UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO
FACULTAD DE INGENIERIA
DIVISION DE ESTUDIOS SUPERIORES
CENTRO DE EDUCACION CONTINUA
DIRECTORIO GENERAL

REGISTRO DE ASISTENTES Y PROFESORES.

NOMBRE DEL CURSO: ____________________ FOLIO [| | | |] 1 5 CLAVE ASOC [|] 6 7

[| | | | |※] 8 13 [...] 14 NOMBRE(S) APELLIDO PATERNO APELLIDO MATERNO 41

REG. FED. CAUS. [ ] 42 51 CED. PROF. [ ] 52 58

TEL. PARTICULAR [ ] 59 65 TEL. OFICINA [ ] 66 72 EXTENSION [ ] 73 76

MARQUE CON UNA CRUZ

ASISTENTE [ ] PROFESOR [ ] [ ] 77 [1] 80

[ ] 8 DOMICILIO PARTICULAR (CALLE. NUMERO Y No. INTERIOR) 41

[ ] 42 COLONIA 71 Z.P. [|] 72 73

ESTADO ____________________ [|] 74 75

TITULO PROFESIONAL ____________________ [|] 76 77 ESPECIALIDAD ____________________ [|] 78 79 [2] 80

[ ] 8 DOMICILIO DE OFICINA (CALLE, NUMERO Y No. INTERIOR) 41

[ ] 42 COLONIA 71 Z.P. [|] 72 73

ESTADO ____________________ [|] 74 75 [3] 80

SOCIACIONES A LAS QUE PERTENECE.

____________________ [|] ____________________ [|]

____________________ [|] ____________________ [|]

APLICACION DE MINICOMPUTADORAS

(del 21 de octubre al 12 de noviembre de 1977)

Fecha	Duración	Tema	Profesor
Oct. 21	17 a 21 h	INTRODUCCION	M. en C. Marcial Portilla Robertson
Oct. 22	9 a 13 h	CIRCUITOS LOGICOS	M. en C. Marcial Portilla Robertson
	14 a 18 h	ARQUITECTURA DE MINICOMPUTADORAS	M. en C. Marcial Portilla Robertson
Oct. 28	17 a 21 h	ENTRADA Y SALIDA DE MINICOMPUTADORAS	Ing. Raymundo Segovia
Oct. 29	9 a 13 h	ACCESO DIRECTO A MEMORIA	M. en C. Manuel Grijalva López
	14 a 18 h	PAGINACION	Dr. Adolfo Guzmán Arenas
Nov. 4	17 a 21 h	INTERFASES	Dr. Víctor Gerez Greiser
Nov. 5	9 a 13 h	PROGRAMACION DE SISTEMAS	Ing. Raymundo Segovia
	14 a 18 h	SISTEMA OPERATIVO EN TIEMPO REAL SELECCION DE UNA MINICOMPUTADORA	Dr. Adolfo Guzmán Arenas
Nov. 11	17 a 21 h	APLICACIONES	Dr. Víctor Gerez Greiser
Nov. 12	9 a 13 h	APLICACIONES EN UNA MINICOMPUTADORA	M. en C. Gertudiz Kurts de Lara Sr. Fernando Lepe Casillas
		CLAUSURA	

'edes. 6, X, 1977.

DIRECTORIO DE PROFESORES DEL CURSO APLICACION DE MINICOMPUTADORAS

DR. VICTOR GEREZ GREISER
PROFESOR TITULAR
INGENIERIA MECANICA Y ELECTRICA
FACULTAD DE INGENIERIA
UNAM
TEL.: 550.52.15 E. 3746

M. EN C. MANUEL GRIJALVA LOPEZ
JEFE DE LA SECCION DE INGENIERIA DE CONTROL
SECCION DE INGENIERIA MECANICA Y ELECTRICA
FACULTAD DE INGENIERIA
UNAM
TEL.: 550.52.15 E. 3751 y 37.52

DR. ADOLFO GUZMAN ARENAS
INVESTIGADOR
INSTITUTO DE INVESTIGACIONES EN MATEMATICAS APLICADAS Y EN SISTEMAS
UNAM
TEL.: 550.52.15 E. 4585 y 4584

M. EN C. GERTUDIS KURTZ DE LARA
LABORATORIO DE CIBERNETICA
FACULTAD DE CIENCIAS
UNA

M. EN C. MARCIAL PORTILLA ROBERTSON
JEFE DE LA SECCION DE COMPUTACION
EDIFICIO DE INGENIERIA MECANICA Y ELECTRICA
FACULTAD DE INGENIERIA
UNAM
TEL.: 550.52.15 E. 3750

ING. RAYMUNDO SEGOVIA
INVESTIGADOR
INSTITUTO DE INVESTIGACIONES EN MATEMATICAS APLICADAS Y EN SISTEMAS
UNAM
TEL.: 50.52.15 E. 4583

'edcs. 7, X, 1977.

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

APLICACION DE MINICOMPUTADORAS

PROGRAMACION DE SISTEMAS

Tomado del libro Minicomputer Systems Organization and Programming. (PDP-11).

Noviembre, 1977

I/O PROGRAMMING

Being able to program a computer to do calculations is of little use if there is no way of getting the results of calculations from the machine. Likewise, the programmer often must supply the computer with information to be processed. A programmer must, therefore, be provided with the means to transfer information between the computer and the peripheral devices that supply input or that serve as a means of output.

In order to perform an I/O function, the programmer must specify what the data are, where they are to go or come from, and how the I/O device is to be controlled. Depending on the small computer being utilized, the I/O function may require the CPU to wait until the I/O operation is complete, or the I/O function may allow the CPU to go on and process other functions while the operation is being performed. When the I/O function holds up the CPU, we say that the I/O operation is *interlocked* with the CPU. When both can be performed simultaneously, we say that I/O is *concurrent* with computation.

Concurrent operation is becoming the standard mode for most small computers. This mode takes several forms. In one form, the concurrent I/O function can operate on data words one at a time. During the operation, the data word is held temporarily in a special register, such as the accumulator.

In another form, the I/O function operates directly between memory and the I/O unit. This mode of operation requires a separate path [called a *direct memory access (DMA)* path] between the memory and the I/O unit. The DMA allows the I/O function to be performed with a minimum of dependency on the part of the CPU.

A third mode of operation allows a large block of I/O information to be passed between an I/O unit and the memory. As support for such *block transfers*, special registers are provided for holding a count of the number of

words yet to be transferred, the current I/O unit and memory address of the data word being transferred, and the data. Once initiated by an I/O instruction, block transfers run concurrently and independently of the CPU until they are completed (i.e., the word count goes to zero or an addressing error occurs).

Whenever there is a DMA path as well as a CPU path to memory, conflicts may arise. Because the I/O requests for memory are time dependent, occur infrequently, and are of short duration, the I/O request is given preference over the CPU request. Such preferential treatment is called *cycle stealing* in that the I/O unit is granted memory cycles at the expense of the CPU.

It should be fairly obvious from this brief introduction that with the various possibilities, I/O programming is very machine dependent. The complexity of the I/O system determines the corresponding complexity of the I/O programming. On the PDP-11, the programming of I/O devices is extremely simple, and no new I/O instructions are necessary for dealing with input/output operations.

The key to the simplicity of I/O programming is the UNIBUS, described in Chapter 3. The UNIBUS permits a unified addressing structure in which control, status, and data registers for *peripheral* devices are *directly* addressed as memory locations. Therefore, all operations on these registers, such as transferring information into or out of them or manipulating data with them, are performed by normal memory reference instructions.

6.1. BASIC I/O PROGRAMMING AND OPERATIONS FOR THE PDP-11

The use of memory reference instructions on peripheral device registers greatly increases the flexibility of I/O programming. For example, information in a device register can be compared directly with a value and a branch made on the result.

```
CMPB    DEV # 'Y        IS CHARACTER = 'Y'
BEQ     YES             YES IT IS
```

In this case the program looks for a "Y" in the keyboard data buffer (TKB) and branches if it finds it. There is no need to transfer the information into an intermediate register for comparison.

When the character is of interest and is to be saved, a memory reference instruction can transfer the character into a user buffer in memory or to another peripheral device. The instruction

```
MOVB    PRB,LOC         SAVE CHARACTER IN MEMORY LOCATION "LOC"
```

transfers a character from the paper tape reader buffer (PRB) into a user-defined location.

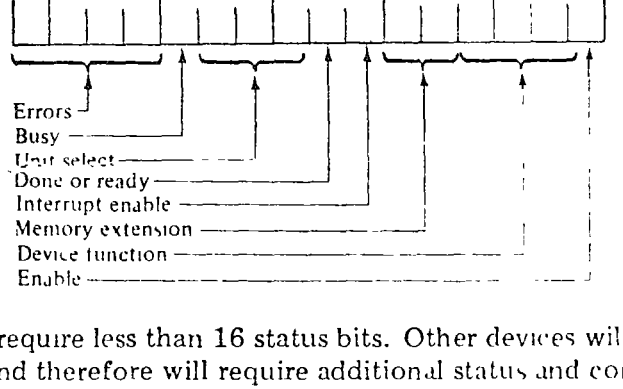
Another aspect of I/O programming is that arithmetic operations may be performed on a peripheral device register that is used for both input and output. Thus there is no need to funnel all data transfers, arithmetic operations, and comparison through other words or general-purpose registers. Instead, the peripheral device register can itself be treated as an accumulator.

6.1.1. Device Registers

All peripheral devices are specified by a set of registers that are addressed as memory and manipulated as flexibly as an accumulator. For each device, there are two types of associated registers:

1. Control and status registers.
2. Data registers.

Each peripheral has one or more control and status registers (CSR's) that contain all the information necessary to communicate with that device. The general form shown here does not necessarily apply to every device, but is presented as a guide:



Many devices require less than 16 status bits. Other devices will require more than 16 bits and therefore will require additional status and control registers.

The bits in the control and status registers are generally assigned as follows:

Bit	Name	Description
15-12	Errors	Generally, there is an individual bit associated with a specific error. When more bits are required for errors, they can be obtained by expanding the error section in the word or by using another status word. Generally, bit 15 is the inclusive OR of all other error bits (if there is more than one). All errors are generally indicated by individual status bits.

11	Busy	Indicates that a step is being performed.
10-8	Unit Select	Some peripheral systems have more than one device per control. For example, a disk system can have multiple surfaces per control, and an analog-to-digital converter can have multiple channels. The unit bits select the proper surface or channel.
7	Done or Ready	The register can contain a DONE bit, a READY bit, or a DONE-BUSY pair of bits, depending on the device. These bits are set and cleared by the hardware, but may be queried by the program to determine the availability of the device.
6	Interrupt Enable	Independently programmable. If bit 6 is set, an interrupt will occur as a result of a function done or error condition.
5-4	Memory Extension	Will allow devices to use a full 18 bits to specify addresses on the bus.
3-1	Device Function Bits	Specify operations that a device is to perform. For example, a paper tape read function could be "read one character." An operation for a disk could be "read a block of words from memory and store them on the disk."
0	Enable	When set, this bit enables the device to perform the I/O device function.

Each device has at least one buffer register, besides the CSR registers, for temporarily storing data to be transferred into or out of the computer. The number and type of data registers is a function of the device. The paper tape reader and punch use single 8-bit data buffer registers. A disk would use 16-bit data registers and some devices may use two 16-bit registers for data buffers.

6.2. BASIC DEVICE FUNDAMENTALS

The two most basic peripheral devices commonly attached to a PDP-11 are the ASR-33 Teletype® and the DEC PC-11 high-speed paper tape unit. Actually, these two devices are really four units in that the teletypewriter keyboard/reader and printer/punch are two separate units, as are the paper tape reader and punch contained in the PC-11.

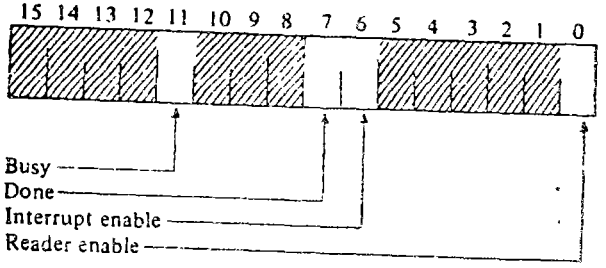
6.2.1. Teletype Keyboard/Reader

The teletype control contains an 8-bit buffer (TKB) which assembles and holds the code for the last character struck on the keyboard or read from the

tape. Teletype characters from the keyboard/reader are received serially by the 8-bit shift register TKB. Upon program command, the contents of the TKB may be transferred in parallel to a memory location or a general register.

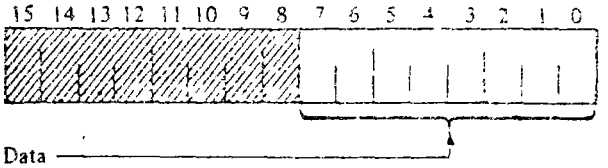
A character is read from the low-speed reader by setting the teletype reader enable bit, (RDR ENB), to a 1. This sets the busy bit (BUSY) to a 1. When a teletype character starts to enter, the control deenergizes a relay in the teletype unit to release the tape feed latch. When released, the latch mechanism stops tape motion only when a complete character has been sensed and before sensing of the next character is started. When the character is available in buffer (TKB), the busy bit (BUSY) is cleared and the done flag (DONE) is set. The keyboard must be read within 18 milliseconds of DONE to ensure that there is no loss of information.

Teletypewriter Keyboard/Reader Status Register (TKS):



Bit	Name	Description
15-12		Not used.
11	Busy	Indicates that the teletype control is receiving a start bit or information bits. Cleared by INIT, set by start bit, cleared after reception of first halt bit. Read only.
10-8		Not used.
7	Done	Character available in buffer. Cleared by INIT, cleared by referencing data buffer, causes interrupt when INTR ENB = 1. Read only. Cleared when RDR ENB is set.
6	Reader Interrupt Enable (INTR ENB)	Interrupts Enable. Enables Error or Done to cause an interrupt. Cleared by INIT.
5-1		Not used.
0	Reader Enable (RDR ENB)	Enables reader (*not keyboard*) to read one character. Cleared by INIT, cleared when legitimate start bit is detected. Low order.

Teletypewriter Keyboard/Reader Buffer (TKB):



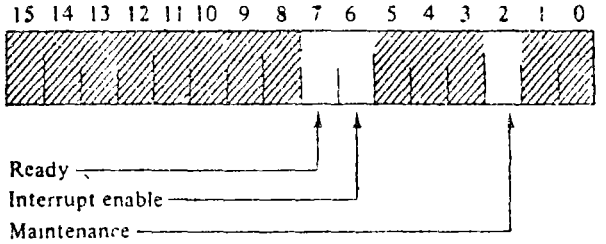
Bit	Name	Description
15-8		Not used.
7-0	Data	Holds character read. Cleared by start bit. Read only.

Any reference to TKB (as word or byte) or TKB + 1 clears DONE. The "unused" and "load only" bits are always read as zeros. Loading "unused" or "read only" bits has no effect on the bit position. The mnemonic "INIT" refers to the initialization signal issued by ON, POWER UP, console START, or RESET.

6.2.2. Teletype Printer/Punch

On program command, a character is sent in parallel from a memory location (or a general register) to the TPB for transmission to the teleprinter/punch unit. This transfer of information from the TPB into the teleprinter/punch unit is accomplished at the normal teletype rate and requires 100 milliseconds for completion. The READY flag in the teleprinter/punch indicates that the TPB is ready to receive a new character. A maintenance mode is provided which connects the TPB output to the TKB input so that the teletypewriter operation may be verified.

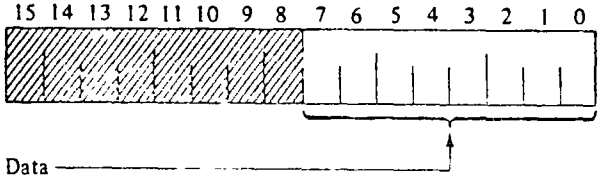
Teletypewriter Printer/Punch Status Register (TPS):



Bit	Name	Description
15-8		Not used.

7	Ready	Punch available. Set by INIT, cleared when buffer is loaded, set when punching complete. Caused interrupt if INTR ENB = 1. Read only.
6	Interrupt Enable (INTR ENB)	Enables READY to cause interrupt. Cleared by INIT.
5-3		Not used.
2	Maintenance	Maintenance function. Disables serial line input from teletype unit and enables serial output of punch to feed into reader buffer. Cleared by INIT.
1-0		Not used.

Teletypewriter Printer/Punch Buffer Register (TPB):



Bit	Name	Description
15-8		Not used.
7-0	Data	Holds character to be punched. Cleared by INIT. Load only.

Any instruction that could modify TPB as a byte or word clears READY and initiates punching. Other references to either byte or word have no effect on the punch.

The four addressable registers associated with the teletype may be read or loaded using any PDP-11 instruction that refers to their address. The address assignments for these registers are as follows:

Register	Address
TKS	177560
TKB	177562
TPS	177564
TPB	177566

When using PAL-11, a direct assignment is made (e.g., TKB = 177562) so that the device registers may be referenced symbolically.

6.2.3. Simple Programming Example

Since the teletype keyboard is treated as a separate unit from the printer, it is necessary to write a simple program to "echo" back to the printer a character typed on the keyboard. This program looks as follows:

```
        TKS=177560              ;DEFINE
        TKB=TKS+2               ;STATUS
        TPS=TKS+4               ;AND BUFFER
        TPB=TKS+6               ;REGISTERS
        .=1000
ECHO    INC     TKS             ;SET READER ENABLED
LOOP1   TSTB    TKS             ;TEST FOR DONE
        BPL     LOOP1           ;GOES NEGATIVE WHEN SET
LOOP2   TSTB    TPS             ;TEST PRINTER READY
        BPL     LOOP2           ;GOES NEGATIVE WHEN SET
        MOVB    TKB,TPB         ;MOVE CHARACTER
        BR      ECHO            ;LOOP AROUND AGAIN
        END     START
```

The value of making the DONE bit line up with the byte boundary is clearly demonstrated in this example. Had it not been set up as the sign bit of the byte, it would have been necessary to copy the status register to a temporary location so that a bit test (BIT) could have been performed, followed by a branch on zero. Since this alternative, although possible, is not as "neat" as the TSTB, it illustrates once again the value of properly designing a computer at both the hardware and software levels.

The setting of the reader enable bit in this program is superfluous. However, by including the instruction to do so the program is generalized in that input is allowed to come from either the keyboard or the reader. Likewise, output can go to either the printer or the punch. All that is necessary is for the user to place a paper tape in the reader and set it to "start," or to turn on the punch, and these paper tape devices become operative (in parallel) with their counterparts (e.g., the keyboard or the printer). Consequently, this one program allows for any legitimate combination of teletype devices to be connected together.

6.2.4. More Complex Octal Dump Program

A programming tool frequently used by assembly language programmers is the *memory dump program*. This program aids the user who is developing or debugging programs by providing him with an octal copy of a program or

The program shown is a memory-to-teletypewriter octal dump routine and illustrates basic I/O programming utilizing the teleprinter. It also illustrates the use of position-independent coding. The need for PIC is dictated, of course, by the necessity of being able to load the dump routine anywhere in memory.

The program begins by typing an "A" character and waiting for the user to type in an octal starting location (up to five digits). The return key causes the program to respond with a line feed and an "N" character, signifying a program request for number of words to be dumped. The second return begins the dump:

```
A1000
N12

001000  010706  005745  112700  000015  004767  000142  112700  000012
001020  004767  000132
```

A flowchart of this program is shown in Fig. 6-1, and the actual program looks as shown in Fig. 6-2.

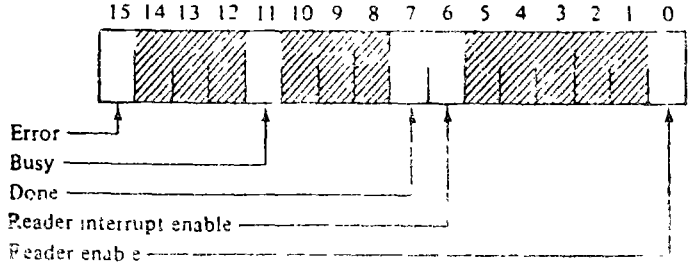
6.2.5. High-Speed Reader/Punch

The high-speed reader/punch consists of two units for reading and punching eight-hole perforated paper tape at, respectively, 300 characters per second and 60 characters per second. Each unit has its own status and buffer registers capable of controlling the transfer of one byte to or from the unit.

Data are recorded (punched) on paper tape by groups of holes arranged in a definite format along the length of the tape. The tape is divided into *channels*, which run the length of the tape, and into *columns*, which extend across the width of the tape as shown in Fig. 6-3.

The status register for the paper tape reader is almost identical in format to the status register for the teletypewriter keyboard/reader. The difference is found in the error bit, which is set by an "out of tape" or "off-line" condition.

Paper Tape Reader Status Register (PRS):



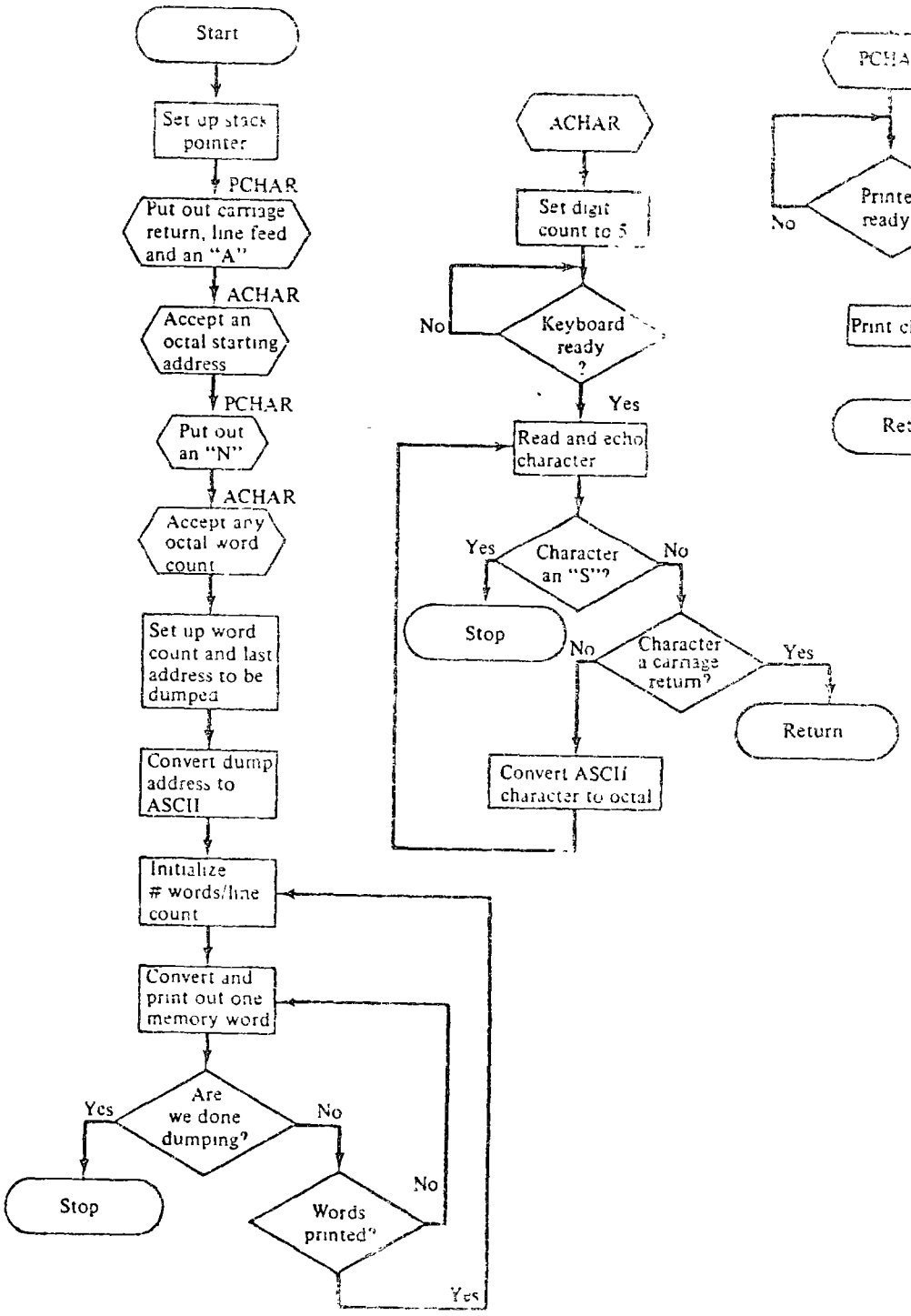


Fig. 6-1

```
;
;       EDUMP - AN OCTAL DUMP PROGRAM
;       WRITTEN BY ELLIOT SOLOWAY 6/1/72
;
;       INPUTS ARE N -- THE NUMBER OF WORDS
;              AND A -- THE STARTING ADDRESS
;
;       OUTPUT IS THE STARTING MEMORY ADDRESS
;              AND THE CONTENTS OF UP TO 8 WORDS
;              OF MEMORY
;
        R0=%0
        R1=%1
        R2=%2
        R3=%3
        R4=%4
        R5=%5
        SP=%6
        PC=%7

        TKS=177560
        TKB=TKS+2
        TPS=TKS+4
        TPB=TKS+6

        CR=15
        LF=12

CORE    MOV     PC,SP           ;SET UP STACK POINTER
        TST     -(SP)
        MOVB    #CR,R0          ;PRINT INITIAL CARRIAGE
        JSR     PC,PCHAR        ;RETURN AND
        MOVB    #LF,R0          ;LINE FEED USING
        JSR     PC,PCHAR        ;PUT CHARACTER SUBROUTINE
ADDR    MOVB    #'A,R0          ;PRINT AN "A"
        JSR     PC,PCHAR
        JSR     PC,ACHAR        ;ACCEPT UP TO 5 OCTAL DIGITS AS ADDRESS
        MOV     R5,R1           ;R1 CONTAINS START ADDRESS
        MOVB    #'N,R0          ;PRINT AN N FOR NUMBER OF WORDS
        JSR     PC,PCHAR
        JSR     PC,ACHAR        ;ACCEPT <= 5 OCTAL DIGITS
        MOV     R5,R2           ;FORM WORD COUNT NUMBER
        ADD     R5,R2           ;TO BE DUMPED
        ADD     R1,R2           ;FORM ENDING ADDRESS
        TST     -(R2)           ;LESS TWO
LOOP1   MOV     PC,R4           ;SET UP RELATIVE ADDRESS
        ADD     #BUF-LOOP1-2+6,R4       ;OUTPUT BUFFER START ADDRESS
        MOV     #CR,R0          ;RESET PRINTER
        JSR     PC,PCHAR        ;CARRIAGE
        MOV     #LF,R0          ;FOR DUMP
        JSR     PC,PCHAR        ;INFORMATION
ARND    MOV     R1,R0           ;CONVERT THE DUMP ADDRESS
        JSR     PC,CNVRT2       ;TO ASCII CHARACTERS
        MOV     #8.,R3          ;NUMBER OF WORDS DUMPED PER LINE
OCTAL   MOV     PC,R4           ;SET UP RELATIVE ADDRESS OF
        ADD     #BUF-OCTAL-2+6,R4       ;BUFFER
        JSR     PC,CNVRT1       ;PRINT ONE WORD
```



```
        TST     (R1)+           ;NEXT ADDRESS TO BE DUMPED
        CMP     R2,R1           ;ARE WE DONE?
        BLT     FUDG            ;YES, PRINT A CR,LF
        MOVB    (R4)+,@#TPB     ;MOVE IT TO TELEPRINTER
COMP    MOV     PC,R0           ;CALCULATE LAST BYTE ADDRESS
        ADD     #BUF-COMP+2+7,R0,   OF BUFFER
        CMP     R0,R4           ;ARE WE DONE?
        BGE     DUMP            ;NO, PRINT ANOTHER CHARACTER
        RTS     PC              ;YES, RETURN
FUDG    MOVB    #CR,R0          ;PUT OUT THE CR
        JSR     PC,PCHAR
        MOVB    #LF,R0          ;AND LF COMBINATION
        JSR     PC,PCHAR
        JSR     PC,PCHAR
        JMP     ADDR            ;RETURN
;
;       CONSTANTS AND DATA BUFFERS
;
FIVE    WORD    0
BUF     WORD    0
        =.+4
        ASCII   /  /
CNT     WORD    0
        END
```

Fig 6-2 (cont)

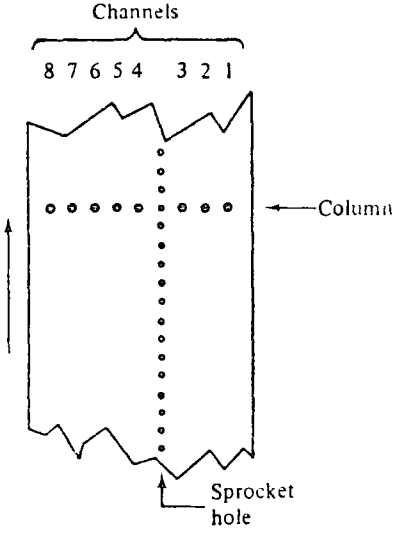
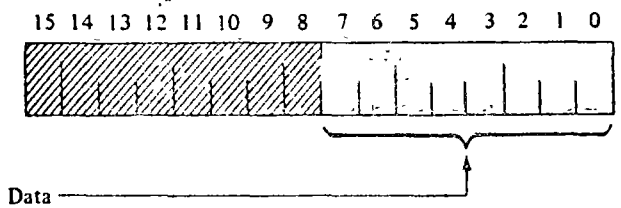


Fig. 6-3 Punched paper tape.

