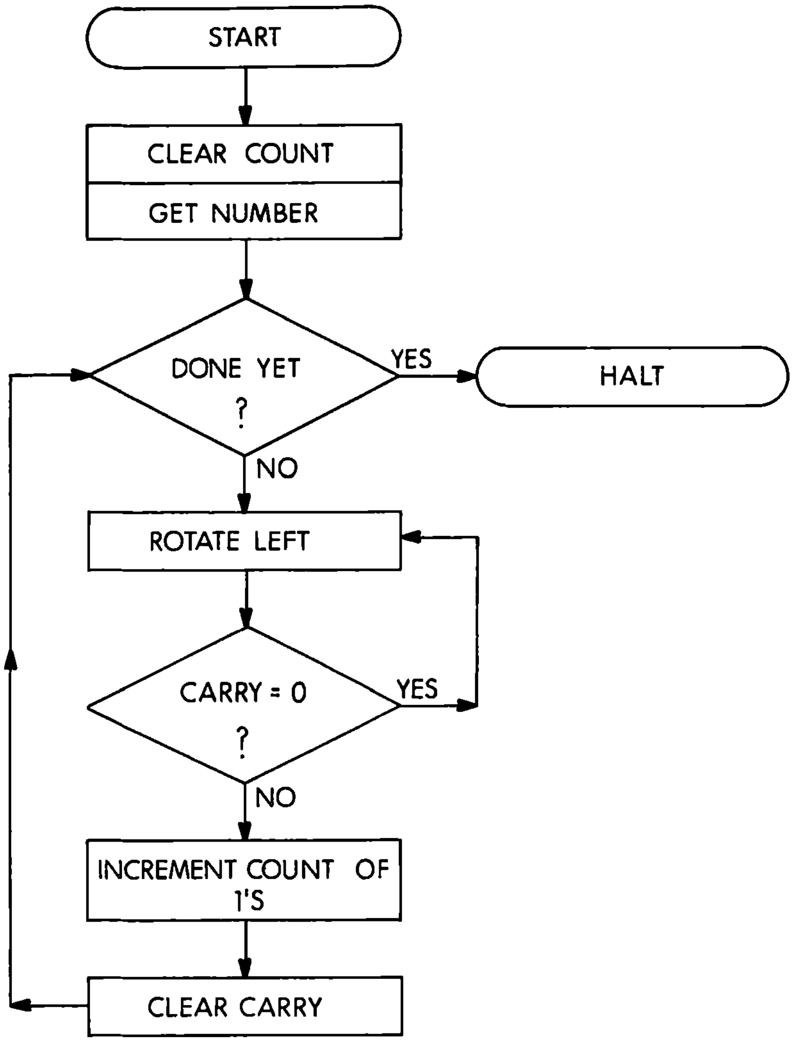


Figure 6-1. Count the Binary One's Program

```
         *4#0
         XRA          /clear AC and carry
         LBA          /set count to 0
         ADM          /add the number from memory
LOOP,    JFZ ROTA
         HLT          /stop if the number is zero
ROTA,    RAL          /rotate one bit into carry
         JFC .-1      /rotate again if bit was 0
         INB          /increment count of 1's
         ORA          /clear carry and set other
                      /condition flip-flops
         JMP LOOP
         $
```

The following points should be observed in the preceding example:

1. The number was checked to see that it was non-zero to begin with. If this check were not made, a zero number would be rotated endlessly by the remaining instructions in the program.

2. Because a rotate right instruction (RAR) would transfer the bits into the carry flip-flop just as the RAL instruction does, either could be used in the above program. Both instructions use a circular shift of the accumulator and carry bits.

3. Because the carry bit is rotated into the accumulator by the rotate instructions, the carry must be cleared each time a 1 is rotated into it.

4. The rotate instructions only affect the carry condition flip-flop. To test for a zero accumulator after rotating, it was necessary to use the ORA instruction. ORA is a very useful instruction and in this program, it served two functions: clearing the carry flip-flop and setting the other condition codes without altering the contents of the accumulator.

6.3 LOOPING A PROGRAM

As many of the examples given have already shown, the use of a program loop, in which a set of instructions is performed repeatedly, is common coding practice. Looping a program is one of the most powerful tools at the coder's disposal. It enables him to perform similar operations many times using the same instructions, thus saving memory locations because he need not store the same instructions over and over. Looping also makes a program more flexible because it is relatively easy to change the number of loops required for differing conditions by resetting a counter. It is good to remember that looping is little more than a jump to an earlier part of the program; however, the jump is usually conditioned upon changing program conditions.

There are basically two methods of creating a program loop. The first method is using an INr instruction to count the number of passes through the loop. INr is usually followed by a JFZ (jump if zero false) instruction to the beginning of the loop. This technique is very efficient when the required number of passes through the loop can be readily determined.

The second technique is to use the conditional jump instructions to test conditions other than the number of passes which have been made. Using this second technique, the program is required to loop until a specific condition is present in the accumulator or carry flip-flop, rather than until a predetermined number of passes are made.

To illustrate the use of an INr instruction in a program loop situation, consider the following program, which simply sets the contents of all addresses from 2#000 to 2#377 to zero.

```
*4#0
XRA             /clear AC
LLA             /set register L to zero
LHI 2           /set register H to 2
LMA             /clear next memory location
                /pointed to by H and L
INL             /bump memory pointer
JFZ .-2         /if not zero; repeat
HLT             /else, halt
$
```

Several points should be carefully noted:

1. The first three instructions initialize the loop, but are not in the loop.
2. The L register is used as the count of the number of passes and also serves as a pointer to the memory location being zeroed. After loca- 2#377 is cleared, the L register goes to zero, and the program continues through the JFZ instruction without branching and executes the HLT instruction. On each previous occasion, it executed the JFZ instruction.
3. Every time a memory reference instruction is used in a program, the H and L registers must be set up to the desired address.

The following program outlines a conditional test to create a loop. The program will search all of memory to find the first occurrence of the octal number 234.

```
          OPDEF           INHL; RST!10; 0
          *10
          INL             /subroutine to increment memory pointer
          RFZ             /called via a restart instruction
          INH
          RET

          *37#0
BEGN,     XRA
          LLA             /initialize memory pointer to zero
          LHA             /offset and zero block
LOOP,     LAM             /get next number into AC
          CPI 234         /compare to 234
```

```
                JTZ EXIT        /if equal, exit from loop
        NEXT,   INHL            /else, increment pointer
                JMP LOOP        /and loop
        EXIT,   HLT
                $
```

The previous example could be utilized in a debugging phase where you are searching for the occurrence of a particular instruction or value. It is interesting to note that the program will search itself as well as all other core memory locations. Also, notice the following points with regard to the example:

1. Since the search would stop when the 234 is reached inside the program, the program should begin near the highest available memory locations.

2. The program could be restarted at location NEXT in order to find the next occurrence of 234_8 after the program had halted.

3. The search time could be reduced by preloading one of the general purpose registers with the search number and then doing a CPr rather than a CPI within the loop. This would save 12 μs for every pass through the loop.

6.4 PROGRAM DELAYS

Because the development of a microprocessor was primarily sparked by a desire for speed in performing control functions, it seems inconsistent and self-defeating to slow the MPS down with program delays. However, there are many occasions when a microprocessor must be told to slow down or to wait for further information. This is because most peripheral equipment, and certainly the human operator, are very much slower than the microprocessor program. A temporary delay may be introduced into the execution of a program when needed by causing the microprocessor to enter one or more futile loops in which it must traverse a fixed number of times before jumping out. It is often necessary to have a microprocessor perform a temporary delay while a peripheral device is processing data to be submitted to the MPS. The delays can be accurately timed so as not to waste any more processor time than necessary.

The following is a microprocessor delay routine using the INB instruction for an inner loop and the INC instruction for an outer loop. The reader should remember when analyzing the example that the MPS represents only positive numbers up to 177_8 or 127_{10}. Therefore, the microprocessor counts up to 127_{10} and then continues to count, starting at the next octal number 200_8, which the microprocessor interprets as -128_{10}. Successive increments of this number will finally bring the count to zero. Thus, a register could be used to count 256_{10} iterations using an INr instruction.

```
                (Main Program)
           .
           .
           .
           XRA          /clear AC
           LCA          /set register C to zero
           LBA          /set register B to zero
OUTR,      INB          /inner
           JFZ .-1      /loop
           INC
           JFZ OUTR
           .
           .
           .
           .
           .
```

The inner loop consists of an INB instruction with an execution time on the MPS of 20 μs (a microsecond is 10^{-6} seconds) and a JFZ instruction with an execution time of 44 μs if the jump is taken and 36 μs when it falls through. Therefore, the inner loop takes 64 μs for one pass, and each time it is entered, the program will traverse it 256_{10} times before leaving. This means that a delay of 16.376 milliseconds (milliseconds is 10^{-3} seconds) has occurred. If, as in the example above, the outer loop is also traversed 256_{10} times, a total delay of 2.564 seconds would be generated. For any given purpose, a desired delay from milliseconds to seconds can be obtained precisely by varying the initial contents of B and C.

A second type of delay, which waits for a device response, is not a timed delay but causes the microprocessor to wait until it receives a response from an external device. This is discussed more fully in the section on Input/Output Coding.

6.5 PROGRAM BRANCHING

Very few meaningful programs are written which do not take advantage of the microprocessor's ability to determine the future course the program should follow based upon intermediate results. The procedure of testing a condition and providing alternative paths for the program to travel for each of the different results possible is called branching a program. We have already used some of these conditional instructions in the previous examples. There are four condition codes that may be tested: carry, zero, sign, and parity. Each condition may be tested for a true state--the flip-flop is set to a one; or for a false state, the flip flop is clear (0). We can jump, call a subroutine or return from a subroutine on any of the above conditions. A typical example of a conditional jump would be a program to compare B and C to switch them if C is larger than B.

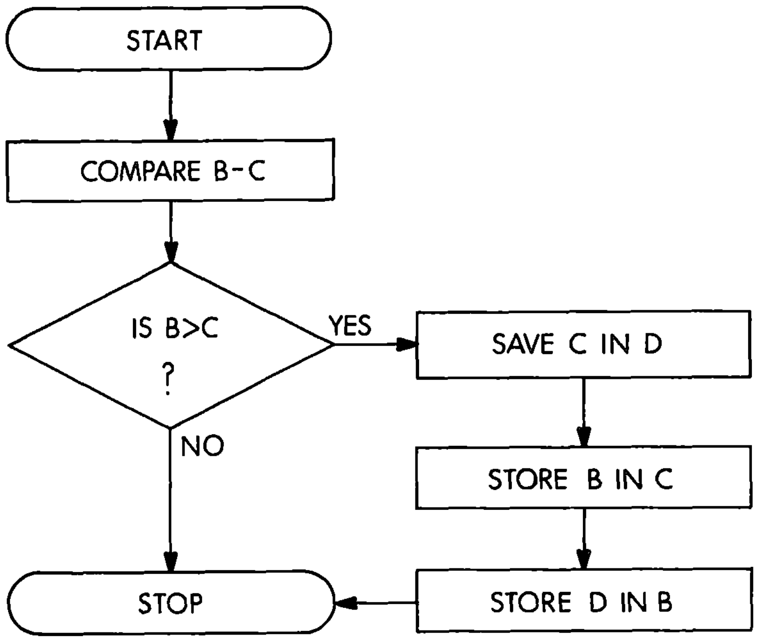


Figure 6-2. Comparing Two Numbers

```
          *4#0
TEST,     XRA
          ADB           /add B to accumulator
          CPC           /compare C to B
          JFC STOP      /STOP if B is not greater than C
          LDC           /otherwise switch
          LCB           /B and C, using D
          LBD           /for temporary storage
STOP,     HLT
          $
```

In MPS, the compare instructions are all performed with one of the values being compared to the contents of the accumulator. Therefore, the first step in the examples was to get B into the accumulator. The CPC instruction caused the zero flip-flop to be set (B = C), the carry flip-flop to be set (B < C), or both zero and carry to be cleared (B > C). If B is less than or equal to C, the JFC instruction will be satisfied and the program will proceed to switch the values of B and C. The concepts illustrated by the above example can be included in a larger program that will take a set of elements and arrange them in increasing order.

