I-85
3/65

PROGRAMMED DATA PROCESSOR - 8

PDP-8 COURSE WORKBOOK

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

PDP-8 COURSE WORKBOOK

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

Copyright 1965 by Digital Equipment Corporation

The PDP-8 Course Workbook was written and edited by the training department of the Digital Equipment Corporation.

CONTENTS

INTRODUCTION

This workbook is written to assist the student in understanding PDP-8 computer systems. By using it as a class text, study guide, and workbook in conjunction with the PDP-8 Handbook (F-85) and Program Writeups, all aspects of the course can be covered and the fundamental concepts of programming the PDP-8 can be mastered.

The workbook describes the PDP-8 and gives sample problems in programming and homework assignments. It is suggested that the student solve the problems and retain them for future reference.

This book is accompanied by a series of lectures and laboratory periods designed to further assist the student in understanding the operating features of the PDP-8 system. During these periods, the uses and capabilities of the software packages provided with each computer installation are discussed. By the end of the course, each student should be able to use all the equipment in the PDP-8 system.

CHAPTER 1

THE CONCEPT OF A DIGITAL COMPUTER

A digital computer can be defined as a device capable of automatically carrying out a sequence of operations on data expressed in discrete (digital) form. Each operation is one of four kinds:

1. An input of data from a device outside the computer.

2. An arithmetic or processing operation on data stored within the computer.

3. An output of data from the computer to the external equipment.

4. A sequence-determining operation in which the computer chooses which of two (or more) alternative sequences of operation it will perform. This choice is based upon the condition of the data in the computer or the status of peripheral equipment at the time of sequence determination.

Using these four operations, present-day computers are capable of performing extended and varied sequences of arithmetic operations. As such, they are very flexible and useful in rapidly executing computations which would take months or years of tedious work if done by hand. However, even in the solution of simple problems, the machine must be told the step-by-step procedure for arriving at the correct solution.

For each machine, such as the PDP-8, there is given set of instructions which may be used to process data and solve problems. The circuitry and wiring within the computer itself determine how each instruction is executed, and the order in which the instructions are carried out is determined by a stored program.

When digital computers, such as the PDP-8, execute an instruction, they do it in discrete steps called cycles. During the first cycle, the fetch cycle, the computer obtains the instruction and determines how to perform it. If the instruction does not require an operand from memory, the instruction will be executed during the fetch cycle. For all instructions which refer to a location in the memory, a second cycle must be performed. This execute cycle actually performs the operation specified by the instruction. In the PDP-8, the cycle time is 1.6 microseconds.

Since digital computers operate at high speeds, the sequence of instructions must be available within microseconds. The essential elements of a system which can prestore data are shown in Figure 1. The instructions are fed into the machine via the input and stored in the memory. The operations and data manipulations are performed in the arithmetic element, and processed data is fed to the output element. All elements can be controlled manually from the console or automatically by the computer itself. The control element is not clearly identifiable. In a machine, control is quite dispersed and often involves a heterogeneous mass of circuitry.

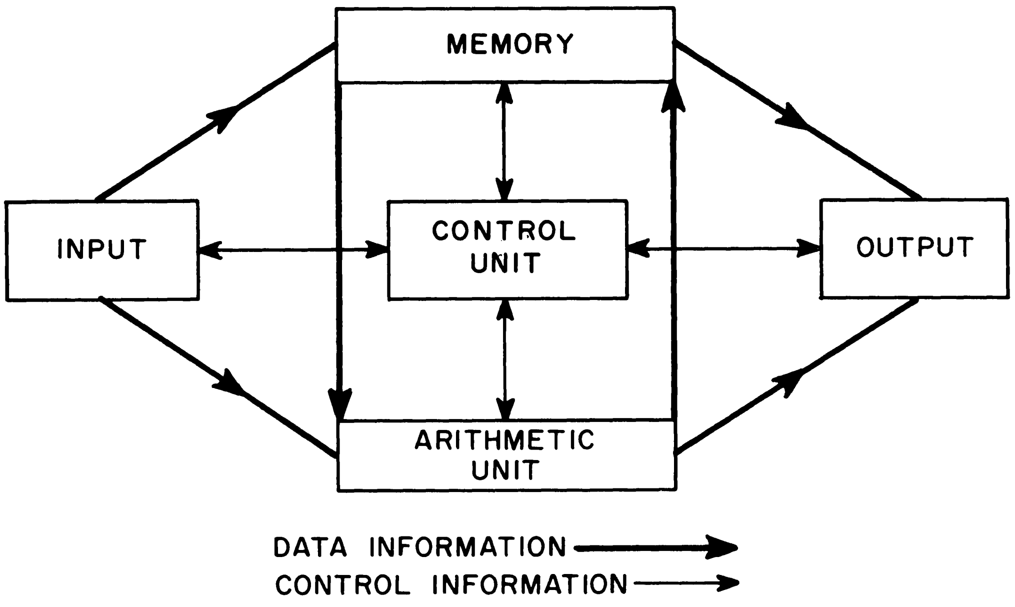


Figure 1   Main Elements of PDP-8 System

This workbook describes the available means for storing a sequence of instructions in memory and causing the computer to execute these instructions.

CHAPTER 2

THE CONCEPTS OF DIGITAL COMPUTER PROGRAMMING

A. GENERAL

Each digital computer has a special code in machine language which tells the computer what to do. The specific codes for the PDP-8 will be described in a later chapter. For the moment, it would be instructive to review the fundamental concepts of digital computer programming. If you do not understand the following concepts, it might be useful to obtain a copy of a reference book in basic programming.

Each area in the memory of the computer which can contain the unit of information referred to as a word (12 pieces of binary information) is called a register. Each register has a number associated with it, called the address, which determines its position in memory.

B. STRAIGHT-LINE PROGRAMMING

This type of programming deals with the step-by-step, uninterrupted sequence of program steps. An example of straight-line programming is shown below:

```
Example:  Program the equation A - B(4-3D)
     0/   Load d
     1/   Mul f (f contains 3)
     2/   Complement
     3/   Add g (g contains 4)
     4/   Mul b
     5/   Complement
     6/   Add a
     7/   Halt
```

NOTE: Each instruction as shown would not necessarily occupy one location in memory. They are shown this way for descriptive purposes only.

In the above example, registers f and g contain the constants 3 and 4, respectively. Registers a, b, and d contain the parameters A, B, and D, respectively. The registers 0 through 7 contain the program itself.

It is important to note that the program will be stored in memory along with the data to be used in the solution of the problem. The computer cannot tell the difference between stored data and instructions. The programmer enables the computer to differentiate between the two.

C. PROGRAM LOOPS

Sometimes it is necessary to repeat a portion of a program each time changing only a small portion of the original program to obtain different results. By using a program loop, the programmer can obtain this repetitive operation.