Bit	Name	Description
15	Error	Indicates one of three possible error conditions: no tape in the reader, reader is off-line, or reader has no power. Disables RDR ENB; causes interrupt if RDR INT ENB = 1.
14-12		Not used.
11	Busy	Indicates that a character is in the process of being read. Cleared by INIT, set by RDR ENB, cleared when character is available in buffer. Read only.
10-8		Not used.
7	Done	Character available in buffer. Cleared by INIT, set when character available, cleared by referencing reader buffer (PRB), cleared by setting RDR ENG; causes interrupt when RDR INT ENB = 1. Read only.
6	Reader Interrupt Enable (RDR INT ENB)	Interrupts enable. Enables the error or done bits to cause an interrupt. Cleared by INIT.
5-1		Not used.
0	Reader Enable (RDR ENB)	Enables reader to fetch one character. Clears done, sets busy, and clears reader buffer (PRB). Operation of this bit is disabled if ERROR = 1; attempting to set it when ERROR = 1 will cause an immediate interrupt if RDR INT ENB = 1. Load only.

The paper tape punch unit behaves much like the teletypewriter keyboard/punch, only at a higher speed. It, too, like the paper tape reader, has an error bit which is set when the punch is "out of tape" or is "off-line."

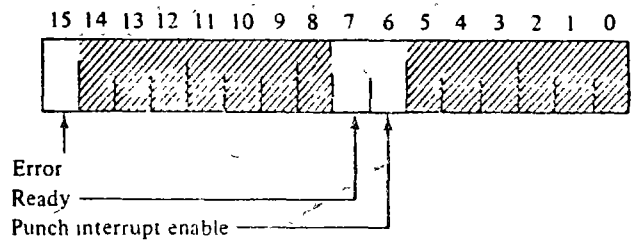
Paper Tape Reader Buffer (PRB):



Bit	Name	Description
15-8		Not used.
7-0	Data	Holds character to be read. Cleared by RDR ENB. Read only.

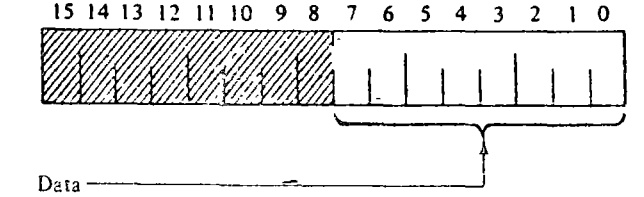
Note: Referencing either high byte or low byte or both bytes clears DONE. Referencing is any operation (read, load, test, compare).

Paper Tape Punch Status (PPS):



Bit	Name	Description
15	Error	Indicates one of two error conditions in punch: no tape in punch, or punch unit out of power. Causes interrupt if PUN INT ENB (or PPS) = 1.
14-8		Not used.
7	Ready	Ready to punch character. Set by INIT, cleared by loading data buffer (see note under PPB), set when punching complete. Causes interrupt when PUN INT ENB = 1. Read only.
6	Punch Interrupt Enable (PUN INT ENB)	Interrupts enable. Enables error or ready to cause interrupt. Cleared by INIT.
5-0		Not used.

Paper Tape Punch Buffer Register (PPB):



Name	Description
Data	Write only. Any instruction that could modify bits 7-0 of PPB clears Ready and initiates punching. An immediate interrupt will occur when punching is initiated if error = 1 and PUN INT ENB = 1.

INITIAL LOAD PROBLEM

When a computer is first received by the customer, its memory is usually
n unloaded state. With the exception of the hardware bootstrap option
cussed later), which may have been purchased with the system, the com-
er "knows" nothing, not even how to accept input. The problem is that
rder to load memory with a user program, there must already be instruc-
s in memory for loading the user program. This seeming contradiction is
n compared to lifting oneself up by one's own bootstraps, and therefore
s the name of the *bootstrap* or *initial load problem*.

One possible solution to this apparent dilemma is to require the CPU to
e some form of *deposit* mechanism which allows the user to deposit
hine language instructions in specified memory locations. This mechan-
includes a way of specifying both the data to be deposited and the
ress in memory of where it is to go.

The *software bootstrap* for the PDP-11 is a sequence of instructions for
ling user programs. The bootstrap utilizes a special paper tape format
self-modification in order to work. The bootstrap loader source program
nown in Fig. 6-4. The starting address in the example denotes that the

```
    000001          R1=%1              ;USED FOR THE DEVICE ADDRESS
    000002          R2=%2               USED FOR THE LOAD ADDRESS DISPLACE
    017400          LOAD=17400                 ;DATA MAY BE LOADED NO LOWE
                                               ;THAN THIS
    017744           =17744            ;START ADDRESS OF THE BOOTSTRAP LOAD
744 016701 START    MOV     DEVICE,R1          ;PICK UP DEVICE ADDRESS
    000026
750 012702 LOOP     MOV     # -LOAD+2,R2       ;PICK UP ADDRESS DISPLACEMENT
    000352
754 005211 ENABLE   INC     @R1                ;ENABLE THE PAPER TAPE READER
756 105711 WAIT     TSTB    @R1                ;WAIT UNTIL FRAME
760 100376          BPL     WAIT               ;IS AVAILABLE
762 116162          MOVB    2(R1),LOAD(R2)     ;STORE FRAME READ FROM TAPE
    000002
    017400
770 005267          INC     LOOP+2             ;INCREMENT LOAD ADDRESS
                                               ;DISPLACEMENT
    177756
774 000765 BRNCH    BR      LOOP               ;GO BACK AND READ MORE DATA
776 000000 DEVICE   0                          ;ADDR OF INPUT DEVICE
    000001           END
```


loader is to be loaded into memory bank zero (a 4K system). It is loac
by hand, using the deposit switch (see Appendix E), into the last 14_{10} me
ory words of the computer.

In operation, the bootstrap actually loads the data read into success
bytes located above the LOAD address. A sample tape input to load d
starting at location 17600 and ending at 17742 would be

```
351 }
351 }
 .  }  leader
 .  }
 .  }
351 }
177 }  lower byte of starting displacement - 1
 .  }
 .  }  date to be loaded
 .  }
301 }
035 }
026 }  byte equivalent to MOV DEVICE,R1
000 }
302 }
025 }  byte equivalent to MOV #.LOAD+2,R2
373 }
XXX}  address between 17600 and 17742 where loaded program begi
```

The necessity for the special leader is dictated by the need to be able t
load an all-zero byte or blank tape. The bootstrap loader starts by loadin
the device status register address into R1 and 352_8 into R2. The next in
struction indicates a read operation in the device and the next two instruc
tions form a loop to wait for the read operation to be completed. When dat
are encountered, they are transferred to a location determined by the sum c
the index word (177400) and the contents of R2.

Because R2 is initially 352_8, the first word is moved to location 17775
and it becomes the immediate data to set R2 in the next execution of th
loop. These immediate data are then incremented by 1 and the progra
branches to the beginning of the loop.

The leader code, plus the increment, is equal in value to the data place
in R2 during the initialization, therefore, leader code has no effect on th
loader program. Each time leader code is read, the pro sor executes th
same loop and the program remains unmodified. The first code other tha

value which acts as a pointer to the program starting location (loading address). Subsequent bytes are read not into the location of the immediate data but into consecutive core locations. The program will thus be read in byte by byte. The INC instruction which operates on the data for R2 puts data bytes in sequential locations and requires that the value of the leader code and the offset be 1 less than the desired value in R2.

The boot overlay code will overlay the first two instructions of the loader, because the last data byte is placed in the core location immediately preceding the loader. The first instruction is unchanged by the overlay, but *the second instruction is changed to place the next byte read, jump offset into the lower byte of the branch instruction. By changing the offset in this branch instruction,* the loader can branch to the start of the loaded program or to any point within the program. The self-modification scheme used not only loads the data but also initializes the bootstrap code and forces a jump to an address 17XXX within the program just loaded.

The key requirement for a deposited bootstrap loader is that it be short in length. Clearly, as the bootstrap program becomes longer, its usefulness decreases as the frustration to deposit it in memory increases. Therefore, another technique is used to bootstrap in user programs.

The alternative technique is to add a *hardware* bootstrap loader to the CPU so that the hardware can perform the initial program load (IPL). The IPL is activated by pushing a "load" button on the CPU, causing a predefined instruction sequence to be executed. This instruction sequence includes both the command sequence for the input device and the specific memory locations into which information is to be placed.

The form that the hardware bootstrap mechanism takes varies from machine to machine. Examples include either reading a data record into memory and executing the first (or last) instruction word read in, or executing an instruction sequence held in read-only-memory (the PDP-11 uses an ROM) or on an alterable "dead-start" panel. Regardless of the method, the result is usually the same, the loading into memory of a short program sequence called the *absolute loader*.

In operation, the absolute loader is a *systems program* for reading input records that contain machine language instructions bound to absolute memory locations. Unlike the bootstrap loader, the absolute loader is capable of reading large amounts of information into various segments of memory. The format of the information is such that each record contains

1. A word count of the number of words in the record.
2. A load address where the first and subsequent words in the record are to be loaded.
3. The words to be loaded.
4. A transfer address for the absolute loader.

Both the absolute and bootstrap loaders are systems programs. Systems programs are those programs which of themselves do not produce useful results but rather aid the programmer in accomplishing his desired objective. Systems programs are written by systems programmers whose job is the support of the users of the system. Systems programs include such things as PAL-11 and EDUMP. Chapter 7 is devoted entirely to discussing the multitude of systems programs available to PDP-11 users

6.4. TAPE AND DISK STORAGE UNITS

Many large-capacity storage devices may be connected to a small computer such as the PDP-11. Two such devices commonly found on this computer are the DECtape (capacity 147,968 words) and the DECdisk (capacity 65,536 words). Since these bulk storage devices require more elaborate programming and control, it is instructive to examine their characteristics and operation.

6.4.1. DECtape Operation

DECtapes consist of 10 tracks arranged in the format shown in Fig. 6-5. On a tape the first five tracks include the timing and mark tracks, plus three data tracks. The other five tracks are identical counterparts and serve to increase system reliability through redundant recording.

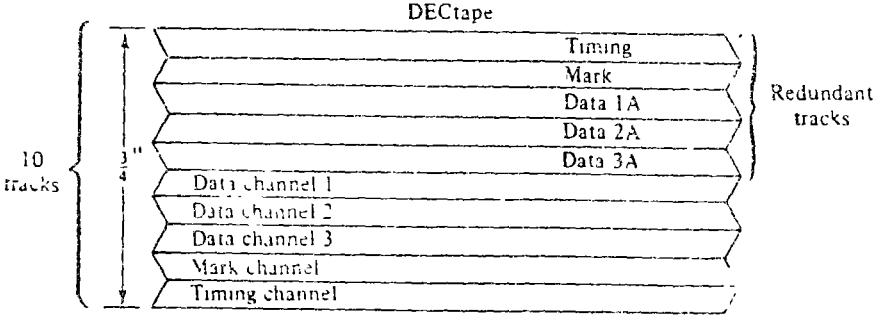


Fig. 6-5 DECtape format.

The timing and mark channels are recorded prior to all normal data reading and writing on the information channels. Information read from the mark channel is used during the reading and writing of data to indicate the beginning and ending of data blocks and to determine the functions to be performed by the system in each control mode. The data in one bit position of

each track are referred to as a line or a character. Since six lines or characters make up a word, the tape can record 18-bit data words. Normally, the 2 extra bits are ignored.

A reel of DECtape is divided into three major areas: end zones, extension zones, and the information zones. The information area consists of blocks of data, containing 256 data words per block. Altogether there are 578 blocks of information (see Fig. 6-6).

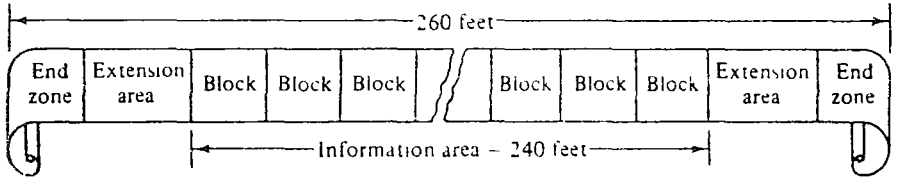


Fig. 6-6 DECtape block arrangement

The blocks permit digital data to be partitioned into groups of words that are interrelated, at the same time reducing the amount of storage area that would be needed for addressing individual words. A simple example of such a group of words is a program. A program can be stored and retrieved from magnetic tape in a single block format because it is not necessary to be able to retrieve only a single word from the program. It is necessary, however, to be able to retrieve different programs that may not be related in any way. Thus each program can be stored in a different block on the tape.

Since DECtape is a fixed address system, the programmer need not know accurately where the tape has stopped To locate a specific point on the tape he must only start the tape motion in the search mode. The address of the block currently passing over the head is read into the DECtape control and loaded into an interface register. Simultaneously, a flag is set and a program interrupt can occur. The program can then compare the block number found with the desired block address and tape motion continued or reversed accordingly.

All DECtape operations are handled by the controller through program instructions. The controller selects the transport, controls tape motion and direction, selects a read or write operation, and buffers data transferred

The controller can select any one of eight commands that control operation of the DECtape system When the system is operated on-line, these commands are used for reading or writing data on the tape and for controlling tape motion. The desired command is selected by the program, which sets or clears bits 03, 02, and 01 in the command register (TCCM) to specify an octal code representing the desired command.

The commands are as follows.

Octal Code	Mnemonic	Function
0	SAT	Stops all tape motion.
1	RNUM	Finds the mark track code that identifies the block number on the tape in the selected tape unit. Block number found is available in the data register (TCDT).
2	RDATA	Assembles one word of data at a time and transfers it directly to memory. Transfers continue until word count overflow, at which time data is read to the end of the current block and parity checked.
3	RALL	Reads information on the tape that is not read by the RDAT function.
4	SST	Stops all tape motion in selected transport only.
5	WRTM†	Writes timing and mark track information on blank DECtape. Used for formatting new tape.
6	WDATA†	Writes data into the three data tracks. 16 bits of data are transferred directly from memory.
7	WALL†	Writes information on areas of tape not accessible to WDAT function.

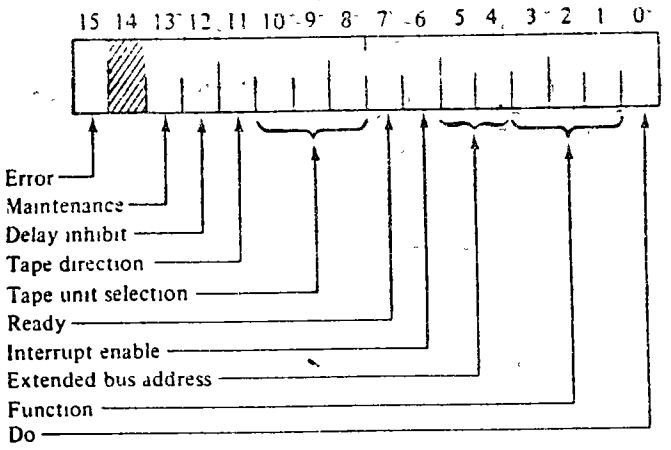
†Switches on the DECtape unit itself must be set in order to prevent accidental overwriting on information already on the DECtape.

All software control of the DECtape system is performed by means of five device registers. They can be read or loaded using any PDP-11 instruction that refers to their address.

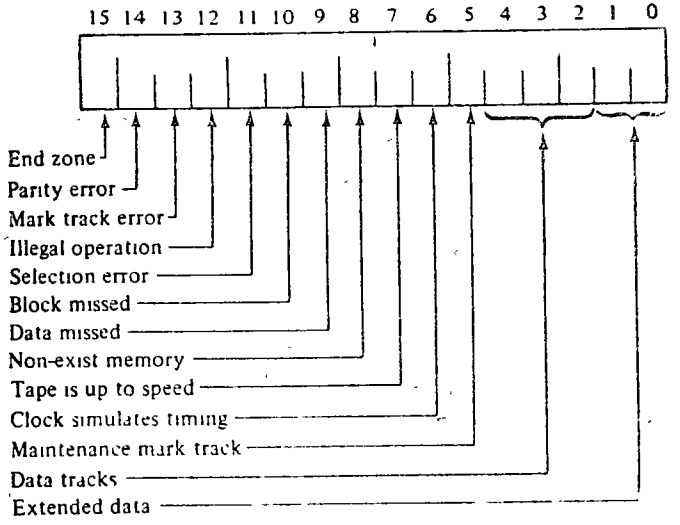
Register	Address
Control and Status Register (TCST)	777340
Command Register (TCCM)	777342
Word Count Register (TCWC)	777344
Bus Address Register (TCBA)	777346
Data Register (TCDT)	777350

The bit utilization for each of these registers is shown Fig. 6-7.

Command register (TCCM)



Control and status register (TCST)



Word count register (TCWC)

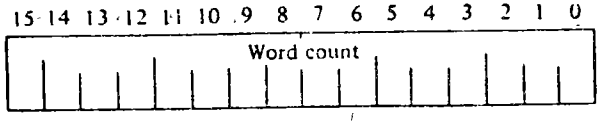
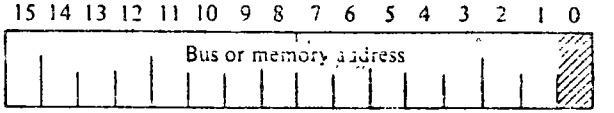


Fig. 6-7

Bus address register (TCBA)



Data registers (TCDT).

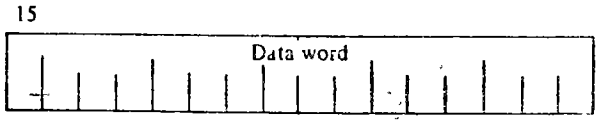
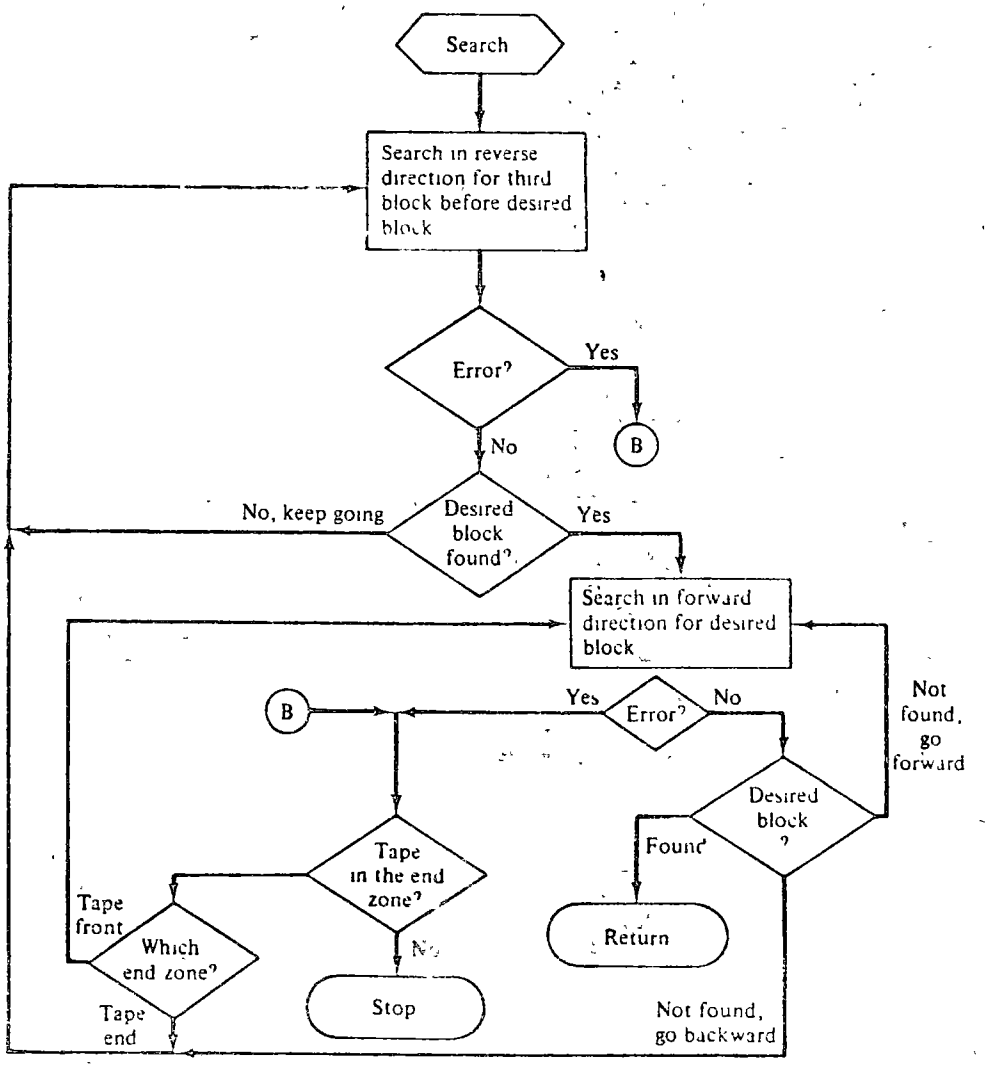


Fig. 6-7 (cont)



6.4.2. Programming Examples

Because DECtapes are organized like disks, they are programmed in much the same fashion. Thus, before one can write in a specified block, the block must be located. A typical method to locate a block is to initiate tape motion and then search for the desired block in either the forward or reverse direction. The search consists of examining each block number as it is read and comparing it to the block number being sought. As soon as a match occurs, reading or writing to the located block may begin.

Although this procedure is relatively simple, several DECtape characteristics must be taken into consideration. First, before DECtapes can be read or written, they must be "up to speed." Thus it takes some time and hence some tape passed over the tape heads before the first block number will actually be read. Second, while waiting for a block number to read after start-up, the tape may be repositioned in the end zone. This error condition requires the tape motion to be reversed so that the tape may be searched in the opposite direction. Third, and finally, having found the desired block, reading or writing must be initiated shortly thereafter, or else the transfer will be unsuccessful and a tape error condition raised.