CHAPTER 7

INPUT/OUTPUT CODING

7.1 INTRODUCTION

Coding a microprocessor to do calculations is of little use unless there is some means of obtaining the result of the calculations from the machine. In most applications, it is also necessary to supply the microprocessor with data before calculations may be performed. A coder must be able to translate information efficiently between the microprocessor and the peripheral devices that supply input or serve as a means of output.

There are two methods for the transfer of information between input/output (I/O) devices and MPS. The first method is programmed transfer, in which instructions to accept or transmit information are included at some point in the program. Programmed transfers are program initiated and executed under program control.

Information may also be transferred via program interrupt, a standard feature of MPS that allows I/O devices to signal the microprocessor when they are ready to transfer information. The microprocessor interrupts its normal flow, jumps to a special routine which processes the information, and then returns to the point at which the main program was interrupted. Program interrupt transfers are device initiated and executed under program control.

Both programmed transfers and program interrupt transfers use the accumulator as the buffer, or storage area, for all data transfers. Since data may be transferred only between the accumulator and the device, only one 8-bit byte at a time may be transferred by programmed transfer or by program interrupt.

7.2 I/O INSTRUCTION FORMAT

Since many different devices could be connected to one microprocessor and each device might transfer information at any time, the instruction must identify the proper device for each transfer. It must also specify the exact nature of the function to be performed. The instructions used to perform programmed data transfers are called input/ output (I/O) instructions. An I/O instruction is an 8-bit byte that has the following format:

	0	1	R	R	M	M	M	1
Bit Position:	7	6	5	4	3	2	1	0

An I/O instruction is divided into three parts: operation code, device selection code, and operation specification bits. Bits 7, 6, and 0 of the instruction contain the operation code. These bits are always set to 0, 1, and 1, respectively. Bits 5 and 4 serve a dual purpose. If they are both 0, the instruction is an INPut instruction. If bits 5 and 4 are not both zero, the instruction is an OUTput instruction. Bits 5 and 4 also combine with bits 3, 2, and 1 to form a 5-bit device selection code. Therefore, one can select 32 different devices, of which 8 are considered input and 24 are considered output.

The states of the condition flip-flops are not affected by executing the I/O instructions. Because INP moves data into the accumulator from an input device and OUT moves data from the accumulator to an output device, it is the coder's responsibility to load data into register A before issuing an OUT instruction and to extract data from A after executing an INP.

7.3 CODING THE UART

As mentioned earlier, one of the most common I/O devices is the Teletype unit, which consists of a keyboard, printer, paper tape reader, and paper tape punch. The Teletype unit can use either the keyboard or the paper tape reader to provide input information to the microprocessor, and either the printer or the paper tape punch to accept output information from the microprocessor. In the MPS system, the Teletype unit is connected to the UART (Universal Asynchronous Receiver/Transmitter) control on the M7341 Processor Module. The UART is an I/O interface on the Processor Module which handles data from a serial port. Three device selection codes are reserved for the UART.

Mnemonic	Device Code	Instruction	Function
INP0	00000	01 000 001	Read data from UART
INP1	00001	01 000 011	Read status from UART
OUT0	01000	01 010 001	Write data to UART

The UART Status Register is set up as follows:

	Error	Data Over-run	Data Avail-able	Trans-mit Buffer Empty	Unused	Unused	Unused	Unused
Bit Position:	7	6	5	4	3	2	1	0

The presence of a bit set in any of positions 4-7 indicates the condition appropriate to that position.

7.4 KEYBOARD/READER INSTRUCTIONS

The following program illustrates using the UART instructions to read one character from the keyboard or paper tape reader. This program does not print the character on the teleprinter. It merely stores the code for the character in memory location IBUF. In this example and the examples that follow, we will be using 7-bit ASCII code (American Standard Code for Information Interchange). The full set of ASCII codes are provided in Appendix C of this guide.

```
        *4#0
READ,   INP1            /read status register into A
        NDI 40          /test for data available
        JTZ READ        /wait for data
        INP0            /read character into A
        NDI 177         /strip off bit 7
        LHI IBUF↑       /set pointer to IBUF
        LLI IBUF
        LMA             /store character in IBUF
        HLT
        *10#0
IBUF,   BLOCK 50; 0     /input buffer, reserved
        $               /for 40₁₀ characters
```

The program begins with an INP1 instruction. If the program is started at location 4#0, it will loop indefinitely between INP1, NDI 40, and JTZ READ until a key on the Teletype unit is pressed or a paper tape is loaded into the reader. As soon as the ASCII code for a character has been assembled in the keyboard/reader buffer register, the data available bit will be set in the UART status register and the program drops out of the waiting loop. The content of the buffer is then transferred into the accumulator (INP0), and the data available bit is cleared.

To store the character in memory, we have to set the H and L registers with the 14-bit address of IBUF. We do this by selecting the high byte of IBUF (IBUF↑) and loading it into the H register and transfer the low byte into the L register. BLOCK is a pseudo-instruction in MLA which reserves a block of memory with IBUF equal to the first memory location. The size of the block in this case is 50_8, and the initial value of each byte in the block will be 0.

7.5 PRINTER/PUNCH INSTRUCTIONS

The following program illustrates using the UART instructions to print out one ASCII character which is stored in memory location OBUF. Since the accumulator is used by the INP1 instructions, the character to be output is typically held in one of the other general purpose registers until the transmit ready bit is detected.

```
         *4#0
PRNT,    LHI OBUF↑       /set up H, L registers
         LLI OBUF        /to point to OBUF
         LDM             /get character into D
         CAL TYPE        /enter type subroutine
         HLT             /halt upon completion
TYPE,    INP1            /read status register
         NDI 20          /test for transmit ready
         JTZ .-3         /wait for ready
         LAD             /put character into AC
         OUT0            /print it
         RET             /return to main line
OBUF,    DATA 101        /stored ASCII 'A'
         $
```

7.6 FORMAT ROUTINES

Input and output routines are often written in the form of subroutines similar to the TYPE subroutine in the previous example. The following program presents a carriage return/line feed subroutine that calls the TYPE subroutine to execute a carriage return and line feed on the teleprinter. Similar subroutines could be written to tab space the carriage a given number of spaces or to ring the bell of the Teletype by using the respective codes for these non-printing characters. If such routines are commonly used in a program, you may want to place them at the starting location of a restart instruction to save time and space in your program.

```
         *30#0
CRLF,    LDI 15          /load ASCII code for
         CAL TYPE        /carriage return into D and print
         LDI 12          /load ASCII code for line feed
         CAL TYPE        /into D and print it
         RET             /return to main line
TYPE,    INP1            /read UART status
         NDI 20          /test for transmit ready
         JTZ .-3         /wait for ready
         LAD             /load character into A
         OUT0            /print it
         RET             /return to statement after CAL
         $
```

7.7 TEXT ROUTINES

The previous examples may be expanded to accept and print more than one character. The following programs illustrate one such expansion. These two programs are compatible in that the characters accepted by the first program may be typed out by running the second program. The first program will accept input characters until a dollar sign ($) is typed at the keyboard. It then stores 000 in the next memory location and halts. The second program types the characters whose ASCII codes were stored by the first program, and halts when a location with contents equal to zero is reached. Both programs use locations beginning at 37#0 as a storage buffer for the ASCII characters. The flowcharts in Figure 7-1 help to illustrate the techniques used in program coding.

Program to Accept ASCII Characters

```
          *4#0
GETC,     LHI BUFF↑      /set up memory pointer
          LLI BUFF
LISN,     INP1           /read UART status
          NDI 40         /test for data available
          JTZ LISN       /wait for data
          INP0           /read character
          NDI 177        /strip off bit 7
          CPI 44         /test for dollar sign
          JTZ DONE       /jump to done if dollar sign
          LMA            /else, store it
          INL            /increment pointer
          JMP LISN       /get next character
DONE,     XRA            /clear AC
          LMA            /store a zero in buffer
          HLT            /halt
          *37#0
BUFF,     HLT            /define 1st location of buffer
          $
```

Program to Print ASCII Characters

```
          *4#100
PUTC,     LHI BUFF↑      /initialize memory pointer
          LLI BUFF
          CAL CRLF       /initialize Teletype carriage
PMSG,     LAM            /get next character into AC
          ORA            /set condition codes
          JTZ EXIT       /exit if zero code set
          LDA            /else, save character in D
          CAL TYPE       /print it
          INL            /increment memory pointer
          JMP PMSG       /loop till zero is detected
```

```
CRLF,      LDI 15          /print carriage return
           CAL TYPE
           LDI 12          /print line feed
           CAL TYPE
           RET             /return
TYPE,      INP1            /read UART status
           NDI 20          /transmit buffer empty?
           JTZ .-3         /no, loop
           LAD             /yes, put character in AC
           OUTO            /print it
           RET             /return
EXIT,      HLT             /halt if last character was 0
           *37#0
BUFF,      HLT             /define start of buffer
           $
```

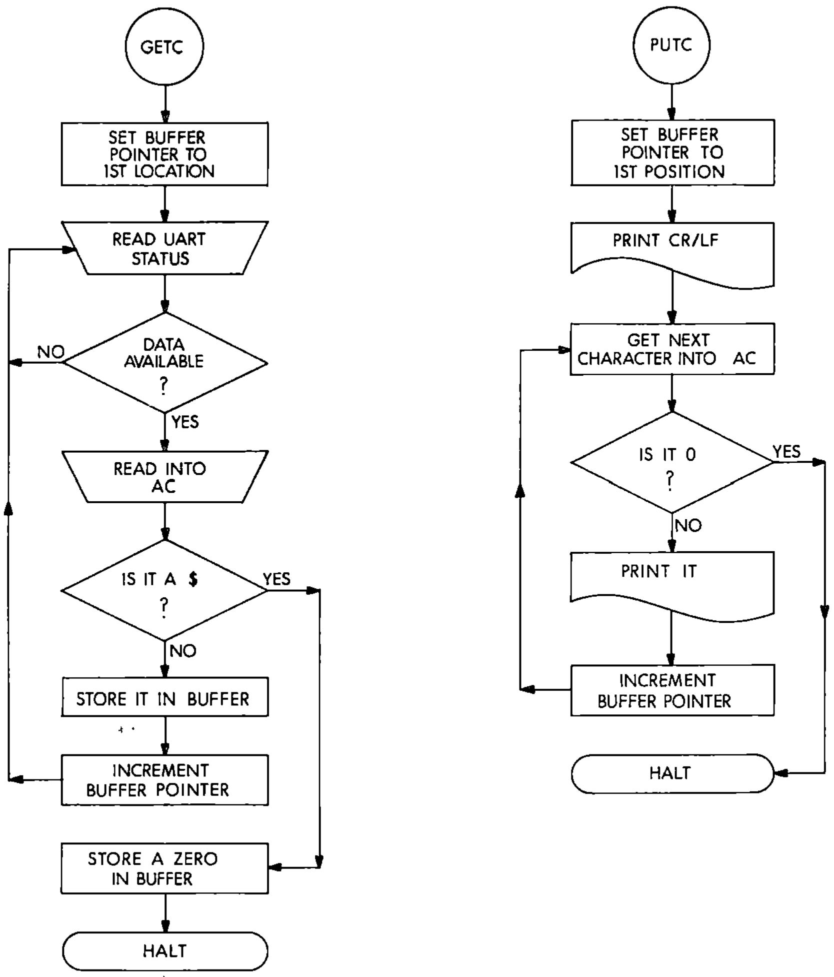


Figure 7-1. Flowchart for ASCII Character Routines

The program to print ASCII characters may be specialized to print a specific message, as in the following example. This print message subroutine (PMSG) is entered with the H, L register pair set to the first location of the text to be printed. TEXT is another pseudo-instruction which the assembler recognizes to store the codes for ASCII characters. The first and last characters of a string are called delimiters and are not stored. They can be any printing character except a left angle bracket (<). Angle brackets are used to enclose a numeric code for an ASCII character that is stored without translation.

