CHAPTER 4
PDP-8/E PROGRAMMING SYSTEMS

GENERAL

This chapter deals with the concepts required to program the PDP-8/E and identifies the system programs available to the user. Two handbooks, INTRODUCTION TO PROGRAMMING and PROGRAMMING LANGUAGES, provide a more detailed treatment and description of the commonly used programming languages and programming systems.

The chapter is divided into 2 sections. Section 1 provides basic programming guidelines and section 2 identifies the various programming systems and commonly used languages available to the user.

SECTION 1

PDP-8/E PROGRAMMING FUNDAMENTALS

Organization of the standard core memory or any 4096-word field of extended memory is summarized as follows:

Total locations (decimal)	0-4095 or 4096
Total addresses (octal)	0-7777 or 10,000
Number of pages (decimal)	0-31 or 32
Page designations (octal)	0-37 or 40
Number of locations per page (decimal)	0-127 or 128
Addresses within a page (octal)	0-177 or 200

Routines using 128 instructions or less can be written in one page using direct addresses for looping and indirect addresses for data stored in other pages. When planning the location of instructions and data in core memory, the following locations are reserved for special purposes:

Address	Purpose
0 (octal)	Stores the contents of the program counter following a program interrupt.
1 (octal)	Stores the first instruction to be executed following a program interrupt.

10 (octal)—17 (octal) Auto-indexing.

MEMORY ADDRESSING

The programmer has 4096 (decimal) locations which he may address. However, as illustrated in Figure 3-2 of Chapter 3, when an instruction is fetched from memory, only bits 5 through 11 contain the address of the data. Addressing is accomplished using octal notation. Therefore the 4096 possible locations require addresses in octal from 0000 to 7777. This means that a total of 12 bits is required to specify an absolute address. So that all locations may be addressed as efficiently as possible, memory is addressed in terms of pages with a coding scheme that allows easy access to any one of the 10,000 octal locations. The page addressing scheme is illustrated in Figure 4-1 which shows the relationship of the 40 octal pages with the 10,000 octal locations. The programmer is interested in only three pages in memory at any one time:

a. The current page
b. Page 0
c. A location on other than the current page or page 0.

Page 0 is used to store commonly used operands and off-page pointers. For instance, the location of an indirect address used by instructions is usually on Page 0.

ABSOLUTE ADDR. (OCTAL)	CORE MEMORY PAGE (OCTAL)	PAGE ADDR. (OCTAL)
7777 7600	37	177 000
7577 7400	36	177 000
7377 7200	35	177 000
7177 7000	34	177 000
6777 6600	33	177 000
6577 6400	32	177 000
6377 6200	31	177 000
6177 6000	30	177 000
5777 5600	27	177 000
5577 5400	26	177 000
5377 5200	25	177 000
5177 5000	24	177 000
4777 4600	23	177 000
4577 4400	22	177 000
4377 4200	21	177 000
4177 4000	20	177 000
3777 3600	17	177 000
3577 3400	16	177 000
3377 3200	15	177 000
3177 3000	14	177 000
2777 2600	13	177 000
2577 2400	12	177 000
2377 2200	11	177 000
2177 2000	10	177 000
1777 1600	7	177 000
1577 1400	6	177 000
1377 1200	5	177 000
1177 1000	4	177 000
0777 0600	3	177 000
0577 0400	2	177 000
0377 0200	1	177 000
0177 0000	0	177 000

Figure 4-1 Memory Addressing Scheme

The format of the Central Processor (CPMA) Memory Address Register establishing the memory page must first be considered. The MA register is an unequally divided 12-bit register in which the least significant bits of the MA (bits 5-11) are called the page address bits and the most significant bits (bits 0-4) are called the page bits. The 12 bits of the Memory Address are established by the program counter (PC) and are re-established each time the PC loads an absolute address for the next instruction. Bits 0-4 are used to establish the memory page as shown in Figure 4-2. Because the pages to be addressed include pages 0-37 (octal), only five bits are required. The first two bits represent numbers from 0 to 3 and the next three represent numbers from 0 to 7. Because the locations to be addressed on any given page include locations 0-177 (octal), only seven bits are required to specify any one location. Bit 5 represents an octal 1 or 0; bits 6, 7, and 8 represent the second octal digit from 0 to 7; bits 9, 10, and 11 represent the last octal digit from 0 to 7. Thus, on the example shown in Figure 4-2, the MA register is addressing page 5, location 73 (absolute address 1273).

When the user first receives his PDP-8/E, he should assume that it has no information content in its memory. Before he can load instructions into memory, he must first perform the initializing and loading procedures described in Chapter 2. The following discussion assumes that the preliminary procedures have been completed and that the programmer now wants to load into core those instructions which will be called upon after a program has been written. His main concern is to decide in which memory locations he desires to place his instructions and in which locations he wishes to place the corresponding data.

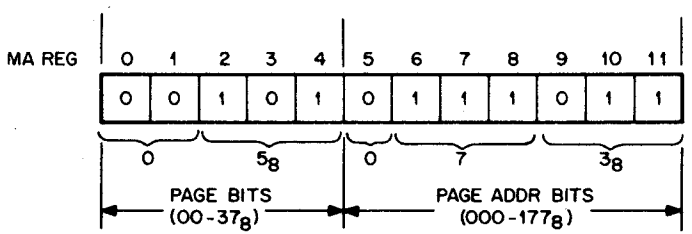


Figure 4-2 Format Establishing Memory Page

Initially, the programmer must load the Central Processor Memory Address Register with an address and then deposit a 12-bit instruction word in the format shown in Figure 4-3.

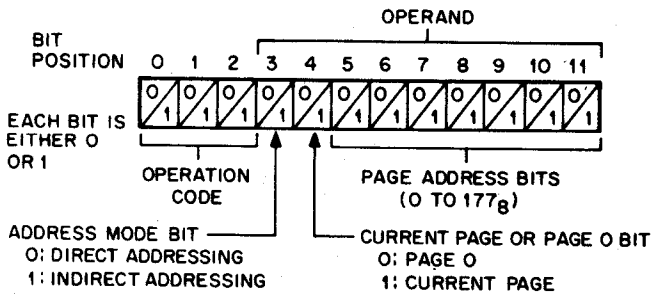


Figure 4-3 Format of a Memory Reference Instruction

The first three bits contain the instruction operation code and have nothing to do with addressing. The last seven bits address the location (from 0 to 177 octal), which will contain either data or a 12-bit address. Those address bits are located in the Memory Buffer Register, and are ineffective until bits 3 and 4 are decoded, at which time the address bits are transferred from the Memory Buffer to the Central Processor Memory Address Register. The page address (the first five bits of the MA register) is determined by whether bit 4 is a 0 or 1 (see Figure 4-4).

Where to place the instruction word or Data word is a very important consideration. At this point, the programmer has three choices in the location of the data:

a. the current page (that page containing the Instruction)
b. page 0
c. a page other than page 0 or the current page.

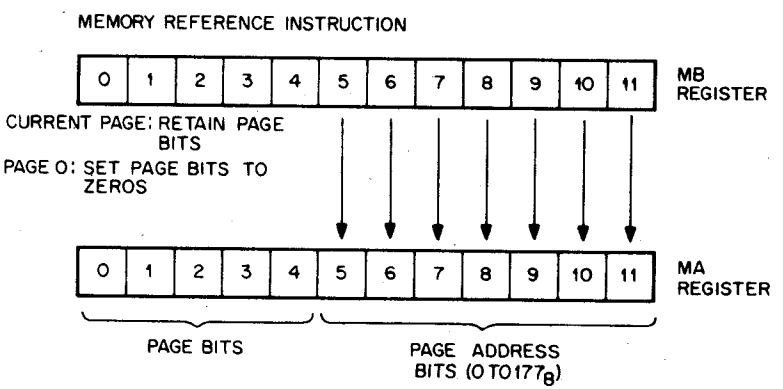


Figure 4-4 Transfer of Address Portion of MRI into MA Register

Current Page—If the programmer desires the data to be located in the current page, he must make bit 4 a 1 in the original instruction word. The logic within the processor causes the first five bits of the MA to remain, and transfers the last seven bits of the MB (the new address within a page) to the last seven bits of the MA Register. This method of updating the MA Register is illustrated in Figure 4-4.

Page 0—Page 0 is commonly used to store operands or address of operands or routines. The programmer must set bit 4 word to 0. The logic within the processor then places all zeros in MA bits 0 through 4 and transfers the content of the last 7 bits of the MB register to the last 7 bits of the MA Register. Thus, the page address is now page 0 and the address within page 0 is some address between 0 and 177 octal. This is illustrated in Figure 4-4.

Addressing A Page Other Than the Current Page or Page 0—The programmer may address a page other than the current page or page 0 by placing a 1 in bit 3 of the original instruction word. As before, the computer then goes to an address on the current page or on page 0, depending on the state of bit 4. The logic within the processor responds to bit 3 being a 1 by going into a defer state for a new address. This procedure is called "Indirect Addressing."

INDIRECT ADDRESSING

In the preceding section, the method of directly addressing 400(octal) memory locations by an MRI was described—namely those on page 0 and those on the current page. This section describes the method for addressing the other 7400(octal) memory locations. Bit 3 of an MRI designates the address mode. When bit 3 is a 0, the operand is a direct address. When bit 3 is a 1, the operand is an indirect address. An indirect address (pointer address) identifies the location that contains the desired address (effective address). To address a location that is not directly addressable, the absolute address of the desired location is stored in one of the 400(octal) directly addressable locations (pointer address); the pointer address is written as the operand of the MRI; and the letter I is written between the mnemonic and the operand. (During assembly, the presence of the I results in bit 3 of the MRI being set to 1.) Upon execution, the MRI will operate on the contents of the location identified by the address contained in the pointer location.

The two examples in Figure 4-5 illustrate the difference between direct addressing and indirect addressing. The first example shows a TAD instruction that uses direct addressing to get data stored on page 0 in location 50; the second is a TAD instruction that uses indirect addressing, with a pointer on page 0 in location 50, to obtain data stored in location 1275. (When references are made to them from various pages, constants and pointer addresses can be stored on page 0 to avoid the necessity of storing them on each applicable page.) The octal value 1050, in the first example, represents direct addressing (bit 3 = 0); the octal value 1450, in the second example, represents indirect addressing (bit 3 = 1). Both examples assume that the accumulator has previously been cleared.

Location	Content	
200	TAD 50	(TAD 50 = 1050_8)
.	.	Instruction
.	.	
50	1275	Data (Number) To Be Acted Upon By
.	.	Instruction Address
.	.	
1275	20	(Content of location 1275 is not used in the execution of the instruction in location 200.)

NOTE: AC = 1275 after executing the instruction in location 200

Location	Content	
200	TAD I 50	(TAD I 50 = 1450_8)
.	.	
.	.	Designates Indirect Addressing
.	.	Instruction
50	1275	Pointer Address
.	.	
.	.	
1275	20	Data (Number) To Be Acted Upon By Instruction Effective Address

NOTE: AC = 20 after executing the instruction in location 200.

Figure 4-5 Comparison of Direct and Indirect Addressing

The following three examples illustrate some additional ways in which indirect addressing can be used. As shown in example 1, indirect addressing makes it possible to transfer program control from off page 0 (or any other page) to any desired memory location. (Similarly, indirect addressing makes it possible for other memory reference instructions to address any of the 4,096(10) memory locations.) Example 2 shows a DCA instruction that uses indirect addressing with a pointer on the current page. The pointer in this case designates a location off the current page (location 227) in which the data is to be stored. (A pointer address is normally stored on the current page when all references to the designated location are from the current page.) Indirect addressing provides the means for returning to a main program from a subroutine, as shown in example 3. Indirect addressing is also effectively used in manipulating tables of data.

EXAMPLE 1

Location	Content	
75	JMP I 100	(JMP I 100 = 5500(octal))
.	.	
.	.	Designates Indirect Addressing Instruction

100	6000	Pointer Address
.	.	
.	.	
.	.	
6000	DCA 6100	Next Instruction To Be Executed
.	.	
.	.	

NOTE: Execution of the instruction in location 75 causes program control to be transferred to location 6000, and the next instruction to be executed is the DCA 6100 instruction.

EXAMPLE 2

Location	Content	
450	DCA I 577	(DCA I 577 = 3777(octal)) Designates Indirect Addressing Instruction
577	277	Pointer Address
.	.	
.	.	
227	nnnn	Data (Number) Stored By Instruction (Effective Address)

NOTES: 1. Memory Location 577 is location 177 of current page. Execution of the instruction in location 450 causes the contents of the accumulator to be stored in location 227.

EXAMPLE 3

Location	Content	
207	JMS I 70	(JMS I 70 = 4470(octal))
210	TAD 250	(The next instruction to be executed upon return from the subroutine.)
.	.	
.	.	
70	2000	(Starting address of the subroutine stored here.)
.	.	
.	.	
2000	aaaa	(Return address stored here by JMS instruction.)
2001	i i i	(First instruction of subroutine.)
.	.	
.	.	
2077	JMP I 2000	(Last instruction of subroutine.)

NOTES: 1. Execution of the instruction in location 207 causes the address 210 to be stored in location 2000 and the instruction in location 2001 to be executed next. Execution of the subroutine proceeds until the last instruction (JMP I 2000) causes control to be transferred back to the main program, continuing with the execution of the instruction stored in location 210.

2. A JMS instruction that uses indirect addressing is useful when the subroutine is too large to store on the current page.
3. Storing the pointer address on page 0 enables instructions on various pages to have access to the subroutine.

PROGRAMMING OPERATIONS

The programmer can make use of any combination of instructions. The following sections describe the more common programming operations.

STORING AND LOADING

Data is stored in any core memory location by use of the DCA (Deposit & Clear AC) instruction. This instruction clears the AC to simplify loading of the next data. If the data deposited is required in the AC for the next program operation, the DCA must be followed by a TAD for the same address. All loading of core memory information into the AC is accomplished by means of the TAD instruction.

The DCA instruction stores the contents of the AC in the referenced location, destroying the original contents of the location. The AC is then set to all zeroes. The following example shows the contents of the accumulator, link, and location 225 before and after executing the instruction DCA 225.

DCA 225

	AC	Link	Loc. 225
Before Execution	1234	1	7654
After Execution	0000	1	1234

The following facts should be kept in mind when using the DCA instruction:

a. The state of the link bit is not altered.
b. The AC is cleared.
c. The original contents of the addressed location are replaced by the contents of the AC.

PROGRAM CONTROL

The Program Counter is used to direct the processor to the next address of the next instruction to be fetched. Therefore, the content of the PC Register is always one more than the content of the Central Processor Memory Address (CPMA) Register. When an instruction has been completed and the processor is ready to go into a new fetch, the content of the Program Counter is transferred into the CPMA Register and the Program Counter with its original address is incremented by +1, thereby pointing to the next sequential address. This procedure is called Program Control because it directs the processor to the next instruction. Because this rigid sequence of instructions is not always desirable for practical programming, the PDP-8/E provides a means of jumping out of this sequence to transfer Program Control from one sequence of instructions to another or to allow the processor to enter a subroutine which is itself a sequence of instructions and re-enter the main program when the subroutine has been completed.

Transfer of program control to any core memory location uses the JMP or JMS instructions. The JMP I and JMS I (indirect address, bit 3 = 1) are used to transfer program control to any location in core memory which is not in the current page or page 0.

The JMS Y is used to enter a subroutine which starts at location Y +1 in the current page or page 0. The contents of the PC are stored in the specified address Y, and address Y + 1 is transferred into the PC. Subroutines or other pages may be entered via an indirect JMS. To exit a subroutine, the last instruction is a JMP I Y, which returns program control to the location stored in Y.

The JMP instruction loads the effective address of the instruction into the program counter, thereby changing the program sequence since the PC specifies the next instruction to be performed. In the following example, execution of the instruction in location 250 (JMP 300) causes the program to jump over the instructions in locations 251 through 277 and immediately transfer control to the instruction in location 300.

```
Location    Content
   250      JMP 300     (This instruction transfers program
    .           .       control to location 300.)
    .           .
   300      DCA 300
    .           .
```

NOTE: The JMP instruction does not affect the contents of the AC or link.

A program written to perform a specific operation often includes sets of instructions which perform intermediate tasks. These intermediate tasks may be finding a square root, or typing a character on a keyboard. Such operations are often performed many times in the running of one program and may be coded as subroutines. To eliminate the need of writing the complete set of instructions each time the operation must be performed, the JMS (jump to subroutine) instruction is used. The JMS instruction stores a pointer address in the first location of the subroutine and transfers control to the second location of the subroutine. After the subroutine is executed, the pointer address identifies the next instruction to be executed. Thus, the programmer has at his disposal a simple means of exiting from the normal flow of his program to perform an intermediate task and a means of returning to the correct location upon completion of the task. (This return is accomplished using indirect addressing, which is discussed elsewhere in this chapter.)

The following example illustrates the action of the JMS instruction:

```
Location    Content
PROGRAM
   200      JMS 350     (This instruction stores 0201 in loca-
                         tion 350 and transfers program con-
                         trol to location 351.)
```



```
   201      DCA 270     (This instruction stores the contents
                         of the AC in location 270 upon return
                         from the subroutine.)
    .           .
    .           .
    .           .
SUBROUTINE
   350      0000        (This location is assumed to have an
                         initial value of 0000; after JMS 350
                         is executed, it is 0201.)
   351      i i i       (First instruction of subroutine)
    .           .
    .           .
    .           .
   375      JMP I 350   (Last instruction of subroutine)
```

The following should be kept in mind when using the JMS:

1. The value of the PC (the address of the JMS instruction +1) is always stored in the first location of the subroutine, replacing the original contents.
2. Program control is always transferred to the location designated by the operand + 1 (second location of the subroutine).
3. The normal return from a subroutine is made by using an indirect JMP to the first location of the subroutine (JMP I 350 in the above example); (Indirect addressing, as discussed in this chapter effectively transfers control to location 201).
4. When the results of the subroutine processing are contained in the AC and are to be used in the main program, they must be stored upon return from the subroutine before further calculations are performed. (In the above example, the results of the subroutine processing are stored in location 270.)