Example: Solve the equation 35(A+B) for all integral values of A starting with 0 and store the solutions in consecutive registers starting at address h.

```
 0/   Load f (f contains 0)
 1/   Add b (b contains B)
 2/   Mul g (g contains 35)
 3/   Deposit into h
 4/   Load 3
 5/   Add c (c contains 1)
 6/   Deposit into 3
 7/   Load f
10/   Add c
11/   Deposit into f
12/   Jump to 0
```

The first four instructions actually solve the problem. The next three instructions increment the address of the storage register by one, and the following three instructions increment the value of A by one. Note that the program steps can be treated as data since they may be modified (see steps 5 and 10). The jump instruction allows the computer to repeat the above routine endlessly.

The jump instruction allows the programmer to break the normal sequence of program steps. In the above example, the jump back to the beginning is said to be unconditional because each time the control circuitry analyzes the instruction, it will always execute the same command, "Return to the beginning." The jump instruction can transfer program execution to any location in the sequence of instructions.

D. PROGRAM BRANCHING

As shown in the previous example, looping is useful in repetitive calculations. It also is used to loop a predetermined number of times and then continue in the main program. This procedure is tallying.

```
Example:  Find the 10th power of 2
     0/   Load f (f contains -10, the tally number)
     1/   Deposit into g
     2/   Load m (m contains 1)
     3/   Deposit into k
     4/   Load k
     5/   Mul h (h contains 2)
     6/   Deposit into k
     7/   Load g (g contains -10)
    10/   Add m (m contains 1)
    11/   Deposit into g
    12/   Skip if zero
    13/   Jump to 4
    14/   Continue
```

In the example above, the first four instructions initialize the routine. In this case, for only one pass through the sequence, they could be left out completely; however, they are inserted to show the necessary steps to initialize the tallying routine. A general rule is to initialize any register which is changed during program execution. The next three steps do the actual program work on the data, and the next four steps check to see if the routine has executed ten loops. If it has not, the tally number will be negative and the routine will continue to loop. As soon as the tally number becomes zero, the program will skip instruction 13 and branch back into the main program. The branch is also useful as it enables the computer to make a decision. For example, if the partial answer to a problem is positive, the computer will continue with the main sequence. If, the partial answer is negative, the computer will jump to a different portion of the program.

Branch instructions may be referred to as conditional skip instructions. They allow the computer to skip or not skip an instruction depending upon existing conditions.

E. SYMBOLIC ADDRESSING

Symbolic addressing is not a concept in programming but is a means by which the programmer can write understandable programs.

The computer uses only machine language and requires that all data and instruction steps be stored in memory in binary numbers. To avoid the use of only ones and zeros in writing programs, it is useful to have a computer program, as the PAL (PDP-8 Assembler), which can interpret mnemonic codes and symbolic addresses by translating them into their binary equivalent.

A symbolic address is an address specified by a letter designation rather than a specific numerical location. The PAL Assembler converts these letters into machine language numbers. The program can begin at any location because all program locations are relative to each other.

A more complete treatment of symbolic addressing on the PDP-8 will follow in Chapter 5. Also, the 3-letter mnemonic names for the instructions themselves (TAD, DCA, ISZ, JMP) will be discussed in a later chapter.

Example: Symbolic Addressing

```
A,  ADD K1            /GET NUMBER
    DEPOSIT INTO K2   /STORE IT
D,  SUBTRACT K3       /SUBTRACT
    MULTIPLY BY K2    /MULTIPLY
    JUMP TO A         /KEEP LOOPING
```

```
K1,  230
K2,  0
K3,  56
```

The above example indicates the format of a program using symbolic addressing.

F. FLOW CHART TECHNIQUES

1. General

Flow charting is a technique to diagramatically represent an initial phase of creating a program. The detail or extent of flow charting is entirely up to the programmer. An experienced programmer might represent a program in two or three distinct steps; whereas a novice might gain more in thinking through a program by using a block for each basic program function. In either case, a flow-chart block should be a condensation, not a representation of each program step.

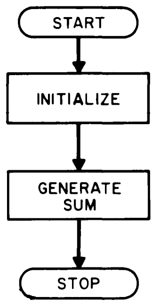
2. Flow Chart Symbols

▭	Used to represent a program function or operation; one entry, one exit.
○	Used to connect two program points; a convenience symbol or chart label, not a program operation.
◇	A decision symbol, one entry, multiple exits, a conditional branch.
⬭	Beginning or stopping point of a program.
→	Representation of flow.

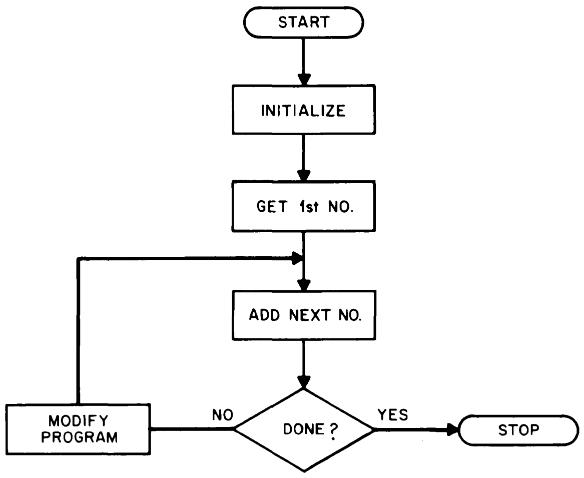
3. Example

Flow chart of a routine to add numbers together.

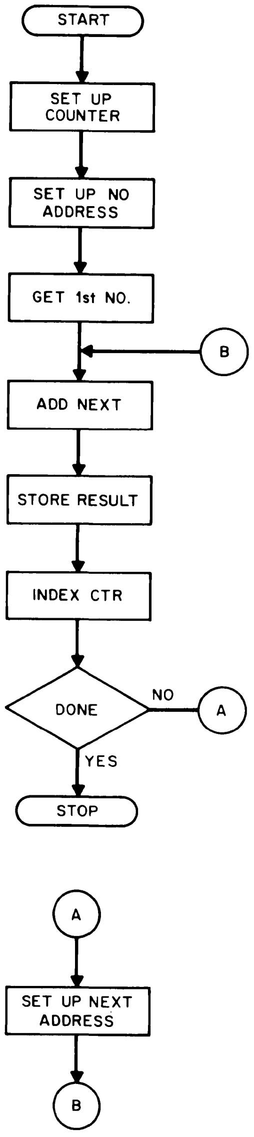
Case I



Case II



Case III



Case I represents a brief diagram. Case II and Case III present more detailed diagrams.

G. SUMMARY

Remember that the function of a flow chart is to assist the programmer. If it is not needed, it may be omitted. A flow chart clearly represents all paths of program flow and gives the programmer a tool to do his program coding.

REVIEW QUESTIONS - CHAPTER 2

These questions are not necessarily covered in the workbook but review basic terms and ideas that the student should understand before continuing with the workbook.