<u>Subroutine to Print the Message, "HELLO!"</u>

```
          *5#0
HI,       LHI MSG↑       /init H, L registers with
          LLI MSG        /address of message
          CAL PMSG       /print message
          HLT            /halt
/Print Message Subroutine
PMSG,     XRA            /clear AC
          ADM            /add character to AC
          RTZ            /return if AC is zero
          LDA            /save character in D
          CAL TYPE       /print character
          INL            /increment low byte
          JFZ PMSG       /loop, if not zero
          INH            /increment high byte
          JMP PMSG       /loop
/Print Character Subroutine
TYPE,     INP1           /read UART status
          NDI 20         /transmit buffer empty?
          JTZ .-3        /no, loop
          LAD            /yes, load character
          OUT0           /print it
          RET            /return
/Define Message
MSG,      TEXT           /HELLO!/<0>
```

With the above routine, messages could be stored anywhere in memory because the L register is tested for zero after incrementing.

7.8 NUMERIC TRANSLATION ROUTINES

The ASCII code for a number must be converted to octal representation before the microprocessor may use the number in calculations. For example, 6 is represented by the 7-bit ASCII code 066. When the Teletype key for 6 is typed, the code 066 is transmitted to the microprocessor upon execution of the next INPO instruction. The method of stripping an ASCII-coded number is to subtract 60 from the character code. This process may be reversed to print out a digit which is stored in memory by adding 60 to the digit. The following programs illustrate these methods. The LISN and TYPE subroutines are the same as in previous examples and are not shown here.

```
          *4#0
NUMI,     CAL LISN        /read number
          SUI 60          /subtract 60
          LEA             /store digit in E
          HLT
          $

          *4#100
NUMO,     LAE             /load digit into AC
          ADI 60          /add 60
          LDA             /store temporarily in D
          CAL TYPE        /print digit
          HLT
          $
```

The routines presented so far have been designed to handle single-digit octal numbers. However, MPS memory locations may contain octal numbers with up to three digits. The following flowcharts and programs illustrate the procedures employed to print out a three-digit octal number which is stored in memory and to accept three octal digits from the Teletype keyboard, convert them to an octal number, and store that number in memory.

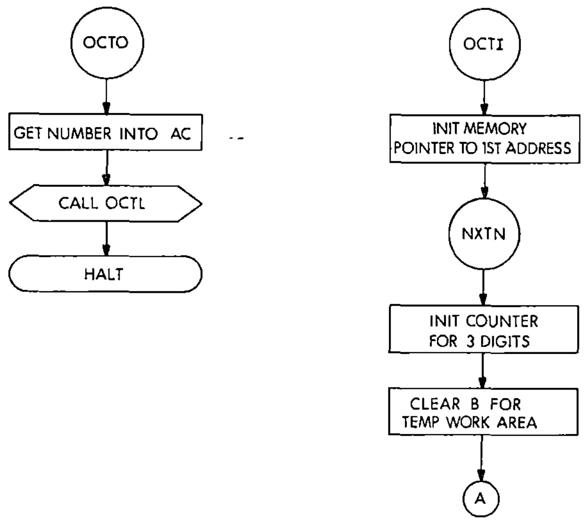

OCTO
GET NUMBER INTO AC
CALL OCTL
HALT
OCTI
INIT MEMORY POINTER TO 1ST ADDRESS
NXTN
INIT COUNTER FOR 3 DIGITS
CLEAR B FOR TEMP WORK AREA
A

Figure 7-2. Three-Digit Octal Number Routine Flowchart

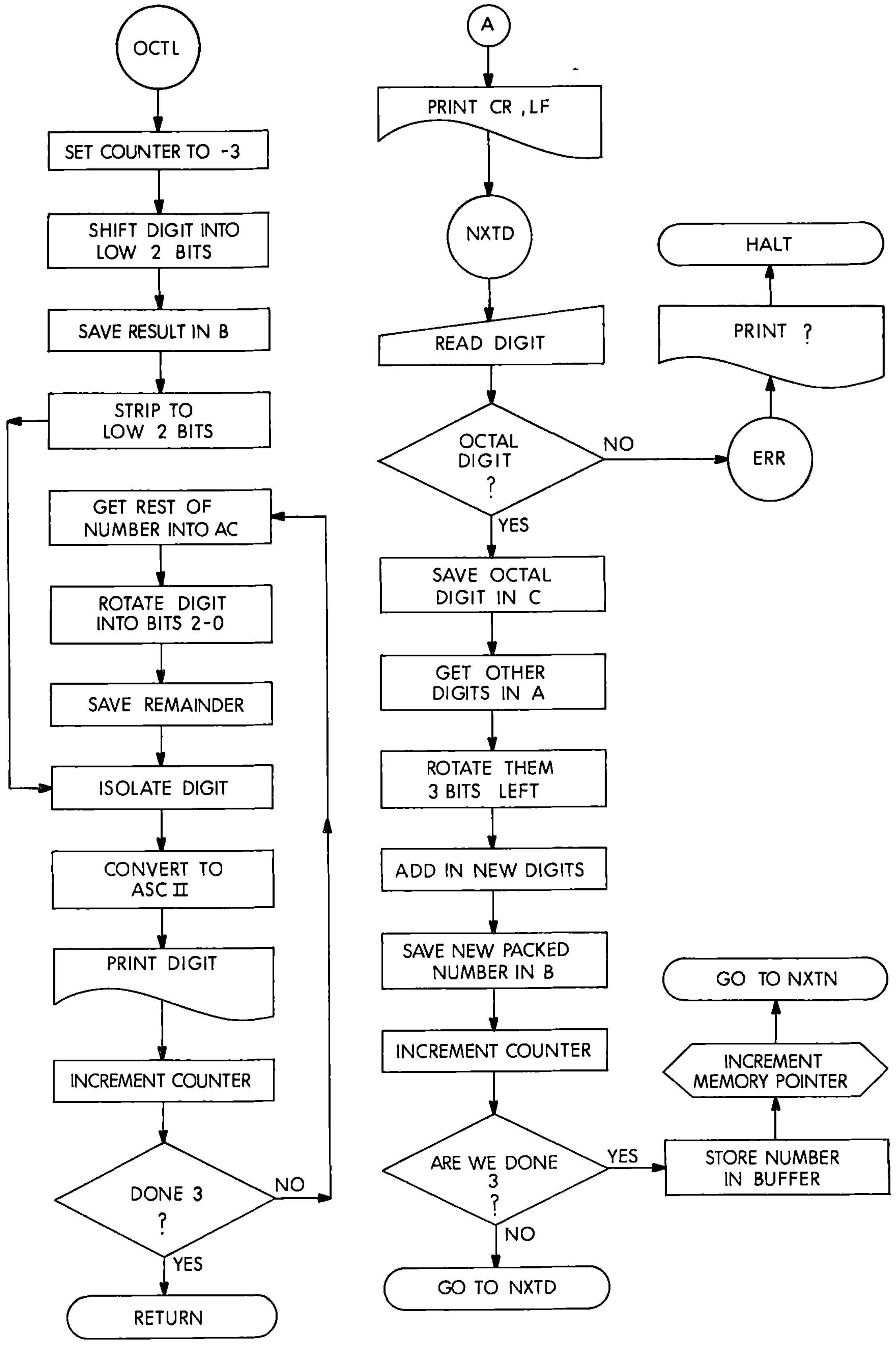


Figure 7-2. Three-Digit Octal Number Routine Flowchart (Cont'd.)

```
          *10#0
/Program to Type a 3-Digit Octal Number
OCTO      LHI NBR↑        /point to number in memory
          LLI NBR
          LAM             /load number into AC
          CAL OCTL        /call output routine
          HLT
/Octal Output Routine
/Called with Octal Number in AC
OCTL,     LLI -3          /set up counter for 3 digits
          RLC             /shift 2 bits left
          RLC             /to bring high digit into bits 1, 0
          LBA             /save result in B
          NDI 3           /strip to low 2 bits
          JMP NXT3
NXT2,     LAB             /retrieve remaining bits
          RLC             /shift left 3 bits
          RLC             /to bring next digit
          RLC             /into bits 2, 1, 0
          LBA             /save remainder
NXT3,     NDI 7           /isolate digit
          ADI 60          /convert to ASCII
          LDA             /store in D
          CAL TYPE        /print digit
          INL             /done 3?
          JFZ NXT2        /no
          RET             /yes
TYPE,     INP1            /read UART status
          NDI 20          /transmit buffer empty?
          JTZ TYPE        /wait for buffer empty?
          LAD             /get digit into AC
          OUT0            /print it
          RET             /return
NBR,      HLT
          $

          *11#0
/Program to Pack and Store 3-Digit Numbers
OCTI,     LHI NBUF↑       /init H, L to starting address
          LLI NBUF        /of Number Buffer
NXTN,     LBI 0           /clear B
          CAL CRLF        /return carriage
```

```
            LEI -3          /set counter for 3 digits
NXTD,       CAL LISN        /get a character
            LCA             /save for later
            NDI 370         /test for number
            CPI 60          /was 0-7 typed?
            JFZ ERR         /no, jump to error routine?
            XRC             /get bits 2-0 into AC
            LCA             /save octal digit
            LAB             /get previous digits
            RAL             /rotate them left three positions
            RAL
            RAL
            ADC             /add new digit
            LBA             /save packed number
            INE             /received 3 digits yet?
            JFZ NXTD        /no, get another
            LMA             /yes, store it
            CAL INHL        /increment memory pointer
            JMP NXTN        /get a new number
ERR,        LDI 77          /type a ?
            CAL TYPE
            HLT
INHL,       INL             /increment H, L pointer
            RFZ             /to memory
            INH
            RET
CRLF,       LDI 15          /type carriage return
            CAL TYPE
            LDI 12          /type line feed
            JMP TYPE
LISN,       INP1            /read UART status
            NDI 40          /data available?
            JTZ .-3         /no, loop
            INP0            /yes, read into AC
            NDI 177         /strip off bit 7
            LDA             /save in D for echo
TYPE,       INP1            /read UART status
            NDI 20          /transmit ready?
            JTZ .-3         /no, wait for ready
            LAD             /load character in AC
            OUT0            /print it
```

```
                    RET
                    *12#0
          NBUF,     HLT             /starting address of buffer
                    $
```

In the last example, several things should be pointed out. Each character read in is immediately echoed back or printed on the Teletype printer. The LISN routine, instead of returning to the main program after the NDI statement, goes directly into the TYPE routine to print the character just read. In the CRLF routine, instead of CALling the TYPE routine after loading the ASCII code for a line feed, a JMP to TYPE is taken. The RETurn instructions in TYPE will get us back to the main program.

CHAPTER 8

INTERRUPTS

8.1 GENERAL

The running time of programs using input and output routines is primarily made up of the time spent waiting for an I/O device to accept or transmit information. Specifically, this time is spent in loops such as:

```
INP1          /read status register
NDI 20        /device ready
JTZ .-3       /no, wait
```

Waiting loops waste a large amount of processor time. In those cases where the microprocessor can be doing something else while waiting, these loops may be eliminated and useful routines included to use the waiting time. This sharing of a microprocessor between two tasks is often accomplished through the program interrupt facility, which is available to MPS users through the External Event Detection Module, M7346. The program interrupt facility allows certain external conditions to interrupt the MPS program. It is used to speed the processing of I/O devices or to allow certain alarms to halt program execution and initiate another routine.

The M7346 External Event Detection Module (EEDM) implements nine levels of priority arbitration. These include application-defined, six-level priority interrupt schemes, an ac/dc power failure detection capability, and the processor control functions of Halt and Restart.

Separate input lines to the EEDM provide for encoding up to six levels of external application-defined event priority. Each of these lines, when asserted, initiates an attempt to jam a one-byte unconditional call (RST) instruction into the Processor Module external event port. If the Processor Module honors or recognizes an interrupt, the RST instruction is fetched and executed. If not, the RST instruction is ignored.