With these points in mind, it is possible to flowchart and code the search procedure as shown in Fig. 6-8. The routine to find a specified block (1) expects the block number wanted to be legitimate and in R0, and (2) finds the block while searching in the forward direction, see Fig. 6-9.

```
        ;DECTAPE SEARCH ROUTINE
        ;R0 CONTAINS DESIRED BLOCK NUMBER
        ;BLOCK FOUND IN FORWARD DIRECTION

        TCST=177340             ;CONTROL/STATUS REGISTER
        TCCM=TCST+2             ;COMMAND REGISTER
        TCDT=TCST+4             ;DATA REGISTER

SEARCH  MOV     R0,BWANT        ;SAVE BLOCK NUMBER
        SUB     #3,BWANT        ;OFFSET TO DESIRED BLOCK
        MOV     #4002,TCCM      ;READ BLOCK NUMBERS IN REVERSE DIR
LOOP1   BIT     #100200,TCCM    ;CHECK READY AND ERROR BITS
        BEQ     LOOP1           ;WAIT FOR READY
        BMI     ERROR           ;FOUND AN ERROR?
        SUB     TCDT,BWANT      ;CHECK BLOCK FOUND
        BLT     SEARCH          ;KEEP SEARCHING BACKWARDS
FORWRD  MOV     R0,BWANT        ;SAVE BLOCK NUMBER
        MOV     #3,TCCM         ;READ BLOCK NUMBERS IN FORWARD DIR
LOOP2   BIT     #100200,TCCM    ;CHECK READY AND ERROR BITS
        BEQ     LOOP2           ;WAIT FOR READY
        BMI     ERROR           ;HAVE AN ERROR?
        SUB     TCDT,BWANT      ;CHECK BLOCK FOUND
        BGT     FORWRD          ;BLOCK NUMBER TOO SMALL
        BLT     SEARCH          ;BLOCK NUMBER TOO BIG
        RTS     PC              ;RETURN WHEN BLOCK FOUND
ERROR   TST     TCST            ;END ZONE ERROR?
        BI      LOOP2           ;IF SO BRANCH
        HALT                    ;OTHERWISE HALT ON ERROR
```



```
LOOP3   BIT     #4000,TCCM      ;TEST DIRECTION
        BNE     FORWRD          ;IF REVERSE, SEARCH FORWARD
        BR      SEARCH          ;IF FORWARD, SEARCH REVERSE
BWANT   0                       ;BLOCK NUMBER
```

Fig. 6-9 (cont.)

When a specified block has been searched for and found, the next thing to do is to transfer information from or to it. The routine shown in Fig. 6-10 uses the SEARCH subroutine to read 100 words from block 50 on DECtape unit 0. The program calls SEARCH, sets up the word count and buffer address, and then waits for the read to be completed. The reader should note that although blocks contain 256 data words, any number of words (up to 256) may be specified in the transfer operation.

```
        ;ROUTINE TO READ 100 WORDS FROM
        ;BLOCK 50, DECTAPE UNIT 0

        R0=%0                   ;REGISTER ZERO
        SP=%6                   ;STACK REGISTER
        PC=%7                   ;PROGRAM COUNTER
        TCCM=177342             ;COMMAND REGISTER
        TCWC=TCCM+2             ;WORD COUNT REGISTER
        TCBA=TCCM+4             ;BUS ADDRESS REGISTER

START   MOV     PC,SP           ;INITIALIZE STACK
        TST     -(SP)           ;   POINTER
        MOV     #50,R0          ;BLOCK 50 TO BE
        JSR     PC,SEARCH       ;   SEARCHED FOR
        MOV     #-100,TCWC      ;COMPLEMENT OF WORD COUNT
        MOV     #BUFFER,TCBA    ;BUFFER ADDRESS
        MOV     #15,TCCM        ;READ DATA FORWARD DIRECTION
LOOP    BIT     #100200,TCCM    ;CHECK ERROR AND READY
        BEQ     LOOP            ;WAIT FOR READY AND NO ERROR
        BMI     ERR             ;BRANCH ON ERROR

ERR     HALT                    ;HALT ON ERROR
BUFFR   .=.+200                 ;SAVE ROOM FOR BUFFER
        END     START           ;END OF ASSEMBLY
```

Fig. 6-10

6.4.3. Disk Operation

Because of the differing requirements for disk storage, many storage alternatives are available to the small computer user. The choice of disk systems spans the range from fast access and fast storage to large storage and medium-access devices. For example, *fixed-head* disk systems are suited for swapping-type devices (e.g., those where the contents of memory and disk must be rapidly exchanged) and scientific applications where fast access and fast

transfer are important. The *moving-head* systems are ideal for large storage requirements where fast access times are less vital.

Before discussing the programming of the disk devices, it is important to understand their basic operation. Generally, all disk devices are organized around flat magnetic surfaces, called *platters*, which look like pancakes. The surface is divided into concentric rings called *tracks* with each track subdivided into *sectors*. The sector is the smallest addressable unit and generally is capable of storing many computer words (e.g., 32 words per sector). Figure 6-11 shows such a disk organization.

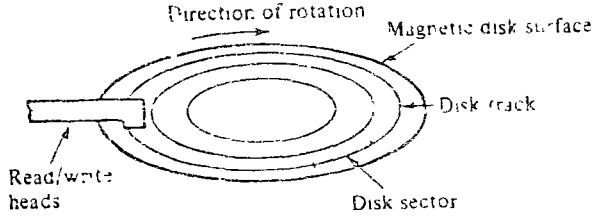


Fig. 6-11 Disk organization.

If the disk platter can be removed from the disk drive mechanism, and another platter used in its place, the removable surface becomes a *disk pack*. Disk packs may consist of one or more platters, with multiple surfaces being stacked vertically on the same shaft, as shown in Fig. 6-12. By logically grouping all the tracks at the same radius on each surface into a *cylinder*, more information is accessible as a unit, thereby effectively increasing the density of the system. However, since disk pack devices are manipulated in a similar fashion to simpler one-surface devices, it is sufficient to consider only the programming of devices typified by Fig. 6-11.

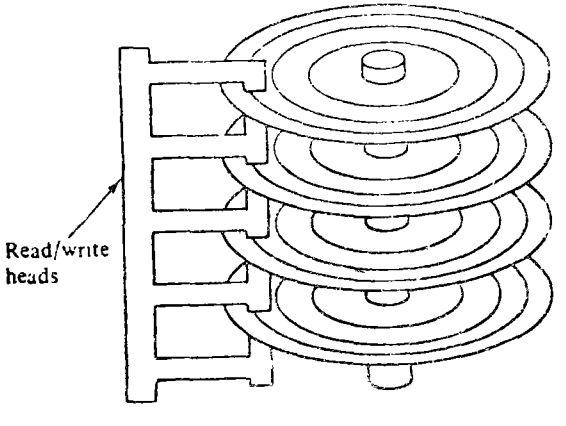


Fig. 6-12 Multisurface disk pack.

Reading or writing disk tracks and sectors can be accomplished in one of two ways. First, there can be one read/write head, which must be positioned over one track or another; this is the moving-head system, mentioned earlier. Second, there can be one read/write head per track, referred to earlier as fixed-head disk system. The advantage of the fixed-head system is that there is no mechanical *seek time* associated with the physical positioning of the head over the appropriate track. Instead, there is a small electronic switching time required to select the appropriate head. The fixed-head system thus requires less time before the accessing of data, but there is a greater cost associated with it because more read/write heads are needed.

Regardless of the type of system, fixed or moving head, there is another delay associated with the disk called *latency*. This is the time it takes for a sector to pass under the read/write head after the appropriate head has been selected. Another name for latency is *rotational delay*, and in a sense it corresponds to the latency of a tape unit while waiting for a particular tape block to come under the tape unit's read/write heads.

Latency time can be reduced by speeding up the rotation of the disk. This also has the effect of passing more information by the read/write head in a given amount of time, thereby increasing the number of characters per second, or *transfer rate* of the device itself. Alternatively, the transfer rate may be increased by just putting more information on a track (e.g., increasing the *density* of information). All these factors, then, density, transfer rate, latency, seek time, fixed/moving head, number of disk surfaces, and so on, must be considered when selecting the appropriate disk system for a particular problem.

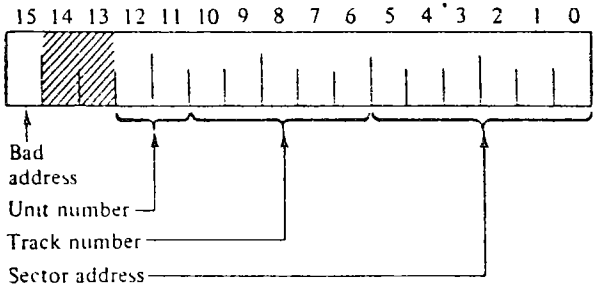
6.4.4 Programming a DECdisk

For simplicity we shall consider a fixed-head DECdisk which has 32 words per sector, 64 sectors per track, and 32 tracks per surface, providing a total capacity of 65,536 words per disk unit. Software control of this DECdisk system is performed by means of eight device registers. Like the registers of other I/O devices, these registers can be read or loaded using any PDP-11 instruction that refers to their address:

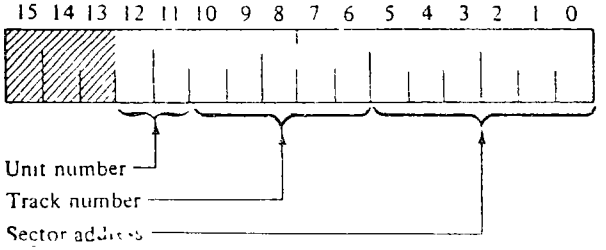
Register	Address
Look Ahead Register (RCLA)	777440
Disk Address Register (RCDA)	777442
Disk Error Status Register (RCER)	777444
Command and Status Register (RCCS)	777446
Word Count Register (RCWC)	777450
Current Address Register (RCCA)	777452
Maintenance Register (RCMN)	777454
Data Buffer Register (RCDB)	777456

The bit utilizations for these registers are shown in Fig. 6-13.

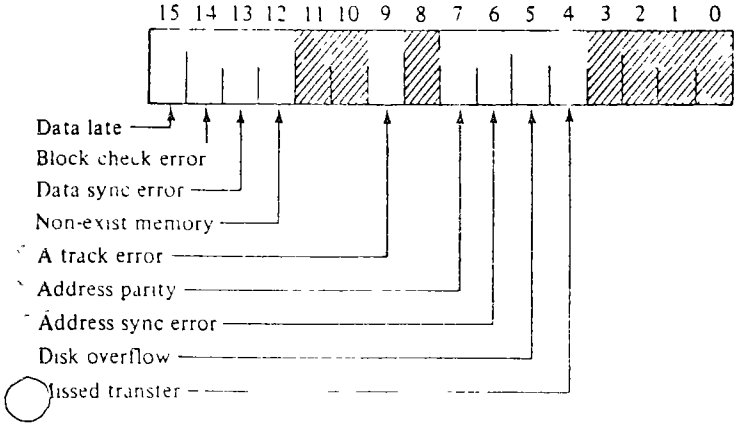
Look ahead register (RCLA)



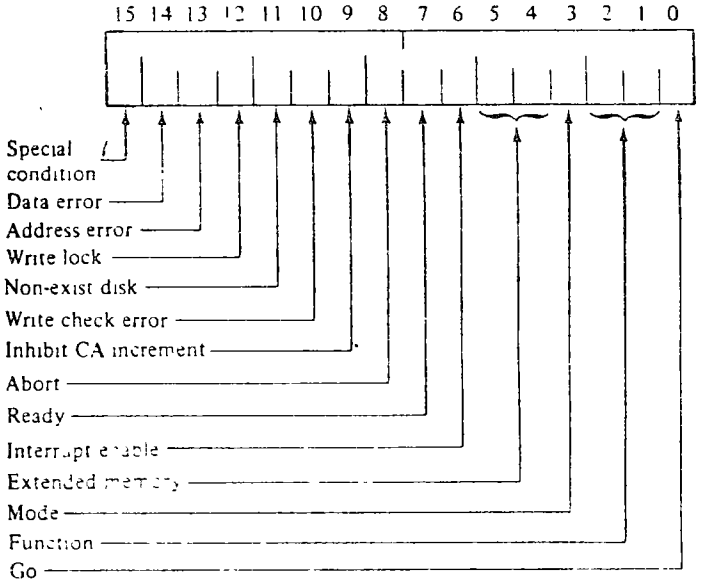
Disk address register (RCDA)



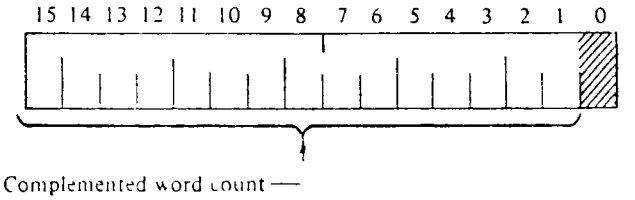
Disk error status register (RCER)



Disk control and status register (RCCS)



Word count register (RCWC)



Current address register (RCCA)

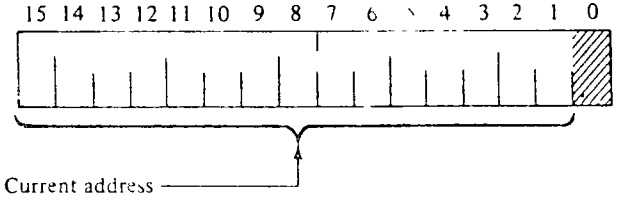
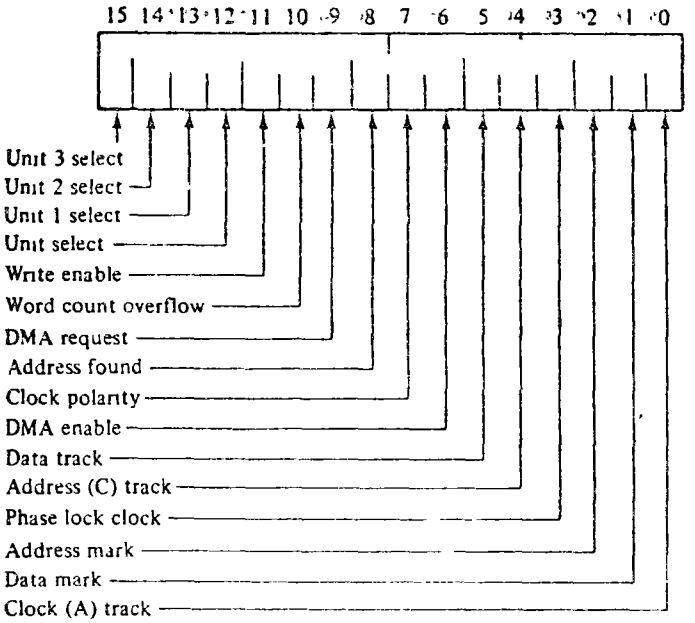


Fig 6-13 (cont.)

Maintenance register (RCMN)



Data buffer register (RCDB)

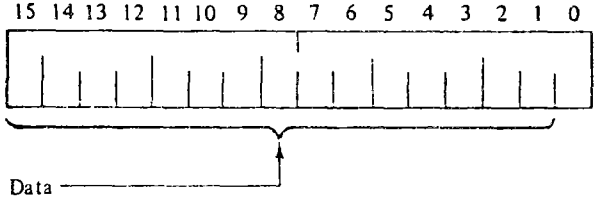


Fig 6-13 (cont.)

Although there are more registers associated with the disk than with the tape unit, programming is easier because the search for a particular sector does not require us to start, stop, or reverse the direction of the disk. Instead, the disk rotates at a constant speed, and all that is necessary is for us to set up the sector address, word count, and buffer address and then wait for the transfer to occur. This sequence of operations can be programmed as shown in Fig. 6-14.


```
;PROGRAM TO READ 100 WORDS FROM
;DISK UNIT 0, TRACK 1, SECTOR 77

RCDA=177442             ;DISK ADDRESS REGISTER
RCER=RCDA+2             ;DISK ERROR REGISTER
RCCS=RCDA+4             ;DISK CSR REGISTER
RCWC=RCDA+6             ;DICK WORD COUNT REGISTER
RCCA=RCDA+10            ;DICK CURRENT ADDRESS REGISTER

START   MOV     #177,RCDA       ;UNIT, TRACK, SECTOR ADDRESS
        MOV     #-100,RCWA      ;WORD COUNT
        MOV     #BUFFR,RCCA     ;BUFFER ADDRESS
        MOV     #5,RCCS         ;READ DISK
        TST     RCCS            ;ANY ERROR?
        BMI     ERROR           ;IF NEG, YES


ERROR   TST     RCER            ;CHECK TYPE OF ERROR
                                ;TAKE APPROPRIATE ACTION


BUFFR   = +200                  ;BUFFER AREA
        END     START           ;END OF ASSEMBLY
```

Fig. 6-14

As for the tape operation, any number of words (up to 65,536) may be transferred, since the disk address register is incremented automatically after each sector is transferred. This process continues both across tracks and even across disk units. Alternatively, if only a portion of the sector (less than 32 words) is desired, the word count register is set accordingly, and only that number of words is transferred to the buffer area.

6.5. PRIORITY INTERRUPT PROGRAMMING

The running time of programs using input and output routines is primarily made up of the time spent waiting for an I/O device to accept or transmit information. Specifically, this time is spent in testing or "polling" the status register of a device and waiting in a loop for a done condition:

```
TEST    TSTB    TKS             ;TEST CSR
        BPL     TEST            ;WAIT FOR DONE
```

Such waiting loops waste a large amount of computer time. In those cases where the computer can be doing something else while waiting, the loops may be eliminated and useful routines included to take advantage of the waiting time. This sharing of a computer between two routines or tasks is accomplished through a program interrupt facility, which is standard on all PDP-11 series computers.

The value of an interrupt facility lies in the ability of the processor to respond automatically to conditions outside the system, or in the processor itself. Unusual conditions occurring at unknown times (such as I/O completion) can generate an interrupt and force the computer to execute an interrupt routine in response to the interrupting action. Thus the user need not poll or test for the occurrence of a condition after the execution of each instruction, but he may write interrupt routines in case they occur.

Basically, an interrupt is a subroutine jump executed by the hardware, as opposed to one written as an explicit software instruction. The interrupt occurs after the execution of an instruction (and before the I-fetch of the next instruction) and must inform the system of the cause of the interrupt. For example, when an interrupt occurs on some machines, an interrupt bit is set in an interrupt status register, indicating what condition raised the interrupt. At the same time, the CPU takes the address of the next instruction from a fixed interrupt location (possibly memory location zero) and begins execution of the interrupt analysis routine at that location.

6.5.1. Interrupt Linkages

Like subroutines, interrupts have linkage information so that a return to the interrupted program can be made. More information is actually necessary for an interrupt transfer than a subroutine transfer because of the random nature of interrupts. The complete machine state of the program immediately prior to the occurrence of the interrupt must be preserved in order to return to the program without any noticeable effects (i.e., was the previous operation zero or negative?). In this way the interrupt will be "invisible" to the interrupted program, since no information, only time, will be lost between the time the running program is interrupted and the time its execution is resumed.

6.5.2. Machine State During Interrupt

The complete machine state of the program immediately prior to the occurrence of the interrupt is generally held in a *processor status word (PSW)*. On computers with sufficiently long memory words, the PSW includes both the condition codes and the program counter. On minicomputers such as the PDP-11, it is necessary to subdivide the PSW into two or more words in order to maintain the *processor status (PS)* and the program counter (PC).

Using one or several words, the technique for handling the interrupt is to replace the current PSW with the interrupt PSW, saving the current PSW somewhere in memory. Diagrammatically this process is depicted in Fig 6-15. The figure shows that two memory locations are required for the interrupt process, plus a register to hold the current PSW.

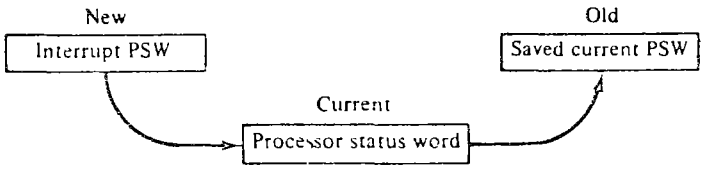


Fig. 6-15 Swapping processor status.

6.5.3. Stacking of Interrupts

One problem with this scheme is that all interrupts use the same swapping technique. Thus, should a second interrupt occur during the execution of the routine to service the first interrupt, the old PSW for the first interrupt will be overwritten and lost. To prevent this, it is necessary to *disable* further interrupts while the interrupt is being serviced. By allocating a bit in the PSW for interrupts enabled/disabled, it is a simple matter to have this bit on in the old PSW and off in the new PSW. Thus when the current PSW becomes the new PSW, interrupts are disabled. When a *return from interrupt (RTI)* occurs, the current PSW becomes the old or saved PSW and interrupts are once more enabled.

During the time in which interrupts are disabled, it is conceivable that other I/O conditions which would ordinarily cause an interrupt may occur. Instead of being allowed to cause an interrupt, these conditions are noted and held in the interrupt status register *(ISR)*. Consequently, when interrupts are enabled, they can cause an interrupt; this guarantees their eventual service.

The interrupt status register serves many purposes. First, it indicates which device has raised an interrupt condition. Second, it saves interruptable conditions during the time that interrupts are disabled. And third, it allows the programmer flexibility in deciding what device to service next after an interrupt has been raised. In particular, this flexibility allows the programmer to decide on the relative priorities of the various interrupts. In this way, under programmer control, when several interrupts occur simultaneously the most critical interrupt may be serviced first.

Allowing the programmer to assign the priorities can lead to problems however. For example, if a high-priority interrupt is raised when interrupts are disabled, there is no way the interrupt can be serviced until the interrupt analysis program is once again executed. Thus it becomes necessary to re-enable the interrupt mechanism during interrupt processing. To do so requires stacking the interrupt return information (the old PSW) and setting the interrupt enable bit. However, one problem connected with the priority of interrupting devices still remains.

6.5.4. Priority Interrupts

Once the interrupt enable bit is set, any device may interrupt. The program to analyze interrupts must therefore examine all the bits in the interrupt status register to choose the highest-priority interrupt to process. Clearly, all that is needed is to allow only higher-priority routines to cause new interrupts, since interrupts at the same or lower levels can wait to be serviced. Thus for programmer convenience, the priority can be built into the hardware and a *priority interrupt* scheme can assign devices to groups within a given priority level. Part of the PSW is used to hold the current priority level, and the loading of the PSW determines the value of the priority level.

Typical PSW and ISR words are shown in Fig. 6-16. These words could serve as the basis for a sophisticated interrupt scheme except for one thing. Although only higher-level priority interrupts are allowed to cause an interrupt, it still is the programmer's job to determine who caused the interrupt. A better scheme would be to let each priority group take its PSW from a different memory location. Thus when an interrupt occurred, it would be known a priori that only certain devices could have caused the interrupt

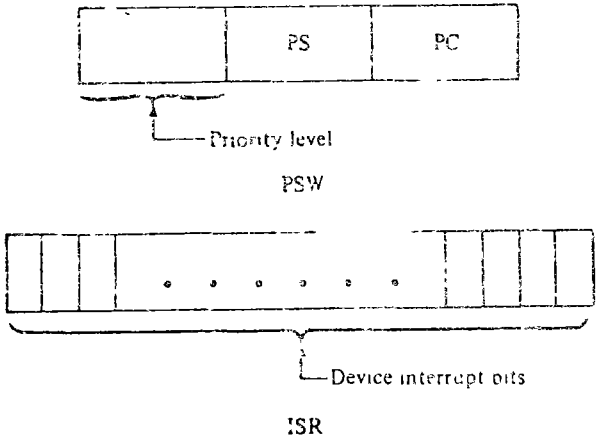


Fig. 6-16 Processor and interrupt status words

6.5.5. Automatic Priority Interrupts

Carrying this idea to its logical conclusion, it should be possible for *each* device to have its own PSW. Thus, given 100 devices, there will be 100 new PSWs (and 100 old PSWs) pointing to 100 potentially different interrupt

service routines. Since each interrupt is uniquely identified, there is no need to have an interrupt status register, and hence no interrupt analysis routine is needed. The resultant savings in time and program space is absorbed, however, by the large number of PSW words that must be reserved in memory.

A modified version of this *automatic priority interrupt scheme* can be found in the PDP-11. This computer uses two words, the processor status word and the program counter, to hold all the machine state information (see Fig. 3-1). Upon interrupt, the contents of the PC and the PS are automatically pushed onto the system stack maintained by the SP (register 6). The effect is the same as if

```
MOV     PS,-(SP)        ;PUSH PROCESSOR STATUS
MOV     PC,-(SP)        ;AND PROGRAM COUNTER
```

had been executed.

The new contents of the PC and the PS are loaded from two preassigned consecutive memory locations called an *interrupt vector*. The actual locations are chosen by the device interface designer and are located in low memory addresses. The first word contains the interrupt service routine address (the address of the new program sequence), and the second word contains the new PS, which will determine the machine status and priority level. The contents of these vectors are determined by the programmer and may be set under program control.

After the interrupt service routine has been completed, an RTI (return from interrupt) is performed. The two top words of the stack are automatically "popped" and placed in the PC and PS, respectively, thus resuming the interrupted program. Because the interrupt mechanism utilizes the stack automatically, interrupts may be nested in much the same manner that subroutines are nested. In fact, it is possible to nest any arbitrary mixture of subroutines and interrupts without confusion. By using the RTI and RTS instructions, respectively, the proper returns are automatic

6.5.6. Reader Interrupt Service

An example of an interrupt operation for the PDP-11 can be found in the routine to read a block of characters from the paper tape reader to a buffer as shown in Fig. 6-17. This code is written in a PIC format and includes setting up the interrupt vector (memory location 70) for the paper tape reader. There are two separate routines. The first, beginning at label INIT, initializes the buffer address pointer and word count in the interrupt routines; then calculates the relocation factor from the offset PRSER-X-2 as follows:

If PC_0 is the PC that was assumed for the program when load at 0, and if PC_n is the current real PC, the calculation is

$$PRSER - X - 2 + PC_n = PRSER - PC_0 + PC_n$$
$$= PRSER + (PC_n - PC_0)$$

since $(X + 2) = PC_0$. As a result, the relocation factor, $PC_n - PC_0$, is added to the assembled value of PRSER to produce the relocated value of PRSER.