1. What is a word? A bit? A register? An address? A location?
2. What is the octal numbering system?
3. If a computer has 4096 memory locations, what is the largest address in octal?
4. List some common input devices for a computer.
5. In location 340 is stored the instruction tad 350.

   In location 350 is stored the number 0005.

   If a program is operating and reaches location 341, what will be the contents of the accumulator?

   What is the operand of the instruction in 340?
6. What are the answers to the following problems?

$$\begin{array}{r} 56_{(8)} \\ +\,17_{(8)} \\ \hline \end{array} \qquad \begin{array}{r} 101_{(10)} \\ +\,101_{(2)} \\ \hline \end{array}$$

CHAPTER 3

INTRODUCTION TO PDP-8 SYSTEMS

A. GENERAL

The Digital Equipment Corporation PDP-8 computer is a high-speed, solid-state, digital computer capable of handling up to 64 input-output devices. It uses a 12-bit word and operates on 2's complement arithmetic. The input-output devices used most often are a perforated-tape reader, a perforated-tape punch and an on-line/off-line teleprinter (Teletype Unit ASR-33). The standard memory for the PDP-8 consists of $4096_{(10)}$ words and can be expanded in increments of $4096_{(10)}$ to $32,768_{(10)}$ words. The basic cycle time is 1.6 microseconds.

B. CENTRAL PROCESSOR ELEMENTS

1. Control Element

The control element is widely dispersed and not precisely definable. Components that fall under the control category are associated with most of the other elements and peripheral equipment. The control element governs the complete operation of the prescribed instruction, and initiation of input-output commands.

2. Arithmetic Element

The arithmetic element is the central part of the computer and may be compared with a desk calculator. It can add (on demand by an instruction) a given number to whatever number currently is contained in itself. It can make simple logical decisions based on the contents of the arithmetic unit. Finally, the arithmetic element can mechanize operations which have no counterpart in ordinary arithmetic. For example, it performs Boolean AND and OR operations. The arithmetic element is a manipulator of numbers and is composed of the accumulator and the link.

a. Accumulator

The accumulator is a 12-bit register that performs addition, logic (negation, and AND), and counting operations. It is connected to the memory module through the memory buffer and can transfer or receive information from the memory module and transfers information from the console. Information from the standard I/O devices and most other input-output devices are transferred into the accumulator through the input mixer. When the data interrupt is used with a high-speed device (drum, mag tape, DECtape), the accumulator is bypassed, and information goes directly through the memory buffer.

b. Link

The link is a 1-bit register used in conjunction with the AC. The link may be rotated along with the AC either to the right or to the left. It may be considered to be left of AC_0 and right of AC_{11}. It may be cleared, complemented, and sensed. The latter feature makes it available as a program flag. The link may be regarded as the overflow register for arithmetic addition. If overflow occurs, the link is complemented.

3. <u>Memory</u>

The memory is not a part of the central processor. However, it is included here as a preview to the description of its associated registers, which are included in the central processor. The memory contains stored information or data to be processed and the instructions of the programs to do the processing. The magnetic core memory of a standard PDP-8 holds 4096 words of 12 binary bits. The memory capacity may be expanded in increments of 4096 words to 32,768 words. Memory may be further supplemented by magnetic tape unit, drum options, or the addressable tape, DECtape.

4. <u>Memory Addressing Element</u>

This element is composed of two discrete units.

a. Memory Address Register

The memory address register contains 12 bits and controls the access to memory while each instruction is being executed. During every 1.6-microsecond memory cycle, the memory address register addresses a single core register in the computer memory. The memory cycle is divided into two portions, a read portion and a write portion. During the read portion, a single 12-bit computer word is read from the addressed core register into the memory buffer. During the write portion of the cycle, the word contained in the memory buffer is written back into the addressed core register. For both read and write portions of the memory cycle, the addressed core register is specified by the contents of the memory address register.

b. Program Counter Register

The program counter is a 12-bit register and may be modified under program control. It indicates to the control circuitry the location of the next instruction to be executed. It can be preset from the switch register by the load address switch. In normal operation, it starts at its preset value and indexes by one each time a program instruction is executed. It can be reset during program execution by a jump instruction. The contents of the program counter are displayed on the console.

5. Memory Buffer Register

The memory buffer register contains 12 bits and is connected to core memory. All information which enters and leaves the memory must come through this register. Information can be transferred between the memory buffer, the accumulator, the instruction register, and the program counter.

6. Instruction Register

The instruction register contains the 3-bit operation code of the instruction currently being performed by the computer. Information enters this register from the MB.

C. INPUT-OUTPUT CONTROL

The input-output control provides control circuitry enabling input-output (I/O) devices to transmit information into or receive information from the central processor. A brief summary of this unit is given in Section B, Chapter 1 of the PDP-8 Handbook (F-85). The discussion of this unit, from a programming point of view, will be left to the chapters on the I/O devices and interrupt facility.

D. COMPUTER CONTROL STATES

The computer performs one memory cycle (1.6 microseconds) in one and only one of four states. These states are:

Fetch
Defer
Execute
Break

The four states of the computer are described in detail in Chapter 1 of F-85.

CHAPTER 4

BASIC PDP-8 INSTRUCTION LIST

A. GENERAL

The list of available instructions for the PDP-8 is shown in Appendix 1 of the PDP-8 Handbook (F-85). The instructions are grouped according to their function. A description of what each instruction does can be found in Section A, Chapter 2 of the PDP-8 Handbook. There are two basic groups of instructions: memory reference and augmented. Memory reference instructions require an operand; augmented instructions do not require an operand.

The addressing technique requires a complete understanding of the page addressing of the PDP-8.

1. Page Addressing

The memory of the PDP-8 is divided into blocks of 128_{10} or 200_8 locations called pages. These pages are not physical pages in memory nor are the areas of the mem- blocked off in any manner. Pages occur as a result of the size of the address portion of the instruction. In order to minimize needless confusion, a number of definitions are given below.

a. Current Page

The page containing the instruction being executed. This page is determined by bits 0-4 of the program counter.

b. Page Address

A 7-bit number contained in bits 5-11 of an instruction which designates one of 200_8 core memory locations. Bit 4 of the instruction indicates the page to which the page address refers.

Bit 4	Page
0	Page 0
1	Current page

Thus, bits 5-11 specify one of the 200_8 locations on the page determined by bit 4.

c. Absolute Address

A 12-bit number used to address any location in core memory.

d. Effective Address

The address of the operand as specified in an instruction or by an absolute address.

To the programmer, the page feature means that there are 400_8 locations available for direct addressing with memory reference instructions. The programmer has 200_8 locations on the page where the program is running and all 200_8 locations of page 0 (absolute locations 0_8-177_8) directly addressable. Any location in the entire 4K of memory may be addressed by the use of indirect addressing in the instruction. The absolute address of the operand is placed in the location specified by bits 4-11 of the instruction.