The EEDM priority logic arbitrates all interrupt assertions and selects the highest priority level asserted, then places the corresponding RST instruction into the Processor Module. The RST instruction associated with each priority level (zero through five) constitutes an unconditional call on one of the six 8-byte subroutines located in the first 48 words of an MPS system memory.

When the interrupt circuitry is enabled and an interrupt request is recognized, the microprocessor automatically disables the interrupt recognition system to lock out all other requests. It then executes a hardware RST instruction to the predetermined location in memory associated with the recognized interrupt. The interrupt service routine calls up the necessary I/O device service routines to satisfy the condition which caused the interrupt, then reenables the interrupt recognition system and executes a RET instruction to resume program execution.

Two OUT instructions are reserved for enabling and disabling the interrupt recognition system. These are device selection codes 11_8 and 12_8 and have the special mnemonics ION and IOF, respectively. Execution of the ION instruction will be deferred until the following instruction has been executed. In this manner, the interrupt recognition system is not enabled until the mainline execution has resumed.

Use of the interrupt system allows a mainline routine to execute without wasting a large amount of time in waiting loops while I/O devices are assembling and transmitting information. The interrupt service routine is entered automatically whenever an I/O device requires servicing under program control.

8.2 CODING AN INTERRUPT

The program presented here consists of a background program which rotates one bit through the accumulator endlessly, and a foreground program, initiated by the interrupt service routine, which accepts and stores temperature readings from an A/D converter. The coding begins with an initialized routine which allocates buffer space to store the incoming data. Once the initialization routine has enabled the interrupt facility, the background program is started.

An interrupt request from the A/D converter will cause the microprocessor to perform the following operations automatically:

1. The interrupt recognition system is disabled.

2. A restart instruction (RST) is executed which performs an unconditional CAL to location 0.

The interrupt service routine then performs the following operations:

1. The contents of the accumulator are saved.

2. The data is read and stored.

3. The contents of the accumulator are restored.

4. The next A/D conversion is initialized.

5. The interrupt recognition system is enabled.

6. Return is taken to the background program.

```
                OPDEF INP3; 107; 0    /read A/D value
                OPDEF OUT3; 127; 0    /init A/D conversion

                *0                    /interrupt service routine
                LDA                   /save contents of AC
                INP3                  /read data from A/D converter
                LMA                   /store data in buffer
                CAL INHL              /increment buffer pointer
                OUT 3                 /start next A/D conversion
                ION                   /enable recognition system
                RET                   /return to background program

                *4#0
        BEG,    LHI BUFF↑             /init buffer pointer
                LLI BUFF
                INP3                  /clear flags
                LAI 1                 /set bit 1 in AC
                OUT3                  /start 1st conversion
                ION                   /enable interrupts
                RLC                   /rotate AC one bit position to left
                JMP .-1               /loop
        INHL,   INL                   /increment memory pointer
                RFZ
                INH
                RET
                $
```

CHAPTER 9

PROGRAMMING EXAMPLE

To summarize the material covered in this guide, the following flowchart (Figure 9-1) and program are presented. The program contains a dummy background routine that simply waits for the next interrupt. The foreground routine is controlling the temperature in a room. The temperature is measured by a thermistor and converted to a binary number by an A/D converter. When the binary number is completely assembled, an interrupt is generated and the service routine is entered at location 0. The incoming temperature is plotted on the console video screen and is compared against preset limits to determine if the room is too cold or too hot. If it is too hot, the heating unit is turned off and the cooling unit is turned on. If it is too cold, the reverse is done. OUT4 sends an 8-bit byte to a controller which turns the heating element on if bit 7 is set and off if bit 7 is clear. The controller will turn on the cooling unit if bit 6 is set and off if bit 6 is clear.