ARITHMETIC OPERATIONS

One arithmetic instruction is included in the order code, the two's complement add (TAD). Using this instruction, routines can easily be written to perform addition, subtraction, multiplication, and division in two's complement arithmetic.

Two's Complement Arithmetic

In two's complement arithmetic, addition, subtraction, multiplication, and division of binary numbers are performed in accordance with the common rules of binary arithmetic. In the PDP-8/E, as in other machines utilizing complementation techniques, negative numbers are represented as the complements of positive numbers, and subtraction is achieved by complement addition. Representation of negative values in one's complement arithmetic is slightly different from that in two's complement arithmetic.

The one's complement of a number is the complement of the absolute positive value; that is, all 1s are replaced by 0s and all 0s are replaced by 1s. The two's complement of a number is equal to one plus the one's complement of the number.

In one's complement arithmetic a carry from the sign bit (most significant bit) is added to the least significant bit in an end-around carry. In two's complement arithmetic a carry from the sign bit complements the link (a carry would set the link to 1 if it were properly cleared before the operation), and there is no end-around carry.

The TAD instruction (see Figure 4-6) performs a binary addition between the specified data word and the contents of the accumulator, leaving the result of the addition in the accumulator. If a carry out of the most significant bit of the accumulator should occur, the state of the link bit is complemented. The add instruction is called a Two's Complement Add to remind the programmer that negative numbers must be expressed as the two's complement of the positive value.

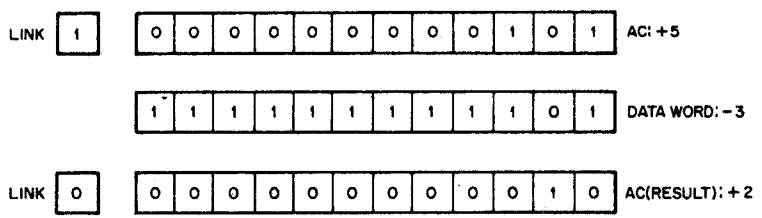


Figure 4-6 Operation of the TAD Instruction

The following points should be remembered when using the TAD instruction:

a. Negative numbers must be expressed as a two's complement of the positive value of the number.
b. A carry out of the accumulator will complement the link.
c. The data word in the referenced location is not affected.

LOGIC OPERATIONS

The PDP-8/E instruction list includes the logic instruction AND. A short routine can be written from this instruction to perform the exclusive OR operation.

Logical AND

The logical AND operation between the contents of the accumulator and the contents of a core memory location Y is performed directly by means of the AND Y instruction. The logical AND performs an AND operation with AC0 and MB0, AC1 and MB1, etc. The result remains in the AC, the original contents of the AC are lost, and the contents of location Y are unaffected.

The AND instruction causes a bit-by-bit Boolean AND operation between the contents of the accumulator and the data word specified by the instruction. The result is left in the accumulator as shown in Figure 4-7.

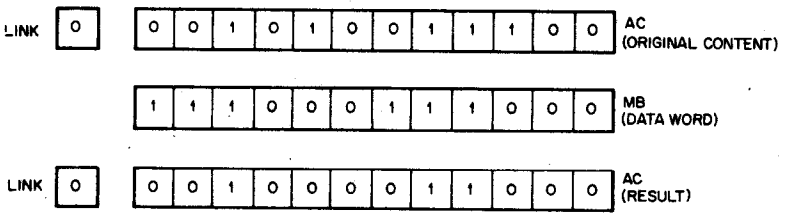


Figure 4-7 Operation of the AND Instruction

The following points should be noted with respect to the AND instruction:

a. A 1 appears in the AC only when a 1 is present in both the AC and the data word (The data word is often referred to as a mask).
b. The state of the link bit is not affected by the AND instruction.
c. The data word in the referenced location is not altered.

Inclusive OR

The Inclusive OR instruction makes use of the MQ register, which is a permanent part of the PDP-8/E. Assuming that value A is in the AC and value B is stored in a known core memory address, the following sequence performs the inclusive OR. The sequence is stated as a utility subroutine called IOR.

```
/CALLING SEQUENCE              JMS IOR
/                              (ADDRESS OF B)
/ENTER WITH ARGUMENT IN AC; EXIT WITH
/LOGICAL RESULT IN AC

/ADDRESS LABEL INSTRUCTION          REMARKS
   IOR,        0
               MQL             /AC TO MQ, CLEAR AC
               TAD I IOR       /GET ADDRESS OF SECOND
               DCA TEM1        ARGUMENT
               TAD I TEM1
               MQA             /IOR MQ TO AC
               ISZ IOR
               JMP I IOR
   TEM1,       0
```

Exclusive OR

The exclusive OR operation for two numbers, A and B, can be performed by a subroutine called by the mnemonic code XOR. In the following general purpose XOR subroutine, the value A is assumed to be in the AC, and the address of the value B is assumed to be stored in a known core memory location.

```
/CALLING SEQUENCE        JMS XOR
/                        (ADDRESS OF B)
/                        (RETURN)
      /ENTER WITH ARGUMENT IN AC; EXIT WITH
            /LOGICAL RESULT IN AC
```

```
XOR,    0
        DCA TEM1
        TAD I XOR
        DCA TEM2
        TAD TEM1
        AND I TEM2
        CMA IAC
        CLL RAL
        TAD TEM1
        TAD I TEM2
        ISZ XOR
        JMP I XOR
TEM1,   0
TEM2,   0
```

An XOR subroutine can be written using fewer core memory locations by making use of the IOR subroutine; however, such a subroutine takes more time to execute. A faster XOR subroutine can be written by storing the value B instead of the address of B, in the second instruction of the calling sequence; however, the resulting subroutine is not as useful as the subroutine given here.

INDEXING OPERATIONS

External events can be counted by the program, and the count can be stored in core memory. The core memory location used to store the event count can be initialized (cleared) by a CLA command followed by a DCA instruction. Each time the event occurs, the event count can be advanced by a sequence of commands such as CLA, TAD, IAC, and DCA.

The ISZ instruction is used to count repetitive program operations or external events without disturbing the contents of the accumulator. Counting a specified number of operations is performed by storing a two's complement negative number equivalent to the number of operations to be counted. Each time the operation is performed, the ISZ instruction is used to increment the contents of this stored number and to check the result. When the stored number becomes zero, the specified number of operations have occurred and the program skips out of the loop and back to the main sequence.

This instruction is also used for other routines in which the contents of a memory location are incremented without disturbing the contents of the accumulator, such as storing information from an I/O device in sequential memory locations, or using core memory locations to count I/O device events.

The ISZ instruction adds a 1 to the referenced data word and then examines the result of the addition. If a zero result occurs, the instruction following the ISZ is skipped. If the result is not zero, the instruction following the ISZ is performed. In either case, the result of the addition replaces the original data word in memory. The example below illustrates one method of adding the contents of a given location to the AC a specified number of times (multiplying) by using an ISZ instruction to increment a tally. The effect of this example is to multiply the contents of location 275 by 2. (To add the contents of a given location to the AC

twice, using the ISZ loop, as shown below, requires more instructions than merely repeating the TAD instruction or using a rotate instruction. However, when adding the contents four or more times, use of the ISZ loop requires fewer instructions.) In the first pass of the example, execution of ISZ 250 increments the contents of location 250 from 7776 to 7777 and then transfers control to the following instruction (JMP 200). In the second pass, execution of ISZ 250 increments the contents of location 250 from 7777 to 0000 and transfers control to the instruction in location 203, skipping over location 202.

CODING FOR ISZ LOOP

Location	Content
200	TAD 275
201	ISZ 250
202	JMP 200
203	DCA 276
.	.
.	.
.	.
250	7776
.	.
.	.
.	.
275	0100
276	0000

SEQUENCE OF EXECUTION FOR ISZ LOOP

		Content After Instruction Execution			
Location	Content	AC	250	275	276
FIRST PASS					
200	TAD 275	0100	7776	0100	0000
201	ISZ 250	0100	7777	0100	0000
202	JMP 200	0100	7777	0100	0000
SECOND PASS					
200	TAD 275	0200	7777	0100	0000
201	ISZ 250	0200	0000	0100	0000
202	JMP 200	(Skipped	during	second	pass)
203	DCA 276	0000	0000	0100	0200

ISZ Instruction Incrementing a Tally

The following points should be kept in mind when using the ISZ instruction:

a. The contents of the AC and link are not disturbed.
b. The original word is replaced in main memory by the incremented value.
c. When using the ISZ for looping a specified number of times, the tally must be set to the negative of the desired number.
d. The ISZ performs the incrementation first and then checks for a zero result.

CODING A PROGRAM

The introduction of an assembler in Chapter 2 enabled the programmer to write a symbolic program using meaningful mnemonic codes rather than the octal representation of the instructions. The programmer could now write mnemonic programs such as the following example, which multiplies 18(10) by 36(10) using successive addition.

```
200     CLA CLL     Initialize
201     TAD 210     Set up a Tally
202     CIA         equal to —18(10) to
203     DCA 212     count the additions of 36
204     TAD 211     Add 36
205     ISZ  212    Skip if Tally is 0
206     JMP 204     Add another 36 if not done
207     HLT         Stop after 18 times
210     0022        Equal to 18(10)
211     0044        Equal to 36(10)
212     0000        Holds the tally
```

Writing the above program was greatly simplified because mnemonic codes were used for the octal instructions. However, writing down the absolute address of each instruction is clearly an inconvenience. If the programmer later adds or deletes instructions, thus altering the location assignments of his program, he has to rewrite those instructions whose operands refer to the altered assignments. If the programmer 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.

Location Assignment

As in the previous program example, most programs are written in successive memory locations. If the programmer assigned an absolute location to the first instruction, the assembler could be told to assign the next instructions to the following locations in order. In programming the PDP-8/E 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. With this improvement incorporated, and with the use of symbolic addresses, the previous example appears as shown in the following example.

```
*200
START,  CLA   CLL
        TAD   A
        CIA
        DCA   TALLY
        TAD   B
        ISZ   TALLY
        JMP   START +4
        HLT
    A,  0022
    B,  0044
TALLY,  0000
```

NOTE: In this example, CLA CLL is stored in location 200 and the successive instructions are stored in 201, 202, etc.

WRITING SUBROUTINES

Included in the memory reference instructions, given in Chapter 3, was the instruction JMS (jump to subroutine). This instruction is a modified JMP command which makes possible a later return to the point of departure from the main program. The JMS instruction automatically stores the location of the next instruction after the JMS in the location to which the program is instructed to jump, thereby enabling a return.

The programmer need only terminate the subroutine with an indirect JMP to the first location of the subroutine in order to return to the next instruction following the JMS instruction. The following simple program illustrates the use of a subroutine to double a number contained in the accumulator.

(Main Program)

```
START,  CLA CLL
        TAD N           Get the number in the AC
        JMS DOUBLE      Jump to subroutine to double N
        DCA TWON        First instruction after the subroutine
          .
          .
          .
N,      nnnn            Any number, N
TWON,   nnnn            2N will be stored here
```

Subroutine

```
DOUBLE,   0000
          CLL RAL       Rotate left, multiplying by 2
          SNL           Did overflow occur?
          JMP I DOUBLE
          RAR           If overflow occurs, display the num-
                        ber to be doubled in the AC and
          HLT           then stop the computer.
```

Notice that the first instruction of the subroutine is located in the second location of the subroutine. Any instruction stored in location DOUBLE would be lost when the return address is stored. Also note that the subroutine as it is written must be located on page 0 or current page, because it is directly addressed. (A subroutine is often located on another page and addressed indirectly as the next example demonstrates.)

The following program multiplies a number in the accumulator by a number stored in the location immediately following the JMS instruction.

Main Program

```
*200
START,          TAD A
                DCA .+3
                TAD B
                JMS I 30
                0000
                DCA PRDUCT
                .
                .
                .
PRDUCT,         0000
A,              0051
B,              0027
*30

                MULT
```

Subroutine

```
*6000
MULT,           0000
                CIA
                DCA MTALLY
                TAD I MULT
                ISZ MTALLY
                JMP .-2
                ISZ MULT
                JMP I MULT
MTALLY,         0000
```

The preceding example illustrates the following important points.

a. The JMS I 30 instruction could be used anywhere in core memory to jump to this subroutine because the pointer word (stored in location 30) is located on page 0, and all pages of memory can reference page 0.
b. The period was used to denote the current location in the instructions DCA .+3 and JMP .−2.
c. Since the result of the subroutine is left in the AC when jumping back to the main program, the next instruction should store the result for future use.
d. The first instruction of the subroutine is in location MULT + 1 since the next address in the main program is stored in MULT by the JMS instruction.
e. The first two instructions of the subroutine set the tally with the negative of the number in the AC.
f. The second number to be multiplied is brought into the subroutine by the TAD I MULT instruction, as it is stored in the location specified by the address that the JMS instruction automatically stores in the first location of the subroutine. This is a common technique for transferring information into a subroutine.

g. The ISZ MTALLY instruction is used in the subroutine to count the number of additions. The ISZ MULT instruction is used to increment the contents of MULT by one, thereby making the return jump (JMP I MULT) proceed to the next instruction after the location that held the number to be multiplied.
h. An interesting modification of the previous program is achieved through defining a "new operation" MLTPLY by including in the coding the statement MLTPLY = JMS I 30. The assembler would make a replacement in such a way that any time the programmer writes MLTPLY the computer would perform a jump to the subroutine and return to the program with the product in the AC.

ADDRESS MODIFICATION

A very powerful tool often used by the programmer is address modification, meaning the inclusion of instructions in a program to modify the operand portion of a memory reference instruction. It is a particularly useful technique when working with large blocks of stored data as illustrated by the two programs that follow (see Figure 4-8).

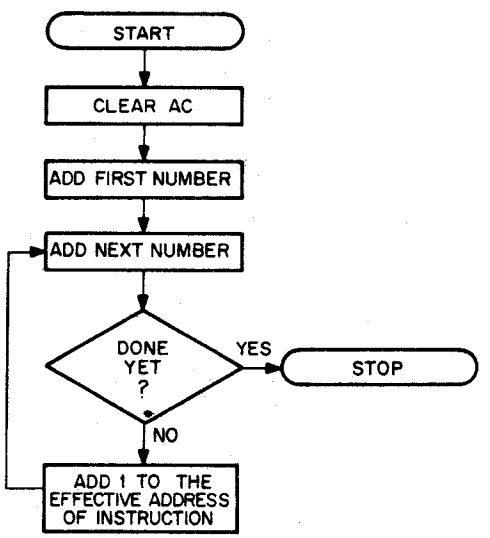


Figure 4-8 Address Modification

```
*200
START,          CLA CLL
                TAD K200
                CIA
                DCA TALLY
                TAD K4000
                DCA NUM
                TAD K4200
                DCA RESULT
```

```
AGAIN,          TAD I NUM
                JMS SQUARE
                DCA I RESULT
                ISZ RESULT
                ISZ NUM
                ISZ TALLY
                JMP AGAIN
                HLT
K200,           0200
TALLY,          0000
K4000,          4000
NUM,            0000
K4200, 4200
RESULT,         0000
*300
SQUARE,         0000
                DCA STORE
                TAD STORE
                CIA
                DCA COUNT
                TAD STORE
                ISZ COUNT
                JMP .—2
                JMP I SQUARE
STORE,          0000
COUNT,          0000
$
```

It will be noted that the first eight instructions are concerned with initializing the program. This initializing enables the stored program to be restarted several times and still operate on the correct locations. If the program had merely incremented locations K4000 and K4200 and utilized those locations for indirect addressing, it would only operate on the correct locations on the first running. On successive runnings, the program would be operating on successively higher locations in memory. With the program written as shown, however, the pointer words are automatically reset. This procedure is often referred to as "housekeeping."

LOOPING A PROGRAM

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

There are two basic methods of creating a program loop. The first method is using an ISZ (2nnn(octal)) instruction to count the number of passes made through the loop. The ISZ is usually followed by a JMP 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 Group 2 Operate Microinstructions 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 link bit, rather than until a predetermined number of passes are made.

To illustrate the use of an ISZ instruction in a program loop situation, consider the following program which simply sets the contents of all addresses from 2000 to 2777 to zero.

```
*200
CLEAR,      CLA
            TAD CONST
            DCA COUNT           /SET COUNT TO —1000.
            TAD TTABLE
            DCA STABLE          /SET STABLE TO 2000.
            DCA I STABLE        /CLEAR ONE LOCATION.
            ISZ STABLE          /SELECT NEXT LOCATION.
            ISZ COUNT           /IS OPERATION COMPLETE?
            JMP .—3             /NO: REPEAT.
            HLT                 /YES: HALT.
CONST,      7000                /2'S COMP OF 1000.
COUNT,      0
TTABLE,     TABLE
STABLE,     0                   /POINTER TO TABLE.
*2000
TABLE,      0
$
```

Several points should be carefully noted.

a. The first five instructions initialize the loop, but are not in it. The location COUNT is set to —1000 at the beginning, and 1 is added to it during each passage of the loop. After the 1000th (octal) passage, COUNT goes to zero, and the program skips the JMP instruction, and executes the HLT instruction. On each previous occasion, it executed the JMP instruction.
b. In the list of constants following the HLT instruction, TTABLE contains TABLE, which is defined below as having the value 2000, and containing 0. Therefore, STABLE contains 2000 initially. In order to understand this point, it must be remembered that an asterisk character causes the first location after the asterisk to be set to the value after the asterisk. Therefore, *in the previous example CLEAR equals 200 and TABLE equals* 2000.
c. ISZ STABLE adds 1 to the contents of location STABLE, forming 2001 on the first pass, 2002 on the second pass, and so on. Since it never reaches zero, it will never skip. This is a very

common use. It is said to be indexing the addresses from 2000 to 2777. (When using an ISZ instruction in this way, the programmer must be certain that it does not reach 0. Follow the ISZ instruction with a NOP if it does reach 0 so that the resulting skip will not modify the program sequence.)

d. For every ISZ instruction used in a program, there must be two initializing instructions before the loop, and there must be a constant and a counting location in a table of constants. This procedure allows the program to be rerun with the counting locations reset to the correct values.