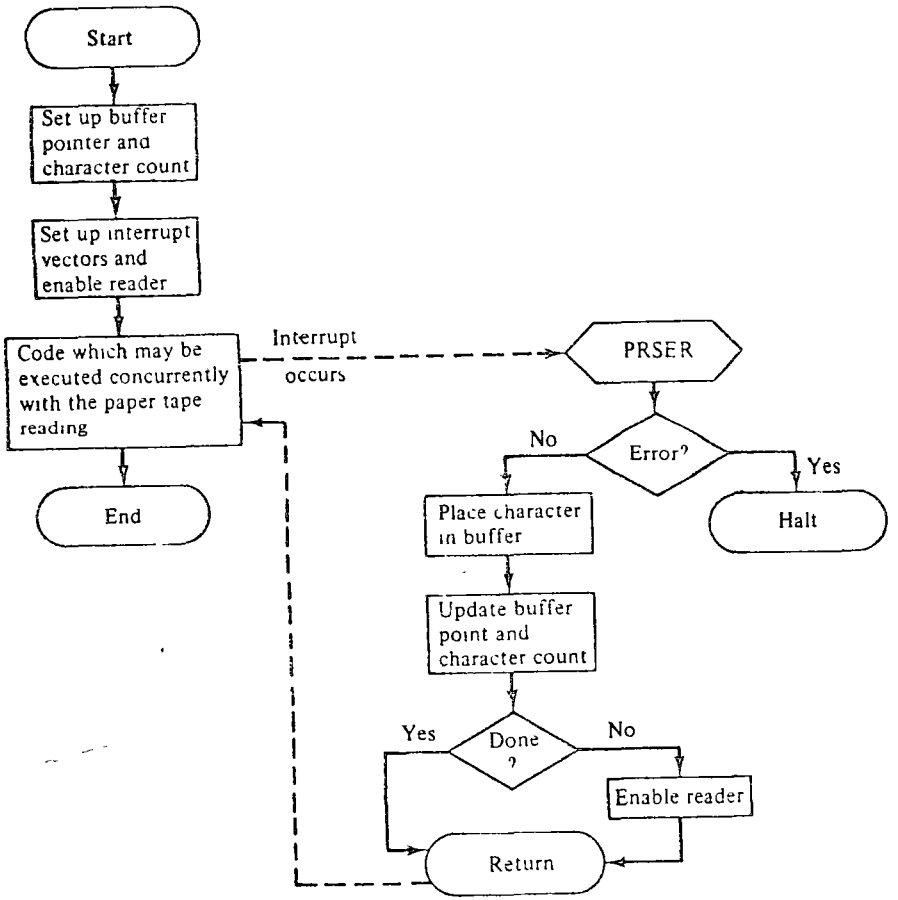


Fig. 6-17

Then it establishes the priority level for the reader and sets it to interrupt after a character has been read. This frees up the CPU so that other code may be executed while the buffer is being filled.

The second routine, the paper tape interrupt service routine, PRSER, is activated each time a character is received. Once activated, the routine stores the character in the buffer, updates the buffer pointer and word count, and resets the interrupt enable bit if more characters are to be read.

logical flow, then, of these two routines looks as shown in Fig. 6-17, and the code is as given in Fig. 6-18.

```
        ;INTERRUPT DRIVEN ROUTINE TO
        ;INPUT CHARACTERS FROM PAPER
        ;TAPE READER

        R0=%0                       ;DEFINE REGISTERS
        SP=%6
        PC=%7
        PRS=177550                  ;DEFINE DEVICE
        PRB=PRS+2                   ;REGISTERS

INIT:   MOV     PC,SP               ;INITIALIZE STACK
        TST     -(SP)               ;TO POINT TO INIT
        MOV     #BUFADR,PTR         ;SET UP BUFFER ADDRESS POINTER
        MOV     #100,CRCNT          ;SET UP CHARACTER COUNT
X       MOV     PC,R0               ;R0=ADDR(X+2)
        ADD     #PRSER-X-2,R0       ;ADD OFFSET
        MOV     R0,@#70             ;SET UP VECTOR ADDRESS
        MOV     #200,@#72           ;STATUS TO PRIORITY 4
        MOV     #101,PRS            ;SET INTR ENB AND RDR ENB

                                    CODE WHICH MAY BE EXECUTED WHILE
                                    BUFFER IS BEING FILLED

BUFADR:  = +100                     ;100 CHARACTER BUFFER
PRSER:  TST     PRS                 ;TEST FOR ERROR
        BMI     ERROR               ;DO ERROR THING
        MOVB    PRB,@PTR            ;STORE CHARACTER IN BUFFER
        INC     PTR                 ;BUMP POINTER
        DEC     CRCNT               ;DECREMENT CHARACTER COUNT
        BEQ     DONE                ;BRANCH WHEN INPUT DONE
        INC     PRS                 ;START UP READER
DONE    RTI                         ;RETURN
ERROR   HALT                        ;STOP ON ERROR
PTR     0                           ;BUFFER POINTER
CRCNT   0                           ;CHARACTER COUNTER
        END     INIT                ;END OF ASSEMBLY
```

Fig. 6-18

6.5.7. Priority Levels and Masking Interrupts

Within a group, any number of devices may cause an interrupt at a given priority level. Since it is conceivable that at any given time a programmer may wish to ignore some of the devices, the hardware usually includes a mechanism to mask interrupts from selected devices.

The PDP-11 uses a simple mechanism to mask device interrupts, by allowing the programmer to clear the interrupt enable bit in the device control and status register. Actually, the interrupt bit is automatically cleared each time the system is initialized (by pushing the START key or executing the RESET instruction) and must be set under program control. However, once set, the bit stays set until cleared.

Another approach to this problem is to use a *mask register*. This register contains a bit for each interruptable group (or bit in the ISR if one exists), and the hardware uses the mask bits by ANDing them to the interrupt bits. Only if the result is a 1 is the interrupt allowed to occur. In effect, the mask *disarms* certain specified interrupts. Still, the mask only disarms interrupts within a group and does not set up any priorities between interrupts or groups.

The need for priorities is demonstrated by the following example program. This program utilizes the teleprinter and the 60-cycle clock on the PDP-11. After being loaded and started, the program types out

```
WHAT TIME IS IT?
```

to which the user responds with a four-digit number. Thereafter, the program, utilizing clock interrupts every 1/60 of a second, keeps track of the time, responding with

```
AT THE BELL THE TIME WILL BE   XX XX XX
```

every time a keyboard character is struck.

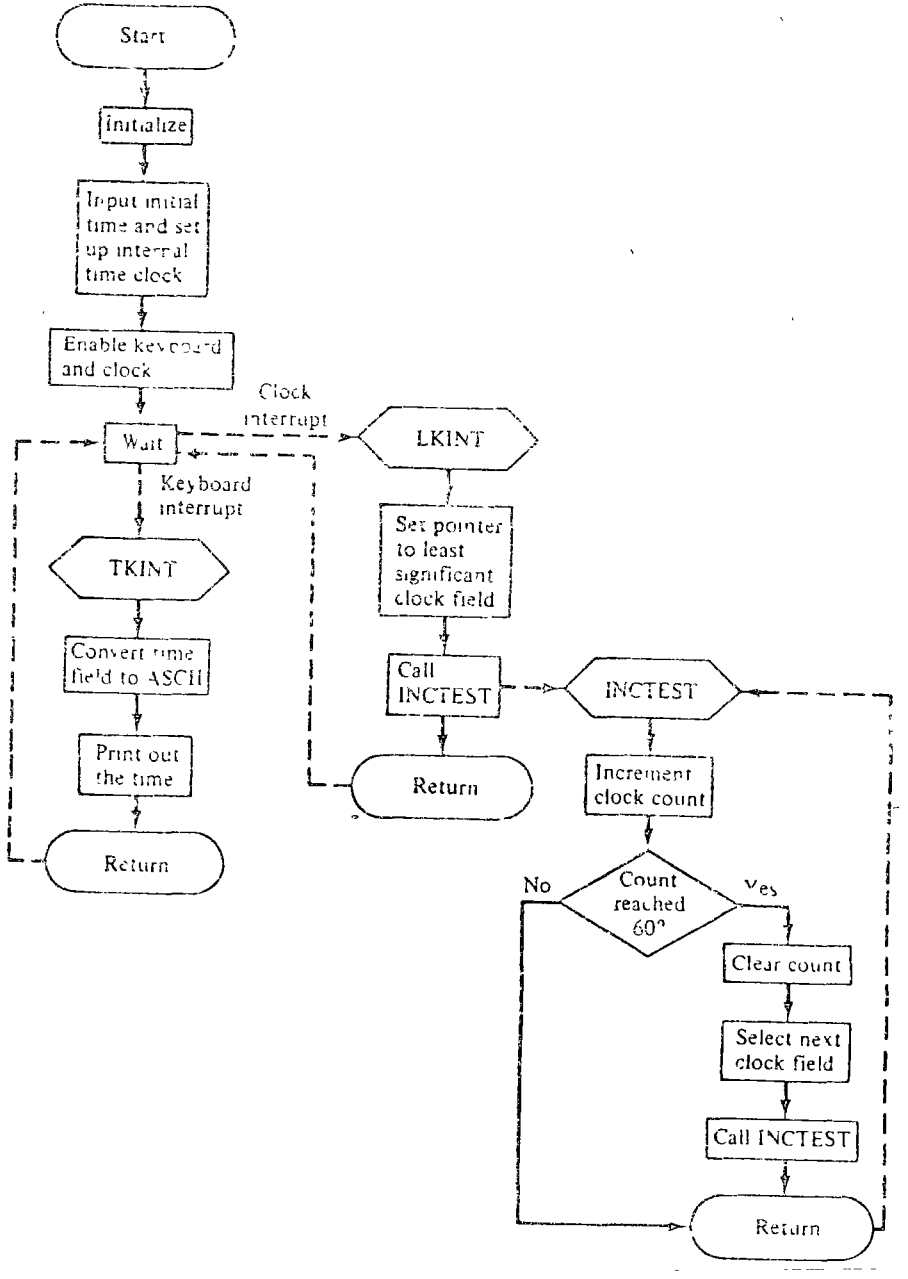
All three devices (keyboard, printer, and clock) are interrupt-driven. Thus while the printer is interrupting to fill its buffer, the clock can be interrupting to tick off another 1/60 of a second. However, the priority of the clock must be greater than that of the printer if ticks are not to be lost. (That this loss of ticks can actually occur can be demonstrated by changing the priority levels set near label NOFIX in the program.)

Priority level is, however, not simply a function of the device. Although each group or device has its own priority level, the running program also has a level. Thus each interrupt that causes an interrupt to occur can raise, lower, or maintain the current priority level of the running program. As a result, if an interrupt occurs at level 7, say, and the interrupt routine does not set the new level at 7, it is quite possible for the higher-level interrupt service routine to be constantly interrupted by lower-level devices. With this in mind, it can be seen that the processor priority level as maintained in the PS word acts as an I/O device interrupt mask.

The various vector addresses and priority levels for the teletype, high-speed reader/punch, and clock on the PDP-11 are as follows:

Device	Vector Address	Priority
Teletype keyboard	60	4
Teletype printer	64	4
High-speed reader	70	4
High-speed punch	74	4
Line clock	100	6

The example in Fig. 6-19 demonstrates the use of priority levels. In addition, it uses recursive programming, and it freely intermixes subroutine stacking with interrupt processing. It is therefore far from a trivial example of the power and flexibility of an interrupt facility on a small computer. A logical flow is included because it provides an overall picture of what the program does. Of particular interest is the clock interrupt routine, which calls on the clock increment subroutine in a recursive fashion.

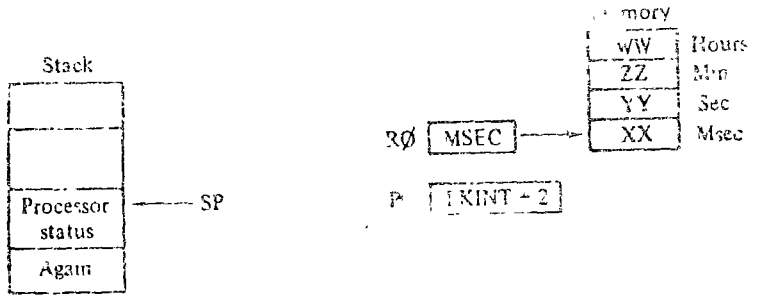


```
        R0=%0
        R1=%1
        R2=%2
        R3=%3
        R4=%4
        SP=%6
        PC=%7
        ,
        TKCSR=177560
        TKDBR=177562
        TPCSR=177564
        TPDBR=177566
        LKCSR=177546
        ,
         =1000
BEGIN   MOV     #500,SP          INITIALIZE SP
        MOV     #M1,R2          ,ASK FOR
        MOV     #EM1,R3         ,THE TIME
        JSR     PC,DPR1         ,PRINT IT OUT
        MOV     #TIME,R2        ,ADDR OF TIME FIELD
        MOV     #4,R1           ,COUNT FOUR CHRS
        MOV     R2,R3           ,PRINT ADDRESSES
NEXTD   INC     TKCSR           ,READ THE TIME
        TSTB    TKCSR           ,TEST FOR A CHARACTER
        BPL      -4             ,WAIT
        MOVB    TKDBR,(R2)      ,PUT IT IN TIME FIELD
        JSR     PC,DPR1         ,PRINT IT
        TSTB    (R3)+           ,NEXT BYTE
        DEC     R1              ,DECREASE COUNT
        BNE     NEXTD           ,KEEP GOING
        MOVB    TIME+1,HOURS    ,LSD OF HOURS
        BIC     #177700,HOURS   ,CLEAR PARITY
        SUB     #60,HOURS       ,CONVERT TO OCTAL
        BIC     #177700,TIME    ,CLEAR PARITY
NEXT    CMPB    TIME,#12        ,ANY TENS?
        BLT     AROUND          ,NO
        ADD     #12,HOURS       ,INCREASE VALUE
        SUB     #1,TIME         ,DEC TENS COUNT
        BR      NEXT            ,ANYMORE?
AROUND  MOVB    TIME+3,MIN      ,GET MINUTES LSD
        BIC     #177700,MIN     ,CLEAR PARITY
        SUB     #60,MIN         ,CONVERT TO OCTAL
        MOVB    TIME+2,R0       ,MUST CORRECT TENS
        BIC     #177700,R0      ,REMOVE PARITY
        MOV     #12,R1          ,ADD 10 DEC
        SUB     #12,R0          ,TEST FOR A ONE
        BMI     NOFIX           ,NO TENS
        BEQ     ADD             ,ONE TEN
MORE    ADD     #12 R1          ,TRY AGAIN
        DEC     R0              ,COUNT THE TENS
        BNE     MORE            ,MORE?
ADD     ADD     R1,MIN          ,ADD IN # OF TENS
NOFIX   MOV     #300,@#2        ,LEVEL 4 INTERRUPT
        MOV     #TKINT,@#60     ,FOR THE TTY KBD
        MOV     #340,102        ,LEVEL 7 INTERRUPT
        MOV     #LKINT,100      ,FOR THE CLOCK
        MOV     #101,TKCSR      ,INIT KBD
        MOV     #100,LKCSR      ,AND CLOCK
AGAIN   WAIT                    ,NOTHING TO DO
```

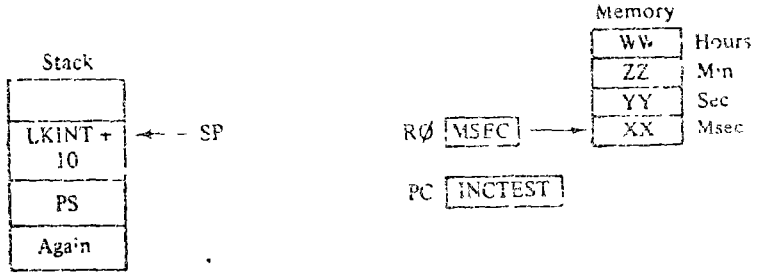


```
LKINT   MOV     #MSEC,R0        ,ADDRESS OF LS FIELD
        JSR     PC,INCTEST      ,RECURSIVE CALL
        RTI                     ,CLOCK UPDATED
INCTEST INC     (R0)            ,ADD ONE
        CMP     (R0),#60        ,REACHED LIMIT?
        BNE     RETURN          ,NO
        CLR     (R0)            ,RESET FIELD
        TST     -(R0)           ,ADDR OF NEXT FIELD
        JSR     PC,INCTEST      ,CALL ME AGAIN
RETURN  RTS     PC              ,RETURN HOME
TKINT   MOV     #M2,R2          ,PRINT OUT
        MOV     #EM2,R3         ;THE TIME
        JSR     PC,DPR1         ,MESSAGE
        MOV     #3,R3           ,NUMBER OF FIELDS
        MOV     #OUT,R2         ,OUTPUT AREA
        MOV     #HOURS,R4       ,FIRST FIELD ADDR
CNVRT   CLR     R0              ,INITIALIZE
        MOV     (R4)+,R1        ,FIRST VALUE
LOOP    CMP     R1,#12          ,ANY TENS?
        BLT     ADDUP            NO
        INC     R0              ,YES, COUNT
        SUB     #12,R1          ,DEC TENS
        BR      LOOP            ,DO IT AGAIN
ADDUP   ADD     #60,R0          ,TENS IN ASCII
        MOVB    R0,(R2)+        ,STORE IT
        ADD     #60,R1          ,UNITS IN ASCII
        MOVB    R1,(R2)+        ,STORE IT
        TSTB    (R2)+           ,SKIP
        DEC     R3              ,LOOP COUNT
        BNE     CNVRT           ,DO IT THREE TIMES
        MOV     #OUT,R2         ,READY
        MOV     #BELL,R3        ,TO PRINT
        MOV     #101,TKCSR      ,RDR ENB
        TST     TKDBR           ,CLEAR DONE BIT
        JSR     PC,DPR1         ,YES
        RTI                     ,DONE AT LAST
DPR1    CMP     R2,R3           ,ARE WE DONE?
        BGT     DPR2            ,YES
        TSTB    TPCSR           ,READY TO PRINT?
        BPL      -4             ,NO
        MOVB    (R2)+,TPDBR     ,PUT IN BUFFER
        BR      DPR1            ,NEXT CHARACTER
DPR2    RTS     PC              ,RETURN
M1       BYTE   15,12           ,CR AND LF
         ASCII  /WHAT TIME IS IT?/
EM1     BYTE    '
        EVEN
TIME    BYTE    0,0 0,0         ,FOUR CHAR TIME
OUT     BYTE    0,0,' ,0,0,' ,0,0
BELL    BYTE    7               ,STRIKE THE GONG
        EVEN
HOURS   WORD    0
MIN     WORD    0
SEC     WORD    0
MSEC    WORD    0
M2      BYTE    15,12
        ASCII   /AT THE BELL THE TIME WILL BE
EM2     BYTE    '
        END     BEGIN
```

To understand how the increment clock routine works, it is necessary to examine the stack after each call. Just after the line labeled LKINT is executed, the symbolic contents of the stack (and R0) will be

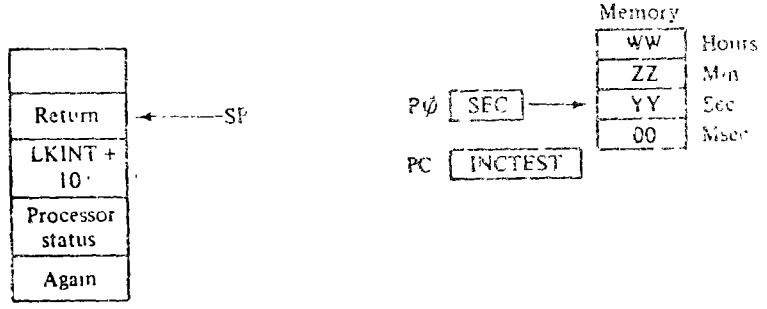


Now, when the subroutine INCTEST is called, the picture changes to



If the value in location MSEC is less than 59, it will be incremented by 1, a return made from the subroutine (e.g., the stack is popped) followed by a return from the interrupt routine (popping off the PC and PS from the stack), and the program will once again wait for an interrupt.

Suppose, however, that the value in location MSEC equals 59. In this case the increment subroutine will now set MSEC = 0, advance the pointer in R0 to point to the location SEC, and call the subroutines INCTEST recursively. At this point the picture looks as follows:



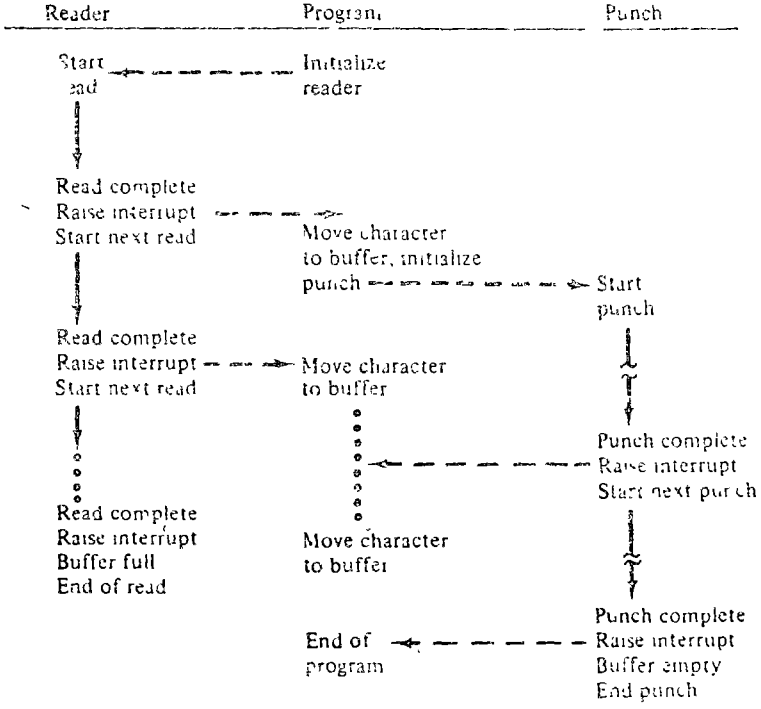
Again the value at the location pointed to by R0 is checked to see if it is 59. If it is, it is zeroed, R0 is advanced to point to the next most significant field, and the routine INCTEST is once again called recursively.

Alternatively, if the value pointed to by R0 is less than 59, we have the case discussed previously, where the value pointed to by R0 is incremented and a subroutine return is made. Since the return causes the instruction following the subroutine call to be executed and this instruction is a return from subroutine, the stack is popped twice (or more) until the return from interrupt occurs, at which point the program waits for a new interrupt to occur. The actual unwinding of the recursive calls becomes quite simple, since it only serves to restore the stack.

6.6. BUFFERING AND BLOCKING

Although basic I/O units (teleprinter and paper tape reader/punch) operate on characters, characters per se are not exactly what the programmer wishes to input or output. Rather, I/O programming is concerned with strings of characters such as 10-digit numbers, people's names, octal representations of memory words, or lines of assembly code. In other words, the I/O consists of *blocks* of data that have a logical connection.

Since the I/O device does not perform I/O in a block fashion, it is generally the programmer's responsibility to *block* characters on input and *deblock* them on output. *Buffers*, which are contiguous blocks of memory, act as repositories for the blocks of data and allow the program to stream



data to or from an I/O unit at a rate consistent with the I/O device. Buffers are of particular use between two I/O devices with dissimilar I/O rates. For example, if the high-speed paper tape reader is six times faster than the high-speed paper tape punch, a buffer can be used to allow simultaneous input and output, provided, of course, that a full buffer terminates input and an empty buffer terminates output.

The overlap of input and output on the high-speed reader punch is shown in Fig. 6-20. Each device is running in interrupt mode at its maximum rate. Because the reader has a higher I/O transfer rate, it will finish first, followed by the punch routine emptying the buffer, and then by termination of the program.

6.6.1. Overlap of Computation and I/O Processing

Another use of buffers and blocking can be found in the overlap of computation and I/O processing. For this situation *double buffering* is used, so that while one buffer is being filled (or emptied), a second buffer is available to the running program. Actually, the number of buffers may be more than two, depending upon their rate of utilization by the program and the I/O device and upon the size of the buffers. In a *balanced system*, the buffer size and number is adjusted so that computation and I/O processing are 100 percent overlapped. When the computation is less than 100 percent, the system is said to be *I/O bound*. Correspondingly, when the system is *computation bound*, the I/O utilization is less than 100 percent.

Whether a system is one way or another depends on many things, including the computer configuration, economic considerations, system load, and so on. These considerations fall into the province of the systems programmer, who is concerned with operating systems design and performance.

6.7. INPUT/OUTPUT PROGRAMMING SYSTEMS

In order to facilitate effective utilization of I/O devices and to assist the user in writing his I/O code, most computer manufacturers provide their computer users with an *input/output programming system (IOPS)*. Such a system

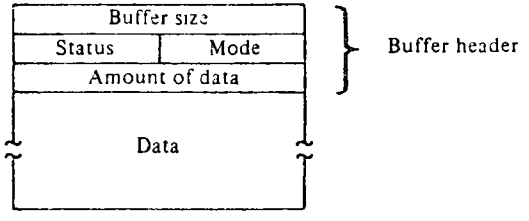
1. Frees the programmer from the details of dealing directly with I/O devices.
2. Provides better I/O organization and service.
3. Facilitates I/O programming through simple assembly language macros.†
4. Provides conformity across various operating enviroments.

In addition, the programmer can use an I/O programming system to allow

1. Asynchronous I/O service.
2. Concurrent (overlapped) I/O operation
3. Device-independent programming.
4. Blocking and buffering.

I/O programming system macros fall into three categories. The fir category includes the initializing commands. These commands initialize bot the IOPS tables and the device interrupts, and assign unit numbers to syste devices. In many respects these commands are similar to the assignment an open commands in FORTRAN, which relate a unit number to a device an open a file on the specified device.

The second category of commands includes all the actual data transfer such as READ and WRITE. Finally, the third category includes the contro commands, such as EOF, WAIT, and RESTART. The latter two categorie include macro statements, which make reference to I/O buffers. I/O buffe are of the form



The size of the buffer is of interest to IOPS in that it cannot allow da to overflow the buffer. On the other hand, the amount of data is of concer to the programmer, since it tells him how much of the buffer is filled. Fo both IOPS and the programmer, the mode and status bits are necessary i order to describe the buffer contents (e.g., ASCII, binary, formatted, u formatted, packed, unpacked) and to indicate the status of the I/O operatio (e.g., complete, EOF, checksum error, truncation upon buffer overflow

6.7.1. Example

An example of the use of IOPS to provide a simple input-process-outpu sequence would include

1. The definition of a buffer.
2. The initialization of the system.
3. A read into the buffer.

4. A wait for the read to be completed.
5. The processing of the data read.
6. A write from the buffer.

This sequence might be coded using IOX (*input/output executive*) for the PDP-11:

```
START   RESET                       ;INITIALIZE IOX
        READ    KBD,BUF,ASCII       ;READ INTO BUFFER
IDLE    WAITR   IDLE                ;WAIT TIL READ DONE


        PROCESS BUFFER


        WRITE   PRT,BUF,ASCII       ;WRITE OUT BUFFER
        END     START
```

The purpose of the macro instruction is to serve as a linkage to the I/O programming system. Thus each macro instruction results in a service call to IOX with the macro arguments being passed to IOX as a means of specifying what is to be done.