Four methods of obtaining the operand at the effective address are used as specified by combinations of bits 3 and 4.

Bit 3	Bit 4	Effective Address
0	0	The operand is on page 0 at the address specified by bits 5 through 11.
0	1	The operand is on the current page at the page address specified by bits 5 through 11.
1	0	The absolute address of the operand is taken from the contents of the location on page 0 designated by bits 5 through 11.

Bit 3	Bit 4	Effective Address
1	1	The absolute address of the operand is taken from the contents of the location on the current page at the page address specified by bits 5 through 11.

The augmented instructions in the PDP-8 require more understanding than the specification of each instruction. Their microprogramming feature lends itself to shorter, more powerful programs.

2. Operate Group

The operate group, in general, performs its actions on the accumulator and link. These instructions do not require a memory reference.

The instructions may be referred to as microinstructions because the performance of an operation is specified by the presence of a 1 in a bit of the instruction. Specific instructions may be logically ORed together to perform a number of operations. The event times indicate the order in which the operations are performed. Two instructions occurring at the same event time can be combined subject to the restrictions given in Section A, Chapter 2 of the PDP-8 Handbook.

Within the operate group itself there are two groups.

Group 1 is principally for clear, complement, rotate, and increment instructions. This group is designated by a 0 in bit 3 of the operate instruction.

Group 2 is used in checking the contents of the accumulator and link to determine sequence of operations to be followed. This group is designated by a 1 in bit 3. The programmer never needs to worry about the bits being in the correct place when using the assembler; mnemonic symbols for the instruction takes care of this. The instructions from Group 1 and Group 2 can never be combined because of the use of bit 3.

Within Group 2, there are two groups of skip instructions. They may be referred to as Group 2a and Group 2b.

<u>Group 2a</u>	<u>Group 2b</u>
SMA	SPA
SZA	SNA
SNL	SZL

Group 2a is designated by a 0 in bit 8; Group 2b, by a 1 in bit 8. Instructions from Group 2a and 2b should never be combined, for the same reason as above.

If the programmer does combine legal skip instructions, it is important to note the conditions under which a skip will occur.

a. Group 2a

If these skips are combined in a single instruction, the inclusive OR of the conditions determines the skip.

SZA SNL

The next instruction is skipped if:

the AC contains 0000, or
the link is a 1, or
both conditions exist.

b. Group 2b

If the skips are combined in a single instruction, the logical AND of the conditions determines the skip.

SNA SZL

The next instruction is skipped <u>only</u> if:

the AC differs from 0000 and the link is 0.

One combination is a skip instruction combined with a CLA instruction. The condition is sensed and the accumulator is cleared to enable a TAD instruction to load the AC in the next instruction.

3. Input-Output Group

The instructions in this class are used to enable information transfers between the central processor and external devices via the interface.

The mass of circuitry which controls the transfer of information to and from the computer is called the input-output control. Specific IOT commands are discussed in a later chapter of this workbook.

REVIEW QUESTIONS - CHAPTER 4

1. If you perform a CMA instruction, are you dealing with 1's or 2's complement arithmetic?
2. Describe the action and use of:
   a. The JMS instruction
   b. the AND instruction
   c. The combined use of DCA and TAD instruction
3. a. What is the contents of the AC after a CLA CMA instruction?
   b. What is the contents of the register containing the microinstruction SPA SNL?
   Would this accomplish the desired function? If not, why not?

In the following programs, the number followed by a slash indicates the address where the given instruction or number is located. The address following any mnemonic instruction is an absolute address.

4. The following program adds some numbers together, stores the total away, and halts. What is the sum and in what location is it stored?

```
1000/          CLA CLL
1001/          TAD 1020
1002/          IAC
```

```
1003/    1450
1004/    RAL
1005/    SNL
1006/    5202
1007/    RAR
1010/    3603
1011/    HLT
1020/    4012
1021/    2000
  50/    1021
```

Change location 1020/ from 4012 to 1012 and do the problem again.

5. The original contents of AC is 1234; the link is 1. The following program is executed. Will the program ever stop? If so, what are the final contents of the AC and the link?

```
1000/    RTL
1001/    DCA 1021
1002/    ISZ 1017
1003/    TAD 1020
1004/    NOP
1005/    DCA 1011
1006/    TAD 1017
1007/    DCA 1014
1010/    TAD 1021
1011/    HLT
1012/    IAC
1013/    RAR
1014/    NOP
1015/    RAR
1016/    JMP 1015
1017/    7401
1020/    7006
1021/    HLT
```

6. The following program is executed. What are the contents of the PC when the computer halts?

```
200/          CLA CMA
201/          DCA 213
202/          ISZ 213
203/          HLT
204/          TAD 214
205/          TAD 215
206/          TAD 216
207/          DCA 221
210/          TAD 201
211/          TAD 217
212/          DCA 213
213/          IAC
214/          NOP
215/          2
216/          400
217/          TAD 5
220/          HLT
221/          JMP 200
```

CHAPTER 5

INTRODUCTION TO THE PDP-8 ASSEMBLER (PAL)

A. GENERAL

The PAL Assembler is a program which translates a symbolic source program tape into a form suitable for execution by the PDP-8 and punches out this form on a binary output tape (binary object tape). The source program permits the programmer to express the operations for the computer to perform in a more legible form than the binary code in which the PDP-8 must receive its instructions.

By using this assembler program (PAL), the programmer may use mnemonic names or symbols for the instructions and assign symbolic addresses in the program. For example, if the programmer uses the characters DCA in his source program, the Assembler will translate them to the value 3000_8 as stored in memory. The Assembler process substitutes the binary value of each symbol for the symbol itself and punches it out on the binary output tape.

During assembly, the assembler program keeps a current location counter, which indicates the address of the register where the next instruction or data word will be stored. Do not confuse this with the program counter which is in the PDP-8 machine. For each word assembled, this address is increased by one. The initial address may be preset to allow assembly at any locations and may be reset at any time during assembly by the appropriate control character in the source language program. Unless otherwise specified, assembly starts in location 200.

The Assembler performs its action in two passes (the source language tape is processed twice to produce the one binary object tape). Certain functions which cannot be handled on the first pass must be handled by the second pass. The output from Pass 1 is a symbol table typed on the ASR-33 showing the tags used in the program and their corresponding octal values, and any error indications.

During Pass 2, the values of the tags are inserted in the corresponding symbolic address within the memory reference instruction, and the binary output tape is generated. The program writeup of the PAL Assembler is available from the program library and should be used during the writing and subsequent assembling of programs.

B. HOW TO ASSEMBLE A SYMBOLIC PROGRAM WITH PAL

The detailed description of the assembly process is given in the program writeup. For the programmer's ease of operation, the assembly procedure is shown as a flow diagram on the following page.