The following program utilizes a Group 2 skip instruction to create a loop. The program will search all of core memory to find the first occurrence of the octal number 1234.

```
*0
NUMBER,     1234
*200
BEGIN,      CLA CLL
            TAD NUMBER
            CIA
            DCA COMPARE       /STORES MINUS NUMBER.
            DCA ENTRY         /SETS ENTRY TO 0.
REPEAT,     ISZ ENTRY         /INCREASES ENTRY.
            NOP
            TAD I ENTRY       /COMPARISON IS
            TAD COMPARE       /DONE HERE.
            SZA CLA
            JMP REPEAT
            TAD ENTRY
            HLT               /ENTRY IS IN AC.
COMPARE,    0
ENTRY,      0
$
```

This example shows that the program searches itself as well as all other core memory locations, and points up the following points:

a. The ISZ entry instruction is used to index the locations to be tested. The next instruction (NOP) is unnecessary; thus, if ENTRY becomes zero during the course of the program, the program will not be affected. It is important to protect against an ISZ instruction going to zero and skipping a necessary part of a program, if the ISZ is being used simply to index.

b. The number to be searched for is stored in location 0, and the search starts in location 1. Therefore, the program will find at least one occurrence of the number, and will halt after one complete pass through memory, if not before.

c. The program could be modified to bound the area of the search. If the contents of ENTRY are set equal to one less than the desired start location and the number being searched for is put in the location following the last location to be searched, the program will search only the designated area of memory.

d. The program could be restarted at location REPEAT in order to find a second occurrence of 1234 after being halted by the first occurrence.

AUTO-INDEXING

The PDP-8/E computer has eight special registers in page 0; locations 0010 through 0017. Whenever these locations are addressed indirectly by a memory reference instruction, the content of the register is incremented before it is used as the operand of the instruction. These locations can, therefore, be used in place of an ISZ instruction in an indexing application. Because of this, these eight locations are called autoindex registers. Autoindex registers act as any other location when addressed directly. The autoindexing feature is performed only when the location is addressed indirectly.

The following examples below are a modification of the first program example in the preceding section with an autoindex register used in place of the ISZ instruction. (The purpose of the program is to clear memory locations 2000 through 2777.)

Carefully notice the difference between the two examples, especially that TABLE now has to be set to TABLE-1 since this is incremented by the autoindexing register before being used for the first time. This point must be remembered when using an autoindex register. The register increments before the operation takes place; therefore, it must always be set to one less than the first value of the addresses to be indexed.

```
*10
INDEX,      0
*200
CLEAR,      CLA
            TAD CONST
            DCA COUNT
            TAD TTABLE
            DCA INDEX
            DCA I INDEX
            ISZ COUNT
            JMP .-2
            HLT
CONST,      7000
COUNT,      0
TTABLE,     TABLE-1
*2000
TABLE,      0
$
```

The memory search example of the preceding section could also be simplified using an autoindex register as shown below.

```
*0
NUMBER,     1234
*10
ENTRY,      0
*200
BEGIN,      CLA CLL
            TAD NUMBER
```

Notice that in this case ENTRY originally equals 0 because its content is incremented before being used to obtain data for the comparison.

```
            CIA
            DCA COMPARE
            DCA ENTRY
REPEAT,     TAD I ENTRY
            TAD COMPARE
            SZA CLA
            JMP REPEAT
            TAD ENTRY
            HLT
COMPARE,    0
$
```

PROGRAM DELAYS

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

The following is a simple delay routine using the ISZ instruction for an inner loop and an outer loop. When analyzing the example it should be remembered that the PDP-8/E represents only positive numbers up to 3777(octal) or 2047(10). Therefore, the computer counts up to 2047 (10) and then continues to count starting at the next octal number 4000(octal), which the computer interprets as −2048(10). Successive increments of this number will finally bring the count to zero. Thus, a location could be used to count from 1 up to 0 by using an ISZ instruction.

```
                (main program)
                .
                .
                .
            TAD CONST       /START OF DELAY ROUTINE
            DCA COUNT
            ISZ COUNT 1     /INNER
            JMP .−1         /LOOP
            ISZ COUNT
            JMP .−3
CONST,      6030            /SETS DELAY
COUNT,      0
COUNT 1     0
```

PROGRAM BRANCHING

Very few meaningful programs are written which do not take advantage

of the computer'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. The Group 2 microinstructions presented previously are most often used for this purpose. The ISZ instruction often referred to as a conditional skip instruction, also provides a branch in a program. This instruction operates upon the contents of a memory location, while the Group 2 microinstructions test the contents of the AC and L.

A typical example of a conditional skip would be a program to compare A and B and to reverse their order if B is larger than A (see Figure 4-9).

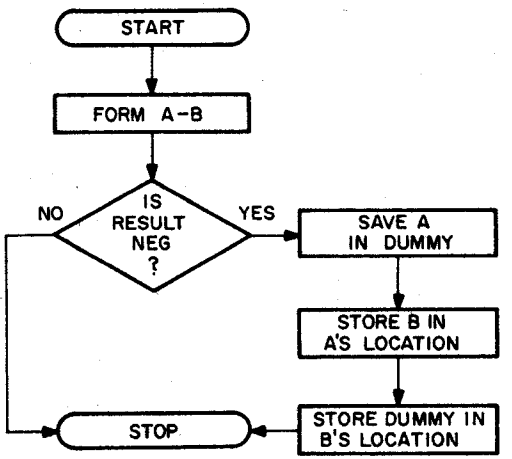


Figure 4-9 Conditional Skip

```
*200
TEST,       CLA CLL
            TAD B           /SUBTRACT B
            CIA             /FROM A
            TAD A           /HERE.
            SMA CLA
            HLT             /STOP HERE IF A IS GREATER
                            /OR EQUAL
            TAD A           /THE REMAINDER OF
            DCA DUMMY       /THE PROGRAM
            TAD B           /DOES THE SWITCH.
            DCA A
            TAD DUMMY
            DCA B
            HLT
A,          1234            /SUBSTITUTE ANY POSITIVE
B,          2460            /VALUES FOR A AND B.
DUMMY,      0
$
```

If A is less than B, their difference will be negative and the HALT will be skipped. The program will proceed to reverse the order of A and B. If A is greater than or equal to B, the program will halt.

MICROPROGRAMMING

Because PDP-8/E instructions of Group 1, Group 2, and Group 3 are determined by bit assignment, these instructions may be combined, or microprogrammed, to form new instructions enabling the computer to do more operations in less time.

Combining Microinstructions

The programmer should make certain that the program clears the accumulator and link before any arithmetic operations are performed. To perform this task, the program might include the following instructions (given in both octal and mnemonic form).

```
CLA        7200 (octal)
CLL        7100 (octal)
```

However, when the Group 1 instruction format is analyzed, Figure 4-10 illustrates the result.

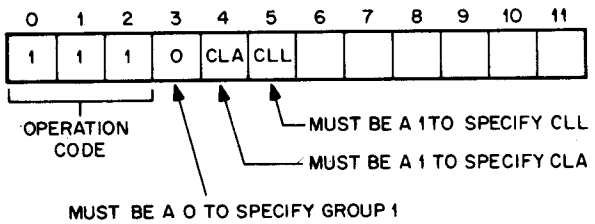


Figure 4-10 Group 1 Bit Assignments

Since the CLA and the CLL instructions occupy separate bit positions, they may be expressed in the same instruction, thus combining the two operations into one instruction. This instruction would be written as follows.

```
CLA CLL    7300 (octal)
```

In this manner, many operate microinstructions can be combined, making the execution of the program much more efficient. The assembler for the PDP-8/E will combine the instructions properly when they are written as above; that is, on the same coding line, separated by a space.

Illegal Combinations

Microprogramming, although very efficient, can also be troublesome for the new programmer. There are many violations of coding which the assembler will not accept.

One rule to remember is: "If you can't code it, the computer can't do it." In other words, the programmer could write a string of mnemonic microinstructions, but unless these microinstructions can be coded

correctly in octal representation, they cannot be performed. To illustrate this fact, suppose the programmer would like to complement the accumulator (CMA), complement the link (CML), and then skip on a nonzero link (SNL). He could write the following.

```
CMA        CML        SNL
```

These instructions require the bit assignments shown in Figure 4-11.

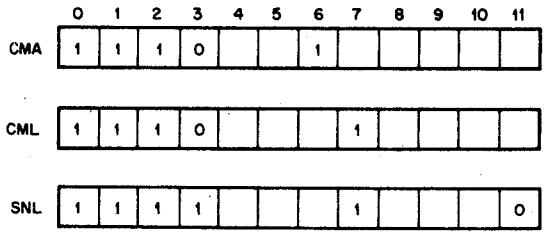


Figure 4-11 Example of Illegal Combinations

The three microinstructions cannot be combined in one instruction because bit 3 is required to be a 0 and a 1 simultaneously. Therefore, no instructions may be used which combine Group 1 and Group 2 microinstructions because bit 3 usage is not compatible. The CMA and CML can, however, be combined because their bit assignments are compatible. The combination would be as follows.

```
CMA CML    7060 (octal)
```

To perform the original set of three operations, two instructions are needed.

```
CMA CML    7060 (octal)
SNL        7420 (octal)
```

Because Group 1 and Group 2 microinstructions cannot be combined, the commonly used microinstruction CLA is a member of both groups. Clearing the AC is often required in a program and it is very convenient to be able to microprogram the CLA with the members of both groups.

```
RAR        7010 (octal)
RTR        7012 (octal)
```

Although he can write the instruction "RAR RTR," it cannot be correctly converted to octal by the assembler because of the conflict in bit 10; therefore, it is illegal.

Combining Skip Microinstructions
Group 2 operate microinstructions use bit 8 to determine the instruction specified by bits 5, 6, and 7 as previously described. If bit 8 is a 0, the instructions SMA, SZA, and SNL are specified. If bit 8 is a 1, the instructions SPA, SNA, and SZL are specified. Thus, SMA cannot be combined with SZL because of the opposite values of bit 8.

OR GROUP—SMA OR SZA OR SNL
If bit 8 is a 0, the instruction skips on the logical OR of the conditions specified by the separate microinstructions. The next instruction is skipped if any of the stated conditions exist. For example, the combined microinstruction SMA SNL will skip under the following conditions:

a. The accumulator is negative, the link is zero.
b. The link is nonzero, the accumulator is not negative.
c. The accumulator is negative and the link is nonzero.

(It will not skip if all conditions fail.) This manner of combining the test conditions is described as the logical OR of the conditions.

AND GROUP—SPA AND SNA AND SZL
A value of bit 8 = 1 specifies the group of microinstructions SPA, SNA, and SZL, which combine to form instructions that act according to the logical AND of the conditions. In other words, the next instruction is skipped only if all conditions are satisfied. For example, the instruction SPA SZL will cause a skip of the next instruction only if the accumulator is positive and the link is zero. (It will not skip if either of the conditions fail.)

NOTES:
1. The programmer is not able to specify the manner of combination. The SMA, SZA, SNL conditions are always combined by the logical OR, and the SPA, SNA, SZL conditions are always joined by a logical AND.
2. Since the SPA microinstruction will skip on either a positive or a zero accumulator, to skip on a strictly positive (positive, nonzero) accumulator the combined microinstruction SPA SNA is used.

Order of Execution of Combined Microinstructions
The combined microinstructions are performed by the computer in a very definite sequence. When written separately, the order of execution of the instructions is the order in which they are encountered in the program. In writing a combined instruction of Group 1 or Group 2 microinstructions, the order written has no bearing upon the order of execution. This should be clear, because the combined instruction is a 12-bit binary number with certain bits set to a value of 1. The order in which the bits are set to 1 has no bearing on the final execution of the whole binary word.

GROUP 1

1. CLA, CLL—Clear the accumulator and/or clear the link are the first actions performed. They are effectively performed simultaneously and yet independently.
2. CMA, CML—Complement the accumulator and/or complement the link. These operations are also effectively performed simultaneously and independently.
3. IAC—Increment the accumulator. This operation is performed third, allowing a number in the AC to be complemented and then incremented by 1, thereby forming the two's complement, or negative, of the number.
4. RAR, RAL, RTR, RTL, BSW—The rotate instructions are performed last in sequence. Because of the bit assignment previously discussed, only one of the five operations may be performed in each combined instruction.

GROUP 2

1. Either SMA or SZA or SNL when bit 8 is a 0. Both SPA and SNA and SZL when bit 8 is a 1. Combined microinstructions specifying a skip are performed first. The microinstructions are combined to form one specific test; therefore, skip instructions are effectively performed simultaneously.
   Because of bit 8, only members of one skip group may be combined in an instruction.
2. CLA—Clear the accumulator. This instruction is performed second in sequence, allowing different arithmetic operations to be performed after testing (see Event 1) without the necessity of clearing the accumulator with a separate instruction before some subsequent arithmetic operation.
3. OSR—Inclusive OR between the switch register and the AC. This instruction is performed third in sequence, allowing the AC to be cleared first, and then loaded from the switch register.
4. HLT—The HLT is performed last to allow any other operations to be concluded before the program stops.

This is the order in which all combined instructions are performed. In order to perform operations in a different order, the instructions must be written separately, as shown in the following example. The following combined microinstruction looks as if it might clear the accumulator, perform an inclusive OR between the SR and the AC, and then skip on a nonzero accumulator.

```
CLA OSR SNA
```

However, the instruction would not perform in that manner, because the SNA would be executed first. In order to perform the skip last, the instructions must be separated, as follows:

```
CLA OSR
SNA
```

Microprogramming requires that the programmer carefully code mnemonics legally so that the instruction actually does what he desires it

to do. The sequence in which the operations are performed and the legality of combinations are crucial to PDP-8/E programming.

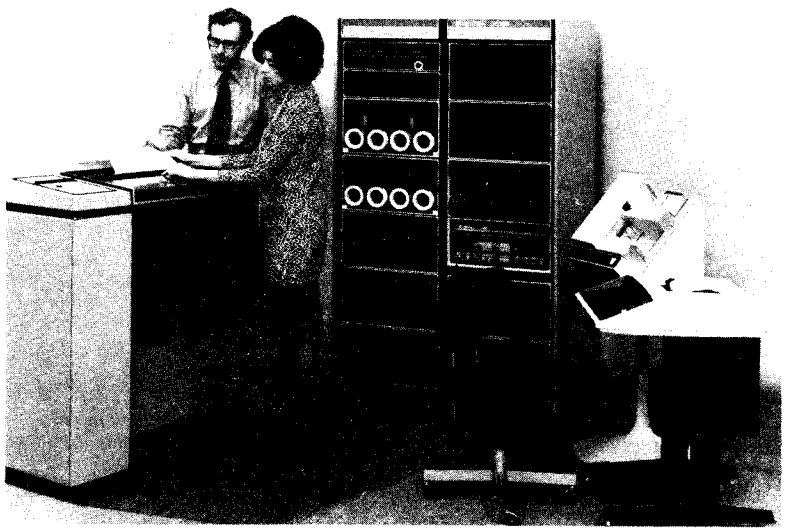
The following is a list of commonly used combined microinstructions, some of which have been assigned a separate mnemonic.

	INSTRUCTION		EXPLANATION
—	CLA	CLL	Clear the accumulator and link.
CIA	CMA	IAC	Compliment and increment the accumulator. (Sets the accumulator equal to its own negative.)
LAS	CLA	OSR	Load accumulator from switches. (Loads the accumulator with the value of the switch register.)
STL	CLL	CML	Set the link (to a 1).
—	CLA	IAC	Sets the accumulator to a 1.
STA	CLA	CMA	Sets the accumulator to a −1.

In summary, the basic rules for combining operate microinstructions are as follows:

a. Group 1 and Group 2 microinstructions cannot be combined.
b. Rotate microinstructions (Group 1) cannot be combined with each other.
c. OR Group (SMA, SZA, or SNL) microinstructions cannot be combined with AND Group (SPA, SNA, or SZL) microinstructions.
d. OR Group microinstructions are combined as the logical OR of their respective skip conditions. AND Group microinstructions are combined as the logical AND of their respective skip conditions.
e. Order of execution for combined instructions is listed below.

Group 1	Group 2
1. CLA, CLL	1. SMA/SZA/SNL (OR group) or SPA/SNA/SZL (AND group)
2. CMA, CML	2. CLA
3. IAC	3. OSR
4. RAR, RAL, RTR, RTL, BSW	4. HLT



For the modern business manager, DEC offers complete data processing capability with on-line input/output devices such as a line printer, DECwriter, card reader, high-speed reader/punch; for mass storage, the DECtape and/or Disk systems are provided with complete file handling programs.

SECTION 2

PDP-8/E SYSTEM PROGRAMS

The Programming System for the PDP-8/E consists of SYSTEM PROGRAMS, UTILITY PROGRAMS, and APPLICATION PROGRAMS, and is complemented with the DECUS LIBRARY (see Figure 4-12).