6.7.2. IOPS Linkage Problem

In a large computer system the expansion of a macro into a service routine linkage requires both a macro assembler and a linking loader. Since neither of these exist as part of the basic software supplied with a small computer system (although they may be available as part of advanced software systems), it becomes the programmer's responsibility to expand the macros into assembly language statements and to link up the program with IOPS.

The technique most often used to link programs with IOPS is through interrupt-producing machine instructions called *service calls (SVC)* or *I/O traps (IOT)*. These program-initiated interrupts are handled like I/O interrupts, and result in the replacement of the current PSW by a new PSW pointing to the I/O programming system. Such SVCs or IOTs have all the advantages of subroutine calls, including arguments passing (the old PC in the PSW points to the first word in the argument list); they also facilitate the functioning of the I/O programming system in that

1. It need not save the processor state (except for the register it uses).
2. It can operate at any priority level it wishes to.
3. It provides a direct linkage between the user program and IOPS through a fixed memory location.

The last point is clearly the most important. The trap instruction effectively does away with the need for a linking loader, since all IOPS calls will be through IOTs, which do not require direct linkage to IOPS. Instead, IOPS need only preload its trap vector so that all IOTs will cause a transfer of control to IOPS, at which point the reason for the IOPS call can be determined. In this regard an I/O trap is analogous to the single level interrupt system already discussed.

6.7.3. Interrupts and Traps

It is worthwhile to digress for a moment and point out that interrupts and traps are not associated only with I/O instructions. Indeed, interrupts may be used to

1. Indicate program faults, such as addressing errors, illegal instruction errors, and abnormal arithmetic results.
2. Handle machine errors, including memory parity checks and automatically detected hardware malfunctions.
3. Flag external conditions, such as power failure and console key interruptions.

Additionally, since these interrupts cause a change in the current PSW, it is possible to utilize an interrupt-generating condition to change the *protection state* of the system.

For example, if I/O instructions are illegal in the *protected* or *user* state, they will cause an interrupt to be raised whenever the computer attempts to execute them. However, should the system be in the *unprotected* or *monitor* state, no interrupt will occur. Consequently, all I/O requests must be handled by an I/O programming system, which is activated by an SVC or IOT instruction that results in a change of state from protected to unprotected mode.

The monitor and user modes permit a structured environment by providing for two distinct states of system operation. Depending upon the state, full or limited memory addressing and instruction execution capabilities are permitted. By making the system state a bit in the PSW, a change of state can occur automatically, thus guaranteeing that all system capabilities may be made available to the interrupt service routine. In Chapter 8, where more advanced operating systems (such as multiprogrammed and time-sharing systems) are discussed, this concept of system state will be discussed more fully.

6.7.4. Programming of a Trap Instruction

Turning back to the use of an I/O programming system, it would be well to examine how such a system might be used on the PDP-11. The earlier

macro program, although quite nice, just does not exist. Instead, the I/O executive IOX requires the same problem to be cast as shown in Fig. 6-21.

```
        RESET=2                 ,ASSIGN IOX COMMAND
        READ=11                 ,CODES
        WAITR=4
        WRITE=12
        KBD=0
        PRT=1
START   IOT                     ;I/O TRAP INSTRUCTION
         WORD   0               ,PERFORM NECESSARY
         BYTE   RESET,0         ,INITILAIZATION

        IOT                     ,TRAP TO IOX
         WORD   BUF             ,SPECIFY BUFFER
         BYTE   READ,KBD        ,AND KEYBOARD READ

IDLE    IOT                     ,TRAP TO IOX
         WORD   IDLE            ,JUMP TO IDLE
         BYTE   WAITR,KBD       ,KBD READ IS FINISHED


        PROCESS BUFFER


        IOT                     ,TRAP TO IOX
         WORD   BUF             ,SPECIFY BUFFER
         BYTE   WRITE,PRT       ;AND PRINTER WRITE

BUF     100                     ,BUFFER SIZE (BYTES)
        0                       ;STATUS/MODE (ASCII)
        0                       ,IOX WILL FILL IN BYTE COUNT
                                ,RESERVE 100 BYTES
         END    START
```

Fig. 6-21

An important point to notice is that since IOX processes buffers, interrupt handling is no longer at the character level but rather at the buffer (filled or empty) level. Since this is consistent with the units of information required (strings of digits, lines of input, etc.), the useability of IOX is clearly demonstrated.

6.7.5. Coroutine Example Utilizing IOX

In Chapter 4 it was mentioned that coroutines were used for I/O processing and represented one of the basic operations to be performed by modern operating systems. The example that follows demonstrates the use of coroutines in a double-buffer I/O scheme which overlaps I/O with computation performing as follows:

Write 01 Read I1 Process I2	} concurrently	Write 02 Read I2 Process I1	} concurrently

The reader should recall that the JSR PC, @(SP)+ always performs a jump to the address specified on top of the stack and replaces that address with the new return address. Thus each time the JSR at B is executed, it jumps to a different location: initially to A and thereafter to the location following the JSR executed prior to the one at B. All other JSR's jump to B+2 (Fig. 6-22). This code, although deceptively short, is a powerful and elegant solution for the programming of double-buffered I/O overlapped with computation. It clearly demonstrates the power and capability of the small computer, on which may be developed time-sharing, real-time, and communications-based systems.

EXERCISES

1. Write a program to type out the message "HELLO?" on the teleprinter.
2. Write a format subroutine for the teleprinter to tab-space the teleprinter carriage. The subroutine is entered with the number of spaces to be tabbed in register R0.
3. Write a program to read columns from the low-speed paper tape reader, punching out each column on the high-speed paper tape punch as three octal digits.
4. Write a subroutine that accepts one to six octal digits from the teleprinter and forms a 16-bit word in R0. As each character is typed, it should be echoed back to the teleprinter. Assume that the line is terminated with a carriage return and that your routine will insert a line feed.
5. Rewrite Exercise 3 to utilize interrupts.
6. Write an interrupt structured program to read 400 characters simultaneously from the high-speed reader, while punching and printing the first 100 characters read. Be careful to terminate the reading while allowing the slower printing and punching devices to complete.
7. Devise a scheme for measuring execution time used by a program. This scheme should be accurate to within 16.6 milliseconds.
8. Code Exercise 3 utilizing IOX.
9. Can Exercise 6 be coded using IOX?
10. Code the coroutine double-buffer example on page 200 so that it can duplicate a paper tape from the low-speed reader to the high-speed punch.

```
        SP=%6
        PC=%7
BEGIN   (DO I/O RESETS, INITS, ETC )

        IOT                     ;READ INTO I1 TO START PROCESS
         WORD   I1
         BYTE   READ.INSLOT
        MOV     @#,-(SP)        ;INITIALIZE STACK FOR FIRST JSR
B       JSR     PC,@(SP)+       ,DO I/O FOR O1 AND I1 OR O2 AND I2

        PERFORM PROCESSING

        BR      B               ,MORE I/O

,END OF MAIN LOOP

,I/O CO-ROUTINES FOLLOW

A       IOT                     ,READ INTO I2
         WORD   I2
         BYTE   READ,INSLOT

        SET PARAMETERS TO PROCESS I1 AND O1

        JSR     PC,@(SP)+       ,RETURN TO PROCESS AT B+2
        IOT                     ;WRITE FROM O1
         WORD   O1
         BYTE   WRITE,OUTSLT
        IOT                     ,READ INTO I1
         WORD   I1
         BYTE   READ.INSLOT

        SET PARAMETERS TO PROCESS I2 AND O2

        JSR     PC,@(SP)+       ,RETURN TO PROCESS AT B+2
        IOT                     ,WRITE FROM O2
         WORD   O2
         BYTE   WRITE,OUTSLT
        BR      A               ,READ INTO I2

         END    BEGIN
```

Fig. 6-22

REFERENCES

I/O programming is very personal in the sense that each computer type has its own I/O instructions and hence I/O idiosyncracies. Books by Flores (1969), Hellerman (1967), and Foster (1970) discuss I/O from the conceptual level, making it more universal in flavor. Others, like this book, treat I/O as it is embodied in a particular machine. For the PDP-11 the best source is the *Peripherals and Interfacing Handbook*, which covers not only I/O devices but also UNIBUS extensions, communication interfaces, and data and control options.

7 SYSTEM SOFTWARE

A comprehensive package of system software accompanies each computer in use today, from the small minicomputer to the large number cruncher. These packages include programs and routines plus associated documentation which allow the programmer to write, edit, assemble, compile, debug, and run his programs, making the full data-processing capability of the computer immediately available.

System software represents the on-going process and continual efforts of system programmers to make the utilization of computers easier, more comprehensible, and less time-consuming than was possible before. Most systems are modular and open-ended, permitting the user to construct specified systems tailored to his particular environment. As such, they act as the buffers or interfaces between the user's needs and the hardware's capability.

We have introduced you previously to three software systems: the assembler, the I/O programming system, and the memory dump routine. Now your attention is directed to those other software systems that assist in the creation and execution of programs—the editor, the macro assembler, and the loader. In addition, since no nontrivial program or system is ever fully debugged or tested, it is worthwhile to conclude our investigation of system software with an examination of testing and debugging techniques.

7.1. EDITOR

The text editor is a powerful context-editing program used to create and modify symbolic source programs and other text material. By means of commands issued from the teleprinter, the editor can be used to create and delete characters, lines, or groups of lines which it maintains in its internal buffer.

Because the editor is on-line in most systems, response to commands is immediate and dynamic.

A good editor is both productive and cost-effective. In use it turns the teleprinter into a very sophisticated typewriter that assists the programmer in the normal "cut and paste" operation of putting a program together. As a result, the editor must not only allow for the insertion and deletion of characters and lines, but it must also be capable of locating symbols, making corrections, and reading or writing blocks of data.

Typical editor commands include the following:

1. INPUT: to enter a new string of characters.
2. DELETE: to delete a string of characters.
3. CHANGE: to replace one string of characters with another.
4. LOCATE: to find the first or *n*th occurrence of a character string.
5. PRINT: to print a string of characters.
6. VERIFY: to print out a string after it has been changed, or located.
7. READ: to fill the editor's internal buffer by reading a block of text from some peripheral device.
8. WRITE: to empty the internal buffer onto a peripheral device.

In addition, there are commands that have to do with the character or line pointer.

Associated with the internal buffer of the editor is a pointer that refers to the line or character in the buffer considered to be the current line or character. The current line or character is defined as the line or character that is being created or edited by the user.

Some editors operate only on lines, some only on characters, others operate on both. If the editor recognizes entire lines it does so by defining a line to end with an especially significant character, such as a carriage return. In this way the editor may assume that each line begins with the character after the terminating carriage return in the last line and ends with the terminating carriage return for the current line.

Various editor requests are provided for moving the current location pointer. These requests include:

1. BEGIN: to position the pointer at the beginning of the buffer.
2. END: to position the pointer at the end of the buffer.
3. NEXT: to position the pointer at the beginning of the next line.
4. LAST: to position the pointer at the beginning of the previous line.

5. FORW: to move the pointer forward one character position.

6. BACK: to move the pointer backward one character position.

In addition, editor commands, such as LOCATE, INPUT, DELETE, and so on, will cause the current location pointer to be repositioned.

There are two response modes in which the editor environment may operate. These are called "normal" and "brief" modes. The normal mode automatically types out each line that has been changed or searched for as the result of an editor request. The brief mode does not respond by typing the edited lines and thus requires the user to issue a verify command (for one line) or a print request (for several lines) in order to see the results of the last command(s).

The editor environment includes two modes of operation: the input and the command modes. The *input mode* specifies that all characters entered are to be treated as input until a special character is recognized as a request for a mode change. The *command mode* implies that the character strings entered are to be treated as requests to the editor.

The sophistication of the editor depends greatly on its operating environment. Large computer systems allow for maximum editor flexibility, including full or abbreviated commands, concatenation of command strings (multiple statements), file manipulating requests, and sophisticated text editing. Small computers generally have very terse, one-letter commands, limited internal buffers, and rigid command formats. Nonetheless, even small computer editors allow sufficient flexibility for creating and modifying source programs.

7.1.1. Example of the Use of a Small Computer Editor

The editor for the PDP-11 is typical of the small computer editor. Requests are entered while the editor is in command mode (each line begins with the editor typing out an *), and they include

1. B: equivalent to BEGIN.

2. ±nA: equivalent to NEXT or LAST depending on the sign. n specifies the number of lines.

3. ±nJ: like A but for characters (e.g., equivalent to FORW and BACK).

4. I: equivalent to INPUT.

5. ±nC: to replace n characters before (−) or after (+) the current pointer position (e.g., equivalent to CHANGE).

6. ±nL: equivalent to PRINT but ±n lines from the current pointer

This subset of editor commands may be used to write the trivial prog given in Fig. 7-1. In the example, the editor is assumed to be running and nonprinting characters are not shown (e.g., carriage return, tab, line fe Additionally, the right-hand comments have been added for the sake readability.

```
*I                                      USER PLACES EDITOR
                                    IN INPUT MODE AND TYPES IN
                                    LINES OF INPUT
        R0=%0
        R1=%1
        MOV     #1,R0
        CLR     R1
        CMP     R0,R1
         END    START

                                            A LINE-FEED TERMINATES
                                    INPUT MODE
*B                                  POSITION POINTER AT BEGINNING
*2A                                 ADVANCE 2 LINES
*1L                                 PRINT THE CURRENT LINE
        MOV     #1,R0
*I                                  BACK TO INPUT MODE

START                               AND ADD A LABEL
*0A                                         TO REPOSITION THE POI-
                                    TO THE BEGINNING OF THE LINE

*1L                                 THE LINE IS LISTED
START   MOV     #1,R0

*15J
*1C                                 THE CHARACTER 0 IS TO
                                    BE CHANGED TO A 0
0

*0A                                 THE POINTER IS REPOSITIONED
*1L                                 AND THE LINE IS PRINTED OUT

START   MOV     #1,R0
```

Fig. 7-1

Although far from exhaustive, this example demonstrates how a si computer editor might work.

7.2. MACRO ASSEMBLERS

The reader has already read how a basic symbolic assembler makes chine language programming easier, faster, and more efficient. In addit

the reader has been presented with the need for and advantage of pseudo-operation instructions for directing the actions of the assembler. Now we shall discuss the advanced features of a macro instruction generator, which is a part of an expanded or *macro assembler*. Note that the keyword is "expanded," since the macro assembler contains all the features normally found in a symbolic assembler plus those necessary to handle macro instruction generation. Thus MACRO-11, the macro assembler for the PDP-11, is a superset of PAL-11, the symbolic assembler, and users of MACRO-11 may write programs that are identical to the programs that they would write for PAL-11.

One of the features of a macro-instruction generator is that it permits easy handling of recursive instruction sequences utilizing the simple technique of parameterization. The generator allows the programmer to create new language elements in order to be able to adapt the assembler to his specific programming applications. In addition, macros may be called inside macros, nested to multiple levels, and redefined within the program.

At this point it might be well to define just what a macro is rather than only what it can do. Very specifically, a macro is an "open routine" which is defined in a formal sequence of coded instructions and, when called or evoked, results in the replacement of the macro call by the actual body of code that it represents. The use of a macro statement does not result in saving memory locations but rather in saving programmer time.

For example, when a program is being written, it often happens that certain coding sequences are repeated several times, with only the arguments changed. It would be convenient if the entire repeated sequence could be generated by a single statement. To accomplish this it is first necessary to define the coding sequence with dummy parameters as a macro instruction, referring to the macro name along with a list of real arguments that will replace the dummy parameters and generate the desired sequence.

Macros must be defined before they may be used. The way to define a macro is to bound the sequence of symbolic instructions with the pseudo-ops .MACRO and .ENDM. For example,

```
.MACRO  MAC  <-- Macro name
LINE1
LINE2        <-- Macro body
LINE3
.ENDM
```

With each macro call (macro order), the macro body is substituted in place of the macro name:

```
ADD     A,B          ADD     A,B
MOV     B,A          MOV     B,A
MAC          ===>    LINE1
MOV     C,B          LINE2
                     LINE3
                     MOV     C,B
```

This replacement process occurs essentially before assembly and can be conceived of as a character-string substitution.

Since the programmer may wish to use the same macro but on different data, macro calls include argument transmission. Thus, if a programmer desires to define a macro instruction "add byte" (ADDB), the following macro definition would suffice:

```
        .MACRO  ADDB X,Y
        MOV     R0,TEMP1        ;SAVE R0
        MOV     R1,TEMP2        ;SAVE R1
        MOVB    X,R0            ;PUTS FIRST BYTE IN R0
        MOVB    Y,R1            ;PUTS SECOND BYTE IN R1
        ADD     R0,R1           ;FORMS RESULT
        MOVB    R1,Y            ;PLACE RESULT IN Y
        MOV     TEMP1,R0        ;RESTORE R0
        MOV     TEMP2,R1        ;AND R1
        BR      TEMP2+2         ;BRANCH AROUND
TEMP1   .WORD   0               ;TEMP LOCATIONS
TEMP2   .WORD   0
        .ENDM
```

7.2.1. Location and Created Symbols

Although it may not have been necessary, the macro body of the preceding example preserved the contents of registers R0 and R1. In doing so, the macro definition developed a serious problem. Each time the macro is called, the symbols TEMP1 and TEMP2 will be redefined, resulting in an assembly error message.

There are, fortunately, two ways out of this dilemma:

1. Parameterize the temporary locations, leaving their definition up to the programmer; for example,

```
.MACRO  ADDB X Y TEMP1 TEMP2
MOV     R0 TEMP1
```

2. Allow the programmer to inform the assembler that certain symbols are known only to the macro and should be replaced by the macro assembler with a *created* symbol, which will be unique for each call of the macro:

```
        .MACRO  ROPE A,?B
        MOV     #6,R0           ;SHIFT COUNT
B       ROR     A               ;ROTATE
        DEC     R0              ;DECREMENT COUNT
        BNE     B               ;LOOP IF NOT DONE
        .ENDM
```

which generates the following code when called:

```
                        E4$   MOV   #E,R0
                              PUF   SUM
RORE    SUM                   DEC   R0
                              BNE   E4$

                        E5$   MOV   #5,R0
                              ROR   VALUE
RORE    VALUE                 DEC   R0
                              BNE   E5$
```

Created symbols are always local symbols between 64$ and 127$. The ocal symbols are created by the macro assembler in numerical order and are generated only when there is no real argument being substituted in place of the dummy argument in the macro definition. If a real argument is specified n the macro call, the generation of a local symbol is inhibited and normal replacement is performed.

7.2 2. Nesting of Macros

Macros may be nested, that is, macros may be defined within other macros. For ease of discussion, levels are assigned to nested macros The outermost macros (those defined directly) are called *first-level macros* Macros defined within first-level macros are called *second-level macros*, and so on. For example,

```
 MACRO  LEVEL1,A,B
ADD     A,B
 MACRO  LEVEL2,C,D
SUB     C,D
 MACRO  LEVEL3 E,F
ADD     E,F
ADD     F,F
 ENDM
CLR     C
 ENDM
CLR     A
 ENDM
```

Level 3 Level 2 Level 1

At the beginning of the macro processing only first-level macros are defined and may be called in the normal manner. Second- and higher-level macros will not yet be defined However, when a first-level macro is called, all its second-level macros become defined. Thereafter, the level of definition is irrelevant and macros at either level may be called in the normal manner. Of course, higher-level macros will not be defined until the lower-level macros containing them have been called.

Using the last example, the following would occur:

Call		Expansion		Comments
LEVEL1	X,Y	ADD	X,Y	Causes LEVEL2
		CLR	X	to be defined.
LEVEL2	I,J	SUB	I,J	Causes LEVEL3
		CLR	I	to be defined
LEVEL3	Y,I	ADD	Y,I	
		ADD	I,I	

If a call to LEVEL3 were made before LEVEL2 defined it, an error would result, since the code expansion would be undefined.

7.2.3. Macro Calls Within Macro Definitions

The body of a macro definition may contain calls for other macros which have not yet been defined However, the embedded calls must be defined before a call is issued to the macro which contains the embedded call

As an example, we consider the macro called SWITCH, which transfers the contents of buffer A to buffer B and vice versa:

```
 MACRO  SWITCH,A,B,TEMP,N
COPY    A,TEMP,N
COPY    B,A,N
 ENDM

 MACRO  COPY,FROM,TO,COUNT,?L
MOV     COUNT,R0
MOV     FROM,TO
DEC     R0
BNE     L
 ENDM
```

7.2.4 Recursive Calls

Although it is legal for a macro definition to contain an embedded call to itself, care must be taken to ensure that the recursive macro expansion will eventually terminate Somehow the assembler must be told that a condition has been detected and that the recursive definition may now stop. The technique used to accomplish this is the conditional assembly statement, although such statements may be used for things other than recursive macro definitions.

7 2 4 1 Conditional Assembly

Conditional assembly directives are most often used to assemble certain parts of a source program on an optional basis. The instruction is of the form

```
IF    cond argument(s)
```

where cond represents a conditional that

1. Tests the value of an argument expression, or
2. Tests the assembly environment; or
3. Determines the attributes of a single symbol or address expression; or
4. Tests the value of character strings.

If the condition is satisfied, that part of the source program starting with the statement immediately following the conditional statement, and including the statements up to the .ENDC (end conditional) assembly directive, are assembled. However, if the condition is not satisfied, the code is not assembled.

Conditional statements may be nested. For each .IF statement there must be a termination .ENDC statement. If the outermost .IF is not satisfied, the entire group is not assembled. If the first IF is satisfied, the following code is assembled. However, if an inner .IF is encountered, its condition is tested, and the code given in Table 7-1 is assembled only if the second .IF is satisfied. Logically, nested .IF statements are like AND circuits. If the first, second, and third are satisfied, the code that follows the third nested .IF statement is assembled.

Table 7-1 Conditional assembly directives

Type	Pseudo-op	Condition
Comparand	IF EQ	argument = 0
	IF NE	argument ≠ 0
	IF GT	argument > 0
	.IF GE	argument ≥ 0
	.IF LT	argument < 0
	.IF LE	argument ≤ 0
Environment	.IF B	Is macro-type argument† blank (i.e, missing)?
	.IF NB	Is macro type argument† not blank (i.e, present)?
Attribute	.IF DF	Is argument symbol defined?
	IF NDF	Is argument symbol undefined?
Character String	.IF IDN	Are two macro-type arguments† identical?
	IF DIF	Are two macro-type arguments† different?

†A macro-type argument is one enclosed in angle brackets (e.g., <A,B C>). Such arguments allow expressions to be treated as single terms.

1. The code generator should put out a BR instruction if the relative distance for the branch is 255 bytes or less. Otherwise, a JMP is generated. The conditional code is

```
BRJUMP  MACRO   LOOP
        IF      DF LOOP
        IF      LT .-255+ -LOOP
BR      LOOP
        ENDC
JMP     LOOP
        ENDC
        ENDM
```

(The .IF DF is necessary since LOOP may be a forward-referenced label.)

2. Code may be saved when using the previously defined ADDB macro when two or more such macros are used in the same program, since TEMP1 and TEMP2 need only be defined once.

```
        MACRO   ADDB,A,B
        MOV     R0,TEMP1
        MOV     R1,TEMP2
        MOVB    B,R0
        MOVB    A,R1
        ADD     R0,R1
        MOVB    R1,B
        MOV     TEMP1,R0
        MOV     TEMP2,R1
        IF      NDF,SW
SW=1
        BR      .+6
TEMP1   WORD    0
TEMP2   WORD    0
        ENDC
        ENDM
```

3. The conditional assembly code may be used to terminate macro recursion.

```
A=0
SUM     MACRO   X,Y
        ADD     Y,Y
A=A+1
        IF      NE A-1
        SUM     X,Y
        ENDC
        ENDM
```

7.2.5. Repeat Blocks, Concatenation, and Numeric Arguments

Occasionally it is useful to duplicate a block of code a number of times in line with other source codes. This is performed by creating a repeat block of the form

```
        REPT    EXPR

        ENDR
```

where expr is any legal expression controlling the number of times the block of code is assembled. For example, to generate a table of ASCII characters, the .REPT could be used as follows:

```
A= A
 REPT    26
 BYTE    A
A=A+1
 ENDR
```

The repeat pseudo-op can also be usefully combined with two other macro features. The first is concatenation. This feature allows the apostrophe or single quote (') character to operate as a legal separating character such that when the ' precedes and/or follows a dummy argument, the ' is removed and substitution of the real argument occurs at that point.

The second feature is the capability of passing a symbolic argument as a numeric string. Such an argument is preceded by the unary operator backslash (\) and is treated as a number. Combining these features, we get the following interesting example:

```
        B=0
         MACRO   INC A,B
        CNT     A,\B
        B=B+1
         ENDM

         MACRO   CNT A,B
A'B     ASCII   /B/
         ENDM
```

This macro pair, when called by

```
 REPT    5
INC     \ B
 ENDR
```

results in the following macro expansion.

```
A0      ASCII   /0/
A1      ASCII   /1/
A2      ASCII   /2/
A3      ASCII   /3/
A4      ASCII   /4/
```

The two macros are necessary because the dummy value of B cannot be updated in the CNT macro. This is because the ASCII characters represent

ing the number are inserted in the macro expansion. Thus in the CNT macro, the number passed is treated as a string argument

7.2.6. System Macros

In any macro assembler there can be found a *system macro* facility. This facility allows the user to access a set of macros that have been predefined for programmer convenience. The system macros are called like any other macro but result in a search of some system library to find the requested definition.

Most often the purpose of calling a system macro is not merely that of substituting a macro body for a macro call. Instead, the macro calls are treated more as subroutine calls on the system to perform such functions as I/O reads and writes, register saving and restoring, and other specialized functions, including using a real time clock or returning control to the monitor after completion of a user program.