C. MISCELLANEOUS

1. Flow Diagrams

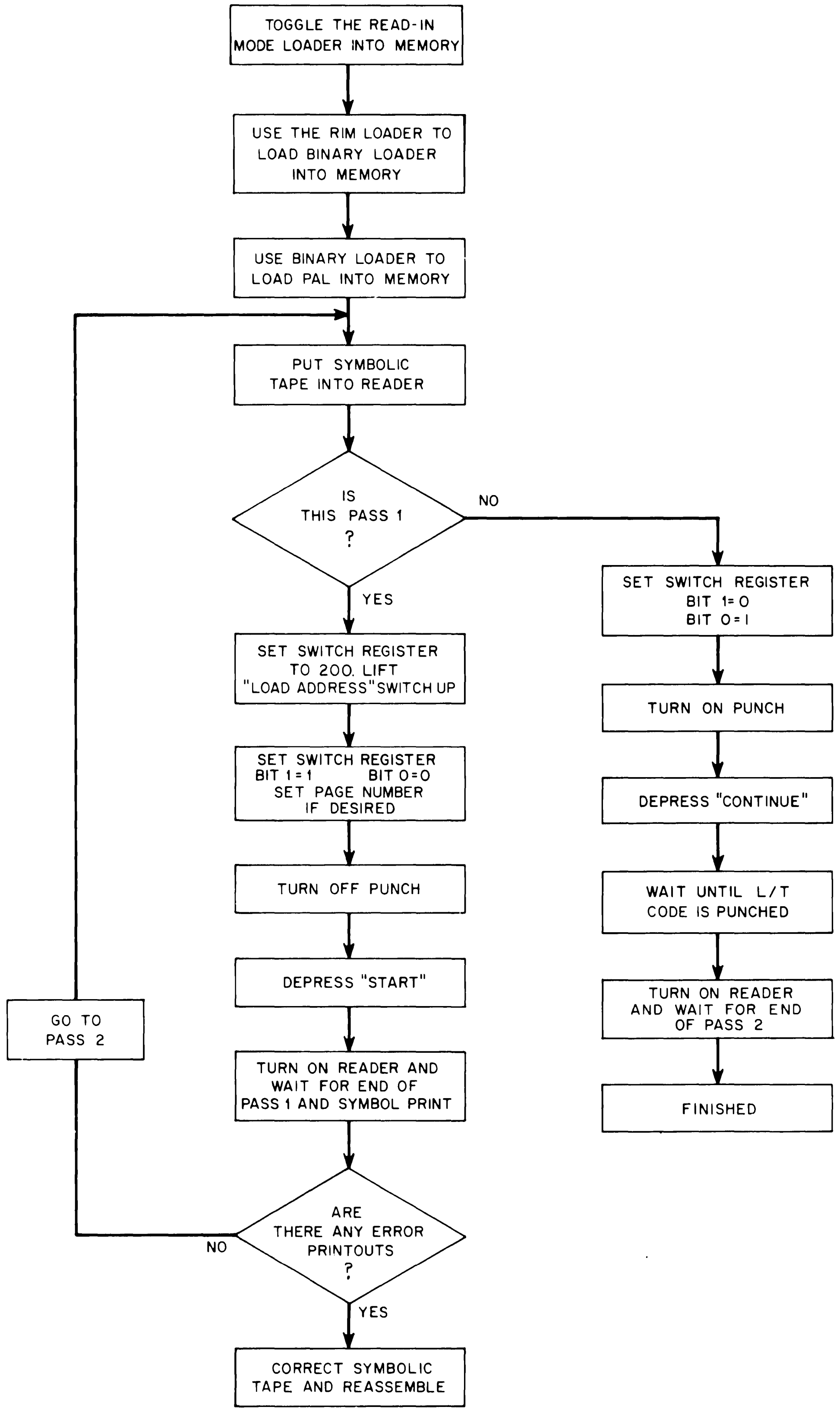
A flow diagram will save the programmer time in writing and executing his program. It will save space in memory if he draws a flow diagram which clearly illustrates the general sequence of operations, the point of which the various sequence determinations should be made, and the program flow following the sequence determination. It is suggested that no coding be done in the flow diagram; only verbal descriptions of operations in the program should be used. Examples of flow diagrams will be given throughout the rest of this workbook.

2. Program Structure

Inexperienced programmers, should draw a flow diagram that clearly illustrates the sequential operation of the program. Then, the instructions should be written in assembler language to accomplish the function described in each block of the flow diagram.

For short, simple programs, such as the ones for homework, the organization in memory is straightforward.

There are three distinct portions to each program. For ease of debugging, keep the areas separate and well marked in the symbolic program. The portions are:


TOGGLE THE READ-IN MODE LOADER INTO MEMORY
USE THE RIM LOADER TO LOAD BINARY LOADER INTO MEMORY
USE BINARY LOADER TO LOAD PAL INTO MEMORY
PUT SYMBOLIC TAPE INTO READER
IS THIS PASS 1 ?
NO
YES
SET SWITCH REGISTER TO 200. LIFT "LOAD ADDRESS" SWITCH UP
SET SWITCH REGISTER BIT 1=1 BIT 0=0 SET PAGE NUMBER IF DESIRED
TURN OFF PUNCH
DEPRESS "START"
TURN ON READER AND WAIT FOR END OF PASS 1 AND SYMBOL PRINT
ARE THERE ANY ERROR PRINTOUTS ?
NO
YES
CORRECT SYMBOLIC TAPE AND REASSEMBLE
GO TO PASS 2
SET SWITCH REGISTER BIT 1=0 BIT 0=1
TURN ON PUNCH
DEPRESS "CONTINUE"
WAIT UNTIL L/T CODE IS PUNCHED
TURN ON READER AND WAIT FOR END OF PASS 2
FINISHED

Initialization

Program

Constants and Variables

The program would be similar to the following for short programs:

(1) Initialization
(2) Program
(3) Constants Variables

Start by writing Part 2 first. As the program is written, generate the constants and variables as they are needed. Any storage register which is reserved for data that may change during the program is referred to as a variable. Then, after the program is written, go back to initialize any registers or instructions which have been changed or modified during the programs. Every program should be written so that it could be run a second time and give the correct result. All program examples will contain three segments clearly indicated.

3. Use of Page 0

Since page 0 is directly addressable from any core page in memory, often-used portions of the program should be put on page 0.

Examples of such portions would be:

Small subroutines (typein and typeout)

Variables

Constants

Temporary storage registers

Subroutine pointers

If the program is contained completely on one page, there is no real advantage to the use of page 0. However, if the program takes several pages, the use of page 0 will be of great value.

D. LOADING PROCEDURES

Since the student in the programming class will soon be running programs on the PDP-8, it is wise to follow the steps in loading and running program. A short description of the loaders and tapes is given below.

1. Read-In Mode (RIM) Loader

The RIM Loader is a program used to load a RIM format tape from the reader into memory.