More than 1000 PDP-8 programs are available to the user. Digital Equipment Corporation's Program Library, for instance, offers more than 700 programs from which to choose. In addition, a library containing programs developed by PDP-8 users, called the "DECUS LIBRARY," is available to all PDP-8 users. These programs cover a wide variety of applications in addition to the application programs developed by DEC. The programming system was designed to simplify and accelerate the process of learning to program. At the same time, experienced programmers will find that it incorporates many advanced features. The system is intended to make immediately available to each user the full, general-purpose data processing capability of the computer and to serve as the operating nucleus for a growing library of programs and routines available to all installations.

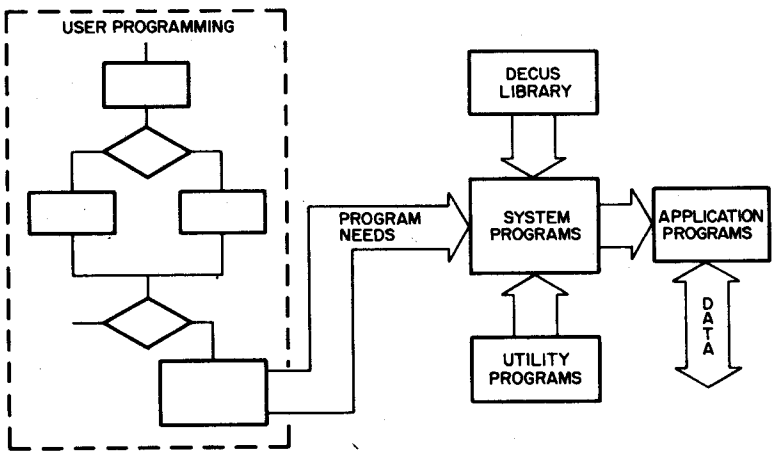


Figure 4-12 PDP-8/E Programming System

PDP-8/E Software Kit

A basic Family-of-8 software kit for the PDP-8/E computer is provided with each basic PDP-8/E system. The user receives one or more paper tapes and the necessary documentation for each program. Those programs provided in the basic software kit are identified by an asterisk (*) in the following program descriptions.

The following material is included in the basic PDP-8/E software kit:

PDP-8/E Small Computer Handbook
Introduction to Programming Handbook, 1970
Programming Languages Manual, 1970
Logic Handbook
PDP-8/E Instruction Cards (2 Cards)
FOCAL-69 Binary Tape and Utility Overlay Tape
PAL-III Symbolic Assembler
DDT Symbolic Debugging Program
ODT Octal Debugging Program (2 Tapes)
Symbolic Editor Program
RIM Punch (ASR-33 version)
Binary Punch (ASR-33 version)
Octal Memory Dump Program
Floating Point Subroutines (4 Tapes)
Software Performance Summary
A Complete Set of Maintenance (Diagnostic) Software

System Programs

The system programs are used, combined with the utility programs and/or the DECUS Library Program, to translate the user's ideas into desired application programs. System Programs include: Monitors, Editors, Assemblers, Compilers, Interpretive Languages, Debuggers, and Loaders.

Monitor Programs

Monitor programs used with the PDP-8/E include PS/8, Disk Monitor, and TSE.

PS/8 PROGRAMMING SYSTEM—PS/8, an 8K programming system, represents a significant advance in software development for small computers with capabilities which were formerly available only on such powerful machines as the PDP-10.

The PS/8 is a program development system for the PDP-8/E with minimum 8K of core and one or more of the following mass storage devices:

a. TC08/TU56 DECtape
b. RF08 Disk
c. RK8 Disk Pack
d. DF32 Disk (64K minimum)
e. TD8-E DECtape (with 12K Core)

The 8K programming system has the following features:

a. It allows the user device-independent access to up to 15 I/O devices, including up to eight DECtapes, up to four Disk units, Teletype, high speed paper tape reader and punch, card reader, line printer, and any other device for which it is possible to write a device handler in one, or in some cases two, pages.
b. The user program may call upon the monitor for several services, including loading device handlers for devices to which the user may assign a name, looking up input files on these devices, creating and closing variable and fixed-length output files on these devices, and getting and decoding a line of input from the console Teletype that identifies input and output files and options.

PS/8 gives the user, in addition to the language processor (8K FOCAL, 8K FORTRAN, PAL-8, and SABR), absolute and relocatable loaders, a symbolic editor, CONVERT (a program to provide file compatibility with the present Disk Monitor System), PIP (Peripheral Interchange Program), and an invisible ODT (Octal Debugging Technique), which allows the programmer to debug programs without giving up valuable core space.

Advantages offered by the PS/8 System include:

a. Device Independence—
   1. Programs can be run using the most effective I/O devices available at a given installation.
   2. As an installation grows, programs need not be rewritten for increased I/O capability.
   3. A user device handler may easily be added.
b. User Interfacing—
   1. User may use any standard I/O device without having to directly program the device.
   2. User may use the system command decoder to vary the I/O used in the programs.
c. Performance—Increased, due to efficient use of storage devices (especially noticeable on DECtape).
d. Expandability—As the system grows, so does the capability of the PS/8 system.
e. The PS/8 Library of Programs.
f. Increased power, derived from the addition of 4K of core to an existing 4K disk monitor system.
g. A program to convert all 4K monitor files to PS/8 files.
h. The capability of accommodating any amount of core storage from 8K to 32K.

Disk Monitor System
A keyboard monitor is available to Disk System users that allows them to save core images on the Disk and restore them to memory. The extensive software package available consists of: FORTRAN Compiler, Program Assembly Language (PAL-D), Editor Program (Editor), Peripheral Interchange Program (PIP), and Dynamic Debugging Technique (DDT-D) Program.

In addition, the user may save and restore his own core images and use the remainder of the available device storage for temporary storage of source or binary data. The monitor system may also be used with DECtape.

Time-Sharing Monitor
TSE (Time-Sharing PDP-8/E) is a general purpose, stand-alone monitor time-sharing system. The TSE can accommodate up to 16 users simultaneously. A minimum of 12K of core memory and an RF08-type disk is required for a comprehensive library of system programs, which provide facilities for compiling, assembling, editing, loading, saving, calling, debugging, and running user programs on line.

The center of a TSE system is a complex of programs called Monitor. Monitor coordinates the operations of the various units, allocates the time and services of the computer to users, and controls their access to the system. The computer works on user programs simultaneously by segregating the central processor operations from the time-consuming interactions of the human users. Execution of various programs are interspersed without interfering with one another and without detectable delays in the responses to the individual user.

Consult the DECUS Library for additional monitor programs.

EDITOR PROGRAM
Symbolic Paper Tape Editor*
The Symbolic Paper Tape Editor program is used to edit, correct, and update symbolic program tapes using the PDP-8/E, the Teletype unit, and/or the high-speed reader. With Editor in core memory, the user reads in portions of his symbolic tape, removes, changes, or adds instructions, and gets back a complete new symbolic tape with errors removed. He can work through the program instruction, spot check it, or concentrate on new sections. A character string search is available. The user can move one or more lines of text from one place to another. The program requires 4K core and a Teletype. Consult the DECUS Library for additional editor programs.

ASSEMBLER PROGRAMS
Assembler Programs used with the PDP-8/E include PAL-III, PAL-D, MACRO-8, and 8K-SABR. The use of an assembly program has become a standard practice in programming digital computers. This process allows the programmer to code instructions in a symbolic language, one he can work with more conveniently then the 12-bit binary numbers that actually operate the computer. The assembly program translates the symbolic language program into its machine code equivalent. The advantages are significant: the symbolic language is more meaningful and convenient to a programmer than a numeric code; instructions or data can be referred to by symbolic names without concern for, or even knowledge of, their actual addresses in core memory; decimal and alphabetical data can be expressed in a form more convenient than binary numbers; programs can be altered more efficiently and debugging is considerably simplified

PAL-III*—PAL-III is a basic assembler allowing symbolic references, origins, and expressions. The output is in a form suitable for input to the binary loader.

PAL-III is a two-pass assembler with an optional third pass; i.e., the symbolic program tape must be passed through the assembler two times to produce the binary-coded tape, and the optional third pass produces a complete octal symbolic program listing, which can be typed and/or punched.

*Part of the basic software package.

PAL-III accepts symbolic program tapes from either the low-speed or high-speed reader and produces the binary tapes on either the low-speed or high-speed punch.

During assembly, the programmer communicates with PAL-III via the switches on the computer console. Switch options are used to specify which pass the assembler is to perform and which reader and punch the assembler should accept input from and punch out on.
PAL-III requires 4K of core memory and a Teletype.

PAL-D—PAL-D incorporates most of the features of both PAL-III and MACRO-8, and is used only in the Disk Monitor System. PAL-D is designed primarily for 4K PDP-8/E computers with disk.

PAL-8—PAL-8 is an extended assembler which runs under the PS/8 Programming System. It includes the best features of PAL-III and MACRO-8 plus a number of additional features:

- Conditional assembly.
- Large symbol table (up to 1800 symbols) (12K core).
- High speed Binary symbol table search.
- Paginated listings with page headings and page numbers.

MACRO-8*—MACRO-8 is an advanced assembler which has the same features as PAL-III, plus the following additional features: user-defined macros, double precision integers, floating-point constants, arithmetic and Boolean operators, literals, text facilities, and automatic off-page linkage generation. To incorporate such features, the size of the user's symbol table was decreased. However, the programmer can increase or decrease the size of the permanent symbol table at the expense of some of the more space-consuming features.

MACRO-8 requires 4K of core memory and a Teletype.

8K SABR—8K SABR (Symbolic Assembler for Binary Relocatable Programs) is an advanced one-pass symbolic assembler. It translates symbolic programs written in the SABR language into binary relocatable code acceptable to the computer. SABR programs are core page independent. Therefore, programs may be written without regard to the 128-word core page of the computer. SABR automatically generates off-page and off-field references for direct or indirect statements. It also automatically connects instructions on one page to those that overflow onto the next. The list of available pseudo-ops is extensive, including external subroutine calling, argument passing, and conditional assembly. SABR offers an optional second pass to produce a side-by-side octal/symbolic listing of the assembled program.

The relocatable binary tapes produced by SABR are loaded into any field of core memory using the 8K linking Loader, as are the comprehensive library of subprograms. These subprograms may be called by any SABR program.

*Part of the basic software package.

The high speed reader and punch is recommended for the 8K SABR program.

Consult the DECUS Library for additional assembler programs.

COMPILER PROGRAMS
Compiler programs used with the PDP-8/E include DIBOL, FORTRAN 4K and 8K, and ALGOL-8. Although the DIBOL program is listed with compiler programs, DIBOL includes a package which contains a Monitor Program, an Editor Program, and utility programs.

DIBOL Software System—DIBOL (**DI**gital Equipment Corporation **B**usiness **O**riented **L**anguage) is the first business language that was designed specifically for a minicomputer. The DIBOL Software System is a complete business oriented software system for implementing business applications such as billing, Accounts Receivable, Inventory Control, Cost Accounting, payroll, Accounts Payable, Sales Analysis, and many other accounting and business management functions. This software system comprises a simple business oriented language, a data management system to provide input, sorting, editing and filing facilities, and a monitor program to organize the DIBOL facilities into a unified system. These components of the software system enable you to develop business applications "your way".

The DIBOL configuration consists of:

a. One PDP-8/E,
b. One high speed paper tape reader and punch,
c. Four DECtapes,
d. One ASR-33,
e. 8K of core,
f. LE8 Line Printer, and
g. The DIBOL Software System Package.

Refer to appendix A for more details on the DIBOL Software System.

4K FORTRAN—The 4K FORTRAN (for FORmula TRANslation) compiler lets the user express the problem he is trying to solve in a mixture of English words and mathematical statements that is close to the language of mathematics and is also intelligible to the computer.

4K FORTRAN consists of a compiler, a debugging aid, and an operating system. The one-pass compiler translates FORTRAN coded symbolic language statements into binary code and produces a binary tape. The debugging aid (Symbolprint) lists the variables used and their locations in core and indicates the section of core used by the compiled program. The program requires 4K of core memory and a Teletype.

Document: 4K FORTRAN Programmer's Reference Manual

8K FORTRAN Compiler—
The PDP-8 Paper Tape System version of 8K FORTRAN has the following features: subroutines, two levels of subscripting, function subprograms, relocatable output, COMMON statements, library subroutines, six types of format specifications, and I/O supervisors. 8K FORTRAN requires the use of the 8K SABR assembler and Linking Loader.

This compiler utilizes all available core from 8K to 32K, and correctly loads programs over page boundaries.

The program requires 8K of core memory, a Teletype and high speed Reader/Punch.

INTERPRETIVE PROGRAMS
Interpretive Programs used with the PDP-8/E include FOCAL and BASIC.

FOCAL-8—FOCAL-8 (FOrmula CALculator) is an on-line conversational language designed for solving complicated calculations. The language consists of short statements and mathematical expressions in standard notation. FOCAL puts the full calculating power and speed of the computer at the user's fingertips without the user having to master the intricacies of machine-language programming. FOCAL is an easy way of simulating mathematical models, plotting curves, handling set of simultaneous equations, and much more.

FOCAL is available in several configurations.

*a) Single-user FOCAL—requires 4K and Teletype.
b) Four-user FOCAL—requires 8K core and four Teletypes (console plus three).
c) Seven-user FOCAL requires 8K core, seven Teletypes (console plus six), and a RF08 or DF32D disk.

BASIC 8
BASIC 8 is a modified version of the algebraic language developed at Dartmouth College. The BASIC language is composed of easy-to-learn English statements and mathematical expressions.

BASIC 8 is available in five versions:

a. EDUSYST-10 One user—requires basic processor and Teletype,
b. EDUSYST-20 Two to five users—requires basic processor with 8K memory and two to five Teletypes.
c. EDUSYST-30 One user Batch—requires basic processor, teletype, and either DECtape or Disk (RF08 or DF32-D),
d. EDUSYST-40 Combination of b and c.
e. EDUSYST-50 Eight to sixteen users—multiple language capability—requires basic processor with 8K memory, DECtape or Disk (RF08 or DF32-D), and eight to sixteen Teletypes.

DEBUGGER PROGRAMS
Debugger Programs used with the PDP-8/E include ODT-8 and DDT-8.

DDT-8 (Dynamic Debugging Techniques)*—On-line debugging with DDT-8 gives the user dynamic printed program status information. It gives him close control over program execution, preventing errors ("bugs") from destroying other portions of his program. He can monitor the execution of single instructions or subsections, change instructions or data in any format, and output a corrected program at the end of the debugging session.

*Part of the basic software package.

Using the standard Teletype, the user can communicate conveniently with the PDP-8/E in the symbols of his source language. He can control the execution of any portion of his object program by inserting breaks, or traps, in it. When the computer reaches a break, it transfers control of the object program to DDT. The user can then examine and modify the content of individual core memory registers to correct and improve his object program.

DDT-8 requires 4K of core memory and a Teletype.

ODT-8 (Octal Debugging Technique)*—ODT-8 allows the programmer to do all the things mentioned in DDT-8 by communicating with his object program using the octal representation of his binary program. ODT-8 occupies less core storage than DDT-8 and can be loaded in upper memory or lower memory, depending on where the binary program resides.

ODT-8 requires 4K of core memory and a Teletype.
Consult the DECUS Library for additional debugging programs.

LOADERS
Read-In Mode (RIM) Loader*
The RIM Loader is a minimum routine for reading and storing information contained in read-in mode coded tapes via the Teletype or high speed paper tape reader.

Binary Loader*
The Binary Loader is a short routine for reading and storing information contained in binary-coded tapes, using the Teletype or high-speed paper tape reader.

The Binary Loader accepts tapes prepared by the use of PAL or MACRO-8. Diagnostic messages may be included on tapes produced when using either PAL or MACRO. The Binary Loader ignores all diagnostic messages. See Appendix A and the DECUS Library listing for additional loader programs.

Linking Loader
The Linking Loader is capable of loading and linking a user's program and subprograms in any field(s) of memory. The Linking Loader has options which can obtain storage map listings of core availability for the user.

The Linking Loader has the capability to search program libraries for subroutines which are referenced by the program in core and load those subroutines needed. A library is a collection of relocatable subroutines (FORTRAN or SABR output) with a directory at the beginning to facilitate searching.

The Linking Loader is capable of loading any number of user and library programs into any field of memory. Several programs are usually loaded into each field. Because of the space reserved for the Linkage Routines, the available space in field 0 is three pages smaller than in all other fields.

*Part of the basic software package.

UTILITY PROGRAMS

PDP-8/E utility programs provide printouts or punchouts of core memory content in octal, decimal, or binary form, as specified by the user. Subroutines are provided for octal or decimal data transfer and binary-to-decimal, decimal-to-binary, and Teletype tape conversion.

Most of these programs require only 4K of core memory and a Teletype. Some of the primary utility programs include data conversion programs and maintenance and diagnostic programs (see Appendix A for a listing of other utility programs).

Data Conversion Programs—Data conversion programs used with the PDP-8/E include the Floating Point Package and Math Function Routines.

Octal Memory Dump—This program enables the user to dump in octal mode any or all data in any memory field to either the Teletype or high-speed paper tape punch. During dumping, the absolute address of each location being dumped is held in the accumulator. When dumping is completed, output devices and memory fields can be changed to dump another section of memory. The program requires one core page.

Mathematical Function Routines

The programming system includes a set of mathematical function routines to perform the following operations: single precision multiplication, division, square root; double precision sine, cosine, multiply, divide; and arithmetic and logical shifts.