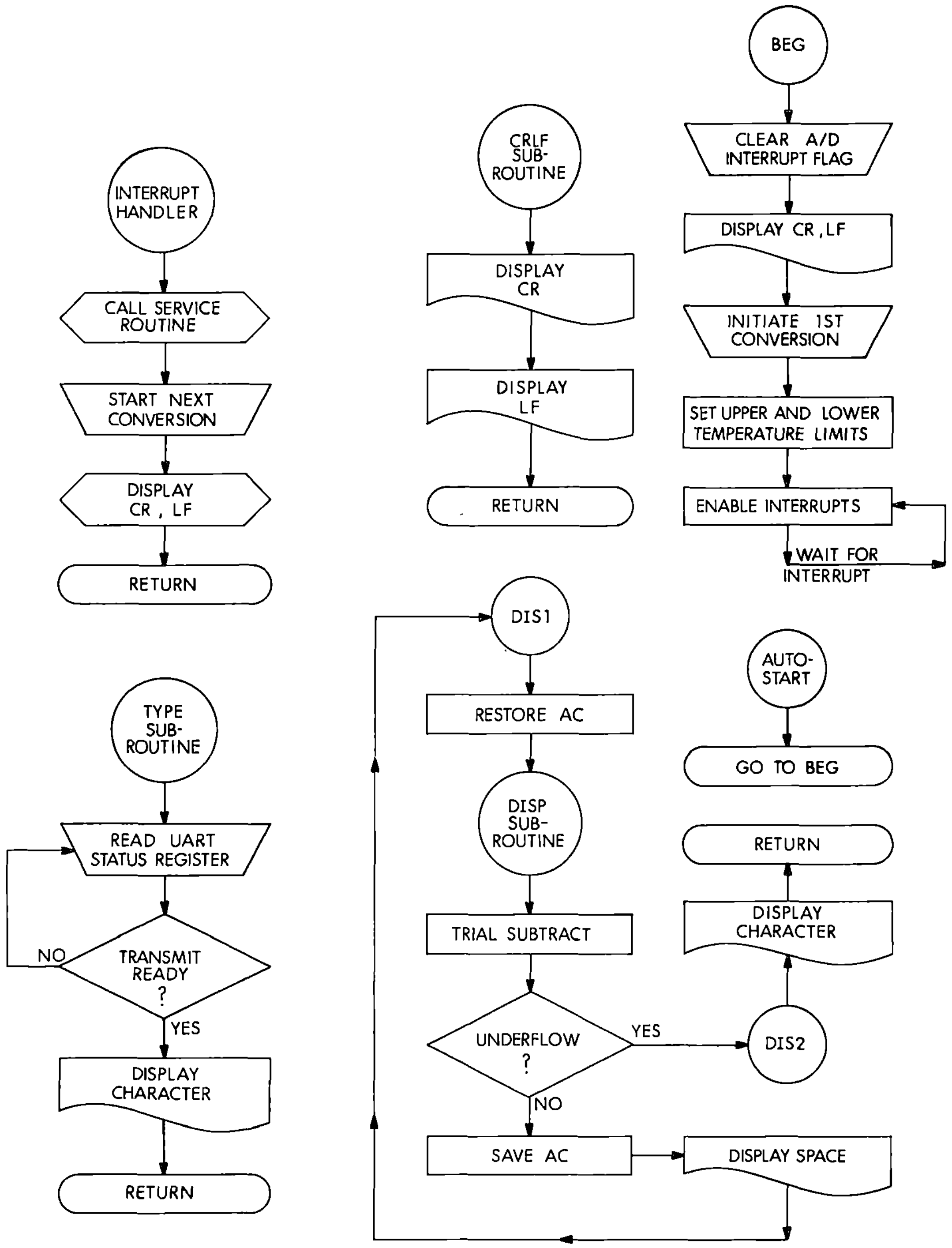


Figure 9-1. Temperature Control Program Flowchart

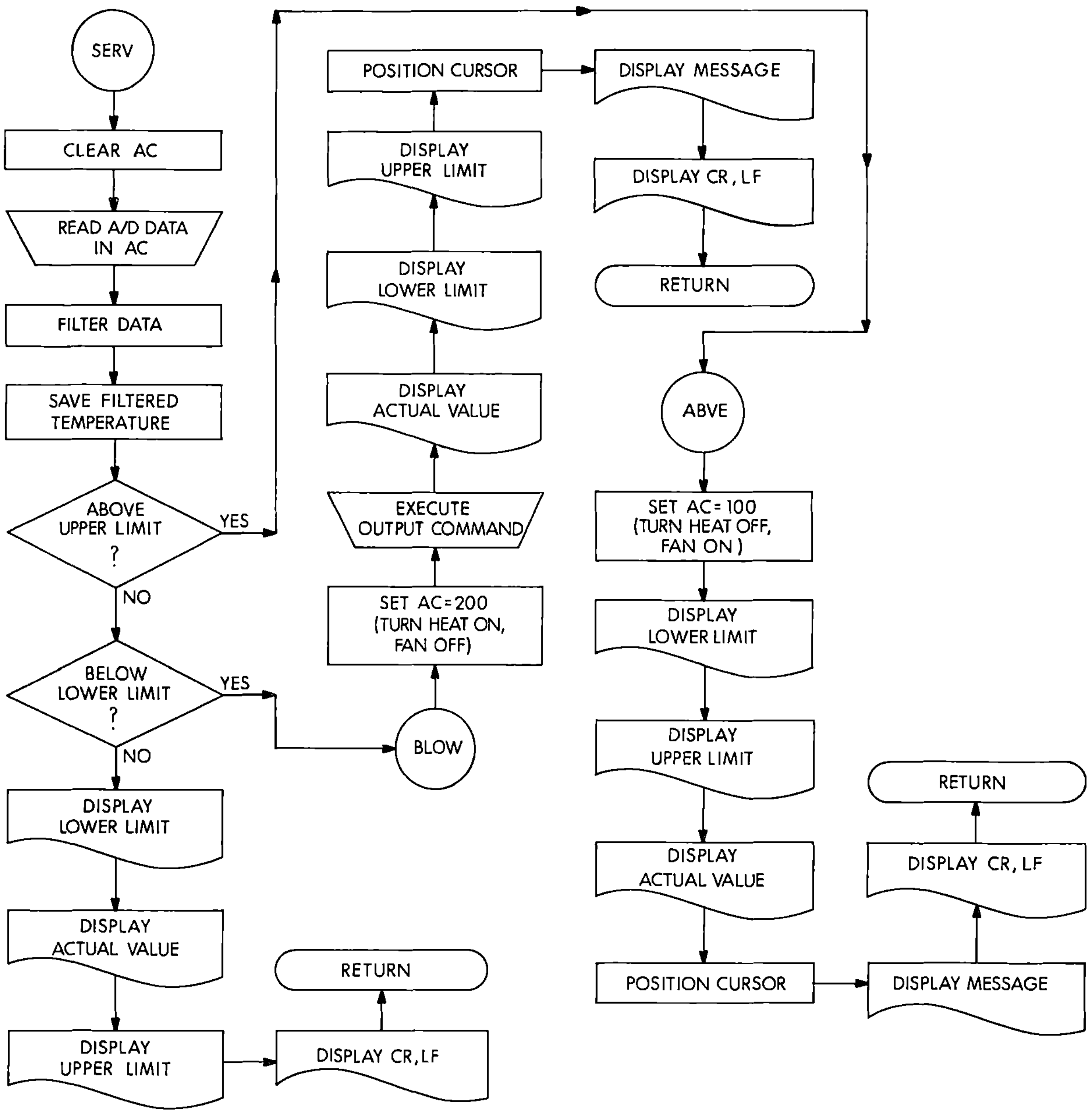


Figure 9-1. Temperature Control Program Flowchart (Cont'd.)

```
/ROOM TEMPERATURE CONTROL PROGRAM
/
        OPDEF TYPE;RST110;0
        OPDEF CRLF;RST130;0
        OPDEF DISP;RST140;0
        OPDEF INP3;107;0
        OPDEF OUT3;127;0
        OPDEF OUT4;131;0

        *0#0
        CAL SERV            /INTERRUPT SERVICE ROUTINE
        LDI 15              /DUMMY CARRAIGE RETURN
        TYPE
        OUT3                /START NEXT CONVERSION
        RET                 /RETURN

/TYPE CHARACTER IN REGISTER D
        INP1                /READ UART STATUS
        NDI 20              /TRANSMIT READY
        JTZ .-3             /NO, WAIT
        LAD                 /LOAD CHARACTER
        OUT0                /DISPLAY CHARACTER
        RET                 /EXIT

        *0#30
        LDI 15              /DISPLAY CR,LF
        TYPE
        LDI 12
        TYPE
        RET

/PLOT CURRENT TEMPERATURE READING
DIS1,   LAL                 /RESTORE A
        SUI 2               /TRIAL SUBTRACT
        JTC DIS2            /UNDERFLOW
        LLA                 /SAVE A
        LDI 40              /NO, DISPLAY A SPACE
        TYPE
        JMP DIS1
DIS2,   LDH                 /GET CHARACTER
        TYPE
        RET
        *0#60
        JMP BEG             /AUTO-START

/START OF PROGRAM
        *0#100
BEG,    INP3                /CLEAR FLAGS
        CRLF                /DISPLAY CR,LF TO INIT PLOT
        OUT3                /INITIATE 1ST CONVERSION
        LCI 120             /SET C EQUAL TO UPPER LIMIT
        LEI 24               /SET E EQUAL TO LOWER LIMIT
        ION                 /ENABLE INTERRUPTS
        JMP .-1             /LOOP FOR INTERRUPT
        JMP .-4             /SAFETY LOOP
```

```
/INTERRUPT SERVICE ROUTINE
SERV,   INP2                /READ DATA INTO A
        LBA                 /SAVE IN B
        CPC                 /TEST UPPER LIMIT
        JFC ABVE            /ABOVE UPPER LIMIT
        CPE                 /TEST LOWER LIMIT
        JTC BLOW            /BELOW LOWER LIMIT
        LAE                 /WITHIN LIMITS, DISPLAY LOWER LIMIT
        LHI 111             /ASCII CODE FOR AN "I"
        DISP
        LAB                 /DISPLAY ACTUAL VALUE
        SUE
        LHI 52              /ASCII CODE FOR AN "*"
        DISP
        LAC                 /DISPLAY UPPER LIMIT
        SUB
        CAL TEST            /TEST FOR OVERLAPPING CHARACTERS
        CRLF
        RET
BLOW,   LAI 200             /TURN HEAT ON, FAN OFF
        OUT4                /EXECUTE OUTPUT COMMAND
        LAB                 /DISPLAY ACTUAL VALUE
        LHI 52
        DISP
        LAE                 /DISPLAY LOWER LIMIT
        SUB
        CAL TEST
        LAC                 /DISPLAY UPPER LIMIT
        SUE
        DISP
        LAI 377             /POSITION CURSOR
        SUC
        ADI 10
        LHI 40              /ASCII CODE FOR A SPACE
        DISP
        LHI BNFF"           /DISPLAY MESSAGE
        LLI BNFF
        CAL PMSG
        CRLF
        RET
ABVE,   LAI 100             /TURN HEAT OFF, FAN ON
        OUT4                /EXECUTE OUTPUT COMMAND
        LAE                 /DISPLAY LOWER LIMIT
        LHI 111             /ASCII CODE FOR AN "I"
        DISP
        LAC                 /DISPLAY UPPER LIMIT
        SUE
        DISP
        LAB                 /DISPLAY ACTUAL DATA
        SUC
        LHI 52              /ASCII CODE FOR AN "*"
        DISP
        LAI 377             /POSITION CURSOR
        SUB
        ADI 10
        LHI 40              /ASCII CODE FOR A SPACE
        DISP
        LHI BFFN"           /DISPLAY MESSAGE
        LLI BFFN
        CAL PMSG
        CRLF
        RET
```

```
/DISPLAY MESSAGE ROUTINE
BNFF,    TEXT /HEAT ON, FAN OFF/
         HLT
BFFN,    TEXT /HEAT OFF, FAN ON/
         HLT
PMSG,    XRA                 /CLEAR AC
         ADM                 /ADD NEXT CHARACTER
         RTZ                 /EXIT IF ZERO
         LDA                 /SAVE IN D
         TYPE                /DISPLAY IT
         INL                 /INCREMENT MEMORY
         JFZ PMSG            /LOOP, IF L NOT ZERO
         INH                 /INCREMENT HI-BYTE TOO
         JMP PMSG            /LOOP
TEST,    CPI 5               /TEST FOR CHARACTER OVERLAPPING
         RTC
         LHI 111             /ASCII CODE FOR AN "I"
         DISP
         RET
         $
```

CHAPTER 10

WHERE DO YOU GO FROM HERE?

You should now be reasonably familiar with the art of communicating with MPS. In essence, you have learned to program a microprocessor. Your task of programming MPS will be made even easier by the availability of the programming tools designed specifically for MPS. They are available in two forms:

1. MPS module programs may be prepared with a small, low-cost PDP-8 minicomputer (excluding the PDP-8/S) with 4K of memory, Teletype terminal, and paper tape reader/punch. The software necessary to program the Microprocessor Series is provided in a Software Kit which consists of the following programs. All programs are provided in paper tape form.

Program	Function
MLE (Microprocessor Language Editor)	Allows editing of source tapes from the Teletype console. Runs on PDP-8*.
MLA (Microprocessor Language Assembler)	Assembles source programs into binary tape format. Runs on PDP-8.*
MRP (Microprocessor ROM Programmer)	Programs PROMs from assembler's binary tape output. Runs on PDP-8*.
MDP (Microprocessor Debugging Program)	Aids in debugging of binary programs. Runs on MPS.
MTD (Master Tape Duplicator Program)	Copies paper tapes. Runs on PDP-8*.
MHL (Microprocessor Host Loader Program)	Loads binary-coded tapes into PDP-8 memory. Runs on PDP-8*.

*Except PDP-8/S.

This package is designated QF500-AB.

2. MPS module programs may also be prepared without the use of a PDP-8, on the MPS Software Development Set, using the 54-slot prewired backplane

and 8K of RAM memory, Teletype terminal, and paper tape reader/punch. The software required is provided also in a Software Kit, which consists of the following programs. Again, all programs are provided in paper tape form.

Program	Function
MLE (Microprocessor Language Editor)	Allows editing of source tapes from the Teletype console. Runs on 8K of MPS memory.
MLA (Microprocessor Language Assembler)	Assembles source programs into binary tape format. Runs on 8K of MPS memory.
MDP (Microprocessor Debugging Program)	Aids in debugging of binary programs. Runs on MPS.

This package is designated QY500-AB.

The two kits are fully described in the following documents:

QF500-AB	MPS User's Handbook
QY500-AB	Checkout Procedure, DEC-BE-UMCPA-A-D

The next step is learning how to apply your new expertise to making MPS work for you--how to solve problems in the industrial world, particularly in the area of monitoring and controlling industrial equipment, for which the MPS is especially applicable.

MPS is a viable alternative to hard-wired circuits for many control-oriented functions. The programmable components are standardized, so you don't have to develop special-purpose circuitry to solve specific problems. Modifications made necessary by product evolution are simplified because the whole focus on the design effort is shifted from hardware to software. Reliability is improved because there are fewer components and interconnections. Uniformity within a series of products is increased. Applications flexibility is enhanced and design time shortened over hard-wired logic; yet the cost is considerably less than the price of a minicomputer.

MPS adaptability ranges from devices which already incorporate some degree of automation, such as numerically controlled machine tools or laboratory blood analyzers, to products such as appliances for which digital control hasn't been considered because of cost or complexity. Used with data terminals, MPS can provide higher levels of interactivity among operators while reducing the traffic burden on communication channels.

Machines in industrial, commercial, and consumer use can be given capabilities for adapting to loads or demands, operating in a variety of modes, and performing monitor and control functions automatically. Interfaced with industrial knitting machines, for example, MPS can implement safety interlocks, count pieces for inventory, select stitching parameters and threads, and detect worn or broken needles.

Here are some typical application areas:

- Industrial Control
  - Machine tool control
  - Material Flow
- Process Control
  - Batch mixing
  - Furnace monitoring
  - Batch weighing
- Small Laboratory Automation
  - Analog and digital instrument data acquisition
  - Blood analyzers
- Data Communications
  - Data concentrators
  - Communications processors
  - Minicomputer Preprocessors
  - Intelligent terminals
- Business Machines
  - Optical character recognition
  - Automatic banking
  - "Smart" copying machines
- Health, Education, and Welfare
  - Environmental control of large buildings
  - Automatic teaching machines
  - Remote pollution-monitoring systems
- Transportation
  - Traffic signal controllers
  - Vehicle recognition scanners
  - Traffic flow monitoring

The product that makes all of this possible, the MPS, consists, in the strictest sense, of the following modular components. These are the pure processor components of MPS.

- M7341 CPU Module
- M7344-YA 1Kx8 Read/Write Memory Module
- M7344-YB 2Kx8 Read/Write Memory Module
- M7344-YC 4Kx8 Read/Write Memory Module
- M7345 Programmable Read-Only Memory
  (PROM, capacity to 4Kx8)

- M7346 External Event Detection Module
- KC341 Monitor/Control Panel
- KMP01 Prewired Backplane (24 slots)
- KMP02 Prewired Backplane (54 slots)

However, in order to fully implement an MPS-controlled monitor/control application, a number of additional functions are required; viz., input/output, data transfer and peripheral device selection. These functions are performed by the following standard DIGITAL modules:

- M1501 8-Bit Bus Input Module
- M1502 8-Bit Bus Output Module
- M7328 External Device Selector (32 devices)

The processor, I/O, and control modules can be assembled in either the 24-slot or the 54-slot prewired backplanes, depending on the magnitude and complexity of the MPS monitor/control system. The backplanes are prewired to the standard MPS and input/output configuration, thus relieving the user of the task of interconnecting the processor, control, and input/output functions. Spare slots in each backplane allow the user the flexibility of adding his own modular electronics.

The referenced modules are all fully documented with technical data sheets and schematics. In addition to this, a number of valuable application notes relative to MPS usage are available. A complete list of support documentation follows:

Title	Document No.
MPS User's Handbook	DEC-08-UMPHA-A-PA
M7341 Processor Module Data Sheet	5804 00874 3985/M
M7344 Read/Write Memory Module Data Sheet	5804 00874 3986/M
M7345 Programmable Read-Only Memory Data Sheet	5804 00874 3987/M
M7346 External Event Detection Module Data Sheet	5804 00874 3988/M
M7328 Evoke Decoder Module Data Sheet	5804 00973 3149/TP
M1501 Bus Input Interface Module Data Sheet	044X 01171 1894/S
M1502 Bus Output Interface Module Data Sheet	0404X 10572 22376/S
M111 Inverter	See 1975-76 Logic Handbook
General Interfacing Techniques for M7341 Microprocessor Module Application Note	5804 00874 4087/Y
MPS Pocket Reference Card	5313 100175 4707/Y

APPENDIX A

BLOCK-OFFSET TO OCTAL CONVERSION

This appendix can be used if it is ever necessary to convert the block-offset notation in assembly language programs and output to octal notation. Only the first and last conversions are given for each block. To convert an offset within a given block, simply add the offset to the starting octal location in the block. For example:

Block	Offset	Octal
11	0	4400
.	.	.
.	.	.
.	.	.
11	377	4777

To convert block 11 offset 227 to octal, simply add 227 to 4400. The correct octal equivalent is thus 4627.