The RIM Loader must be toggled initially into memory from the console of the computer but should never need to be toggled again.

In order to avoid destruction by the maintenance program (Maindec series), the RIM Loader should be toggled into locations 20 through 40. The program and description for loading it are found in F-85, Appendix 5. The RIM Loader can then be used to load the same program into the top end of memory to free page 0 for use in programming. Thus the RIM Loader may be located in locations:

0020-0040	(for maintenance program)
7700-7720	(other)

The starting address for the respective RIM Loaders is 20, or 7700.

To load a tape in RIM Format using the RIM Loader:

a. Check to see if the RIM Loader program is correct in memory. If not, toggle it in.

b. Put RIM Format tape in reader.

c. Always put leader/trailer code over reader head - never blank tape.

d. Set SR to 20, or 7700 depending upon the location of the loader program.

e. Depress the LOAD ADDRESS switch.

f. Depress the START switch.

g. Computer is now running and waiting to load any program from any tape which is in RIM Format <u>regardless</u> of the starting address of that program.

h. Turn on the reader and wait until tape is completely read in. When the reader stops, the program is in memory.

From the programmer's viewpoint, the RIM Loader is <u>usually</u> used to load the Binary Format Loader.

2. <u>Binary Format (BIN) Loader</u>

The BIN Loader is a <u>program</u> used to load a PAL Binary Format tape from the reader into memory.

The BIN Loader tape is loaded by the RIM Loader program.
A description of the BIN Loader is given in Appendix 5 of F-85.
The BIN Loader is loaded into location 7721-7777.
The starting address for the BIN Loader is 7777.

To load a tape in PAL Binary Format using the BIN Loader:

a. Put PAL Binary Format tape into reader.

b. Set SR to 7777.

c. Depress LOAD ADDRESS switch.

d. Depress START switch.

e. The computer is now running and waiting to load a program from any tape that is in PAL Binary Format regardless of the starting address of that program.

f. Turn on the reader and wait until the tape is read in. When the reader stops, the program is in memory.

NOTE: If the AC does not contain 0 when the computer halts, the tape has read in incorrectly and should be loaded again.

The BIN Loader is used to load:

PAL

Symbolic Tape Editor

Octal Debugging Tape

Any tape from Pass 2 of the PAL Assembler

Any other tape from the program library in PAL Binary Format

When the program is loaded correctly from the PAL Binary Format tape into memory, determine the starting address (SA) of the program from the label on the tape.

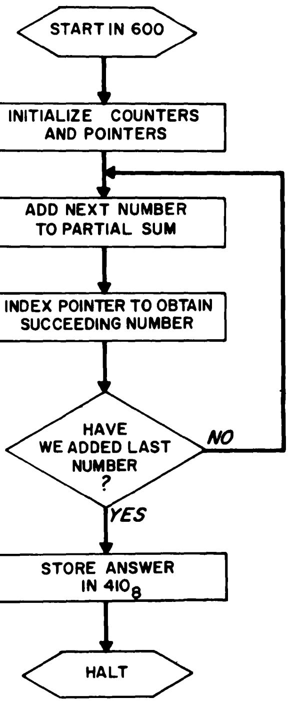
Start the program which was just read in by putting the starting address (SA) into the switch register, depress LOAD ADDRESS and START.

E. EXAMPLE PROGRAM FOR ASSEMBLER

1. Problem

Write a routine using a program loop and a branch condition to add up numbers stored in location 200_8 through 207_8, and store the answer in location 410_8. Write the program starting in location 600.

2. Flow Diagram


START IN 600
INITIALIZE COUNTERS AND POINTERS
ADD NEXT NUMBER TO PARTIAL SUM
INDEX POINTER TO OBTAIN SUCCEEDING NUMBER
HAVE WE ADDED LAST NUMBER ?
NO
YES
STORE ANSWER IN 410$_8$
HALT

3. Program Listing

```
/ADD UP NUMBERS
*600
BEGN, HLT
/LOAD 200-207 WITH NUMBERS.  HIT "CONTINUE" ON CONSOLE
/
/THE NEXT FIVE INSTRUCTIONS INITIALIZE THE ROUTINE
      CLA            /CLEAR THE ACCUMULATOR
      TAD M10        /LOAD AC WITH -10 TO INITIALIZE TALLY
      DCA CNTR       /DEPOSIT IN COUNTER
      TAD 2HUN       /LOAD AC WITH FIRST ADDRESS IN BUFFER
      DCA BUFF
/
/THE NEXT SEVEN INSTRUCTIONS ARE THE PROGRAM ITSELF
ADDR, TAD   I BUFF   /ADD NEXT NUMBER FROM BUFFER
      ISZ   BUFF     /INDEX POINTER
      ISZ   CNTR     /INDEX COUNTER.  IS IT ZERO?
      JMP   ADDR     /NO, JUMP TO ADD NEXT NUMBER
      DCA   I ANSR   /YES, DEPOSIT TOTAL IN 410
      HLT            /STOP. HIT "CONTINUE" TO REPEAT PROGRAM
      JMP   BEGN+1
```

```
/
/THE NEXT THREE REGISTERS CONTAIN THE CONSTANTS FOR THE PROGRAM
M10,     0-10          /NEGATIVE TALLY NUMBER
2HUN,    200           /FIRST ADDRESS IN BUFFER
ANSR,    410
/
/THE NEXT TWO REGISTERS ARE RESERVED FOR THE VARIABLES IN THE PROGRAM
CNTR,    0
BUFF,    0
/
$
```

4. Output From the Assembler

```
PASS 1:  SYMBOL TABLE
BEGN     600
M10      615
CNTR     620
2HUN     616
BUFF     621
ADDR     606
ANSR     617

PASS 2:  BINARY TAPE AND "PRINTED MESSAGE (?)"

BF<:

*<*?& ?7
```

The printed message output has no relevance to the programmer. The binary codes which are being punched on the tape in binary format simply coincide with the codes for the teleprinter output. The same buffer is used for output to the punch and printer.

REVIEW QUESTIONS - CHAPTER 5

1. Ten numbers are stored in locations 200_8 through 211_8. The letter B is stored in 212_8. Write a routine starting in location 600_8 which will determine how many of the ten numbers are larger than B and store this quantity in location 500_8. Use a flow diagram to lay out the program.

2. Deposit 7777_8 in all odd memory locations, and 0000 in all even memory locations, starting with the address set on the switch register, and ending 1000_8 locations later. Use a flow diagram as above. Assume that the SR setting is such that the program will not be destroyed.