Floating Point Package*

The Floating Point Package permits the PDP-8/E to perform arithmetic operations that many other computers can perform only after the addition of costly optional hardware. Floating point operands retain the maximum precision available by discarding leading zeros. In addition to increasing accuracy, floating point operations relieve the programmer of scaling problems common in fixed-point operations, a particularly advantageous feature to the inexperienced programmer. The floating point subroutines and interpreter permit the programmer to encode arithmetic operations to either six or ten decimal digits of precision as easily as though the machine had floating point hardware. Also included in the package are input and output conversion routines.

Any common storage reserved by the programs being loaded is allocated in field 1 from location 200 upwards. The space reserved for common storage is subtracted from the available loading area in field 1. The program reserving the largest amount of common storage must be loaded first.

The Run-Time Linkage Routines necessary to execute SABR programs are automatically loaded into the required areas of every field by the Linking Loader as part of its initialization. The user needs to know nothing more about these routines than the particular areas of core they occupy.

*Part of the basic software package.

MAINTENANCE AND DIAGNOSTIC PROGRAMS

A complete set of standard diagnostic programs is provided (see Appendix A) to simplify and expedite system maintenance. Program descriptions and manuals permit the user to effectively test the operation of the computer for proper core memory functioning and proper execution of instructions. In addition, diagnostic programs to check the performance of standard and optional peripheral devices are provided with the devices.

Consult the DECUS Library for additional utility programs.

THE DECUS LIBRARY

Although each PDP-8/E is delivered to the user complete with an extensive program kit, the DECUS Library provides the user with a continually growing assortment of various programs which are available for general use. DECUS includes a wide variety of system programs, utility programs, and application programs. A listing of the programs available from the DECUS Library can be obtained from the Program Library, Digital Equipment Corporation, Maynard, Mass. 01754.

APPLICATION PROGRAMS

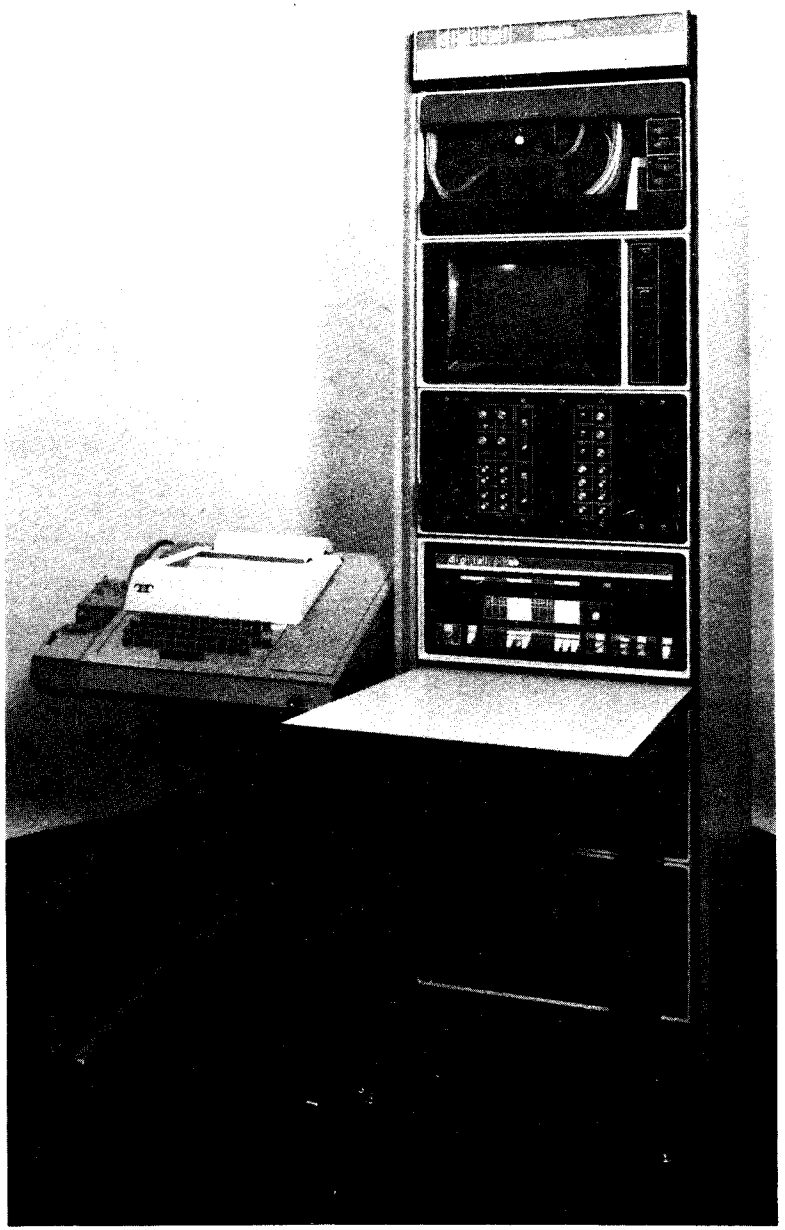
Integrated Application Packs are special-purpose software packages which provide a specific solution to one aspect of a general problem. By using the proper application pack and additional hardware and/or software, the user can have a computer system customized to fit his particular requirements. Although numerous programs are offered by the DECUS Library, some of the commonly used application programs include:

Display 8
IDAC 8
LAB 8/e
EDUSYSTEMS
DIBOL
Quick Point 8
Typeset 8
PHA 8
Chromatographic Data Processing (CDP)

LAB-8/E System Software

The LAB-8/E is a major step forward in low-cost laboratory computing. At a cost lower than most special purpose instruments, LAB-8/E includes an analog-to-digital converter, real time clock with three Schmitt triggers, point plot display control, and Digital's newest small general purpose computer, the PDP-8/E.

*Part of the basic software package.



LAB-8/E

The LAB-8/E is designed to be used as a total laboratory system, not a computer with laboratory-type peripherals. The peripherals have been designed to plug into the laboratory option cabinet H945. The Schmitt triggers may start the clock, the clock may start the analog-to-digital converter, the analog-to-digital converter may increment the multiplexer, and so on. This kind of flexibility lets you configure an interactive LAB-8/E system from a PDP-8/E and lets you expand the system to suit your particular needs, in most cases without the expense of extra cabinets, space, and cabling.

A significant feature of the LAB-8/E is its software. Programs developed for over 11,000 "family of 8" computers are compatible with the PDP-8/E. Special groups of applications software developed for the LAB-8/E allow many users to put their system to work on the day it arrives. Users may share their programs through DECUS, one of the world's largest computer users groups.

The LAB-8/E is being used in biomedical research for EEG, ECG, EMG, Behavior Studies, diagnostic assistance, patient monitoring, and similar applications. In analytical instrumentation, LAB-8/E is used for NMR work, electrochemistry, kenetic studies, reporting, automation of instruments, etc. In engineering and science, the LAB-8/E is used in simulation techniques, and laboratory applications in physics, biology and psychology. In industrial testing, LAB-8/E is used for material testing, sound and vibration analysis and real-time data acquisition and data analysis.

The LAB-8/E features some unique software packages such as BASIC and Advanced Averages, Auto and Cross Correlation, NMR Averager, Simulator, Fast Fourier Transforms, Histrogram programs, DAQUAN (data acquisition and analysis), and real-time BASIC, the easy to learn, conversational language. Refer to the LAB-8/E Users Handbook, DEC-LB-HRZA-D for more information.

INDAC SOFTWARE FOR IDACS-8 SYSTEMS

INDAC software package is designed for real time data acquisition and control applications. This field proven software presents the total system capability to a design or process engineer by integrating various process interfaces and allowing English like statements for data input/output in a real time environment.

INDAC system software fully supports INDAC language. It is a BASIC like language with special commands for program scheduling as a function of time, sequence or external event. Communication between process variables and computer is achieved by much simplified input/output commands.

The elements of INDAC software are:

- SYSTEM BUILDER—GENDAC—A conversational program allowing system configuration for a specific installation. It allows adding *new I/O handlers, library routines, etc. to an INDAC system.*
- INDAC COMPILER—The compiler converts a source program into object code, which is executable at runtime. It provides extensive diagnostics to help debug source program.

- INDAC EXECUTIVE—The executive schedules various tasks at runtime, supervises allocation of disk and core, handles clock interrupts, controls the flow of data from input to output devices, and allows on-line communications between the user and INDAC system.

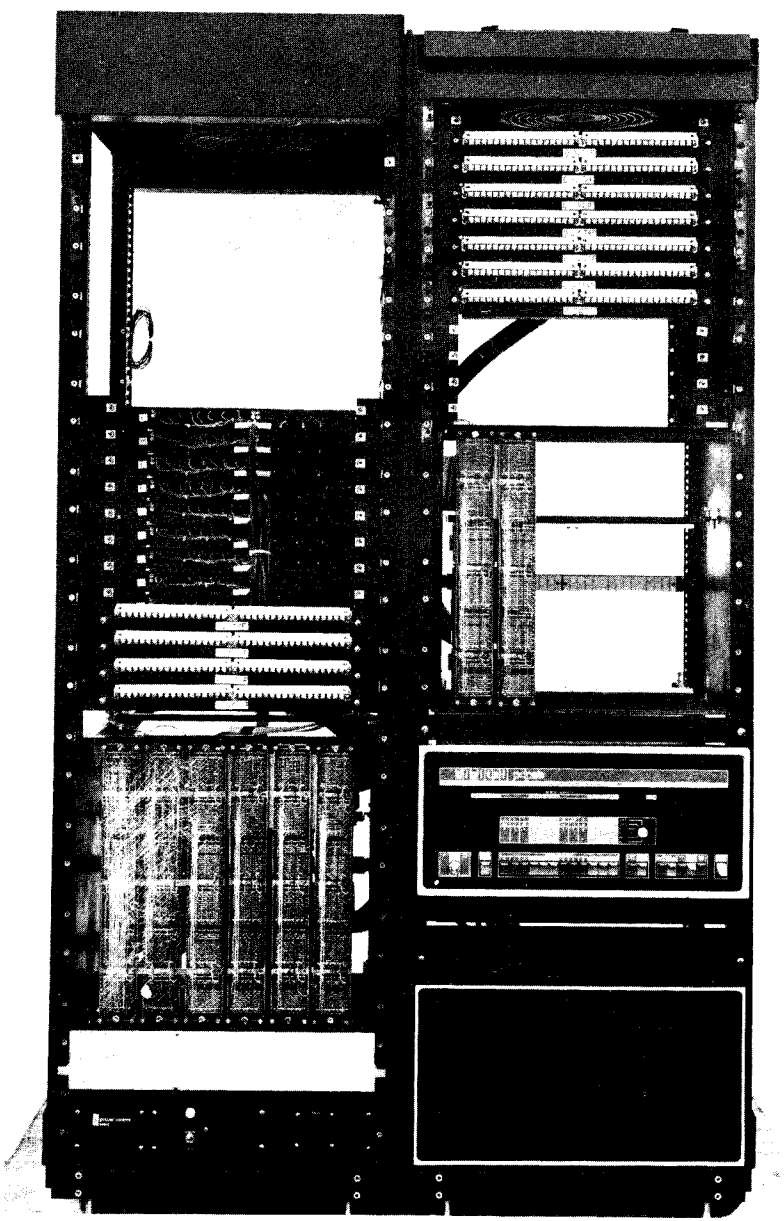
- I/O HANDLERS LIBRARY—The input/output handlers library services various devices in INDAC system, which can be addressed with simple GET or SEND statements. The devices supported include the complete line of analog input, digital input/output and analog output sub-systems besides standard peripherals.

- LIBRARY—The library contains the arithmetic and transcendental functions such as sine, cosine, arctangent, or log, as well as conversion routines for commonly used thermocouples.

- SUPPORTING SOFTWARE—INDAC support programs are unique in the field of small computer systems. INDAC program preparation is achieved by powerful PDP-8 disc monitor system. User can establish and maintain files for source programs, edit and compile them, in a file-to-file operation. Other support programs help core examination for system configuration and other related debugging tasks.

INDAC software has been proven in various field installations for applications such as:

Quality control and testing of air-conditioner valves
Performance testing of internal combustion engines
Control of semi-conductor diffusion furnaces



INDAC-8 System

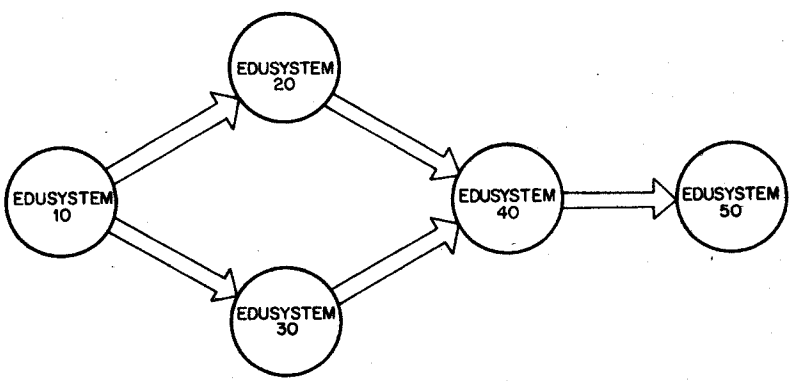
EDUSYSTEMS 10 THROUGH 50

DEC's Edusystem series is ready to serve the needs of schools—now and for the future.

The Edusystem series is a reliable, proven, expandable, and instructionally complete classroom computer system. Designed as a complete instructional package, Edusystems 10 through 50 offer a range of classroom computer systems that can update the calculator user to computers or fill the most ambitious computer science and administrative needs of the modern school.

The PDP-8 family of computers is the nucleus of the system. Time-proven curriculum materials and a truly comprehensive library of application programs in all subject areas are brought together in the classrooms by Edusystems as a total instructional package for the beginner or sophisticated teacher-user.

Each Edusystem is completely expandable to a higher system; no non-functioning leftovers or obsolescent units. The expansion route for Edusystems is diagrammed below:



The Edusystem-10 graduate, who requires a system capable of involving a much greater number of students, has two choices: (1) Edusystem-20 or (2) Edusystem-30. Both are excellent systems and the choice comes in tailoring the system to the individual school.

Edusystem-20 can accommodate up to five on-line terminals for time-sharing BASIC on one PDP-8/E Computer. The benefit here is a very great degree of interaction between student and the computer, accompanied by increased motivation.

Edusystem-30 offers a capability of "batching" mark-sense cards that students mark with an ordinary pencil. This relieves the bottleneck of students waiting to sit down and type their programs. The benefits here are stored or "saved" programs and throughput, more students actually getting involved with "hands on" computer experience. With an optional DECwriter and accompanying paper-tape reader, Edusystem-30 can realistically process up to 100 programs each hour.

The multiple terminals of Edusystem-20 allow students to interact with the computer in exercising simulation programs. Population growth, genetic change, environmental pollution, and Civil War battles are only a few examples of simulation programs that encourage the student to learn by doing.

Edusystem-30 is a true miniature of the gigantic systems in commercial and scientific computer centers.

For those who need more:

- Edusystem-20 offers even greater throughput;
- Edusystem-30 is for users who want multiple terminals for greater student involvement;
- Edusystem-40—a combination of Edusystem-20 and Edusystem-30.
- Edusystem-50 is the top of the PDP-8 based Edusystem line. Edusystem-50 offers the school well versed in computer sciences a true computer center installation. Sixteen simultaneous users may be handled by Edusystem-50 either at the computer site or over telephone lines.

Edusystem-50 is versatile; each of the many users has his choice of computer language. BASIC, FOCAL, ALGOL, FORTRAN, and PAL (an assembly language) are available to any user at any time.

In all of these Edusystems, the design criteria are reliability, ease of operation and the production of a complete instructional program. Each Edusystem includes a teacher's library of curriculum material that includes:

Edusystem User's Guide; DEC
Teach Yourself BASIC, I and II; Tecnica
Basic BASIC; Hayden
A FOCAL Primer; Cornell University
Computer Methods in Mathematics; Addison-Wesley
Computer Assisted Math Program; Scott-Foresman
CAMP, First Course
CAMP, Second Course
CAMP, Algebra
CAMP, Geometry
CAMP, Intermediate Mathematics
Teacher's Guides

Problem Solving with the Computer; Entelek
Computers in the Classroom; DEC
Problems for Computer Mathematics; DEC
An Introduction to Computer Science; Scott-Foresman
with a Teacher's Commentary

Fundamentals of Digital's Computers; Howard W. Sams
Introduction to Programming; DEC
Programming Languages; DEC
"Introducing BASIC with the Overhead Projector"; DEC
23 viewgraph transparencies and teacher's guide
BASIC Application Programs, Sets I, II, III, IV and V; DEC
Program listings and descriptions in all subject areas
BASIC Simulation Programs, Volumes I through VI; Huntington Project and DEC

EDUSYSTEM-10

Edusystem-10 is a low-cost, general-purpose instructional computer system—ideal for the school just getting started with computers.

Edusystem-10 is a logical follow-on to the use of simple electronic calculators in the Math and Science Labs, with the added ease and power of the English-like control language BASIC.

Edusystem-10's BASIC allows students with no experience to perform calculations the minute they sit at the terminal and to develop their first simple program during their first class period.

Edusystem-10 is composed of:

- Hardware
  - PDP-8/e Computer with 4K (4096) words of core storage
  - Power Fail Detect and Auto Restart
  - Hardware bootstrap loader
  - ASR-33 Teleprinter with paper-tape reader/punch
- Software
  - BASIC compiler and Teacher's Curriculum Materials
  - FOCAL
  - FORTRAN*
  - PAL III (Assembly)*
- Input/Output
  - 1 interactive terminal

* Available with optional high-speed, paper-tape reader/punch

EDUSYSTEM-20

Edusystem-20 is a low-cost time-sharing system that simultaneously supports two to five terminals. Edusystem-20 is intended for school systems intending to use the computer with large classes of students or in several classes at the same time.

A typical distribution of five terminals is:

a. Three terminals in the Math Lab
b. One terminal in a Science Lab
c. The fifth terminal operating over telephone lines at a second school.