Typical system macro calls look as follows:

```
READ    PAR1,PAR2

WRITE   PAR3,PAR4
```

where PAR1, PAR2, ... are parameters associated with the macro call. The actual expansion of the macro looks as follows.

```
MOV     #PAR1,-(SP)
MOV     #PAR2,-(SP)
EMT     4

MOV     #PAR3,-(SP)
MOV     #PAR4,-(SP)
EMT     2
```

On the PDP-11, the *EMT (emulator trap) instruction* serves as a call to the system monitor. Thus the effect of making a read/write macro call is the stacking of parameters and the turning over of control to the monitor, which

7.2.7. Power of the Macro Assembler

Macro assemblers, which possess the features of nested definitions, conditional code generation, and recursive calls, provide a capability more powerful than a subroutine facility. The reason is that the macro assembler allows code generation at translation time so that the actual program generated fits the applications for which it was intended. Thus, unlike the subroutine, it does not require extensive testing of conditions that may occur at execution time because the code was generated to handle only those cases that were known to occur.

An example in the use of such macro assemblers can be found in system generators. System generators are parameterized macro programs that allow the user to define his particular operation environment as arguments to the program. The program may then be assembled, and produces as output machine language programs tailored to his installation. Such programs do not test to see how much memory or what options are available; instead, such information is already embedded in the operating environment code. As a result, instructions for testing memory size or whether or not a printer is available need never be executed.

Finally, a powerful use of macros can be found in totally parameterized macro programs. The instructions in such programs are either macro calls or macro definitions based entirely on previously defined macros. Thus the macro programmer need never know what the actual machine instructions are or what they are capable of doing. Indeed, the programmer need not know anything at all about the host computer, since the macro expansion is based on character strings and does not depend on the generated result.

A classic example of the use of such a macro-generation scheme can be found in the implementation of SNOBOL4 by its designers and users. This language is written as a macro-generation implementation and only requires that each macro be defined for the host computer. Once each macro is defined, the macros, along with the SNOBOL4 system, may be assembled into a running SNOBOL4 interpretor.

7.3. THE LOADER

The initial load problem was discussed in Chapter 6 in connection with the bootstrap loader. The bootstrap loader, although sufficient for loading short programs, was not general or flexible enough for loading long programs. Instead, that task falls on the *absolute loader*.

The absolute loader is a system program which enables the programmer to load his programs into any available memory locations, in any order. It is used to load programs that are in absolute binary (i.e., fixed to absolute memory locations) or PIC format. Having completed its task, the absolute loader will either halt or transfer control to the start of the newly loaded program.

The absolute loader is usually loaded by the bootstrap into the uppermost area of available memory. In this way it may be preserved across user or system program loads so that it can be available without reloading. Of course, when writing programs, the user must be aware of what memory locations the absolute loader (and the bootstrap if it resides in memory) occupies so that it will not be altered by his program(s).

An absolute program as seen by the absolute loader consists of one or more blocks of data. Each block may include

1. A start-of-block indicator.
2. A record count of the number of bytes, words, and so on, to be loaded
3. A load address.
4. The information to be loaded.
5. A block checksum.

Although the first and last items are not absolutely necessary, they occur frequently in block requirements for small computer loaders.

The start of block indicator is used to indicate that a load block follows. In this way nonloader data may be mixed with loader information. For example, a small computer with only a teletype as a system I/O device may put both the assembly listing and the binary loader tape out to the teletype punch, and leave it up to the loader to separate the two.

The block checksum is used as error indicator for the loader. As each load record is generated by the assembler it is added (logically) to the checksum, which eventually becomes part of the load block. During normal program loading, the checksum is again computed, and if this new value does not agree with the block checksum of the block data, a load error is indicated and the loader halts. Thus the block checksum serves to guarantee that the load operation has been performed correctly.

The rest of the loader block fields are used as shown by the flowchart in Fig. 7-2. Note that the last load address may or may not be used as a transfer address upon completion of the load process. This decision depends on whether the assembly program terminated with a

```
END     LABEL
```

or simply an .END. One way of indicating this difference, which is used by the absolute loader for the PDP-11, is to make the load address even or odd, depending on its being a transfer address or not.

As an alternative to taking the load address from the load block, it should be possible to indicate the load address by use of the computer console switches. This capability allows PIC programs to be loaded in memory

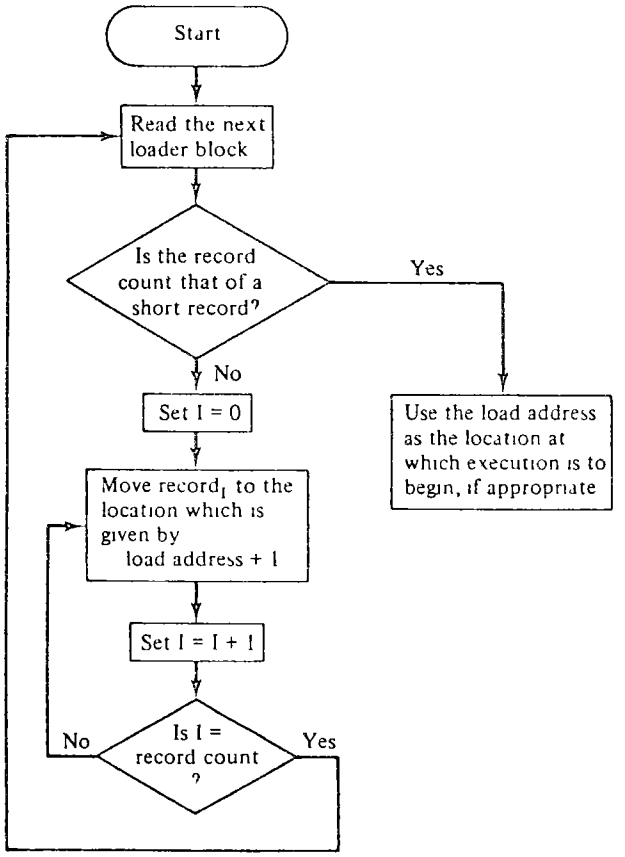


Fig. 7-2

locations different from the relative load addresses given in the load blocks. PIC programs are thereby *relocated* into new memory positions by the simple process of making the actual load address for each block be the sum of the two addresses provided.

7.3.1. Relocation of Programs

Relocation of PIC programs by the absolute loader turns out to be not only useful but necessary. For example, it allows the user to control the loading of the dump routine so that it may be placed in a location of memory that does not overlap the area to be dumped. More generally, such relocation of PIC programs makes it possible for the user to write separate PIC segments, which may be combined in memory to form one large program.

However, making the programmer write all his relocatable programs in PIC format is unduly restrictive. Instead, it seems much more sensible to

leave the mechanical process of relocation up to the computer since it can easily handle the problem. Consequently, the programmer is encouraged to write all his programs in a relocatable form.

As the FORTRAN programmer knows, each FORTRAN program and subprogram requires a separate compilation by the FORTRAN translator. The following are advantages of this requirement:

1. Errors discovered in one FORTRAN program (or subprogram) require only that that program and not all others be recompiled.

2. Absolute addresses need not be assigned at translation time. Thus programs are prevented from arbitrarily overlaying each other. This flexibility also allows subroutines to change size without influencing the placement of other routines or affecting their operation.

3. Separate translations allow the same symbols to be used in different source programs.

4. Once translated, subroutines may be placed for general use in a *library* for future use without retranslation.

Fortunately, these advantages apply to assembly language programming as well, provided that a relocatable assembler and a linker/loader are available as system programs.

Up to now we have not really considered how subroutines are linked and loaded with their calling routines. By default, the absolute assembler would be used to assemble all programs and subprograms together, determining which portions of memory each routine is to occupy, and maintaining the subroutine entry addresses in the assembler's symbol table. However, should we have decided to assemble each routine separately, we would have been faced with the tasks of keeping track of what memory is to be allocated to which routine and what addresses need to be adjusted (e.g., the address portion of the

```
JSR     REG.SUBR
```

instruction must be modified to point to the entry point of the subroutine).

The relocatable assembler and linker/loader mechanize this process for us in the following way. First, the assembler produces object code as if it were to be loaded starting at location zero. Second, the assembler flags each *relative* address and data word so that the linker/loader will know what parts of the program will be affected by relocation. Third, the assembler allows the programmer to declare certain symbols *global* symbols. A global symbol is either defined in a program (as a label or by direct assignment) or it is assumed to be defined in some other separately assembled program. In the first case the global is called an *entry symbol*; in the second case it is called an *external symbol*.

7.3.2. Linking and Loading

As the start of the linking and loading process, the linker/loader receives the following information from the relocatable assembler:

1. Object code.
2. Relocation information about the individual fields in the object code
3. Relative assembly address of the first instruction or datum in the load module.
4. Global entry point and external reference symbols.
5. Length of the load module.

This information assists the linker/loader in developing a *load map* detailing what programs have been loaded, how long they are, where they reside in memory, and what other programs they require. The linker/loader will attempt to load programs until all programs are loaded and no new ones are required, or programs are found to be missing.

Some programs may be part of a user or system library. Such libraries include already translated user routines and intrinsic functions such as SIN, COS, TAN, EXP, LOG, and so on. These programs must be located by the linker/loader through a *directory* which describes the routine, its entry points (some routines such as SIN and COS may share common code), what other routines this routine may need, the length of the routine, and where the routine is to be found. A typical load map and directory are shown in Fig. 7-3.

Load Map

Routines present	Memory address	Length	Routines required	Calling addresses
MAIN	150	1025		350
SUB1	1175	300	SIN EXP	1220, 1345 1415

Directory

Routine	Entry points	Called routines	Length	Physical location
SIN	SIN, COS	— —	268	Desk address
MATH	EXP, LOG SQRT	FLTPT, CLOG	500	Paper tape (external)

Fig. 7-3 Load map and directory.

The actual process of linking and loading is generally handled by one of two possible techniques. The first is called the *transfer vector method* and utilizes a technique similar to the jump table example presented in Chapter 4. By making each external routine call result in a transfer into a jump table, the loader can eventually fill in the address where the called routine has been loaded. Figure 7-4 shows how the assembler code for the PDP-11 could be used to produce relocatable code that includes a jump table to the called routines MUL and DIV. After loading, these table entries will contain jump instructions to the actual starting locations for MUL and DIV.

Assembler code		Relocatable output				Program in memory (with MUL at 200, DIV at 300)		
		0	MUL	0		100	JMP	200
GLOBL	MUL,DIV	2	DIV	0		102	JMP	300
ADD	A,B	4		ADD	50,52	104	ADD	150,152
JSR	PC,MUL	12		JSR	PC,0	112	JSR	PC,100
SUB	B,C	16		SUB	52,54	116	SUB	152,154
JSR	PC DIV	24		JSR	PC,2	124	JSR	PC,102
JSR	PC,MUL	30		JSR	PC 0	130	JSR	PC 100
.				.		.		
.				.		.		
.				.		.		

Fig. 7-4 Loading process using the transfer vector technique.

The linking loader method attempts to avoid the one level of indirectness of the transfer vector technique. It therefore creates a linked list of all calls to the external routine and preserves this list until such time as the relative load address of the external routine is known. At that time, the linking loader traverses the linked list, building up direct calls to the external routine(s).

Figure 7-5 shows the same PDP-11 code being linked and loaded as in Fig. 7-4, except that the linking loader technique is used in the figure. The relocatable output of the assembler includes a linked list of all references to the same external routine, with the list terminating in a null [shown by a dash (-) in the figure].

The basic difference between these two techniques is that the transfer vector method resolves links during loading, while the linking loader does it before loading. The output of the linker part of the linking loader is, therefore, one complete load module, which is loaded by the relocatable loader part.

Assembler code			Relocatable output			Program in memory (with MUL at 200, DIV at 300)	
			MUL	L24			
GLOBL	MUL,DIV		DIV	L20			
ADD	A,B	0	ADD	44,46	100	ADD	144,146
JSR	PC,MUL	6	JSR	PC,–	106	JSR	PC,200
SUB	B,C	12	SUB	46,50	112	SUB	146,150
JSR	PC,DIV	20	JSR	PC,–	120	JSR	PC,300
JSR	PC,MUL	24	JSR	PC,L6	124	JSR	PC,200
°			°			°	
°			°			°	
°			°			°	

Fig. 7-5 Loading process using linking loader technique.

In either case, the results are the same:

1. Object modules are relocated and assigned absolute addresses.
2. Different modules are linked together and global symbols are correlated between those modules which define them and those which use them
3. A load map is produced, displaying the assigned absolute addresses.

thus allowing the programmer to assemble his program and subprograms separately.

7.4. DEBUGGING TECHNIQUES

One of the maxims of programming seems to be that no program of any degree of complexity will run correctly the first time it is executed The problem is that a symbolic program can be assembled correctly and still contain logical errors, that is, errors that cause the program to do something other than what is intended. Although the assembler can check for and detect syntactic errors, it cannot detect logical errors. Consequently, logical errors are usually detected only when the program is run on a computer.

Determining whether or not a program has a logical error is sometimes difficult in itself. A computer is generally used to solve the kinds of problems that require involved calculations, which preclude knowing much about the answers generated. As a result, only when answers are grossly incorrect

is the programmer sure that a logical error exists. When seemingly small errors or results that cannot be measured against known values appear, the programmer is faced with the difficult task of deciding whether or not his program is indeed incorrect. And given a large, complicated program, the programmer may not be able to test all conceivable cases that could be generated, thus causing him to accept on faith that his program does work, 'until proved wrong'

Assuming that a logical error is known to exist, the problem becomes that of determining its cause. Several techniques for this are available:

1. Taking a memory dump of all locations that affect the results.
2. Using the console switches and lights to monitor program execution.
3. Tracing the program as it is executed.
4. Producing intermediate output as the results are generated.

Taking a memory dump, although often helpful, is both static and after the fact. By the time the dump is taken, the error may have caused all pertinent information, including itself, to be altered or eliminated.

Alternatively, the programmer, having the machine to himself, may use the console switches to examine specific locations while stepping through the program instruction by instruction Besides the difficulty in both interpreting binary console displays and translating them into symbolic expressions related to the user's program listing, this technique is extremely time-consuming and very tiring. A better technique would be to place a halt in the program just before the section of code which is to be checked so that the magnitude of the operation may be reduced. Of course, this requires the programmer to know where to place the halt.

A better technique would be to let the computer print out the program instructions and results as they are being interpreted by some trace program. This, too, is a time-consuming process, but only on the part of the computer, since the programmer need not be present while the trace is being run. Some computers, the PDP-11, for example, even have a T-bit in the processor status word to assist in tracing instructions. This bit, when set, causes a processor trap at the end of each instruction execution, greatly facilitating the tracing process.

If computer time is a matter of concern, the programmer is faced with having to trace only selected variables or locations. Either a trace routine is used, or the programmer himself generates intermediate output which indicates that a certain variable has changed value or a specific location has been branched to or referenced.

The programmer can, of course, while sitting at his desk using the program assembly listing, mentally execute his program. This method is frequently used with very short programs, but only with very short ones.

Human memory cannot retain every step and instruction in even a fairly short program; it cannot match a computer memory.

What is needed to debug a user program conveniently and accurately is a service program that will assume the tasks the programmer would have to perform if he used the console switches, took a memory dump, and/or selectively traced his program. Such a facility is known as a *dynamic debugging program (DDP)*.

On a small computer, the DDP takes the form of a conversational system program. It provides the user with a convenient means for debugging and closely monitoring the operation of their programs. In fact, the DDP acts both as a program supervisor and as a binary editor.

Through commands issued to the DDP via the teletype, the user is able to: (1) start a program, (2) suspend its execution at predetermined points, (3) examine and modify the contents of memory words and registers, and (4) make additions and corrections to the running program using either symbolic or octal code. Commands are of the following forms:

1. OPEN: to examine and/or modify contents.
2. CLOSE: to go on to another OPEN or DDP operation.
3. MODE: to establish the type of in or out modes of operation.
4. BREAKPOINT: to suspend the execution of the program at a predetermined point.
5. SEARCH: to search for a particular occurrence of a bit pattern (e.g., an address, a constant, or an instruction).
6. LIMIT: to establish the limits (memory addresses) of the search
7. BEGIN: to start execution of the user program at a specified location
8. PROCEED: to continue execution after a breakpoint interruption

Like all other system programs discussed in this chapter, the sophistication of the dynamic debugging program depends on its operating environment

7.4.1. Example of a Debugging Session

ODT-11 (On-line Debugging Technique) for the PDP-11 is typical of a small-computer dynamic debugging program. Like the PDP-11 editor, ODT has a command mode that is indicated by an asterisk being printed out by the system. Basic commands include

1. n/: opens word n.
2. cr : a carriage return to close an open location.
3. n;G: begins execution at location n.
4. n;B: sets a breakpoint at location n.
5. ;P: proceeds from a breakpoint.
6. $n/: opens register n.

Given the following trivial assembly language program

```
                .=1000
                R0=%0
                R1=%1
START:  MOV     #1,R0
        CLR     R1
        CMP     R0,R1
        HALT
        .END
```

(no label follows .END, since ODT will begin execution of the program), then using ODT-11, the following dialogue may be had (comments have been added for readability):

```
*1004/005001                    EXAMINE THE CLR INSTRUCTION

*$1/000000 123456               CHANGE THE CONTENTS OF R1

 1004;B                         PLACE BREAKPOINTS AT
*1010;B                         LOCATIONS 1004 AND 1010

*1000;G                         BEGIN EXECUTION AT START

E0 001004                       BREAKPOINT OCCURS

*$0/000001                      CHECK R0 AND R1
*$1/123456                      AND THEN
*;P                             PROCEED

E1 001010                       NEXT BREAKPOINT

*$1 000000                      CHECK R0 AND
*$0 000001                      AND R1 AGAIN
```

Although this example is rather brief, it does give the reader some idea of what a dynamic debugging program does. When faced with a typical small computer, with its often-limited number of display lights and means for examining memory or processor registers, the programmer quickly seizes the opportunity to use a DDP rather than probe memory and measure program progress through the console.

7.5. OPERATING ENVIRONMENTS

Having dealt with computers, including their organization and programming, we should now consider their operating environment. After all, from

the user's point of view, the purpose of the computer is to assist the user in the mechanics of solving problems. Thus the operating environment greatly influences how the user is able to solve his problems. This subject forms the content of Chapter 8.

EXERCISES

1. What are the differences between an editor used for program creation and one used for manuscript creation? What types of commands might you find in one or the other?
2. Using the PDP-11 program editor as an example, list its good and bad features. Then give a suggested remedy for each of its bad features.
3. Expand the macro call

```
SWITCH  BUF1 BUF2 SPARE 10
```

for the macro definition given above.

4. Rewrite the macro definitions for SWITCH so that the intermediate storage array TEMP need only be one word long.
5. Define the macro BSS X which is to reserve a block of storage locations X bytes long.
6. Develop a macro that can perform multiplication through recursive calls to the macro body, which performs shifting and adding.
7. Write a program to implement the absolute loader function as flowcharted in Fig. 7.2.
8. What features are missing from ODT-11 as described in the text? Describe a method for implementing them.
9. What difference is there between an on-line debugging package and a continuous trace program?
10. Develop a procedure for implementing a dynamic dump routine which produces a selective dump of specified memory and register contents upon call, without affecting the results of the running program that calls it.

REFERENCES

One of the best references for text editing can be found in the survey article by Van Dam and Rice (1971). Of course, for a particular system, one should read the appropriate manual, such as the PDP-11 *Edit-11 Text Editor*. Similarly, the manuals *Macro-11 Assembler*, *ODT-11R Debugging Program*, and *Link-11 Linker and Libr-11 Librarian* cover the topics of macros, outline debugging, and linking/loading for the PDP-11. However, the books by Gear (1969), Wegner (1968), and Stone (1972), as well as the survey by Kent (1969), are excellent treatments of macro assemblers, while the survey article by Presser and White (1972) is an equally well done presentation of linkers and loaders.

8 OPERATING SYSTEMS

Today it is inconceivable that a medium-to-large computer could exist without an operating system for its users. Indeed, even small or minicomputers can and do have sophisticated disk and tape operating systems as part of manufacturer-supplied software. For this reason, almost all programmers will, at one time or another, come face to face with an operating system environment.

Operating systems, if properly designed, exist for the users' convenience. They serve to bridge the gap between the needs of the user and the characteristics of the hardware. In this capacity, they directly assist the user in solving his problems through simplified programming and more efficient computer operation. However, to a large extent, the user never knows what the operating system is really doing. Instead, the user sees the system in terms of the services it provides for: program preparation, translation, execution, and debugging.

In order to understand what a computer system is all about, it is necessary to understand the system components and their organization. These components, computer hardware and software, were discussed in previous chapters. This chapter is thus concerned with the general job of organization as it is performed by the operating system. However, since our concern has been with small computer systems, we shall continue that interest as we take a look at rather specialized operating environments which exist for this class of machines. Because of the limited resources available, operating systems for small machines tend to be more constrained than for their larger computer system counterparts. Nonetheless, the same principals and concepts apply, the chief difference being that of the relative emphasis placed on the various system components.

8.1. VERY BASIC COMPUTER SYSTEMS

At the very least, every small computer comes complete with a paper tape system. In this environment, very reminiscent of the early days of computing, the input and output of programs and data are performed manually by the user via a paper tape reader and punch. The user communicates with, and receives printed output from, system and user programs through the teletypewriter device connected to the machine.

Even though the loading of programs is performed manually, a paper tape system normally contains a comprehensive software package of commonly used system programs which provide the user with complete facilities for writing, editing, translating, debugging, loading, and running his own programs. Since system programs have already been covered in Chapter 7 and earlier chapters, the reader is familiar with the capabilities of such a paper tape system.

Unless the reader has had the experience of using a paper tape system, he is not likely to realize how unsatisfactory and trying it can be. Operating such a basic system requires the user to take his coded program and manually perform the following operations:

1. Load and execute the paper tape editor.
2. Produce paper tape source programs using the editor
3. Load and execute the paper tape assembler.
4. Translate the editor produced source program.
5. Load and execute the binary object program produced by the assembler.
6. Debug the program, repeating the first five steps as necessary.

Each step presumes that the software bootstrap and absolute loaders remain intact during successive program loads and executions. Unfortunately, this is not usually the case, and more often than not, the beginning programmer will load both loaders at one time or another.

Manual control of the operating environment is clearly inconvenient. It involves manipulating and maintaining numerous paper tape programming systems, and it results, in general, in the inefficient use of the hardware. Consequently, a more automatic level is desirable, and this level of control is found in the typical general-purpose disk operating system (DOS).

8.2. COMPONENTS OF A DISK OPERATING SYSTEM

The addition of a *secondary storage system device* (e.g., a disk) is what makes the disk operating system a comprehensive operating environment for

both the development and execution of user programs. User programs and data, along with system programs, can all reside on the disk and other secondary storage devices, to be loaded into memory under program control. Instead of loading and reloading paper tapes, the DOS user can perform the same functions by issuing *commands* to the system. These commands not only provide user services (such as program loading) but also provide for efficient program and system management. Typical commands and their functions are shown in Table 8-1 for the PDP-11 disk operating system.

Table 8-1 System commands.

Command	Function
GET	Load a program
RUN	Load and begin a program
DUMP	Remove a program from memory
BEGIN	Start execution of a program
STOP	Halt the current program
CONTINUE	Resume execution of a halted program
END	End input from a device
LOGIN	Identify user to system
FINISH	Log off system
ASSIGN	Assign a physical device to a dataset

Commands, however, are only the outward manifestation of an operating system. To gain an understanding of how its functions and facilities are provided, it is necessary to consider the components of the system and their organization. Since one of the most important functions of an operating system is the effective management of its information structures (e.g., programs and data), it is important to understand the basic informational unit of the system. This unit is called a *file*.

8.2.1. Files—Organization and Access

A file is a collection of related records or data items treated as a unit. The word "file" is thus used in the general sense of "any collection of information items similar to one another in purpose, form, and content." For example, a program may be a file, just as a data structure (called a *dataset*) or even some system program such as an editor or assembler, may be. Unfortunately, the same word *file* is also generally applied to external storage media, such as disks and tapes, when what is really meant is *file-oriented devices*.

Each file-oriented peripheral device has a *file structure*, which represents the method of recording, linking, and cataloging data files. The file structure dictates the organization of the file on the device and the method of file access. This organizational structuring is important because a file can be

effective for a user application only if it is designed to meet specific user requirements. Such factors as size, activity, and accessibility must be considered when determining the structure of a file.

The way in which a file is organized upon a storage media depends upon the way in which the user normally expects to create and later process it. Three methods that have been used are: (1) contiguous allocation, (2) linked list allocation, and (3) indexed allocation. Each of these methods is shown in Fig. 8-1.

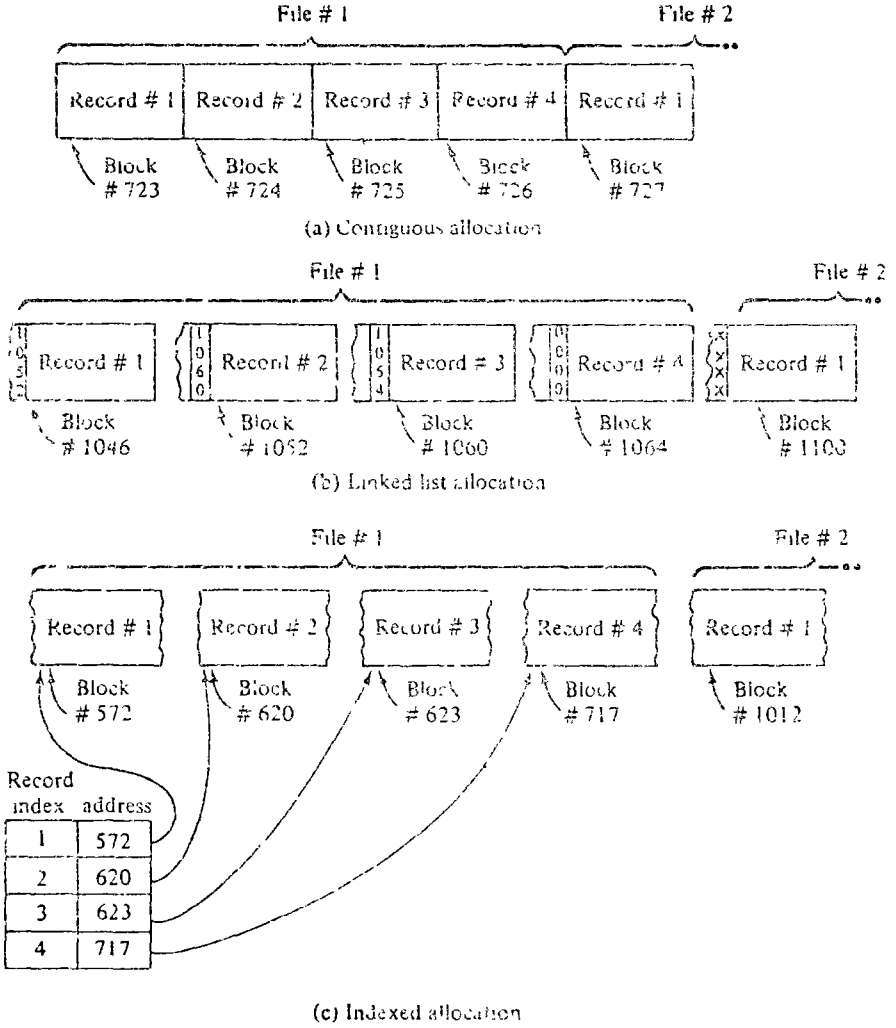


Fig. 8-1 File organization.