3. Write a subroutine called MOVE to move a block of data from one area in memory to another area in memory. The subroutine is entered with the number of words to be relocated in the AC. The initial address of the existing block of data and the initial address of the new location are located in the two registers immediately following the JMS MOVE instruction. Return to the main program with the AC cleared. Use auto-index registers.

```
          .
          .
          .
     TAD N              /NUMBER OF WORDS TO BE MOVED IN AC
     JMS MOVE           /SUBROUTINE CALLED
     IA1                /INITIAL ADDRESS OF ORIGINAL DATA BLOCK
     IA2                /INITIAL ADDRESS OF MOVED DATA BLOCK
          .             /RETURN WITH AC CLEAR
          .
          .
```

CHAPTER 6

INTRODUCTION TO THE PDP-8 SYMBOLIC TAPE EDITOR

A. GENERAL

The PDP-8 Symbolic Tape Editor allows the programmer to correct or generate symbolic tapes in ASCII code with the PDP-8 computer from the keyboard of the ASR-33 teleprinter.

Upon command, the Symbolic Tape Editor will punch out the corrected or generated symbolic program on paper tape in a format acceptable to the PAL Assembler. This program greatly reduces the tedious correction of symbolic tapes using the ASR-33 off line.

For a complete description of the Editor, see the PDP-8 Symbolic Tape Editor writeup.

B. MODES OF OPERATION

There are two ways in which characters typed on the ASR-33 are interpreted, depending upon the mode of the Editor. There are two modes of operation—command and text mode.

1. Command Mode

Characters typed by the programmer on the ASR-33 will be interpreted as commands telling the Editor to perform, or enable the programmer to perform, operations on the text of the program. Commands to the Editor must take one of the following forms with 0, 1, or 2 arguments where ↵ is the nonprinting character representing RETURN.

```
    X↵
   nX↵
 n,mX↵
```

If an incorrect form is typed, the Editor will type back a ? and return to listen for the next command.

NOTE: When the RETURN key is depressed while the Editor is operating, the LINE FEED code will be generated by the program.

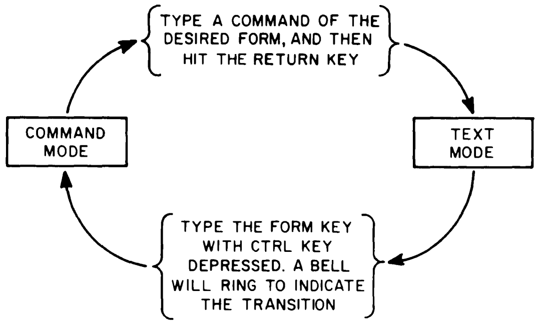
2. Text Mode

Characters typed by the programmer on the ASR-33 will be interpreted as text to the symbolic program. The text will be inserted in the program in the manner described by the command immediately preceding it.

3. Transition Between Modes

When the Editor program is started, it is in command mode (the program will be waiting for a command).

The means of transferring between modes is shown pictorially below:


TYPE A COMMAND OF THE DESIRED FORM, AND THEN HIT THE RETURN KEY
COMMAND MODE
TEXT MODE
TYPE THE FORM KEY WITH CTRL KEY DEPRESSED. A BELL WILL RING TO INDICATE THE TRANSITION

NOTE: After the operator issues a command to insert, change, or append text to his program, the Editor will remain in the text mode until the CTRL/FORM key combination is depressed. This combination generates a special character called form feed, which tells the Editor to return to command mode. The Editor returns to the command mode automatically without bell-ringing after executing a delete or list command.

C. LOADING SEQUENCE TO CORRECT A SYMBOLIC TAPE

1. Load the binary format loader.

2. Load Symbolic Tape Editor with binary format loader.

3. Use locating 0176_8 as the starting address for the Editor.

4. Put symbolic tape of program to be corrected in reader.

5. Type R ↵; turn on reader.

6. If tape has no form feed code on end, hit CTRL/FORM combination after tape has read in.

7. Editor is now waiting for first command.

D. SUMMARY OF COMMANDS

1. Input Commands

A ↵	Append incoming text from teleprinter.
R ↵	Append incoming text from reader.

2. Editing Commands

nI ↵	Insert following text before line n.
K ↵	Delete (kill) entire page of text.
nD ↵	Delete line n of text.
n,mD ↵	Delete lines n through m inclusively.
L ↵	List entire page of text.
nL ↵	List line n of text.
n,mL ↵	List line n through m of text.
nC ↵	Change line n of text.
n,mC ↵	Change line n through m of text.

3. Output Commands

P ↵	Punch entire text.
nP ↵	Punch line n of text.
n,mP ↵	Punch line n through m of text.
F ↵	Punch form feed.

E. SEQUENCE FOR PUNCHING OUT A CORRECT SYMBOLIC TAPE

1. Give the desired punch command: P↵, n,mP↵ or F↵. The computer will halt.

2. Turn on the punch.

3. If desired, switch the ASR-33 to LOCAL and generate as much Leader/Trailer code as needed. Switch teleprinter to ON-LINE.

4. Hit Continue on computer console.

5. If desired, generate Leader/Trailer.

6. When tape is punched, turn off punch before typing next command. Otherwise, the codes for the letters will appear on the symbolic tape.

7. Punching out the symbolic program does not delete it from memory. To read another tape into the buffer, delete the entire page of text (K↵).

F. EXAMPLE TO SHOW USE OF SYMBOLIC TAPE EDITOR

```
/ADD UP NUMBERS
*600
BEGN, HLT
/LOAD 200-207 WITH NUMBERS.  HIT "CONTINUE" ON CONSOLE
/
/THE NEXT FIVE INSTRUCTIONS INITIALIZE THE ROUTINE
          CLA          /CLEAR THE ACCUMULATOR
          TAD M10      /LOAD AC WITH -10 TO INITIALIZE TALLY
          DCA CNTR     /DEPOSIT IN COUNTER
          CLA          /CLEAR THE ACCUMULATOR
          TAD 2HUN     /LOAD AC WITH FIRST ADDRESS IN BUFFER
          DCABUFF
/
/THE NEXT SEVEN INSTRUCTIONS ARE THE PROGRAM ITSELF
BEGN,     TAD  BUFF    /ADD NEXT NUMBER FROM BUFFER
          ISZ  BUFF    /INDEX POINTER
          ISZ  I CNTR  /INDEX COUNTER.  IS IT ZERO?
          JMP  BEGN     NO, JUMP TO ADD NEXT NUMBER
          DCA  ANSR     YES, DEPOSIT TOTAL IN 410
          HLT          /STOP.  HIT "CONTINUE" TO REPEAT PROGRAM
          JMP  BEGN+1
```

```
/
/THE NEXT THREE REGISTERS CONTAIN THE CONSTANTS FOR THE PROGRAM
M10,    0-10          /NEGATIVE TALLY NUMBER
2HUN,   200           /FIRST ADDRESS IN BUFFER
ANSR    410
/
/THE NEXT TWO REGISTERS ARE RESERVED FOR THE VARIABLES IN THE PROGRAM
CNTR,
BUFF,
$
```