Edusystem-20, as with Edusystem-10, generally is operated in a problem-solving role. The multiple terminals of the Edusystem-20 also make the system ideal for simulation programs.

Significantly larger and more complex programs can be run on Edusystem-20 and the practical limit to the size of the student programs is a function of the number of terminals being used and the amount of core storage available. Edusystem-20 BASIC offers extended features such as a complete EDIT command allowing simple debugging of student programs.

Edusystem-20 is composed of:

- Hardware
  - PDP-8/e Computer with 8K or more of core storage
  - Power Fail Detect and Auto Restart
  - Hardware Bootstrap Loader
  - 1 to 5 ASR-33 Teleprinters
- Software
  - Time-shared BASIC and Teacher's Curriculum Materials
  - Time-shared FOCAL
  - FORTRAN*
  - PAL III (assembly)*

EDUSYSTEM-30

Edusystem-30 is a true miniature of the massive commercial and university computer centers. Edusystem-30 offers stored programs on a mass storage device, "batch" operation with pencil marked cards and sophisticated BASIC software.

Edusystem-30 has **throughput.** As many as 50 student programs can be run in one hour when the output device is a standard ASR-33 teleprinter, and over 100 programs can be run in one hour with an optional DECwriter. Edusystem-30 gives the lower dollar-per-student figure possible.

Commonly used utility programs can be stored on the mass storage device, and students need only enter data cards.

Edusystem-30 is a natural base for expansion to a PS/8 programming system for computer science and administrative work.

Edusystem-30 consists of:

- Harware
  - PDP-8/e Computer with 4K of core storage
  - A mass storage device (disk or DECtape*)
  - Power Fail & Auto Restart
  - Optical Mark Card Reader
  - ASR-33 Teleprinter
  - Optional DECwriter or line printer
- Software
  - Batch BASIC, cards, templates and Teacher's Materials
  - FOCAL
  - FORTRAN**
  - PAL III (assembly)**
- Input/Output
  - Card reader/teleprinter
- *Environmental Requirements*
  - Control of excessive dust, temperature and humidity (formal computer room not required).

* TC08
** Available with optional high-speed, paper-tape reader/punch

EDUSYSTEM-40

Edusystem-40 is the ideal school-wide instructional computer system operating under one language. Edusystem-40 offers all the advantages of both Edusystems-20 and -30.

Edusystem-40 can "batch" mark-sense cards for the large volume of instructional use, and then provide interactive terminal use for intensive work with a smaller number of either advanced or slower students. The interactive terminal allows the slower student the motivation and infinite patience of the computer, while the advanced student uses it as a means of investigation and expression.

Edusystem-40 software is provided in card format to make use of the high-speed input of the card reader.

Edusystem-40 consists of:

- Hardware
  - PDP-8/e Computer with 8K or more of core storage
  - Mass-storage device (disk or DECtape*)
  - Power Fail and Auto Restart
  - Hardware Bootstrap
  - Optical Mark Card Reader
  - 1 to 5 ASR-33 Teleprinters
  - Optional DECwriter or Lineprinter
- Software
  - Batch BASIC with cards, templates and Teacher's Materials
  - Time-Sharing BASIC
  - Time-Sharing FOCAL
  - FORTRAN**
  - PAL III (assembly)
- Input/Output
  - 1 to 5 interactive terminals, or card reader/teleprinter
- *Environmental Requirements*
  - Control of excessive dust, temperature and humidity (formal computer room not required)

* TC08
** Full standard FORTRAN II available with 64K of disk or DECtape configuration (PS/8)

EDUSYSTEM-50
Edusystem-50 is a true time-sharing system that offers multiple languages to 16 simultaneous users. Edusystem-50 supports from 4 to 16 terminals in BASIC, FOCAL, ALGOL, FORTRAN or PAL (assembly) at the same time.

The computer science class might be using 6 terminals and PAL, while the Math Lab is using 8 terminals operating with BASIC, and the Physics class is using 2 terminals with the simplified FORTRAN.

Edusystem-50's BASIC allows users to maintain files on DECtape and output through the optional line printer. Edusystem-50 can do administrative and instructional work at the same time.

Edusystem-50 consists of:

- Hardware
  - PDP-8/E Computer with 12K or more core storage
  - 262K disk storage
  - DECtapes*
  - Time-Sharing hardware/software
  - Clock
  - 1 to 16 ASR-33 Teleprinters/remote lines
  - Optional Line Printer
  - High-Speed Paper-Tape Reader/Punch
  - Power Fail Detect and Auto Restart
  - Hardware Bootstrap Loader
- Software
  - Time-Shared 8 (TSS/8) Monitor System
  - BASIC, FOCAL, ALGOL, FORTRAN-D, PAL-D
  - System Manager's Guide and documentation
  - User's Guides
  - Teacher's Curriculum Materials
- *Input/Output*
  - 1 to 16 interactive terminals and optional line printer
- Environmental Requirements
  - Control of excessive dust, temperature and humidity (formal computer room not required)

TYPESET-8 COMPUTERIZED SYSTEM
Typeset-8 is a computerized system that produces punched paper tape containing all the hyphenation, justification and format commands needed to drive just about any Typesetting machine on the market.

A perforator operator sits down at the same perforating machine he has always used. When he is through typing, he feeds the tape into a Photoelectric reader, which transfers the combined instructions and marked up copy to paper tape for processing. Letter spacing, word spacing, and end-of-line decisions are unnecessary; therefore, manual justification is completely eliminated. As the perforated tape is being read, the computer simultaneously processes the instructions and text, as the Typeset-8 punch perforates the output tape.

Hot Metal
When the output tape is placed in the tape reader of the linecasting machine, it is processed in the same manner as a manually prepared tape. As the perforated tape is being read, the linecasting machine drops the proper mats to produce lines of justified tape.

Photo Composition (cold type)
Photo composition and associated new processes for composing a complete type form ready for plate making and the press begins and ends in the same way as the older and more conventional methods of printing from hot metal. The main difference is that the input tape contains slightly more complex format codes to produce varying typefaces, styles, kerning, leading, and column widths.

Business
In addition to typesetting, DEC offers a business package designed to computerize many areas of administrative activities with considerable time-saving and cost-saving features.

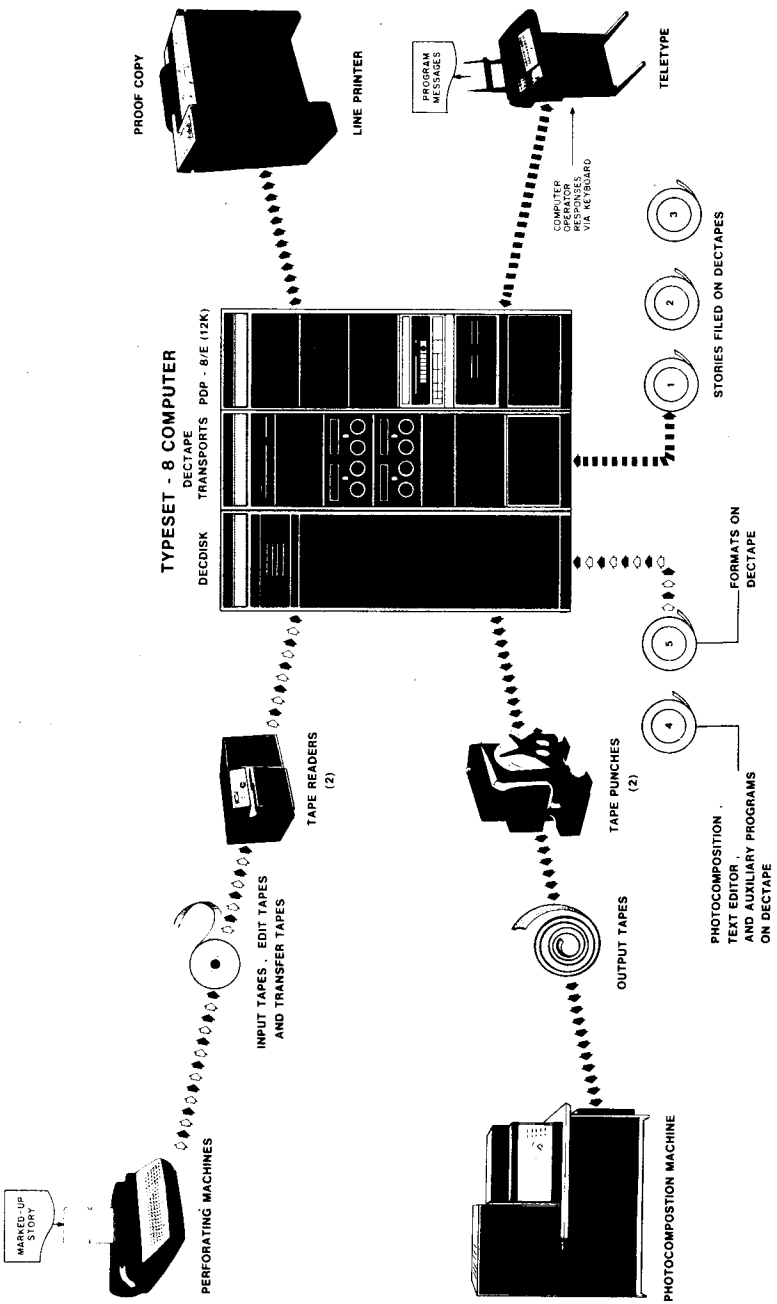
DEC's Business Package include 5 systems: Payroll, Circulation, Advertising, Accounts Payable, and General Ledger. These systems are made to order for your business. They are intended to do the detail work and give the manager time to manage. For example, with DEC's Business Package:

- You can print a complete payroll for 500 employees in less than three hours.
- You can have detailed sales analysis reports on circulation and advertising.
- Your accountant can indicate the due bills he doesn't want to pay; and your computer automatically pays the rest.
- *You can use Business Package reports to determine if, why, and* where business is increasing or decreasing.
- You can pinpoint your troublespots.
- You will have time to manage.

STORAGE AND EDIT SYSTEM

The PDP8/E Storage and Edit System is designed for customers who need the capability to retain text on a storage media for an indefinite period of time. The stored information can be corrected, reformatted, updated, and reworked until a final proof is accepted by the editorial staff. Proofing is done through copy output on a line printer, thus eliminating the expense, time, and manual intervention involved in using a photocomp machine as a proofing device.

During initialization of the system, the "master system tape" is placed on a DECtape transport and then transferred to the disks by the executive. The master tape is then removed, and three storage tapes mounted. With the system in this configuration (i.e., 3 storage tapes), immediate access can be gained to 100 stories, consisting of approximately 1 million TTS characters or 21,000 11-pica lines. Additional transports can be added to increase the immediate access by 7000 lines/transport. A directory of stories currently on the system is retained on the storage tapes and can be requested for printout through the monitor. Files that are retained for an extended length of time and updated less frequently can be stored on separate tapes and incorporated into the system when needed.



CHAPTER 5

PROGRAMMED DATA TRANSFERS

GENERAL

The nature of the IOT instruction was explained in Chapter 3, programming the Teletype was defined in Chapter 4, and the discussion of data transfers as viewed from the peripheral will be discussed in Chapters 9 and 10. This chapter deals with programmed data transfers as viewed from the processor. This discussion is directed to the major considerations of setting up the mechanism for transferring data to and from the processor.

Three types of data transfers are offered by the PDP-8/E processor to receive, store, and transmit data between one or more peripherals and the processor. These transfers are called:

a) Programmed I/O Transfers,
b) Programmed Transfers using the Program Interrupt Facility,
c) Data Transfers using the Data Break Facility.

The first two types of transfers, described in this chapter, are controlled by the program, while the data break transfers are controlled by each peripheral. Data break transfers are discussed in Chapter 6.

PROGRAMMED DATA TRANSFER vs. DATA BREAK

Most input/output (I/O) transfers are controlled by the computer program. Such an information transfer requires perhaps five times as much computer time as does a data break transfer. However, in terms of real time, the duration of a programmed transfer is rather small, due to the high speed of the computer, and is usually well within the limits required for laboratory or process control instrumentation. To get maximum benefit from the control features of the PDP-8/E, the user should utilize programmed data transfers in most cases. Moreover, the peripheral control circuits are usually simpler and less expensive than are those of data break transfer peripherals.

PERIPHERAL REQUIREMENTS

Programmed data transfers use the processor accumulator (AC) register as an intermediate storage point between core memory and a buffer register within the peripheral. If an output data transfer is to be performed, the computer program causes data to be loaded from core memory into the AC. The program then uses an input/output transfer (IOT) instruction to, first, select the desired peripheral and, second, direct the peripheral to generate certain control signals, which cause the data in the AC to be placed onto the OMNIBUS DATA lines. The peripheral, whose control module monitors these DATA lines, then strobes the data into its input buffer register and prepares to process the information. Each transfer from computer to peripheral is handled in this manner.

If the transfer is from peripheral to computer, the process is reversed.

The IOT instruction selects the peripheral and directs it to generate control signals. The peripheral then places the data in its output buffer register on the OMNIBUS DATA lines. A processor timing signal strobes the information into the AC. The program may then either deposit the data into a memory location or use it in some other way.

Each peripheral connected to the OMNIBUS transfers data in the manner described. The bus system of I/O transfers, by which many peripherals monitor the OMNIBUS signal lines, imposes the following requirements on the peripheral equipment:

a. Each peripheral must contain a device selector circuit which monitors Memory Data (MD) lines 3 through 8 of the OMNIBUS. During an IOT instruction, these MD lines carry a selection code which is unique for each peripheral and which must be decoded by the peripheral's device selector.
b. Each peripheral must contain gating circuits which monitor the MD9-11 lines of the OMNIBUS. During an IOT instruction, these MD lines carry command signals that the peripheral must translate into transfer control signals.
c. Each peripheral must contain gating circuits at the input of a receiving register and at the output of a transmitting register (the functions of these registers may be realized with a single buffer register if desired). These gating circuits must be capable of strobing data into or out of the registers when triggered by a command from the device selector.
d. Each peripheral must contain a "Busy/Done" flag (a flip-flop) and gating circuit, which together can assert the OMNIBUS SKIP line when commanded by an IOT instruction. When the flag is set, the peripheral is ready for a word transfer.

PRINCIPLES OF PROGRAMMED I/O TRANSFERS

The simplest and most straightforward type of transfer is the programmed transfer. Rather than responding to an interrupt request and checking each flag to find out which device made the request, the program remains in a continuous wait loop. This method is convenient when the purpose of the processor is to service one or more peripherals. If the user desires to do processing in addition to servicing peripherals, he should employ the interrupt facility.

The IOT Instruction

The nature of the IOT instruction is summarized as follows:

a. PROCESSOR OPERATION CODE—IOTs use operation code 6 and are one-cycle augmented instructions.
b. DEVICE SELECTION—The middle six bits of the instruction word are used for device selection; each peripheral decodes these bits and responds to the IOT only if it sees its particular number or device code. (There is an obvious corollary to this statement. Peripherals generally do not share the same device code.)
c. DEVICE OPERATION CODE—The last three bits of the IOT are used to tell the device what to do. The method varies, depending on whether the device plugs into the OMNIBUS or is at-

tached to the External I/O Bus interface. The end result is the same, however, in that the last three bits define the operation to be performed. The usage is as follows:

Binary Value			Octal Value	Conventional Usage
Bit 9	Bit 10	Bit 11 (LSB)		
0	0	1	1	Sampling flags, skipping
0	1	0	2	Clearing flags, clearing AC
1	0	0	4	Reading, loading, and clearing buffers

Flags

In the control section of every I/O device is a flag, or status flip-flop. The flag is cleared when the computer is first turned on. It can be cleared by one of the device's IOTs, and is set whenever the device finishes its operation. The state of this flag can also be tested, usually by an IOT that causes the next instruction to be skipped if the flag is set.

Input devices (from peripheral to processor) set flags when they have data to be serviced by the processor.

Output devices (from processor to peripheral) receive an IOT instruction that clears the flag. When the device has processed the data received from the processor, it then sets the flag (to let the processor know that it is ready for a new instruction).

In order to understand the purpose of a flag, consider the following situation:

A program requires that a number be entered into the program via the Teletype keyboard. Obviously, the operator might elect to take a lot of time deciding what number to enter. Equally obviously, the program must wait until that number has been entered; otherwise, wrong computation will take place. Thus, a synchronization problem exists. The processor must wait until the operator has made up his mind.

This problem is easily overcome by making use of the flag. The flag sets when the keyboard electronics has a character available in its buffer. Furthermore, the program can find out when the flag sets by means of the skip IOT. All that has to be done is to use the following instructions:

```
Sometimes Called a "Wait Loop"
        KSF      /SKIP IF THE KEYBOARD FLAG IS SET.
        JMP .-1  /IF YOU DIDN'T SKIP THE LAST TIME, TRY AGAIN.
                 /EVENTUALLY THE PROGRAM WILL GET PAST THE JUMP INSTRUCTION.
                 /WHEN THE OPERATOR STRIKES A KEY.
```

Note that "JMP .—1" means "jump back to previous instruction." Assuming that the flag becomes set, the program then clears the flag and brings the character into the AC. If it does not clear the flag at this time, the device will not get a second character. Also, the program must

move the character into the accumulator so that it can process the character (store or operate upon it). This is illustrated by:

```
.
KCC      /CLEAR THE KEYBOARD FLAG.
KRS      /BRING THE CHARACTER INTO THE AC.
.
.
```

A more convenient method of accomplishing this in one instruction as illustrated by the following:

```
KSF      /WAIT FOR THE FLAG TO SET.
JMP .-1
KRB      /CLEAR THE FLAG AND GET THE CHARACTER.
.
.
```

Data Transfers

As seen from the example above, data is moved from a peripheral into the AC under program control. However, the input transfer may be an OR with the previous contents of the AC, or it may be a jam transfer, depending on the design of the peripheral: therefore, the user should consult the detailed IOT listing for the specific peripheral (given in Chapter 7).