In the small computer system, file structure organization is not usually left up to the user but is predefined for the various peripheral devices. Similarly, the method of access is system defined and is ultimately connected with the file structure. The most usual access technique involves *sequential access* to both the data file and the individual data records. This access method is characteristic of unidirectional devices, such as magnetic tape, although other devices, such as disks, may be organized so as to permit sequential access.

Sequential access is a storage retrieval technique in which a file and the records within it must be retrieved in the sequence in which they physically occur. Sequential access, when applied to the process of locating the beginning of a file or a data record within the file, means that the time required for such access is dependent on the necessity for waiting while nondesired files or records are processed in turn.

Traditionally, contiguous allocation is used to implement sequentially accessed files on sequentially organized devices such as magnetic tape. After each record is processed, the next record is immediately available, since positioning of the physical media will leave that record positioned at the read/write head of the device.

As an alternative to the sequential organization, the linked-list organization may be used for *direct-access devices* such as disks, where the time to search for and locate the next record is insignificant in processing the file. The linked-list structure has the advantage over the contiguous allocation of allowing files to grow larger with time by simply linking in a new record to the end of the list. This is not in general possible for the contiguous allocation, since the next block may already have been allocated.

Another technique for accessing a file is *random access*. Random access of a file and records within the file means that the time required for such access is independent of the location of the file or record relative to other files or records on the medium. Thus the order of retrieval of file information is unimportant and can be ignored.

Again, two possibilities exist when file access is random. These are contiguous allocation and indexed allocation. By knowing where the contiguous file begins, random accessing occurs in much the same fashion as element accessing occurs for a one-dimensional array. The limitation of the contiguous allocation remains the same, however. files cannot, in general, expand in length with time.

The use of an index into the file allows both random access and growth with time. Thus this method of allocation is preferred over contiguous allocation unless access time is important. Like linked-list allocation, indexed allocation requires that the location of the next record be fetched before the actual record may be accessed.

As a third alternative to the two access methods presented, an intermediate method may be used. This method, normally employed on disk and disk-like devices, allows a file to be accessed randomly while the file's data records may be accessed sequentially. This access method is called *indexed-sequential* and uses the indexed organization with more than one data record per block.

Indexed-sequential organization is well suited to those applications where it is necessary to access sets of records randomly but individual records of the set sequentially. A typical example of such an application would be a personnel records file where having found the records for a certain employee, it is necessary to update these employee records in a sequential fashion

8.2.2 Directories

Having provided a file structure, and having specified its access method, the next problem is how the file is located by the system once stored away. One method that could be used is to keep track of the device addresses so that each file can be retrieved directly. The use of absolute addresses is not very acceptable, however, for much the same reasons that absolute addresses are avoided in symbolic programming Instead, symbolic names must be associated with each file so that the files may be referred to by their *file names*

To provide a connection between the file names and their device locations, a *file directory* or table of contents for each directoried file device must be part of the system. The file directory will contain not only the unique name of the file and its starting address on the device, but also its file structure, including, if necessary, a pointer to an index table. Figure 8-2 shows a directoried data access for a sequentially organized file that can be

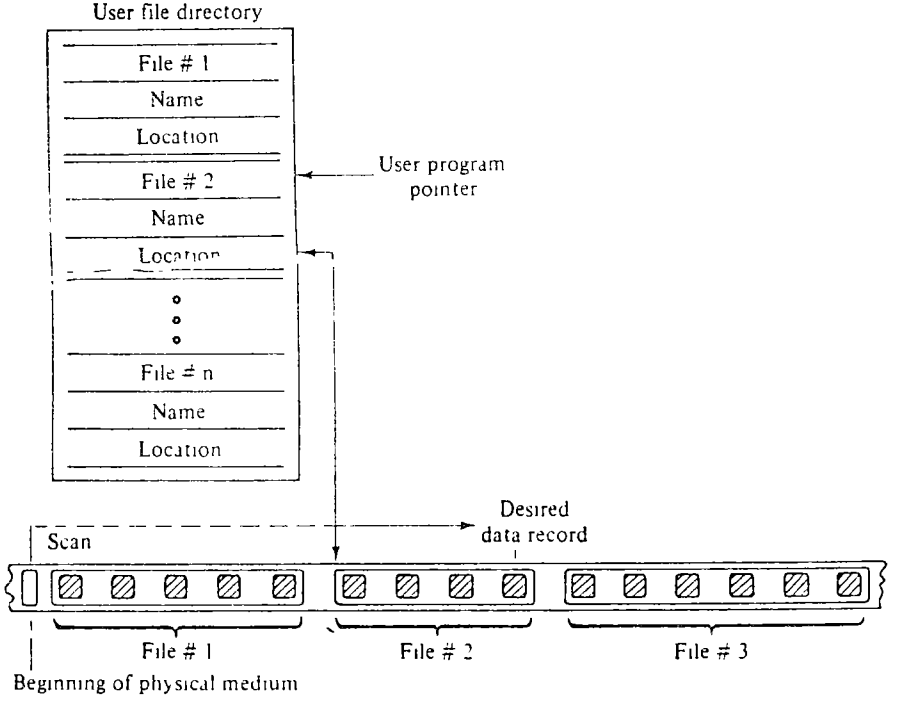


Fig. 8-2 Directoried data access

randomly accessed. Devices such as tape cassettes and DECtapes have this capacity by which the transport may search to a known location before it begins processing the file.

When directoried data files are removable from the system, it is necessary to preserve the directory of files between uses. To do so requires that the directory be stored on the physical media, in a fixed location, along with the files it points to. As part of these directories, *bit maps* are maintained both to indicate which device blocks each file occupies and to show all occupied blocks.

File structures that employ a directory allow simpler and, in the long run, faster access to a file (e.g., the beginning of a file). This is a distinct advantage over those devices which do not use a directory and must therefore rely on a file's position relative to other files in order to locate it.

8.2.3. Multilevel Directories

When two or more people share the same device (such as the system disk) for storing files, problems may arise because of duplicate file names. Since both will have access to the same set of files, one user may accidently modify or destroy another's file by simply not knowing that the file name used was already assigned. The solution to this dilemma is to establish a separate user file directory for each system user. The separate directory will therefore allow each user to name a file without regard to the names chosen by others.

The basic mechanism for locating user files on a shared device requires a two-level file directory, as shown in Fig. 8-3. Each user has a unique code that must be provided whenever the LOGIN command is used. This code serves to identify a particular entry into the master file directory, which is actually a pointer to the user's file directory.

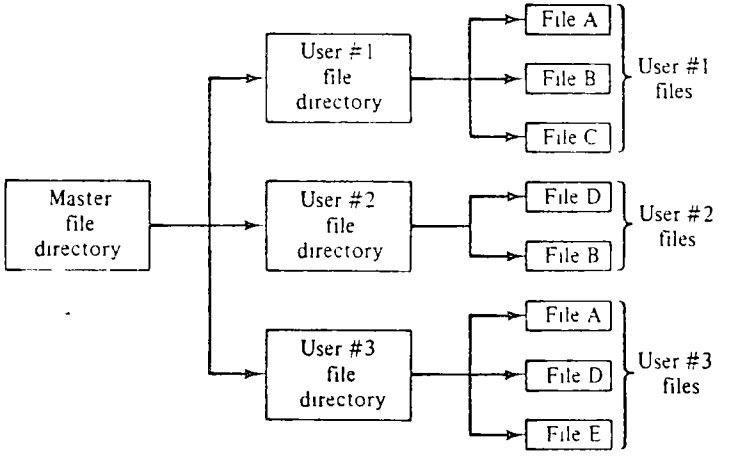


Fig. 8-3 Master and user file directories.

8.2.4. Problems of Control

Now that we have described an elaborate scheme to define the files within the system, it becomes obvious that one of the central functions of an operating system is control. For example, the file system represents one large facility that must be controlled in its allocation of peripheral device space and its storage and retrieval of file information on the peripheral devices. Fortunately, the function of the file system is precisely the minimization of the potential problem.

As far as the user is concerned, devices themselves are not of primary interest; the datasets of files that reside on them are. Thus a simple and useful extension may be made by broadening the concept of a file to encompass all information sets, including devices. In this manner it is easy to conceive of a paper tape reader as an information set (and hence a file) of a very special type. By *special type* is meant that the information set must be handled in only a very limited fashion (e.g., output not allowed to an input device). With suitable limitations, there is no reason why all devices cannot be conceived of as being file-like information sets. All that is necessary is to recognize and build into the system the fact that it is not possible to treat all files uniformly (e.g., not all files can be read from, written to, or rewound).

Both the system and the user can treat I/O devices uniformly as information sets. Within the system, however, there must be an interface between the system or user-created program and the external world of I/O devices. The purpose of the interface is to minimize I/O programming for the DOS user in the same way that the I/O programming system (IOPS) simplified I/O programming for the basic system user, as discussed in Chapter 6. In that chapter it was pointed out that the IOPS routines served to relieve the user of the burden of I/O service, file management, overlapping I/O considerations, and unnecessary device dependence. The last point, device independence, is especially important in a disk operating system, where an I/O service routine for a nonfile-structured device, such as the paper tape reader, must ignore (rather than declare as an error) a command to "seek a file" (which is required for all file-structured devices prior to issuing read commands).

Clearly, the central function of IOPS is to establish the information path between the system and the device. This requirement is met somewhat independently of the user, since one of the goals of IOPS is to minimize the user knowledge required. And since the user is to be spared the task of writing system software to perform I/O, standard system routines (called device handlers) must be a central part of IOPS. These routines perform the functions of

1. Driving the I/O devices.
2. Manipulating files on the devices.
3. Allocating/deallocating storage space on the devices.
4. Maintaining current records about user requests and device status.
5. Coordinating peripheral activities (such as buffering and blocking) as required by the I/O.

As pointed out in Chapter 7, system macros are the means of communication between user programs and I/O device handlers. These programmed I/O commands are also referred to as *requests*. Table 8-2 lists typical requests for the DOS-11 (PDP-11 DOS) system. Not all requests shown in the table actually perform I/O. Some, such as .OPEN and .CLOSE, merely serve to initialize a dataset or file for subsequent I/O processing. Others, such as .BIN2D, .RADPK, and .ALLOC, perform auxiliary operations on the data or the data file.

Table 8-2 Programmed requests.

Programmed Request	Function
OPEN	Open a dataset
.ALLOC	Allocate a sequential file
.CLOSE	Close a dataset.
.DELET	Delete a file
.LOOK	Search a directory for a file
.READ	Read from a device.
.WRITE	Write on a device
.WAIT	Wait for device completion
.BIN2D	Convert binary to decimal ASCII
.BIN2O	Convert binary to octal ASCII
.D2BIN	Convert decimal ASCII to binary
.O2BIN	Convert octal ASCII to binary
.RADPK	Radix-50 ASCII pack
.RADUP	Radix-50 ASCII unpack

8.2.5. File Management Utility

Although program requests for file management provide the basic functions needed to utilize files, it is inconvenient to have to write a program every time one wishes to manipulate files. Thus most operating systems include a system software package for the transfer of data files from one I/O device to another, while performing simple editing and control functions as well. This package, known as PIP (*P*eripheral *I*nterchange *P*rogram) on the PDP-11, handles all data and file formats found in DOS-11 so as to

1. Transfer a file or group of files from one device to another.
2. Merge files into a single file.

3. Delete, update, rename, or replace files.
4. Allocate file space and initialize whole devices.
5. Print listings of file directories.
6. Handle file protection.

In effect, the file utility package provides at the user level the same sort of services that IOPS provides at the program level. Users need only enter commands to the PIP program and it will decode the command and perform the desired function. For example, the user might wish to make a backup copy on DECtape of an existing disk file. To do so he would run the PIP program and then, in response to PIP's request for a command (indicated by a # sign), type in

```
#DT1 BACKUP SRC<DC0 MYFILE SRC
```

The new file, named BACKUP.SRC, would then be a copy of the original file, called MYFILE SRC.

To examine the directory for a certain device, the command to PIP would be

```
#LP <DC0 /DI
```

indicating that a *directory* listing of the contents of disk unit zero is to be produced on the *line printer*. This listing would appear as

```
DIRECTORY DC0  [  1,1  ]

25-NOV-73

MONLIB CIL    401C 09-SEP-73 <377>
MACRO  OVR     28C 19-NOV-73 <233>
LINK11 OVR     72C 19-NOV-73 <233>
MACROP LDA     97  19-NOV-73 <233>
LINK   LDA     56  19-NOV-73 <233>
PIP    LDA     36  19-NOV-73 <233>
EDIT   LDA     50  19-NOV-73 <233>
LIBR   LDA     30  19-NOV-73 <233>
PIPOV0 OVR     18  19-NOV-73 <233>
PIPOV1 OVR     14  19-NOV-73 <233>
PIPOV2 OVR     18  19-NOV-73 <233>
PIPOV3 OVR     12  19-NOV-73 <233>
ODT    OBJ     37  19-NOV-73 <233>
TEST   BAK      4  23-NOV-73 <000>
TEST   PAL      5  23-NOV-73 <000>

TOTL BLKS    878
TOTL FILES    15
```


where the fields indicate the file name, file size, file creation date, and file protection code. Also included are the total number of files and blocks in use for the "user identification code," [1,1].

It is important to bear in mind that a file management utility is a system software program in the same sense as were the programs in Chapter 7. The goal of such software programs is to provide routines that assist the user in solving his problems. As a consequence, these programs do not in and of themselves produce useful results but rather allow the user to utilize the hardware available to him effectively.

8.2.6. Device Independence

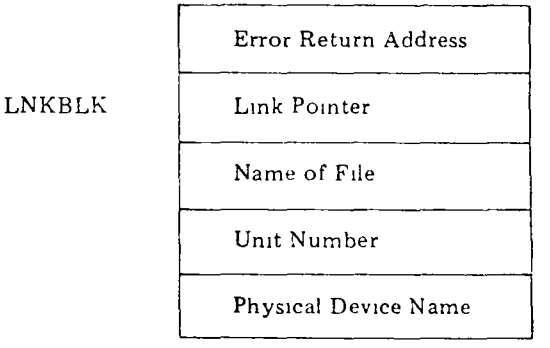
As the reader may recall, system macros are implemented using interrupt-generating instructions. For example, a write macro operation

```
WRITE   LNKBLK,BUFHDR              ;WRITE DATASET
```

in DOS-11 is expanded into

```
MOV     #BUFHDR,-(SP)              ;STACK
MOV     #LNKBLK,-(SP)              ;MACRO ARGUMENTS
EMT     2                          ;PERFORM EMULATOR TRAP
```

The two arguments of the macro call are called the *linkblock* and the *buffer header*. The buffer header serves to define the data buffer as described for IOX earlier in Chapter 6. The linkblock serves to establish the connection between the data file (logical device) and the physical device. A linkblock is defined as follows:



The first two entries are used by the system for error processing and initializing the dataset. The next entry, the logical name of the dataset, is used to associate a logical file with a physical device The function of the ASSIGN command is to fill in this entry. Finally, the last two entries serve to specify the standard name of the physical device associated with the file.

Ordinarily, a programmer specifies I/O devices as he writes the program. However, there are circumstances when he will want to change the device specifications when his program is run. For example:

1. A device that the user specified when he wrote his program is not in operation at run time, but an alternative device is available.

2. The programmer does not know the configuration of the system for which he is writing, or does not wish to specify it (i.e., he is writing a general-purpose package).

Through the use of the linkblock, the ASSIGN command, or by assuming the default condition, the programmer can write programs that are *device-independent*. From the user's point of view, such device independence results in very flexible programming.

8.2.7. Monitor

The user communicates with the system in two ways: (1) through keyboard instructions, which have been referred to as commands, and (2) through programmed macro instructions. In both cases the effect is to initiate a *control program* or routine which loads a file, makes a correspondence between a logical file and a physical device, opens a dataset, writes onto a device, etc.

Clearly, the control programs must work in mutual harmony if the system is to operate successfully. Although much of the system can be conceived as the sequencing of one program or *task*† after the other, it is possible to have two tasks operating in parallel (e.g., an I/O operation and a computation). Thus a master control program called the *monitor*, which can be responsible for the entire operating system and all of its component parts, is needed.

The monitor must be responsible for the initiation, maintenance, and termination of all other programs. It coordinates program-to-program and task-to-task transitions and processes the communications among the user, the system, and the many control programs. It also must act on monitor calls, validate and transmit I/O calls to device handlers, supervise data and file manipulations, and provide error diagnostics.

There are basically three sections of a monitor. (1) the permanently resident monitor, (2) the nonresident monitor, and (3) the system loader. The resident monitor remains in memory when system or user programs are running and acts as the interface between the program and the system's facilities.

The user/operator may alter the structure of the resident monitor via commands to the nonresident monitor. The nonresident monitor allows the user to alter many key parts of the system, in order to set up the system for the next program. Normally, at the end of a particular program, the computer user or the program itself returns control to the nonresident monitor. At that point the user issues new commands to set up the system for the next program to be run.

The system loader builds the resident monitor according to prior commands to the nonresident monitor. It loads all system programs and all handlers for those system programs from the system disk, and these programs in turn allow the user to edit, assemble, load and link, perform file manipulations on, execute, debug, and so on, his programs. Since the purpose of the system loader is basically to set up the system (e.g., by loading system programs and setting them into execution), it is completely invisible to the user.

8.2.7.1. Monitor/User Interaction

The console teletypewriter is the primary user-system interface for DOS program control. This control is implemented by commands to the monitor which cause system and user programs to be loaded and executed (as described in Section 8.2), by commands that perform special services, and by control character commands that provide system control while running user or system programs.

Most of the monitor commands must be issued prior to loading programs and are interpreted by the nonresident monitor, since it is not, in general, necessary to keep the command recognizer in memory during system or user program execution. However, during program execution, a small set of keyboard commands must be available for general program control. These commands are interpreted by a portion of the teletypewriter's I/O device handler (which is part of the resident portion of the monitor) and are used to control program start and restart, dumping of memory, and reloading of the nonresident monitor.

Since the monitor and any program running under it must share the same console teletypewriter, the user must specify whether the given keyboard input is intended for the monitor or for the operating program. Consequently, the modes of operation are determined by the first character entered. All characters following a special control character (a CTRL/C for DOS-11) are interpreted as monitor commands and are passed to the monitor for execution. All other characters are assumed to be for the operating program, and the characters will be buffered until required by the program.

8.2.7.2. Monitor Organization

Figure 8-4 illustrates the data flow and general organization of the monitor. Although most of the functions of the various modules have already been described, several require further comment.

†A task is a well-defined unit of work that competes for the resources of the system (e.g., memory, files, I/O). Stated more simply, it is a program or routine with known inputs and outputs.

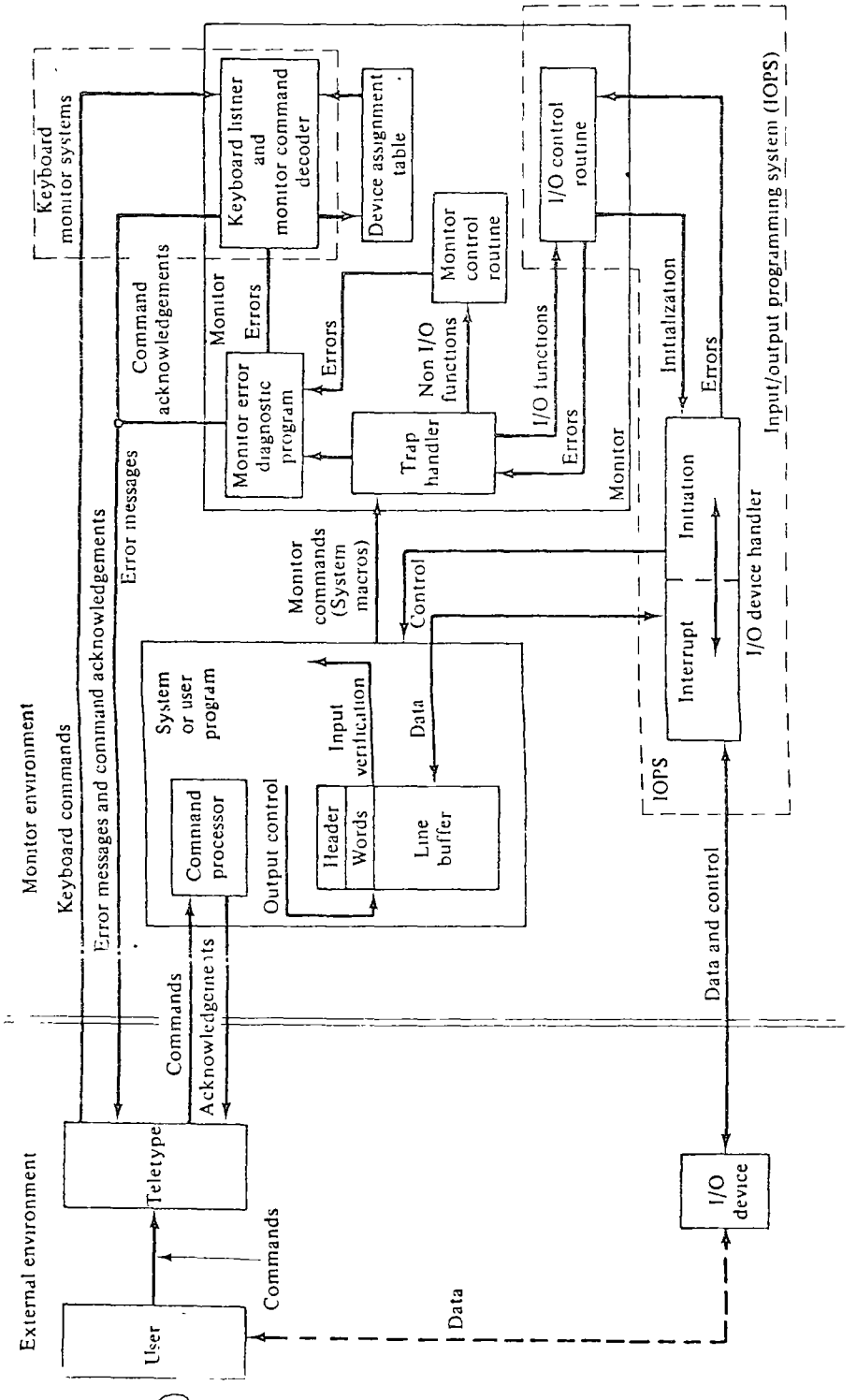


Fig. 8-4 Command, control, and data flow in monitor environment.

The first modules of interest are the command processor and the monitor command decoder. Both the user and the monitor share the same control program, called the *command string interpreter*. This routine preprocesses the specification for whatever user or system program it was called by. By having one routine for both the system and the user, one common format for input and output dataset specifications to a program is provided through a single monitor routine.

The next module of interest is the one labeled *device assignment table (DAT)*. This table is used to store the data from each ASSIGN entry, since device/file specification by console assignment can occur at any time, even before the program that requires the new assignment is loaded. The DAT is set up in a similar format to the linkblock (as shown below) and resides within the monitor so that its entry may be checked whenever the program under execution calls for dataset initialization.

Logical name of dataset
Physical device name
Unit number
File name

The use of a device assignment table can be illustrated by the following example. Before being run, a user program wishes to assign a DECtape file FREQ.BIN to a dataset called FRQ The ASSIGN command would be used:

```
ASSIGN, DT FREQ BIN, FRQ
```

where the commas act as separators and the colon separates the name of the physical device (DT for DECtape) from the file name.

At first, the use of the ASSIGN command, the device assignment table, and the linkblock may seem strange. However, the FORTRAN programmer should be able to recognize that these new commands and tables are nothing more than a new solution to an old problem. In FORTRAN, when performing reads and writes, the programmer must write statements of the form

```
READ  (u,f)  I/O list
WRITE (u,f)  I/O list
```

where u represents a unit and f a FORMAT statement label. Usually there are default values for u, and reading a data card is performed on unit 5 [e.g., READ (5,10) A,B,C] while printing a line occurs on unit 6 [e.g., WRITE, (6,20) A,B,C]. However, when file-oriented devices such as tapes are used,

some form of an assign command (or control card) must be used to equate the unit numbers to their particular devices. This, of course, associates all the files on the device media with the unit number, and it is up to the user to separate out the various files.

Going one step further, it would be very nice to be able to associate a particular file on a particular device with a unit number. As long as the device has a directory associated with it, this is a relatively simple process. For example, a programmer could issue the command

```
ASSIGN, DISK MYFILE, 6
```

to assign the file, MYFILE, on DISK1 to unit 6. Alternatively, instead of using unit numbers, dataset names may be used, so the command becomes

```
ASSIGN, DISK1 MYFILE, MYDATA
```

allowing the user's program to refer to the file by its dataset name, MYDATA, rather than by a unit number.