Block	Offset	Octal
0	000	0000
.	.	.
.	.	.
.	.	.
0	377	0377
1	0	0400
.	.	.
.	.	.
.	.	.
1	377	0777
2	0	1000
.	.	.
.	.	.
.	.	.
2	377	1377
3	0	1400
.	.	.
.	.	.
.	.	.
3	377	1777
4	0	2000
.	.	.
.	.	.
.	.	.
4	377	2377

Block	Offset	Octal
5	0	2400
.	.	.
.	.	.
.	.	.
5	377	2777
6	0	3000
.	.	.
.	.	.
.	.	.
6	377	3377
7	0	3400
.	.	.
.	.	.
.	.	.
7	377	3777
10	0	4000
.	.	.
.	.	.
.	.	.
10	377	4377

Block	Offset	Octal	Block	Offset	Octal
11	0	4400	23	0	11400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
11	377	4777	23	377	11777
12	0	5000	24	0	12000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
12	377	5377	24	377	12377
13	0	5400	25	0	12400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
13	377	5777	25	377	12777
14	0	6000	26	0	13000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
14	377	6377	26	377	13377
15	0	6400	27	0	13400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
15	377	6777	27	377	13777
16	0	7000	30	0	14000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
16	377	7377	30	377	14377
17	0	7400	31	0	14400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
17	377	7777	31	377	14777
20	0	10000	32	0	15000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
20	377	10377	32	377	15377
21	0	10400	33	0	15400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
21	377	10777	33	377	15777
22	0	11000	34	0	16000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
22	377	11377	34	377	16377

Block	Offset	Octal	Block	Offset	Octal
35	0	16400	47	0	23400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
35	377	16777	47	377	23777
36	0	17000	50	0	24000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
36	377	17377	50	377	24377
37	0	17400	51	0	24400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
37	377	17777	51	377	24777
40	0	20000	52	0	25000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
40	377	20377	52	377	25377
41	0	20400	53	0	25400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
41	377	20777	53	377	25777
42	0	21000	54	0	26000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
42	377	21377	54	377	26377
43	0	21400	55	0	26400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
43	377	21777	55	377	26777
44	0	22000	56	0	27000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
44	377	22377	56	377	27377
45	0	22400	57	0	27400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
45	377	22777	57	377	27777
46	0	23000	60	0	30000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
46	377	23377	60	377	30377

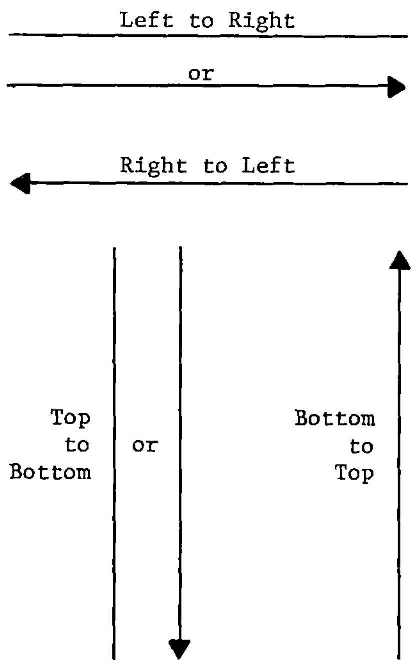
Block	Offset	Octal	Block	Offset	Octal
61	0	30400	71	0	34400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
61	377	30777	71	377	34777
62	0	31000	72	0	35000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
62	377	31377	72	377	35377
63	0	31400	73	0	35400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
63	377	31777	73	377	35777
64	0	32000	74	0	36000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
64	377	32377	74	377	36377
65	0	32400	75	0	36400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
65	377	32777	75	377	36777
66	0	33000	76	0	37000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
66	377	33377	76	377	37377
67	0	33400	77	0	37400
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
67	377	33777	77	377	37777
70	0	34000			
.	.	.			
.	.	.			
.	.	.			
70	377	34377			

APPENDIX B

FLOWCHART SYMBOLS

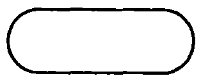
The following is a partial list of flowchart symbols which can be used to diagram the logical flow of a program. The symbols may be made sufficiently large to include the pertinent information.

REPRESENTATION OF FLOW



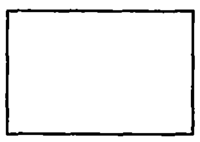
The direction of flow in a program is represented by lines drawn between symbols. These lines indicate the order in which the operations are to be performed. Normal direction of flow is from left to right and top to bottom. When the flow direction is not from left to right or top to bottom, arrowheads are placed on the reverse direction flowlines. Arrowheads may also be used on normal flowlines for increased clarity.

TERMINAL

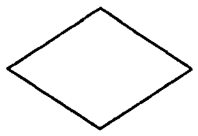
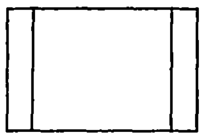
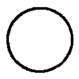
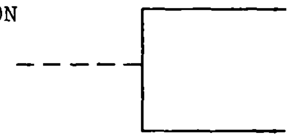
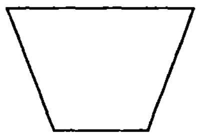
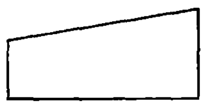
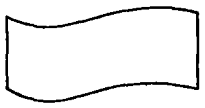
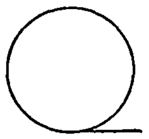


The oval symbol represents a terminal point in a program. It can be used to indicate a start, stop, or interrupt of program flow. The appropriate word is included within the symbol.

PROCESSING



The rectangular symbol represents a processing function. The process which the symbol is used to represent could be an instruction or a group of instructions to carry out a given task. A brief description of the task to be performed is included within the symbol.

DECISION		A diamond is used to indicate a point in a program where a choice must be made to determine the flow of the program from that point. A test condition is included within the symbol and the possible results of the test are used to label the respective flows from the symbol.
PREDEFINED PROCESS		This symbol is used to represent an operation or group of operations not detailed in the flowchart. It is usually detailed in another flowchart. A subroutine is often represented in this manner.
CONNECTOR		The circular symbol represents any entry from or an exit to another part of the program flowchart. A number or a letter is enclosed to label the corresponding exits and entries. This symbol does not represent a program operation.
ANNOTATION		An addition of descriptive comments or explanatory notes for clarification is included within this symbol.
INPUT/OUTPUT		This symbol is used in a flowchart to represent the input or output of information. This symbol may be used for all input/output functions, or symbols for specific types of input or output (such as those which follow) may be used.
MANUAL INPUT		This symbol may be used to represent the manual input of information by means of online keyboards, switch settings, etc.
PUNCHED TAPE		The input or output of information in which the medium is punched tape may be represented by this symbol.
MAGNETIC TAPE		This symbol is used in a flowchart to represent magnetic tape input or output.

APPENDIX C

7-BIT ASCII CODE

Octal Code	Char.	Octal Code	Char.	Octal Code	Char.	Octal Code	Char.
000	NUL	040	SP	100	@	140	
001	SOH	041	!	101	A	141	a
002	STX	042	”	102	B	142	b
003	ETX	043	#	103	C	143	c
004	EOT	044	$	104	D	144	d
005	ENQ	045	%	105	E	145	e
006	ACK	046	&	106	F	146	f
007	BEL	047	’	107	G	147	g
010	BS	050	(	110	H	150	h
011	HT	051	)	111	I	151	i
012	LF	052	*	112	J	152	j
013	VT	053	+	113	K	153	k
014	FF	054	,	114	L	154	l
015	CR	055	-	115	M	155	m
016	SO	056	.	116	N	156	n
017	SI	057	/	117	O	157	o
020	DLE	060	0	120	P	160	p
021	DC1	061	1	121	Q	161	q
022	DC2	062	2	122	R	162	r
023	DC3	063	3	123	S	163	s
024	DC4	064	4	124	T	164	t
025	NAK	065	5	125	U	165	u
026	SYN	066	6	126	V	166	v
027	ETB	067	7	127	W	167	w
030	CAN	070	8	130	X	170	x
031	EM	071	9	131	Y	171	y
032	SUB	072	:	132	Z	172	z
033	ESC	073	;	133	[	173	{
034	FS	074	<	134	\	174	|
035	GS	075	=	135	]	175	}
036	RS	076	>	136	↑	176	~
037	US	077	?	137	←	177	DEL

APPENDIX D

SUMMARY OF ASSEMBLER PSEUDO-INSTRUCTIONS

Pseudo-Instruction	Function
$	Signals end of assembly language program.
PAUSE	Causes pause in Assembler processing until CONTinue switch is pressed.
*expression	Specifies initial program location counter and can be used to reset current location counter.
OCT	Sets radix for subsequent numbers in program to octal (base 8).
HEX	Sets radix for subsequent numbers in program to hexadecimal (base 16).
DEC	Sets radix for subsequent numbers in program to decimal (base 10).
EXPUNGE	Deletes instruction symbol table.
OPDEF mnemonic;value;type	Allows programmer to define own instructions according to value and type given.
label, DATA n0;n1;n2;nm	Assigns values to incremental memory locations.
label, BLOCK size[;initial[;increment]]	Assigns a block of memory of the size given with values of zero, an initial value, or a set of increments.
label, TEXT∇ literal∇	Specifies ASCII character strings and/or numeric representations of ASCII characters to be included in a program.
label, ADDR a0;a1;a2;...;am	Assigns address constants to memory locations.

APPENDIX E

SOLVING THE PROBLEM

The following example might be a suggested approach for the problem of monitoring and controlling the temperature in an office building.

A. <u>The Application</u>

1. Monitor and control building temperature.
2. Monitor fire alarm system in building.
3. Monitor intrusion alarm system in building.

B. <u>Application Features</u>

1. Continuous monitoring of building thermostats.
2. Adjustment of controller setpoints on a local basis.
3. Automatically starts and stops heating and cooling equipment.
4. Provides out-of-limits alarming and corrective action.
5. Maintains log of building operation.

C. <u>Application Components</u>

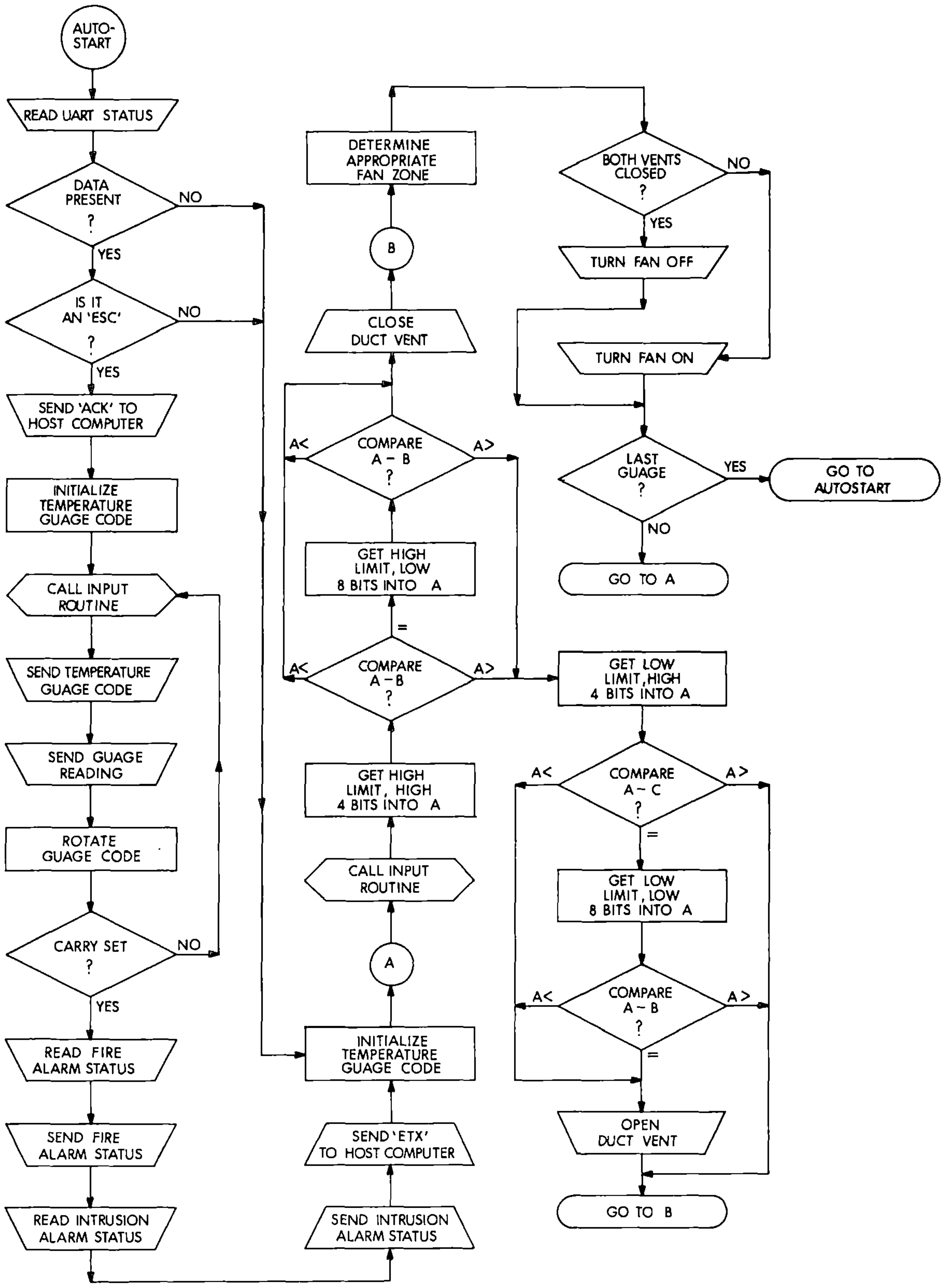
1. Eight thermostats monitoring room temperature points.
2. Eight burglar alarm switches to monitor intrusions.
3. Eight over-temperature thermocouple devices to monitor a fire situation.
4. Four fans to distribute heating air or cooling air.
5. Eight ducts which can be opened or closed for zone heating or cooling.
6. Water-flow detection switch to monitor inadvertent operation of sprinkler valves.
7. Fire alarm audible signal to be turned on after detection of fire condition.
8. Visual indicators on console to advise of intrusion or fire situation.