Output transfers work in a similar manner. Data is loaded into the AC using the TAD instruction. Then an output IOT is given to move the data to the peripheral and (usually) to initiate some sort of operation. The flag is used slightly differently, but it serves much the same purpose. Consider the following example involving the Teletype printer:

```
.
TAD DATA  /GET A CHARACTER TO BE PRINTED
TLS       /SEND IT TO THE PRINTER ELECTRONICS, CLEAR
          /THE FLAG, AND SAY "GO."
TSF       /HANG AROUND UNTIL THE FLAG SETS, SO YOU
JMP .-1   /DON'T ACCIDENTALLY TRY TO PRINT TWO
.         /DIFFERENT CHARACTERS AT THE SAME TIME.
.
```

If the printer flag is set ahead of time, the flag may be tested before doing the "TLS." This technique speeds up the program, since the computer can be doing instructions instead of just waiting for the flag.

```
TFL                 /SET PRINTER FLAG
.
.                   /GET CHARACTER
TSF
JMP .-1
TLS                 /PRINT IT
.
```

The user should not try to be exotic and microcode skip and clear IOTs into one instruction. It won't always work. Trouble could develop if the flag happened to set between the skip and clear functions, because the program might have cleared the flag before it found out the flag had been set.

PRINCIPLES OF PROGRAM INTERRUPTS

GENERAL

There may come a time when waiting for flags consumes too much valuable time. What is needed is some way of ignoring peripherals until they need attention.

For instance, the device flag sets when a new character is available if the device is an input device. If the device is an output device, the flag sets when the device is ready to receive a new character. Wouldn't it be desirable to service peripherals only if one or more flags happen to set?

To accomplish this, all peripherals OR their flags onto a special line called the Interrupt Request line. If this line becomes ground, it means that there's a flag (and hence a device) somewhere that needs attention. However, the flag must **demand** attention; not just pull a line which can be tested with an IOT. The "demand attention" circuitry is known as the interrupt system, and works as follows:

1. The interrupt system is automatically turned off when the computer is first turned on.
2. The interrupt system is turned on by an IOT (ION), but the actual enabling is delayed by one instruction. (The reason for this will be shown later.)
3. If the interrupt is enabled and some device's flag gets set, the processor **automatically** responds by executing a JMS to location zero and simultaneously turning off the interrupt system. Note that this JMS is hardware-generated.
4. In case it is needed, there's an IOT (IOF) which turns off the interrupt system.

The interrupt system is a simple piece of hardware, but has far-reaching program implications. Let's examine some of those implications.

Coding a Program Interrupt. Assume, for example, that the operation is reading and punching information using the PC8/E Reader/Punch combination. Between reading and punching, the processor is doing a simple, unrelated program. Once things get started, the procedure that follows is indicated in figure 5-1.

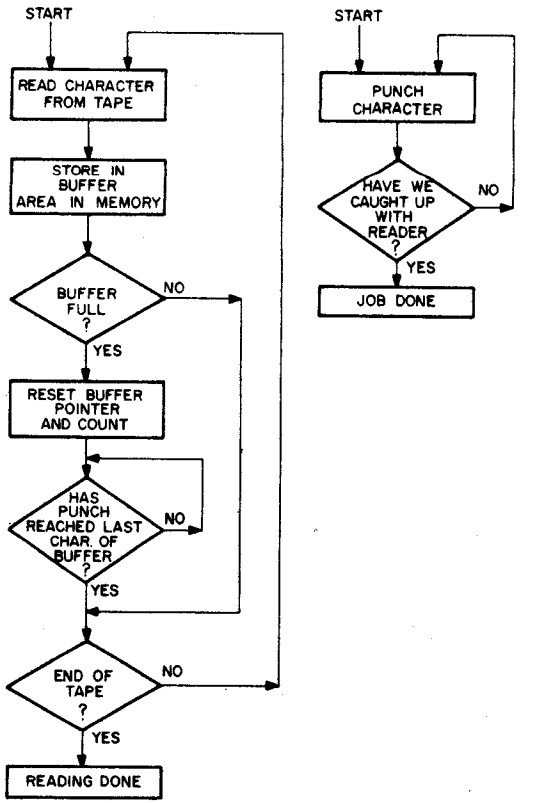


Figure 5-1 Program Interrupt Example

```
/The Interrupt Service Routine

0,      xxxx            /WHERE THE RETURN ADDRESS IS
                        /STORED WHEN THE INTERRUPT OC-
                        /CURS.

1,      JMP I.+1        /THIS INSTRUCTION GETS TO THE
2,      FLAGS           /FLAG TEST ROUTINE WHICH IS
                        /USUALLY ON OTHER THAN PAGE 0.

/The flag-testing and restoration routine

FLAGS,  DCA AC          /SAVE AC
        RAL
        DCA LNK         /AND LINK
        RSF             /CHECK READER FLAG
```

```
            SKP
            JMP RDR
            PSF             /AND PUNCH FLAG
            SKP
            JMP PUNCH
            KCC             /IF WE GET HERE, SOMEONE PROB-
                            /ABLY STRUCK A TELETYPE KEY
            TCF             /OR SOME OTHER STRANGE INTER-
                            /RUPT EXISTS.
DISMIS,     CLA CLL         /HERE'S HOW WE EXIT
            TAD LNK         /RESTORE LINK
            RAR
            TAD AC          /AND AC,
            ION             /TURN ON THE INTERRUPT
            JMP I 0         /AND USE THE ONE-CYCLE DELAY
                            /TO GET THE PC LOADED.

/THE READER SERVICE ROUTINE. WE KNOW THE FLAG IS SET AL-
/READY, OR WE WOULDN'T BE HERE.

RDR,        RRB             /GET CHARACTER FROM READER,
                            /CLEAR FLAG
            DCA I 10        /SAVE IN BUFFER
            ISZ COUNT1
            JMP RDGO        /BUFFER NOT FULL
            TAD BUFF        /BUFFER FULL
            DCA 10
            TAD KCOUNT
            DCA COUNT1
            JMP DISMIS      /NOTICE-WE DID NOT MOVE
                            /READER.
RDGO,       RFC             /MOVE READER
            JMP DISMIS      /AND EXIT

/THE PUNCH SERVICE ROUTINE

PUNCH,      TAD I 11        /GET WORD FROM BUFFER.
            PLS             /PUNCH IT AND CLEAR FLAG
```



```
            CLA
            TAD 11          /COMPARE ADDRESS PUNCHED
            CIA
            TAD 10          /WITH READER ADDRESS
            SNA CLA
            JMP FINISH      /ADDRESSES EQUAL, SO JOB DONE.
            ISZ COUNT2
            JMP DISMIS
            TAD KCOUNT      /TOP OF BUFFER, SO RESET
            DCA COUNT2      /COUNTER
            TAD BUFF
            DCA 11          /AND POINTER.
            JMP RDGO        /AND RESTART THE READER.

/OK—THAT TAKES CARE OF THE INTERRUPT HANDLER. NOW COMES
/THE MAIN PROGRAM.

START,      RFC             /START READER
            PLS             /AND PUNCH
            TAD KCOUNT      /INITIALIZE ALL VARIABLES
            DCA COUNT1
            TAD KCOUNT
            DCA COUNT2
            TAD BUFF
            DCA 10
            TAD BUFF
            DCA 11
            ION             /AND TURN ON INTERRUPT
            ISZ PHONEY      /THIS IS A TRIVIAL NULL
            JMP .—1         /JOB WHICH SLOWLY INCREMENTS
            IAC             /THE AC.
            JMP .—3
FINISH,     HLT

/CONSTANTS
```

```
KCOUNT,    -100
BUFF,      377
/VARIABLES
COUNT1,    0
COUNT2,    0
AC,        0
LNK,       0
PHONEY,    0
```

The reader stops automatically when tape runs out of the reader. When the punch pointer becomes equal to the reader pointer, the job is finished. Notice how the one-instruction delay built into the ION instruction allows the computer to obtain the return address from location 0 before new interrupts can occur.

In the above example, the background null job (the job being done when interrupts were not being serviced) was unrelated to the interrupt routine. In the more general case, the background job is some sort of processing job operating on the buffer after reading and before punching. Clearly, the programming problem is more severe. The reader must be ahead of the processing, which must be ahead of the punching. Such problems are beyond the scope of this chapter, and the reader should, therefore, consult Chapter 5 of "Introduction To Programming."

THE PDP-8/E System fits just about anywhere. The user who begins with an inexpensive system can later convert his table-top system to a larger cabinet configuration and still retain most of his hardware.

CHAPTER 6

DATA BREAK TRANSFERS

GENERAL

Data break (sometimes called Direct Memory Access or DMA) is another form of data transfer used with high speed devices. Generally, a mass-storage device utilizes this form of data transfer. In the case of a high-speed device such as a disk, it becomes desirable to transfer a block of information at the fastest possible transfer rate. A block of data containing up to 4K words would require one data break for each word transferred.

Data break is the process of stealing memory cycles for the purpose of adding data to memory or removing data from memory without disturbing the major registers within the processor. Data breaks, therefore, stall the processor only during that period of time when some action is required by the processor, such as transferring data in or out of memory.

Data break is desirable when a large quantity of words is to be transferred. Transferring one 12-bit word is a relatively easy task to accomplish. If a word is to be transferred out to some peripheral, the only requirement would be to stop the program and jump to some subroutine that would address memory and transfer data out to that peripheral. Therefore, all that would be needed is a program subroutine, a memory address, and a 12-bit register to receive the 12-bit word at the peripheral end. It would obviously be more practical to use a programmed type of data transfer rather than a data break transfer.

To use data break transfers effectively, a series of words should be transferred and the peripheral should be considered a high-speed peripheral. To accomplish this, however, the process becomes more complicated. For example, if 50 words are to be transferred, 50 addresses are necessary, along with a means of telling the processor when the 50 words have been transferred. Thus, an address incrementing device is required to add a number to the current address and a word count mechanism is required to count the number of transfers.

THE BASIC DATA BREAK SYSTEM

The purchaser of a PDP-8/E and a data break peripheral receives, in addition to his basic programming package, subroutines corresponding to the data break device and a Programmer's Reference Manual to initially set up and program the device. Such subroutines might include the Interrupt Service Routine, the Search Subroutine, and the Read and Write Subroutines, depending upon the type of data break peripheral. These are first loaded into memory to provide the necessary IOTs required for addressing and initializing transfer control. The user then calls these subroutines.

The basic data break system is illustrated in Figure 6-1. The user calls for the data break, and the software routine initializes the data break transfer. The peripheral control contains a break priority circuit, a current address register, a word count register, and a data register.

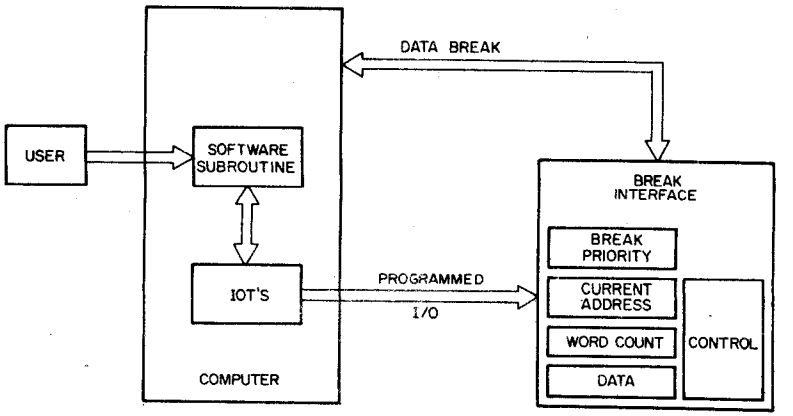


Figure 6-1 General Block Diagram of Basic Data Break Interfaces

Its function is to address memory, count the number of word transfers and receive or transmit data.

Current Address (CA) Register

The Current Address Register provides the ability to sequentially address a series of memory locations by incrementing before each transfer. Initially, the CA register is loaded with an address that is one location before the desired address. The register is then incremented (a one is added to the register) and the address is placed onto the memory data lines.

Word Count (WC) Register

The Word Count Register serves to count the number of data words in a block of data that is transferred from or to a peripheral. The two's complement of the number of words to be transferred is placed in the register initially. The register increments at each word transfer until the WC register contains zeros. When the WC register overflows, a signal is generated which clears the enable circuits.

Data Break Priority

Up to 12 data break devices can be simultaneously attached to the PDP-8/E. Each device is assigned a line on the OMNIBUS for use in determining priority.

Bit 0 represents the highest priority, and bit 11 represents the lowest. When a device makes a break request, the priority of the other devices also making a break request is tested. This guarantees that a faster and higher priority device will make the first data transfer.

Data Register

Data transfer occurs between the device Data Register and the Memory Buffer Register. The data register only serves either to receive or to transmit data.

Data Break Configurations

Two types of data break configurations (single-cycle and three-cycle) may be used on the PDP-8/E.

Single-cycle devices contain the WC and CA registers in the peripheral. Once data transfer begins, the processor stalls only once (one cycle is stolen) while the data transfer takes place.

Three-cycle devices use memory to contain the WC and CA registers. The processor then must stall one cycle for the word count, one cycle for the current address, and one cycle for the word transfer.

ONE-CYCLE DATA BREAK TRANSFERS

A simplified block diagram of the one-cycle data break operation is illustrated in Figure 6-2, and a flow chart showing the interaction between the processor and the device is illustrated in Figure 6-3. Because this is a one-cycle data break device, Figure 6-2 shows the word count and current address registers in the peripheral.

Figure 6-3 divides the chart in terms of actions required by the processor and actions required by the device. The "flow of events" time periods are divided to reflect the INITIAL SET-UP, DATA TRANSFER, and EXIT.

Initial Set-Up

The Program Subroutine initializes the word count (WC) and current address (CA) hardware registers located in the peripheral and generates IOTs to enable the device control logic and specify the direction of transfer.

Data Transfer

The 12-bit word count register becomes incremented just prior to a word transfer and provides an overflow signal to the control logic after the register is incremented to all zeros.

The 12-bit current address register is incremented at the same time as the WC register. If an overflow in the WC register occurs, the device clears the enable circuits after the current word has been transferred. Before the current address can be placed on the memory address lines, a priority test must be made. If the device is the highest priority, the contents of the CA register are placed on the memory address lines.

After the priority test, the break device generates CPU disabling signals which tell the processor to stall while a word is transferred into or out of memory. While the contents of the CA registers are placed on the MA lines, a memory direction signal line (transfer into or out of memory) is enabled. The transfer of data occurs between the memory buffer register and the device data register.

At the end of the memory cycle, the CPU disabling signals are removed and the processor resumes operation. If the device is still enabled (WC overflow has not occurred), it then repeats the data transfer cycle. If WC overflow has occurred, the device begins an EXIT.