Returning to Fig. 8-4, the last module to be explained is the one labeled *trap handler*. Both at the user level and internally within the monitor, the standard method by which a monitor routine is accessed is through a trap or monitor calling instruction. For the PDP-11, the EMT instruction is useful because its lower byte is not considered in the hardware decoding operation, it can therefore be used for a software code to identify the module required and avoids the use of a second word (e.g., as shown, a call for an I/O .WRITE is EMT 2 or 104002). By using the stack to pass arguments, the monitor call (though the EMT ensures that the called module has complete freedom in its use of registers and that the necessary handler for this instruction has the opportunity to control all communication paths throughout the system (e.g. EMT is an interrupt-generating instruction). This control facility is a particular advantage to the small computer operating system which must *swap* monitor routines in and out of memory and maintain complete control of user and system programs, all without the aid of any special hardware.

8.2.7.3. *Monitor Residency Table*

An important part of the trap handler is the *monitor residency table (MRT)*, which supplies two types of information for the trap handler:

1. It shows which monitor routines are resident in memory, either permanently or for the duration of a program run, and where they are loaded currently.

2. It acts as a directory to the remaining routines as stored within the monitor library on the system device, to enable immediate access when one of these routines must be brought into memory.

For DOS-11, the table is a set of one-word codes and is ordered in the sequence of those codes starting at 0.

The format of each word in the table (see Fig. 8-5) shows the current location of the monitor routine it represents, using the fact that for a valid PDP-11 address for execution access, bit 0 must be 0 (i.e., a word boundary).

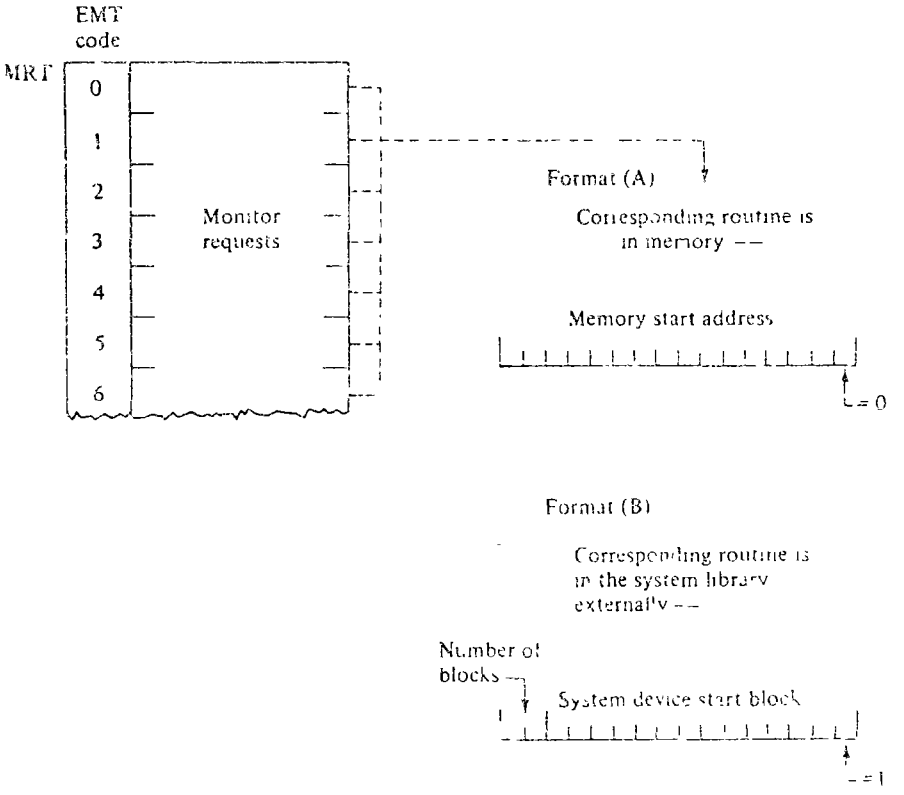


Fig. 8-5 Monitor residency table format

The state of the table depends on which routines must remain within the computer memory at all times, because they control the system generally, and which routines may reside upon the system device, because they perform ephemeral tasks. By using the system loader, nonresident routines may be loaded when required and can later be removed when their purpose is served. In this way, available memory space need not be used by the system (e.g., the monitor) but may be made available to the user. Clearly, this is a necessary requirement for the small computer user who has a machine with somewhat limited memory space.

8.2.7.4. *Monitor Memory Organization*

From the previous sections it is clear that certain monitor routines/modules must be resident in memory at all times. These routines determine the minimum allocation of the computer's memory, as shown in Fig. 8-6. The modular structure of the monitor allows the user to determine which modules are to be resident and which modules are to be swapped from the disk. In the latter case, it should be noted that a temporarily loaded routine occupies a reserved area within the monitor (the swap area) and does not require that a part of a program be swapped out first. This means that no restrictions need be placed upon the activities of a program as might be the case if part of its area were potentially removable.

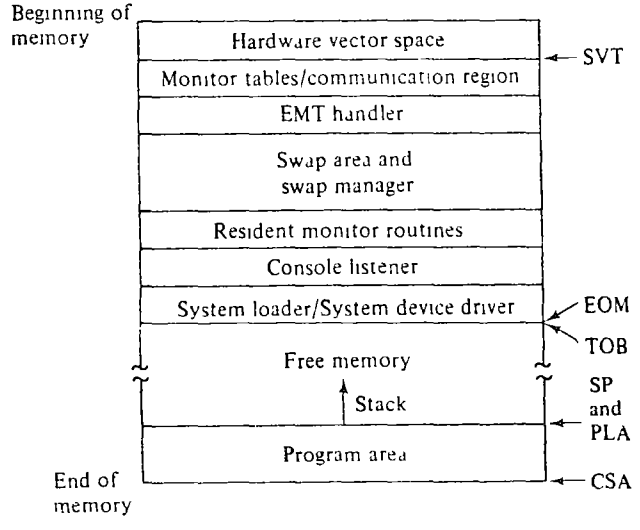


Fig. 8-6 Memory allocation

Despite the fact that swapping can be accomplished fairly quickly from the disk, it still takes a finite time, and the user who has memory to spare may prefer to make use of it. Modularity of the monitor routines again helps, in that

1. If a particular module is required so frequently by the user(s) of the system, that module can be added to the list of those already part of the permanently resident monitor, or

2. If a module is particularly appropriate to one application, the routine can be loaded with the program concerned so that the routine is resident for the duration of the run.

8.2.7.5. *Dynamic Memory Management*

Another feature of the monitor as shown in Fig. 8-6 is its dynamic buffer allocation scheme for free memory management. This scheme postpones the allocation of memory for the purpose of I/O service until a running program actually requires it. Only then are the buffers allocated and the I/O drivers loaded, and when they are no longer required, their memory space is released. The allocation and deallocation of memory, being dynamic, means that the basic memory map varies with time. A typical memory map during program execution would appear as shown in Fig. 8-7.

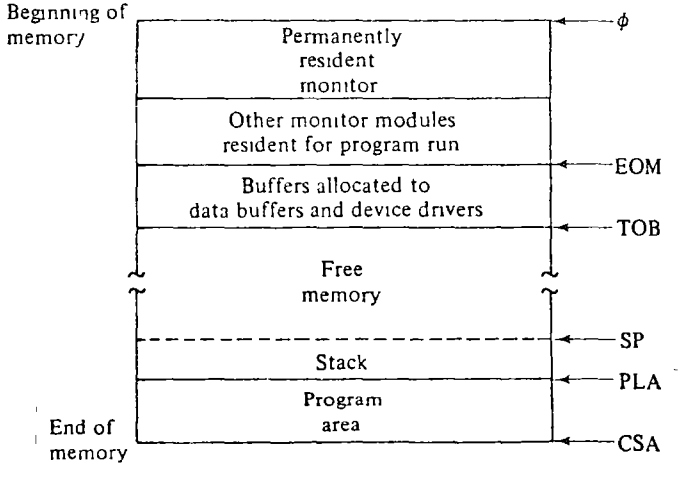


Fig. 8-7 Memory during program run.

Another consequence of dynamic memory allocation is that all the modules that take advantage of this feature must be independent of the positions they occupy. Position-independent coding presents no problems for a computer such as the PDP-11, but allowing independent modules to intercommunicate does. What is needed is a *system vector table (SVT)*, which provides a common area for the storage of information on the state of the system at any time. In particular, the SVT must contain pointers to the other parts of the system which provide such information as the end of the monitor, the start of the monitor residency table, the name of the loaded program, and so on, as shown below in Fig. 8-8. These pointers were previously indicated on Figs. 8-6 and 8-7, allowing the reader to go back and interpret their use.

8.2.8. Use of Operating Systems

Having discovered something about the internals of a typical disk-based operating system, it is worthwhile to examine how an operating system meets the more general needs of its users. As might be expected, the needs tend to be rather diverse, and since it is clearly impossible to view operating

Symbol	Meaning	Purpose
EOM	End of monitor	Dynamic origin for free-core buffer space
TOB	Top of buffers	Dynamic end of allocated buffer space
CSA	Core size available	Set on initialization to highest memory address
PLA	Program load address	Set only when a program is in core (lowest point loaded)
SCW	System configuration	Reserved for bit switches to indicate available facilities
BAT	Beginning of D A T	Set only if a device assignment table is established
MUS	Monitor/user switch	Low byte 1 = program loaded, − 1 = program stopped High byte 1 = program running, − 1 = program waiting
PSA	Program start address	Set if program in core to address in source END (or 1 if none)
RSA	Restart address	Set by program for RESTART at console keyboard
PGN	Program name	6 character value associated with source program
MRT	MRT start address	Used for access to the monitor residency table
DDL	DDL start address	Likewise for the device driver list
MSB	MSB start address	Used for access to the main swap buffer

SVT

Fig. 8-8 System vector table contents.

systems tailored to each user's application, operating system designers generate systems that meet the needs of particular application areas. These areas and their operating environment may be broadly classified as one of the following:

1. Batch and time-sharing systems.
2. Real-time control systems.
3. Data-based systems.
4. Computer communications systems

Batch processing and time-sharing systems are familiar operating environments for most computer system users. These general-purpose programming systems are best suited for the programmer who wishes to develop

and execute his programs. From the standpoint of the small computer disk operating system, these systems represent add-on capabilities to the basic DOS environment.

Real-time control systems are designed for operating environments where many tasks must be maintained and controlled as events occur that are external to the computer. These systems must be capable of scheduling the real-time programs (called *tasks*) performing the input or output of necessary task information, communicating to the human operator what is happening, and performing such other functions as required for a real-time, multiprogrammed operation. A typical example might be found in a process control application.

The distinguishing feature of the data-based system is clearly the enormous amounts of information that must be managed. This information must be readily available to the user who queries the system, and must be well protected against accidental loss or unauthorized intrusion. Like the real-time system, emphasis is placed on program use rather than program development or testing.

Computer communications systems are often likened to electrical power utilities and natural gas networks. In both cases the system presents itself as a vast web of interconnected units capable of almost indefinite growth so that as the customer load increases, the system can be expanded without limit both by adding extra units and by connecting with other utilities to draw on their unused capacity. Such systems require well-defined interfaces and interconnection structures.

What follows in this chapter is an examination in more detail of each of these operating systems. Since the subject of this book is the small computer, emphasis will be on what types of small computer operating systems have been developed, and what function they serve.

8.3. BATCH AND TIME-SHARING SYSTEMS

Given the capabilities of the small computer plus the added flexibility of the disk-based operating system, it is not too difficult to develop a *batch processing system*. The batch processor is actually an additional control program within the monitor which allows user commands to come from the same device as the user programs. By placing user commands and programs together to form *jobs*, the system is capable of running many jobs consecutively without requiring operator intervention.

Special monitor commands (in addition to nonbatch commands) are used to

1. Enter batch mode.
2. Define and separate jobs.

3. Indicate that data follow.
4. Indicate the end of the job.
5. Terminate batch mode.

These commands form what is called a *job control language*, and if the batch input device is a card reader, they are punched into *job control cards*.

Besides the batch system, it is not uncommon to find *time-sharing systems* on small computers. Rather generally defined, a time-sharing system is one that provides many users with simultaneous access to a central computing facility.

A time-sharing system is, in fact, a *multiprogrammed* computer which allows its multiple users to share system resources in such a fashion that each user thinks that he is getting individual attention. The system is multiprogrammed in that several user programs will be simultaneously resident in memory at any given time. Each program receives a quantum of computer time, called a *time slice*, during which it may perform computations. Should it use up its time slice, or reach a point where further computation is not possible (e.g., some I/O information is needed), the CPU will be turned over to another program. This transfer of control is handled rapidly since the next program to gain the CPU will already be in memory.

The time-shared operation of a computer implies sharing the computer's time and space resources on a dynamic, and hence temporary, basis. Several (or all) user programs may be memory resident, while others may be in the process of being loaded from or to auxiliary mass storage. Indeed, if memory is not large enough to hold all the user programs and data, it will be necessary to swap user information in and out from the auxiliary storage upon demand.

The time-sharing operating system (TSOS) requires a sophisticated set of control programs to handle the sharing of system resources, the time slicing, the storage allocation and program relocation, and the basic servicing of users, besides the types of operations normally associated with a disk operating system. One of these control programs, the *scheduler*, has primary responsibility for both the basic servicing of the users and the optimal uses of the system resources. Each time the monitor gains control, it utilizes the scheduler to determine which program is to be put into execution next and what user swapping must occur if it is to keep the system busy and the users satisfied.

Because of heavy demands placed on the computer, it is often necessary to limit the flexibility of the TSOS. The most flexible TSOS is an *open, conversational system* that gives the user direct access to all the facilities (including I/O devices) of the operating system. *Closed, conversational systems* usually limit the user to specific languages and systems. *Remote program entry systems* are the most restricted form of time sharing, in that the user is capable of preparing and submitting programs from remote terminals, but he may not interact with the running program and he must wait for the program to run to completion before accessing the generated results.

The closed, conversation TSOS is the most common form of a small computer time-sharing system. Usually eight-to-sixteen users are able to program in a higher-level language, taking advantage of most of the system resources. Generally speaking, it is not the computer power which greatly limits the number of users but the amount of memory that is available to the system.

8.4. REAL-TIME CONTROL SYSTEMS

Real-time control systems are designed for handling data in a time that is consistent with the response time demanded by the process that generated the information. Such systems operate in a multiprogrammed environment with the real-time monitor controlling and supervising a large number of memory- or disk-resident programs and tasks. This control and supervision allows the tasks to share memory and disk space, I/O device handlers, and resource allocation and use.

The execution of the many tasks is determined by software priorities, hardware interrupts, timing algorithms, and requests from other tasks. Additionally, the user of the real-time system can install new tasks on-line, establish their software priority, and request their activation at any time with an automatic reactivation at a periodic interval of time thereafter.

The actual system response time for a task request depends mainly on whether or not another task is running at a higher-priority level. To prevent high-priority tasks from executing too long, a *watchdog timer* is often used to guarantee that all tasks are serviced. This timer is set at the start of each task with the maximum duration that a task may run, at a particular priority level, before being suspended or dropped.

The real-time monitor controls and executes all input and output operations. This is one of the areas of direct concern to the real-time user, since most real-time applications are characterized by a large amount of I/O. Indeed, tasks are initiated or suspended by the occurrence of some I/O operation.

8.4.1. Real-Time Programming

Programming for real-time control is generally performed in either assembly language or a higher-level language, usually FORTRAN, with extensions to allow real-time monitor calls. Program development can be done on-line with the real-time monitor, although the amount of memory available or the sophistication of the system may require off-line development.

Real-time programs rely heavily on system macro calls to schedule, queue, run, suspend, synchronize, and so on, tasks within the system. Often the data that are collected by the task is simply stored to be analyzed later under a general-purpose programming environment.

8.5. DATA-BASED SYSTEMS

Operating as a text-oriented information utility, the data-based system allows a large number of users to access a common data base. Problems such as order entry systems, automated medical records, seat reservations, information directories, and catalog searching represent prime candidates for implementation of data-based systems.

The conversational environment in which such systems are designed to operate typically demands little computer processing power, but tends to demand large storage facilities. When data are entered, the system must check its legality, decide where to file it, and select an appropriate response to be given to the user. None of these actions requires large amounts of processing.

When data are fetched and reports are generated, there will be a manipulation of information and/or the accessing of data from peripheral storage devices, in order to assemble the required data. Still, only a small amount of processing is necessary to actually format and produce the report. As a consequence of the small demand for the central processor, such system can be time-shared between a large number of users.

Although most of the data within the system may be potentially accessed at any time, large volumes of data need be available only for low-level, low frequency usage. Thus the important aspect of these systems is the availability of large-capacity peripheral storage devices such as disks, drums, and data cells. Further, an effective data management system must use the storage effectively, minimizing the amount of storage utilized and providing fast and efficient data retrieval.

8.5.1. Effective Data Management

Features and techniques used to provide effective data management include:

1. Storing data in a hierarchical tree structure so that the most frequently accessed material can be optimally located in the structure.

2. Simultaneously updating and retrieving information.

3. Allowing dynamic restructuring of a structure during use.

4. Allocating space within the system as required rather than on a static basis.

5. Optimally mapping a data structure onto a peripheral device and retrieving it or rewriting it only as needed.

6. Making the system device-independent to avoid reprogramming.

7. Operating the system in a reentrant manner so that one copy may be shared by all users.

8. Keeping most of the system and user tasks resident in memory to minimize swapping.

One of the most time-consuming aspects of developing information system programs involves the optimal interfacing of the user and the system within a particular application area. Much attention must be given to human engineering and to the modification and revision of the techniques available to the user for the storage and retrieval of system data. In addition, the protection and security of the information itself must be guaranteed.

8.5.2. Storage, Manipulation, and Access of Data

The complexity, validity, security, and variety of the data that must be handled in a data-based information system impose a number of requirements on the system. A considerable amount of information will be input in the form of text strings of variable length. In processing these data, the system will often be required to check their syntax and even determine, where possible, their semantic content against some established limits.

When the information is accessed and possibly manipulated, the system must check to see if the user has been given such privileges. Consequently, each system user must have some capability/clearance list which can be compared with the list attached to the data he wishes to access and which will prevent unauthorized access or transformation. The security and privacy so gained will often be selective and data-dependent.

One way to aid the system in protecting itself is to make it a closed, conversational time-sharing system. Users may only make responses to predefined system requests, and may not write, test, or debug general-purpose programs. Additionally, the terminals for such systems may be designed so as to require push-button responses to "canned" messages displayed by the system. Alternatively, higher-level languages may be used to construct more complex search patterns or data structuring, but such languages should be executed interpretively so that system integrity may be preserved.

8.6. COMPUTER COMMUNICATION SYSTEM

The computer communication system operates as an interconnected network of independent computer elements which communicate with each other

and share resources. As a component of these networks, the small computer generally serves as a dependent system that acts either as a data communicator or a data concentrator.

As a data communicator, the small computer serves as one of the following:

1. A device for the storing and subsequent forwarding operation of network messages.
2. A message translator and formator.
3. A controller for a large machine which it interfaces to the network
4. A data entry system for providing remote job entry to a processing facility.

As a data concentrator, the small computer serves as

1. A multiplier that processes many low-speed terminals locally, concentrating the data into one medium-speed communications line to a large system.
2. A message buffer, communications line control, and character-to-message assembler/disassembler for low-speed devices connected to it.

In both applications, the small computer offers a powerful, low-cost alternative to hard-wired communications controllers on the front end of large computer systems. And since these small computers are general purpose machines with character-handling instructions and powerful interrupt structures, they may be programmed to

1. Route messages.
2. Provide code and speed conversions.
3. Handle line and error control.
4. Compress data and format messages.
5. Automatically identify terminals and their characteristics.
6. Provide time and date stamping of messages.
7. Establish communications automatically.
8. Preanalyze messages before transmission.
9. Provide editing, tabulation, and other formating services.

8.6.1. Communications Software

Manufacturer-supplied software comes in two forms: complete systems, often referred to as *turn-key systems*, which may be installed and placed in

operation immediately, and *modular systems*, which consist of both hardware (including the computer and special communications hardware) and special-application software programs, such as device drivers and communication executives

Turn-key systems do not require the user to program the computer. Indeed, some of these systems are supplied with read-only memories, which cannot be accidentally destroyed and which have been specifically programmed to perform a fixed sequence of instructions. On the other hand, modular systems are used as a base on which the user can build special-purpose systems tailored to his needs.

Within the modular systems there will be interrupt service routines, terminal applications programs, and system control/interface packages. Utilizing these routines, the user tailors his system to his specific application, thereby minimizing the amount of hardware and software required.

REFERENCES

Many good books on the subject of operating systems can be found. However, most of them, like Watson (1970), Katzan (1973), Donovan (1972), and Organick (1972), are concerned with the features and structure of particular systems (e g , OS/360, Multics, and XDS-940). The notable exceptions are Hansen (1973), Cohen (1970), and Denning and Coffman (1973). Unfortunately, the latter three books tend to be more mathematical and theoretical in nature and may not be as useful as those geared to specific implementations. For a general treatment of modern operating systems, the reader should peruse Denning's (1971) survey article.

DIRECTORIO DE ALUMNOS DEL CURSO APLICACION DE MINICOM-

PUTADORAS 1977

FRANCISCO ALMADA V.
Sría. de Agricultura y Recursos Hidráulicos
Jefe de la Ofic. de Inform. y Datos
Reforma 35-11°
México 1, D.F.
Tel.: 591.03.83

Plutarco Elías Calles 1362-402
México 13, D.F.
TEl.: 539.05.65

ARTURO AMPUDIA PALMA
Asbestos de Méx., S.A.
Jefe de Sistemas
Carr. Circunv. Km. 12.5
Tlalnepantla, Edo. de Méx.
Tel.: 565.01.00

Horticultura 188
México 2, D.F.
Tel.: 526.18.70

SALVADOR BARRA ARIAS
Centro de Cálculo
Fac. de Ing. UNAM
Tel.: 548.65.60 E.261

Av. El Caporal Andador 6 No.25
México 22, D.F.
Tel.: 594.27.16

FRANCISCO J. BECERRA SANTIAGO

GUILLERMO CAÑIZO LECHUGA
INFONAVIT
Jefe de la Ofi. de Inform Téc.
Bca. del Mto. 280
México 20, D.F.
Tel.: 524.52.33

Lamartine 404
México 5, D.F.
Tel.: 545.20.41

ARTURO CUADROS REYES
Sría. de Agricultura y Recursos Hidráulicos
Analista de Sistemas
Reforma 35-11°
México 1, D.F.
Tel.: 591.03.83

Antonio Sola 78-1°
México 11, D.F.
Tel.: 553.88.22

MARIO DIAZ OTERO
Sría. de Agricultura y Recursos Hidráulicos
Jefe de la Ofi. de Programas
Reforma 69-10
México 1, D.F.
Tel.: 566.17.

Floricultura 239
México 2, D.F.

GUIDO EBERGENY BELGODERE
Cía. de Luz y Fza. del Centro S.A.
Jefe de Ctas. Especiales
Melchor Ocampo 171
México 17, D.F.
Tel.: 546.39.35

Claveles 211
Fracc. la Florida
Edo. de México
Tel.: 546.39.35

SERGIO FUENTES LOMELI
C.F.E.
Superintendente de Const.
Subestación Texcoco
Texcoco, Tex.
Tel. 4.10.18

Club. Cuicaealli 6
Circuito Cronistas
Cda. Satélite, Ed. de Méx.
Tel.: 572.39.80

ALBERTO GARCIA ADALID
Inst. de Invest. Eléctricas
Jefe de la Unidad de Cómputo
Leibnitz 14-901
México 5, D.F.
Tel.: 511.68.64

Villaseñor No. 5 Circ. Geógrafos
Cda. Satélite, Edo. de Méx.
Tel.: 562.16.37

JORGE GARCIA CAMACHO
Fac. de Ing. UNAM
Analista Programados
México 20, D.F.

Netzahualcoyotl 117
México 14, D.F.
Tel.: 577.13.30

ARTURO GUTIERREZ NAVARRO
PEMEX
Marina Nal. 329
México 17, D.F.
Tel.: 531.61.89

Taxqueña 1818 C-23
México 21, D.F.
Tel.: 544.93.53

MARTIN HIDALGO WONG
S.A. R.H.
Reforma 69-10°
México 1, D.F.
Tel.: 566.17.91

ZACARIAS SALVADOR LESSO ROCHA
Cía. Elevadores Otis, S.A. de C.V.
Abedules 75
México 4, D.F.
Tel.: 541.60.00 E. 132

ALEJANDRO LOPEZ ARECHICA
Aseguradora Hidalgo, S.A.
Ejercito Nal. 180
México 5, D.F.
TEL.; 592.39.07

Morelos 7-B
México 21, D.F.
Tel.: 554.96.46

ALEJANDRO LOPEZ MUÑOZ
López, Goñí y Cía. S.A.
Director General
Gante 15-116
México 1, D.F.
Tel.: 585.33.55

J. Fernández de Lizardi 60
Circ. Novelistas
Satélite, Edo. de Méx.
Tel.: 562.25.30

HERIBERTO OLGUIN ROMO
Fac. de Ing, UNAM
TEL.: 548.65.60 E. 261

Odontología 69-401
México 20, D.F.
Tel.: 548.18.60

HECTOR RIVERA MARTINEZ
Centro de Cálculo
Fac. de Ing. UNAM

Campo Encantada 39
México 16, D.F.
Tel.: 352.25.51

CARLOS ROJAS TOLEDO
SARH
Dirección de Construcción

JOSE MARIA SALCEDO LOREDO
Srία. de Agricultura y Recursos Hidráulicos
Jefe de la Ofic. de Programas de Obra
Reforma 69 -10°
México 1, D.F.
Tel.: 566.17.91

Nte. 64 No. 4789-18
México 16, D.F.

GUADALUPE ZAMORANO LIMON
Srία. de Agricultura y Recursos Hidráulicos
Reforma 69-10°
México 1, D.F.
Tel.: 535.13.26

U. Habit. Patos Mz. 4-Edif. O Depto.1
México 1, D.F.
Tel.: 522.57.84

MANUEL ZARATE CHAVEZ
Cía. Elevadores Otis, S.A. de C.V.
Abedules 075
Sta. Ma. Insurgentes
México 4, D.F.
Tel.: 541.60.00

Calle Encinos Mnz. 252 L.8
Villa de las Flores
Tel.: 541.40.00 E. 132

'edcs. 9, XI, 77.