<u>Solution</u>

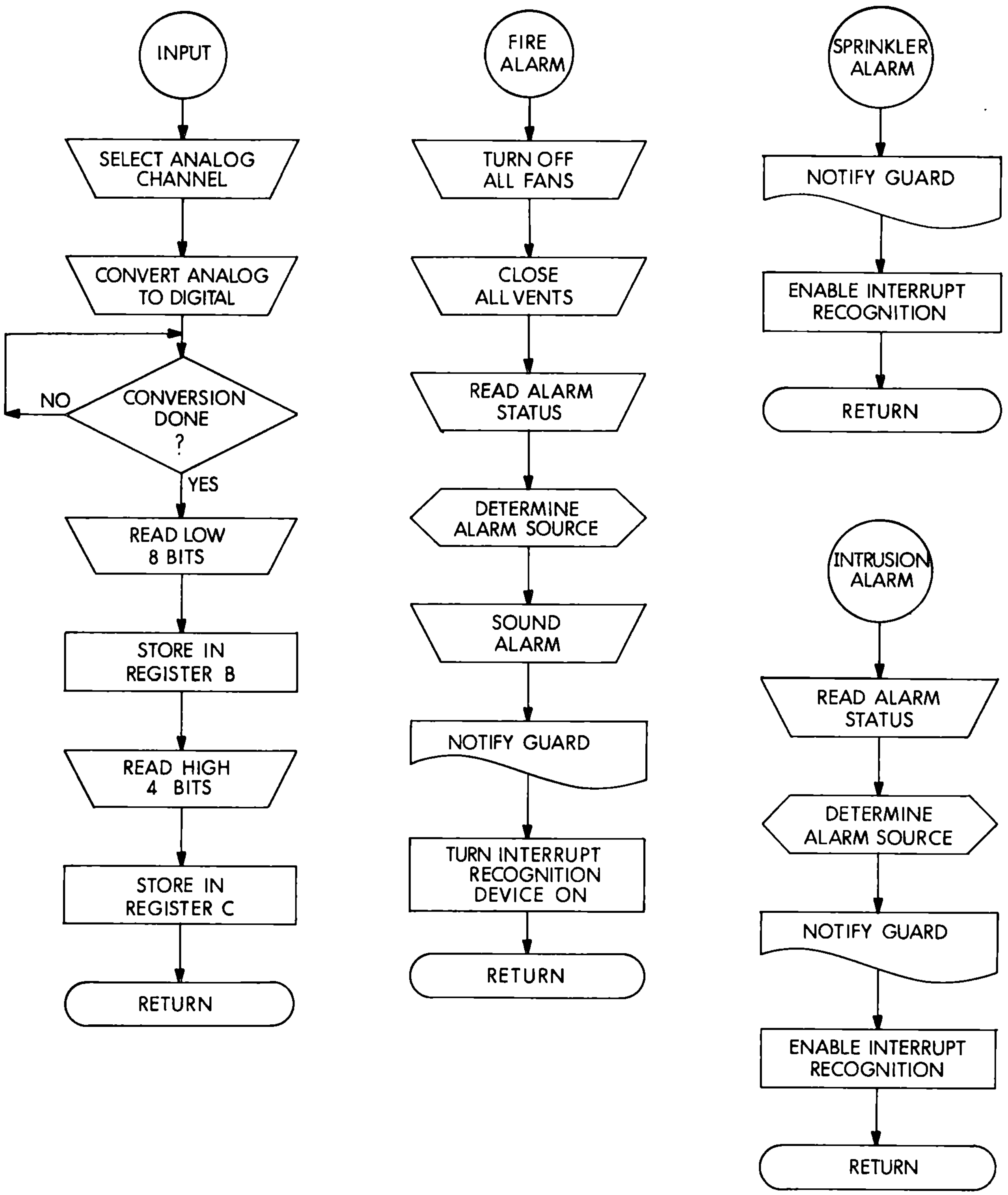
In order to reach the programming solution to this application, we must do the following things:

1. Draw a system flow diagram for the application.
2. Define the software considerations.

1. System Flow Diagram



1. System Flow Diagram (Cont'd.)

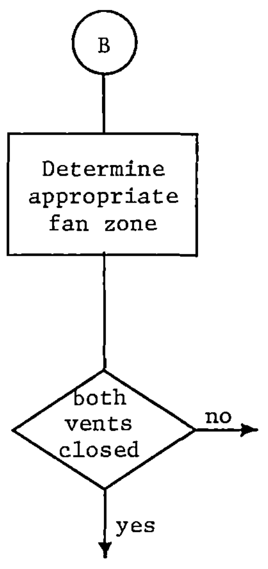


2. Software Considerations

The system flow diagram defines all the operations to be performed by the system software. This particular operation determines that only PROM memory will be required in this system. Internal processing can be done in the internal registers on the CPU module.

The PROM memory will store all instructions, system constants, algorithms, subroutines, as well as the bootstrap program which, on an initial power-up, initializes all system interfaces and starts the machine program. The PROM memory requirements are estimated to be not more than 1K total locations.

A sample section of the system flow diagram has been isolated to show, typically, the types of instructions and number of memory storage locations required to implement a number of typical operations in this system.



```
B,          LAD           /get current gauge bit
            NDI 3         /zone 1?
            JFZ ZONE 1    /yes
            LAD
            NDI 14        /zone 2?
            JFZ ZONE 2    /yes
            LAD
            NDI 60        /zone 3?
            JFZ ZONE 3    /yes
ZONE 4,     LAE           /must be ZONE 4
            NDI 300       /both vents closed?
            JFZ ON4       /no
            LAI FOF4      /set command word
            OUT 34        /turn fan off, this zone
            JMP NXT
ON 4,       LAI FON4      /set command word
            OUT 24        /turn fan on, this zone
            JMP NXT
            .
            .
            .
            .
            .
```

This particular program segment requires 18 memory locations.

GLOSSARY

ABSOLUTE ADDRESS — A binary number that is permanently assigned as the address of a memory storage location.

ACCUMULATOR — One of the 8-bit Index Registers in which arithmetical and logical operations are performed. Abbreviated AC.

ADDRESS — A label, name, or number which designates a location in memory where information is stored.

ADDRESS REGISTER — A register that is used as a pointer to a memory address.

ARGUMENT — A variable or constant which is given to a subroutine; a variable upon whose value the value of a function depends; the known reference factor necessary to find an item in a table or array.

ARITHMETIC UNIT — The component of a computer where processes such as addition, subtraction, multiplication, and division are performed, and operands and results are stored temporarily.

ASCII — American Standard Code for Information Interchange. A standard code using a character set consisting of 8-bit coded characters; used for data interchange among data processing communication systems.

ASSEMBLE — To translate, on a one-to-one basis, from a symbolic program to a binary program by substituting binary operation codes for symbolic operation codes and absolute addresses for symbolic addresses.

ASSEMBLER — A program which translates symbolic language into machine language and assigns memory locations for variables and constants.

BAUD RATE — A unit of signaling speed. In a message in which all characters have the same length, one baud corresponds to a rate of one signal element per second, usually one bit per second.

BINARY — Pertaining to the number system with a radix, or base, of two; uses only two digits: zero (0) and one (1).

BIT — The contracted form of binary digit; a character used to represent one of the two digits in the binary number system. The MPS normally handles 8 bits of data at a time.

BLOCK-OFFSET — A method of describing the absolute address of a given byte of memory. The MPS uses 14-bit addresses. Since the MPS is an 8-bit machine, the 14-bit address must be stored in two consecutive bytes. The low 8 bits are stored in one byte called the offset. The high 6 bits are stored in the next byte called the block. Each block contains 256 bytes of memory. The specific byte within each block is identified by the offset.

BRANCH — A point in a routine where one of two or more choices is made under control of the routine.

BREAKPOINT — A point in a program at which the program can be halted and the results to that point analyzed or preserved for a later continuation of the program. Used primarily in debugging.

BUG — A mistake in the design or implementation of a program resulting in erroneous operations.

BYTE — In the MPS, a group of 8 consecutive binary digits (bits) operated on as a unit.

CALL — To transfer control to a specified routine.

CARRY FLIP-FLOP — A bistable device (capable of assuming either one of two steady states) indicating a carry from the most significant bit in the operation.

CLEAR — To erase the contents of a storage location by replacing the contents with all zeros.

CODING — To write instructions using symbols meaningful to the computer, or to an assembler or other language processor.

COMPLEMENT — <u>One's</u>: To replace all 0 bits with 1 bits and vice versa. <u>Two's</u>: To form the one's complement and add 1.

CONDITION CODE — One of the flip-flops that preserve the results (negative, zero, carry, or parity) of the instruction just completed.

CONTROL — The computer component that affects the carrying out of instructions in the proper sequence, the interpretation of each instruction, and the application of the proper commands to other sections and circuits in accordance with the interpretation.

DATA — A general term used to denote any or all facts, numbers, letters, and symbols. It connotes basic elements of information which can be processed by a computer.

DEBUG — To detect, locate, and correct mistakes (bugs) in a program.

DELIMITER — A character that separates, terminates, and organizes elements of a statement or program.

DESTINATION REGISTER — The register that contains the data to be replaced by or added with the source register. The result of an operation will be placed in the destination register.

DOUBLE-PRECISION — Pertaining to the use of two MPS bytes to represent one number. In the MPS, a double-precision result is stored in 16 bits.

DUMP — To copy the contents of all or part of memory, usually onto an external storage medium.

END BLOCK — The last block of data on the binary output tape that is generated by the assembler. It is identified by a byte count of exactly six.

EXTERNAL STORAGE — A separate facility or device on which data usable by the MPS is stored (such as paper tape or cassette).

EXCLUSIVE-OR — A logical operator having the property that if P is a statement and Q is a statement, then the proposition P exclusive-OR Q is true if either but not both statements are true, false if both are true or both are false. P exclusive-OR Q is often represented by $P \oplus Q$, $P \forall Q$. Contrast with logical OR.

FLAG — A variable or register used to record the status of a program or device. In the latter case, also called a device flag.

FLIP-FLOP — A circuit or device containing active elements, capable of assuming either one of two stable states at a given time.

FLOWCHART — A graphical representation of the operations required to carry out a data processing operation.

HARDWARE — Physical equipment, e.g., mechanical, magnetic, electrical, or electronic devices. Contrast with software.

IMBEDDED LITERAL — A literal that is used in place of a symbol within a source statement.

INITIALIZE — To set counters, switches, and addresses to zero or other starting values at the beginning of, or at prescribed points in, a computer routine. Similar to clear.