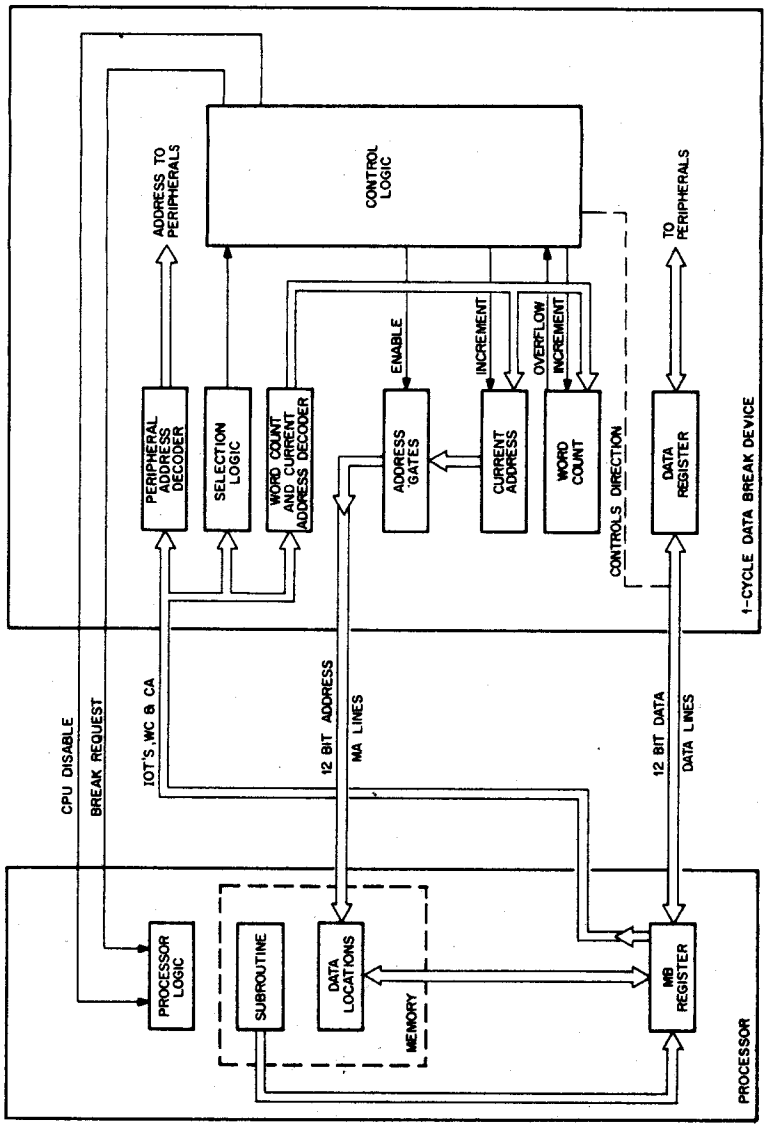


Figure 6-2 1-Cycle Data Break Simplified Block Diagram

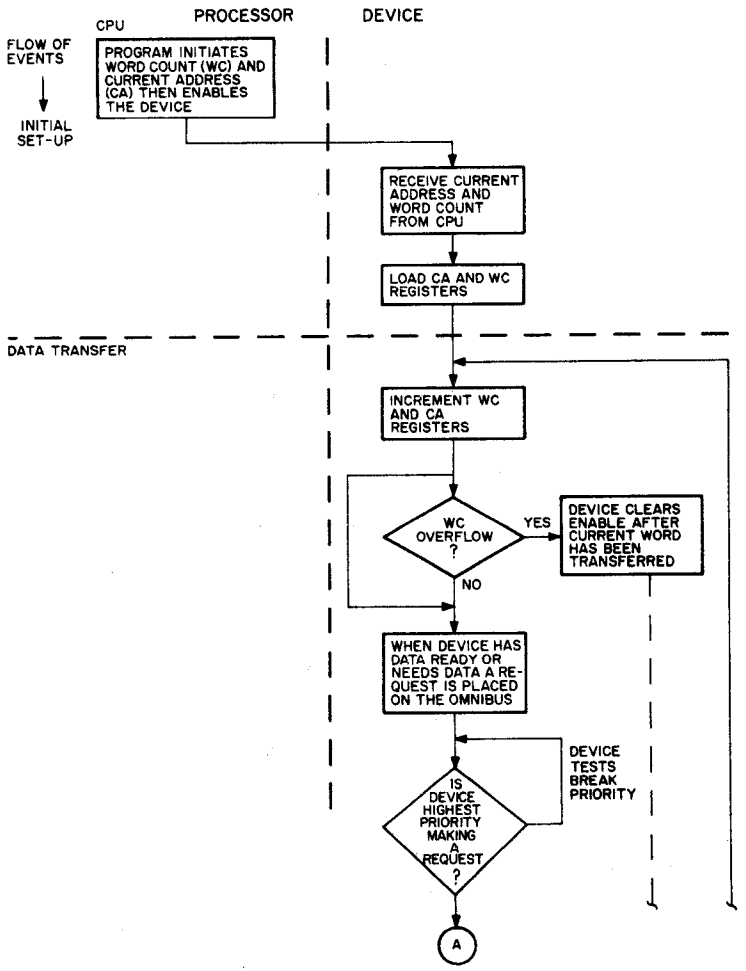


Figure 6-3 1-Cycle Data Break Flow Chart

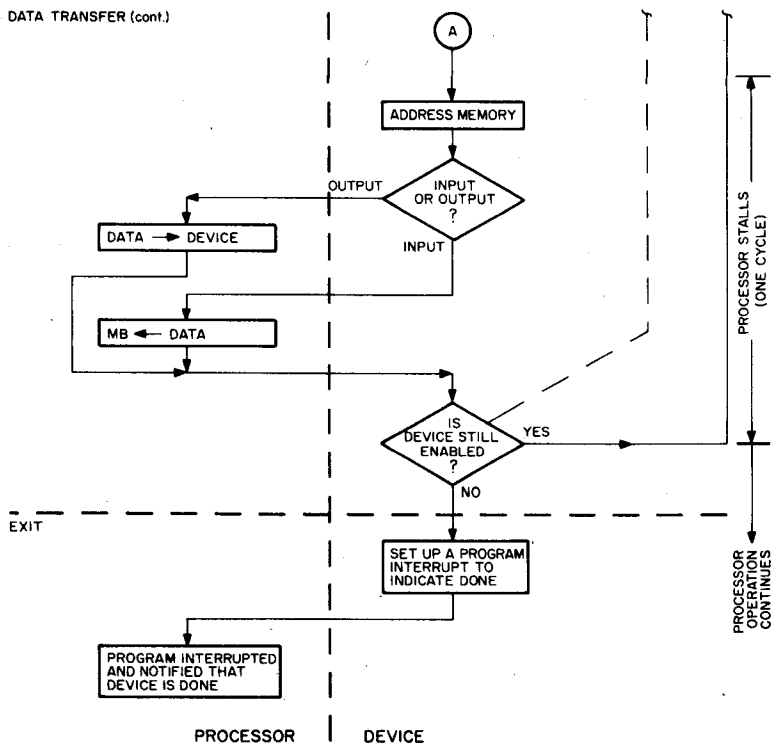


Figure 6-3 1-Cycle Data Break Flow Chart (Continued)

EXIT
To exit, the device sets a flag indicating that the block transfer has been completed. The processor then responds to this flag in the manner described in Chapter 5.

THREE-CYCLE DATA BREAK TRANSFERS

A simplified block diagram of the three-cycle data break operation is shown in Figure 6-4, and a flow chart illustrating the interaction between the processor and the device is shown in Figure 6-5. The WC and CA registers are located in memory. Therefore, the program loads the WC register with the required word count and the CA register with the address where the data will be either stored or retrieved.

The flow chart in Figure 6-4 is divided into time periods to reflect the INITIAL SET-UP, WORD COUNT, CURRENT ADDRESS, DATA TRANSFER, and EXIT. Because the word count and current address registers are located in memory, two additional data break cycles are required. Incrementing of both the CA and WC registers is accomplished in the processor.

Initial Set-Up
For a three-cycle operation, the data break subroutine must load the word count and current address memory locations. In addition, IOTs to enable the data break control logic and peripheral addresses are generated by the subroutine and transferred to the device.

Once the device decodes the IOTs, a break request is initiated and a priority test is performed. If the device has the highest priority, it generates a group of CPU disable signals and the system begins a word count cycle.

Word Count
The memory address of the word count register is hard-wired in the peripheral. This address is gated into the break address register and placed onto the memory address lines. An overriding line to the address register forces a zero into bit 11 (for an even address).

The processor then fetches the contents of the word count register to the memory buffer register. A one generated by the control logic is forced into bit 11 of the data bus, and transferred to the memory buffer register via the adder circuits. The resulting addition is tested for overflow. The contents of the memory buffer register are immediately placed into the word count register. If word count overflow occurred, the device clears its enable after the current word has been transferred. Just prior to entering the current address cycle, the priority is tested again.

Current Address
Updating the Current Address Register is accompilshed in the same manner as Word Count. However, instead of a zero being forced into bit 11 of the break address register, a one is used.

At the end of the current memory cycle, the contents of the MB register are written back into the CA register and also transferred to the device break address register. On the next memory cycle, the device break

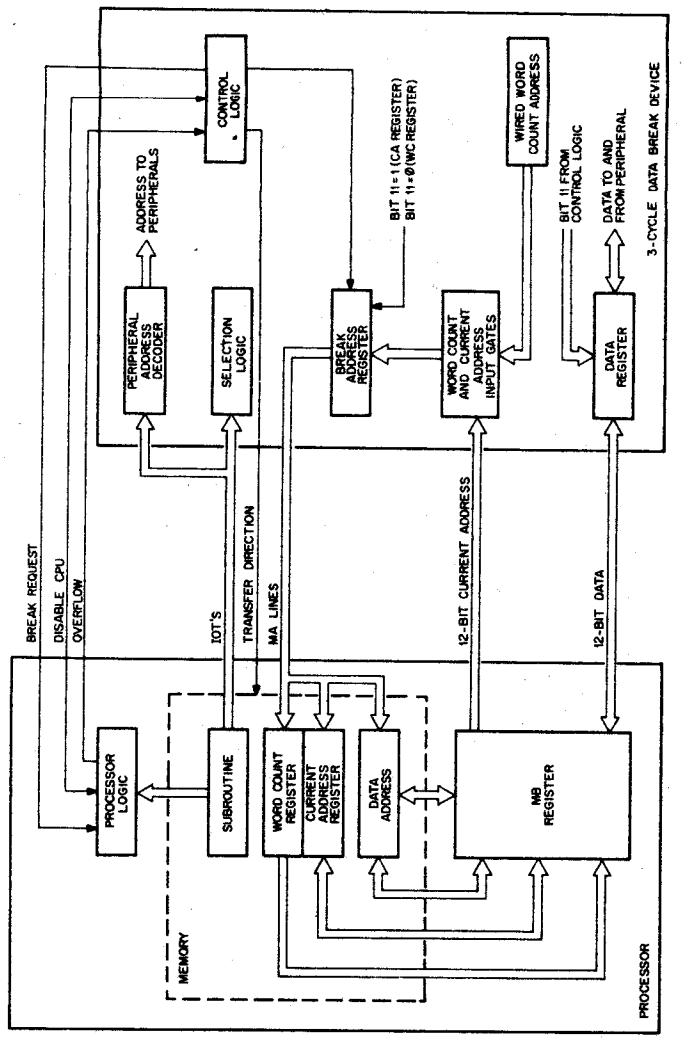


Figure 6-4 3-Cycle Data Break Simplified Block Diagram

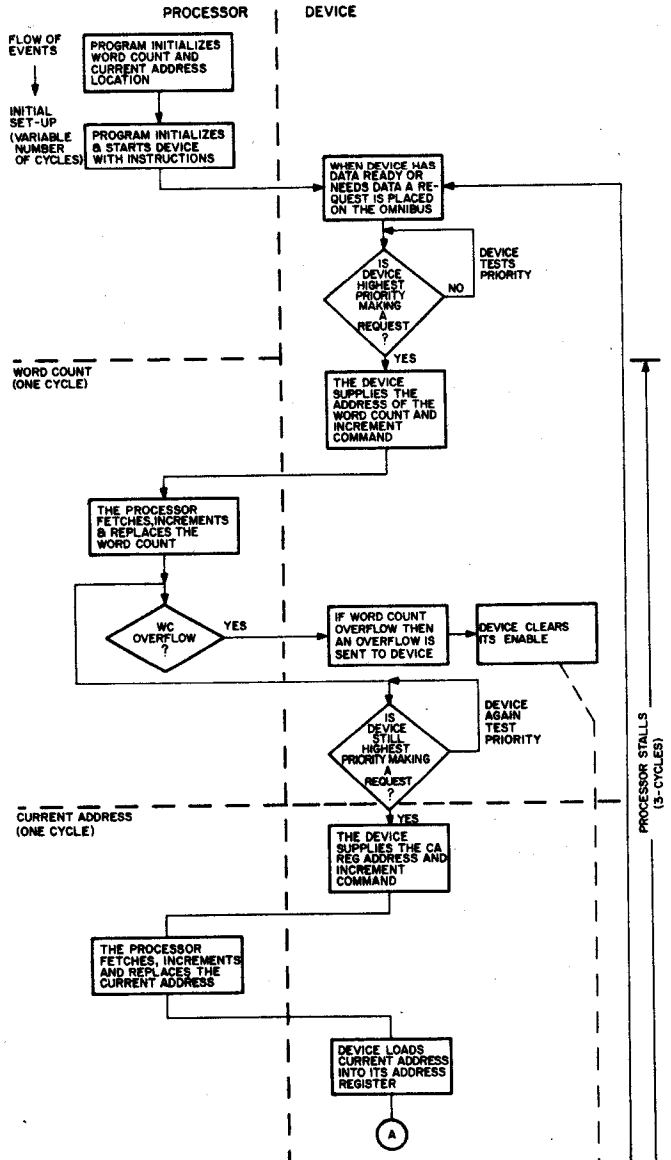


Figure 6-5 3-Cycle Data Break Flow Chart

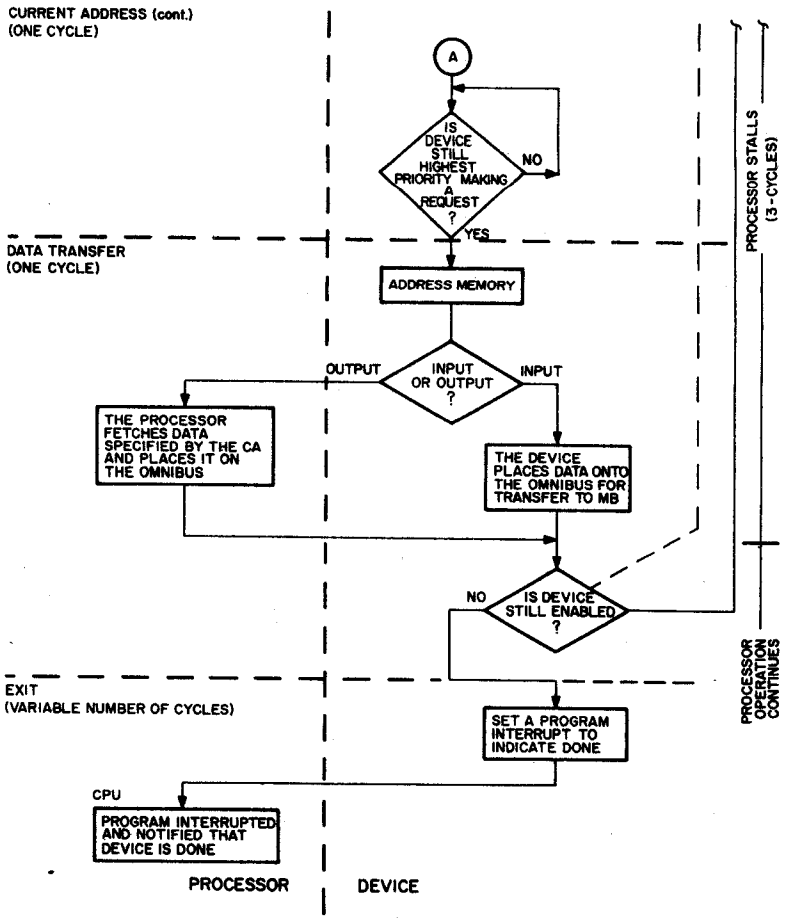


Figure 6-5 3-Cycle Data Break Flow Chart (Continued)

address register points to the data address and the system is now ready to transfer data. Just prior to addressing the data location, another priority test is made.

Data Transfer

When the current address is placed on the address lines, a transfer direction signal is generated, and data is placed onto the OMNIBUS data lines. The data transfer is between the device data register and the *memory buffer register. The CPU disabling signals* are removed at the end of the cycle. If the WC overflow signal has not been received, the device is still enabled. A break request is again initiated and the word count—current address—data transfer cycle is repeated. A word count overflow causes the device to set its device flag and to begin an exit.

Exit
A flag checking subroutine terminates the data break.

PROGRAMMING EXAMPLE
An example of programming for the data break operation is given below. This example deals with programming a DF32 Disk, and provides a subroutine only to transfer a block of words between Disk and Memory. It does not include the subroutines which are necessary to determine the word count, starting address in memory, and disk address. This example serves to illustrate the software required with data break.

BLOCK TRANSFER SUBROUTINE
Subroutine Call

```
        •
        •
        •                   /PRECEDING INSTRUCTION OF THE MAIN
                            /PROGRAM
JMS I DRW                   /DRW CONTAINS ADDRESS OF DISK ROUTINE
DEA & MEA                   /CONTENTS TO BE PLACED IN THE DISK
                            /AND MEMORY EXTENDED ADDRESSES
CA                          /CONTENTS TO BE PLACED IN THE CURRENT
                            /ADDRESS REGISTER
WC                          /CONTENTS TO BE PLACED IN THE WORD
                            /COUNT REGISTER
MEMDIR                      /CONTENTS CONTAINS A 2 FOR READ AND 4
                            /FOR WRITE
DA                          /CONTENTS TO BE PLACED IN THE DISK
                            /ADDRESS
        •                   /NEXT INSTRUCTION IN MAIN PROGRAM
        •
        •
        •
DRW, DISK
DISK,   0                   /RETURN POINTER
        TAD I DISK          /TRANSFER CONTENTS TO BE PLACED IN
                            /DEA and MEA TO AC REGISTER
        ISZ DISK            /MOVE POINTER TO NEXT LOCATION OF CALL
                            /(CA)
        DEAL                /LOAD DISK EXTENDED ADDRESS
        CLA                 /CLEAR AC
        TAD I DISK          /TRANSFER CONTENTS TO BE PLACED IN CA
                            /REGISTER TO THE AC REGISTER
        ISZ DISK            /MOVE POINTER TO NEXT LOCATION (WC)
        DCA I DISKCA        /STORE UPDATED CURRENT ADDRESS
        TAD I DISK          /TRANSFER THE CONTENTS TO BE PLACED
                            /IN THE WC REGISTER TO AC
        ISZ DISK            /MOVE POINTER TO NEXT LOCATION (MEM
                            /DIR)
        DCA I DISKWC        /STORE UPDATED WORD COUNT
        TAD I DISK          /TRANSFER THE CONTENT OF MEM DIR (2
                            /OR 4) TO AC
        ISZ DISK            /MOVE POINTER TO NEXT LOCATION (DA)
```



```
        TAD IOT             /ADD 6601 TO THE AC TO BUILD READ OR
                            /WRITE IOT
        DCA GO              /DEPOSIT IOT TO BE EXECUTED
        TAD I DISK          /TRANSFER DISK ADDRESS TO AC
        ISZ DISK            /MOVE POINTER TO THE NEXT MEMORY
                            /LOCATION (MAIN PROGRAM)
GO,     0                   /EXECUTE IOT TO LOAD DISK ADDRESS
                            /REGISTER AND BEGIN TRANSFER
        DFSC                /SKIP ON COMPLETION FLAG
        JMP .-1             /CHECK FLAG AGAIN
        DFSE                /SKIP ON NO ERROR FLAG
        JMP ERR             /JUMP ERROR
        JMP I DISK          /DISK CONTAINS ADDRESS OF NEXT
                            /INSTRUCTION OF MAIN PROGRAM
DISKCA, 7751
DISKWC, 7750
IOT,    6601
```