INPUT/OUTPUT — A section providing the means of communication between the MPS and external equipment or other computers. Input and output operations involve units of external equipment, certain registers in the computer, and portions of the computer control section. Abbreviated I/O.

INSTRUCTION — A command that causes the MPS to perform a specific operation. Usually also contains the values or locations of the operands required for the operation.

INSTRUCTION REGISTER — In the MPS, one of the 8-bit index registers that decodes each instruction, in turn, to set up the internal circuitry to execute that instruction. Abbreviated IR.

INSTRUCTION SET — A group of mnemonics that are contained in the permanent symbol table of the assembler.

INTERRUPT — An event that breaks the normal operation of the program being executed and passes control to a specific location; generally accompanied by saving the state of the interrupted routine so that control can return later.

JUMP — An instruction that specifies the location of the next instruction and directs the MPS. A jump is used to alter the normal sequence control of the MPS. Under certain conditions, a jump may be contingent upon manual intervention.

JUMP, CONDITIONAL — An instruction which, if a specified condition or set of conditions is satisfied, is interpreted as an unconditional transfer. If the condition is not satisfied, the instruction causes the MPS to proceed in its normal sequence of control. A conditional transfer also includes the testing of the condition.

JUMP, UNCONDITIONAL — An instruction that switches the sequence of control to some specified location. Not contingent on any specified condition.

LEADER — The section of paper tape containing the punched code 200_8 that precedes the binary program.

LEAST SIGNIFICANT DIGIT — The right-most digit of a number. Abbreviated LSD.

LITERALS — Constants that are used in the operand field of a source statement. Ex: LHI 20.

LOAD — To place data into internal storage.

LOCATION — A place in storage or memory where a byte of data or an instruction may be stored.

LOCATION COUNTER — A counter kept by an assembler to determine the address assigned to an instruction or constant being assembled.

LOGICAL AND — A logical operator which has the property that if P is a statement and Q is a statement, the proposition P AND Q is true if both statements are true, false if either is false or both are false. Truth is normally expressed by the value 1, falsity by 0. The AND operator is often represented by a centered dot (P · Q), by a no sign (PQ), by an inverted "u" or logical product symbol (PnQ), or by the letter "x" or multiplication symbol (PxQ). Note that the letters AND are capitalized to differentiate between the logical operator AND and the word and in common usage.

LOGICAL OPERATOR — A symbol which indicates that the two terms of an expression connected by it are to be combined logically.

LOGICAL OR — A logical operator having the property that if P is a statement and Q is a statement, then the OR of P and Q is true if either is true, false if both are false. P OR Q is often represented by P+Q, PVQ. Contrast with exclusive-OR.

LOOP — A sequence of instructions that is executed repeatedly until a terminal condition prevails.

MACHINE LANGUAGE — Information that can be directly processed by the MPS, expressed in binary notation.

MNEMONIC — An alphanumeric designation, easy to remember and commonly used to represent an MPS instruction, e.g., JMP for jump.

MOST SIGNIFICANT DIGIT — The left-most digit of a number. Abbreviated MSD.

MULTIPLEX — To interleave or simultaneously transmit two or more messages on a single data channel.

NESTING TO LEVELS — A level within a program is that portion of code that has a common program counter. In MPS, there are 8 possible levels of code, a main level and seven sublevels. When a subroutine is called, control is transferred to the next sublevel of coding. The ability to jump to these sublevels is called nesting.

OCTAL — Pertaining to the number system with a radix, or base, of eight; uses the digits zero through seven.

OPERAND — A quantity entering or arising in an instruction. An operand may be an argument, a result, a parameter, or an indication of the location of the next instruction, as opposed to the operation code or symbol itself. It may even be the address portion of an instruction.

OVERFLOW — The condition which arises when the result of an arithmetic operation exceeds the capacity of the allotted storage space.

POINTER — A location containing the address to another location.

POST-OPERATOR — An operator that appears after the expression it is to operate on. Ex: In the expression LHI BUF↑, the up-arrow is a post-operator.

PRIORITY ARBITRATION — The process of determining which interrupt, when more than one occurs simultaneously, should be serviced first.

PROGRAM — (1) The complete sequence of instructions and routines necessary to solve a problem. (2) To plan the procedures for solving a problem. This may include the analysis of the problem, preparation of a flowchart, preparing details, developing subroutines, allocation of memory locations, and testing and debugging.

PROM — Programmable Read-Only Memory is any type which is not recorded during its fabrication but which requires a physical operation to program it. Some PROMs can be erased and reprogrammed through special physical processes.

PSEUDO-INSTRUCTION — An instruction to the assembler; an operation code that is not part of the MPS instruction set.

PUSH-DOWN STACK — An area of read/write memory set aside by the processor for temporary storage of subroutine interrupt service linkage. The stack uses the "last-in, first-out" concept.

RADIX — The quantity of characters for use in each of the digital positions of a numbering system. In the most common numbering systems, the characters are some or all of the Arabic numerals, as follows:

System Name	Characters	Radix
Binary	0, 1	2
Octal	0, 1, 2, 3, 4, 5, 6, 7	8
Decimal	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	10

Unless otherwise indicated, the radix of any number is assumed to be 10. For positive identification of a radix 10 number, the radix is written as a subscript to the expressed number; i.e., 126_{10}. The radix of any nondecimal number is expressed in similar fashion; e.g., 11_2 and 15_8.

RAM — Random Access Memory is any type of memory with both read and write capability.

READ — To acquire data from a source.

READ, NON-DESTRUCTIVE — A reading of the information in a register without changing that information.

REDEFINITION WARNING — A warning message given by the assembler when the user attempts to redefine a symbol.

REGISTER — A hardware device used to store one byte (8 bits) of information.

RELATIVE ADDRESSING — A method of expressing an address relative to the current value of the location counter.

RING BUFFER — An area of read/write memory within the processor capable of storing 8 14-bit addresses.

RING COUNTER — A 3-bit register whose value is the location of the currently active program counter. (The "top" of the push-down stack.)

ROM — Read-Only Memory is any type which cannot be rewritten; ROM requires a special operation to permanently record instruction and/or data patterns in it. Compare with PROM memory.

ROUTINE — A set of instructions arranged in proper sequence to cause the computer to perform a desired task. A program or subprogram.

RUN — A single, continuous execution of a program.

SCRATCH PAD — A section of memory that is used for temporary storage of intermediate results or pointers.

SEPARATOR — A character that separates, terminates, and organizes elements of a statement or program.

SHIFT — A movement of bits to the left or right frequently performed in the accumulator. In MPS, also referred to as "rotate."

SIGNED NUMBER — A binary representation in which the most significant bit is reserved for the sign.

SINGLE-PRECISION — The precision used when operations are performed using single 8-bit bytes to represent a quantity.

SOFTWARE — (1) The collection of programs and routines associated with a computer, e.g., compilers; library routines. (2) All the documents associated with a computer, e.g., manuals, circuit diagrams. Contrast with hardware.

SOURCE REGISTER — The register that contains the data to be loaded or added to a destination register. The contents of the source register remains unchanged after an operation.

SUBROUTINE — A series of computer instructions which perform a specific task for another routine. It's distinguishable from a main routine in that it requires, as one of its parameters, a location specifying where to return to in the main program after its function has been accomplished.

SYMBOL TABLE — A table in which symbols and their corresponding values are recorded.

SYMBOLIC LANGUAGE — A coding system in which the mnemonics, or symbols, for the MPS instruction set are used instead of the actual binary machine language.

SYMBOLIC ADDRESS — A set of characters used to specify a memory location within a program.

TEXT BUFFER — A section of memory where text will be stored temporarily until the program is ready to operate on it.

TRAILER — The section of paper tape containing the punched code 200_8 that succeeds the binary program.

UNDEFINED SYMBOL — A symbol appearing in the operand field of a source statement that has not appeared as a label or in a direct assignment statement.

UNDERFLOW — The result of an attempt to subtract a positive number from a positive number of a lesser magnitude.

WORD — In the MPS, synonymous with byte.

WRITE — To record data in a register or location.

Communicating
with DIGITAL's
MPS microprocessor

Reader's Comments

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our publications.

What is your general reaction to this guide? In your judgment is it complete, accurate, well organized, well written, etc.? Is it easy to use? ______

What features are most useful? ______

What faults do you find with the guide? ______

Does this guide satisfy the need you think it was intended to satisfy? ______

Does it satisfy *your* needs? ______ Why? ______

Would you please indicate any factual errors you have found. ______

Please describe your position. ______

Name ______ Organization ______

Street ______ Department ______

City ______ State ______ Zip or Country ______

FOLD HERE

DO NOT TEAR–FOLD HERE AND STAPLE

FIRST CLASS
PERMIT NO. 152
Marlboro, MA 01752

BUSINESS REPLY MAIL—NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

Digital Equipment Corporation
Logic Products
One Iron Way
Marlborough, MA 01752

Attn: MPS Mktg. / M59

digital
LOGIC
PRODUCTS

DIGITAL EQUIPMENT CORPORATION, Component Group Headquarters: 1 Iron Way, Marlborough, Mass. 01752, Telephone: (617) 481-7400

SALES AND SERVICE OFFICES

DOMESTIC — ARIZONA, Phoenix and Tucson • CALIFORNIA, Los Angeles, Monrovia, Oakland, Ridgecrest, San Diego, San Francisco (Mountain View), Santa Ana, Sunnyvale and Woodland Hills • COLORADO, Englewood • CONNECTICUT, Fairfield and Meriden • DISTRICT OF COLUMBIA, Washington (Lanham, Md.) • FLORIDA, Orlando • GEORGIA, Atlanta • ILLINOIS, Chicago (Rolling Meadows) • INDIANA, Indianapolis • IOWA, Bettendorf • KENTUCKY, Louisville • LOUISIANA, Metairie (New Orleans) • MASSACHUSETTS, Marlborough and Waltham • MICHIGAN, Detroit (Farmington Hills) • MINNESOTA, Minneapolis • MISSOURI, Kansas City and St. Louis • NEW HAMPSHIRE, Manchester • NEW JERSEY, Fairfield, Metuchen and Princeton • NEW MEXICO, Albuquerque • NEW YORK, Albany, Huntington Station, Manhattan, Rochester and Syracuse • NORTH CAROLINA, Durham/Chapel Hill • OHIO, Cleveland, Columbus and Dayton • OKLAHOMA, Tulsa • OREGON, Portland • PENNSYLVANIA, Philadelphia (Bluebell) and Pittsburgh • TENNESSEE, Knoxville • TEXAS, Austin, Dallas and Houston • UTAH, Salt Lake City • WASHINGTON, Bellevue • WISCONSIN, Milwaukee (Brookfield) • INTERNATIONAL — ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Canberra, Melbourne, Perth and Sydney • AUSTRIA, Vienna • BELGIUM, Brussels • BOLIVIA, La Paz • BRAZIL, Puerto Alegre, Rio de Janeiro and São Paulo • CANADA, Calgary, Halifax, Montreal, Ottawa, Toronto and Vancouver • CHILE, Santiago • DENMARK, Copenhagen • FINLAND, Helsinki • FRANCE, Grenoble and Paris • GERMANY, Berlin, Cologne, Hannover, Hamburg, Frankfurt, Munich and Stuttgart • HONG KONG • INDIA, Bombay • INDONESIA, Djakarta • ISRAEL, Tel Aviv • ITALY, Milan and Turin • JAPAN, Osaka and Tokyo • MALAYSIA, Kuala Lumpur • MEXICO, Mexico City • NETHERLANDS, Utrecht • NEW ZEALAND, Auckland • NORWAY, Oslo • PHILIPPINES, Manila • PUERTO RICO, Santurce • SINGAPORE • SPAIN, Barcelona and Madrid • SWEDEN, Gothenburg and Stockholm • SWITZERLAND, Geneva and Zurich • TAIWAN, Taipei and Taoyuan • UNITED KINGDOM, Birmingham, Bristol, Dublin, Edinburgh, Leeds, London, Manchester and Reading • VENEZUELA, Caracas • YUGOSLAVIA, Ljubljana •

PRINTED IN U.S.A. EK 01028 75 070/-58 15 05