An assembly using PAL was attempted on the above program. On Pass 1, the Assembler typed out BEGN DT and halted. The program listing indicated that BEGN was duplicated. The Symbolic Tape Editor program was read in with the binary loader and the following sequence of commands was issued. Note that the CTRL/FORM combination and RETURN are nonprinting. Assume that the programmer used these keys correctly.

```
R
15L
BEGN,   TAD  BUFF    /ADD NEXT NUMBER FROM BUFFER
15C
ADDR,   TAD  BUFF    /ADD NEXT NUMBER FROM BUFFER
18L
        JMP  BEGN    NO, JUMP TO ADD NEXT NUMBER
18C
        JMP  ADDR    NO, JUMP TO ADD NEXT NUMBER
14, 19L
/THE NEXT SEVEN INSTRUCTIONS ARE THE PROGRAM ITSELF
ADDR,   TAD  BUFF    /ADD NEXT NUMBER FROM BUFFER
        ISZ  BUFF    /INDEX POINTER
        ISZ  1 CNTR  /INDEX COUNTER.  IS IT ZERO?
        JMP  ADDR    NO, JUMP TO ADD NEXT NUMBER
        DCA  ANSR    YES, DEPOSIT TOTAL IN 410
P
```

After typing P↵ , the complete program listing was typed out but it was not included here. As mentioned in Chapter 5, anything that is punched is typed. Punching out the text does not delete it. If an error occurs in punching, simple correct the error in the usual manner and punch out another tape.

The corrected tape was assembled, and the output of Pass 1 yielded the following symbol table.

BEGN	600
M10	616
CNTR	621
2HUN	617
ADDR	607
BUFF	621
NO	612
JUMP	UA
TO	UA
ADD	UA
NEXT	UA
NUMB	UA
ANSR	UA
YES	613
DEPO	UA
TOTA	UA
IN	UA

There are three things to note.

1. CNTR and BUFF were assigned the same location. This indicates that a violation occurred. The program listing indicates that the tags CNTR and BUFF are the only words in a statement. (Read the official PAL writeup section entitled "Language Details - 3(c)" to see that this is an illegal format.)

2. Symbolic address ANSR is an undefined address. This indicates that the tag ANSR was not interpreted as such. The program listing indicates that the programmer left out the comma.

3. Comments YES and NO were interpreted as tags. Words following interpreted as address.

Observation of listing shows that the control character / was omitted.

The following is the sequence of instructions used with the Symbolic Tape Editor to correct the listing (see next page). The listing as generated during punchout is also shown. It should be noted that this program is still incorrect, it assembled correctly as the symbol table shows. However, the program will not run as the programmer expects. This program will be corrected as an example using ODT in the next chapter.

```
/-7L
/THE NEXT TWO REGISTERS ARE RESERVED FOR THE VARIABLES IN THE PROGRAM
.=0028
29, 30C
CNTR,    0
BUFF,    0
26L
ANSR     410
26C
ANSR,    410
26, 30L
ANSR,    410
/
/THE NEXT TWO REGISTERS ARE RESERVED FOR THE VARIABLES IN THE PROGRAM
CNTR,    0
BUFF,    0
18L
         JMP    ADDR        NO, JUMP TO ADD NEXT NUMBER
18, 19C
         JMP    ADDR        /NO, JUMP TO ADD NEXT NUMBER
         DCA    ANSR        /YES, DEPOSIT TOTAL IN 410
P
/ADD UP NUMBERS
*600
BEGN,    HLT
/LOAD 200-207 WITH NUMBERS.  HIT "CONTINUE" ON CONSOLE
/
/THE NEXT FIVE INSTRUCTIONS INITIALIZE THE ROUTINE
         CLA                /CLEAR THE ACCUMULATOR
         TAD    M10         /LOAD AC WITH -10 TO INTIALIZE TALLY
         DCA    CNTR        /DEPOSIT IN COUNTER
         CLA                /CLEAR THE ACCUMULATOR
         TAD    2HUN        /LOAD AC WITH FIRST ADDRESS IN BUFFER
         DACBUFF
/
/THE NEXT SEVEN INSTRUCTIONS ARE THE PROGRAM ITSELF
ADDR,    TAD    BUFF        /ADD NEXT NUMBER FROM BUFFER
         ISZ    BUFF        /INDEX POINTER
         ISZ    I CNTR      /INDEX COUNTER.  IS IT ZERO?
         JMP    ADDR        /NO, JUMP TO ADD NEXT NUMBER
         DCA    ANSR        /YES, DEPOSIT TOTAL IN 410
         HLT                /STOP.  HIT "CONTINUE" TO REPEAT PROGRAM
         JMP    BEGN+1
/
/THE NEXT THREE REGISTERS CONTAIN THE CONSTANTS FOR THE PROGRAM
M10,     0-10               /NEGATIVE TALLY NUMBER
2HUN,    200                /FIRST ADDRESS IN BUFFER
ANSR,    410
```

```
/
/THE NEXT TWO REGISTERS ARE RESERVED FOR THE VARIABLES IN THE PROGRAM
CNTR,    0
BUFF,    0
/
$
BEGN     600
M10      616
CNTR     621
2HUN     617
ADDR     607
BUFF     622
ANSR     620
```

CHAPTER 7

INTRODUCTION TO THE PDP-8 OCTAL DEBUGGING TAPE

A. GENERAL

ODT is the debugging program for the PDP-8 computer which allows the programmer to correct binary tapes produced by the PAL Assembler from the console of the ASR-33 teleprinter. This program allows the programmer to:

1. Print out the contents of the registers in his program in octal.

2. Insert correction in octal.

3. Run his program all under the control of ODT and punch out a corrected binary tape.

Refer to the program writeup for a complete description.

B. LOADING PROCEDURES

1. Load binary object tape from PAL into memory with binary loader.

2. Load ODT into memory with binary loader.

3. Start ODT at 1200 or 7200 depending on edition use.

C. RESTRICTIONS

1. No more than one breakpoint can exist at one time. The last breakpoint specified will be the one that will be in effect when ODT runs the program.

2. A breakpoint may not be inserted at a JMS instruction.

3. The interrupt cannot be on when using ODT to debug a program. The interrupt portions of the program may be debugged using ODT. However, no ION instructions should be encountered in the debugging. If they are, they should be replaced temporarily by NOP instructions.

If a programmed halt is encountered in the object program, simple start ODT running again from the computer console and proceed as before.

D. PUNCH COMMANDS

The sequence of instructions for punching a program which has been debugged and is in core is as follows:

1. Turn Teletype to off line. Turn punch ON. Punch leader code (200) by depressing the CTRL, SHIFT, REPT. and @ keys simultaneously.

2. Turn Teletype on line, turn punch OFF.

3. Type in low address of memory area to be punched followed by a semi-colon. Type in high address and then depress ALT MODE key. At this point, the computer will halt.

4. Turn on punch, depress CONTINUE on console and the specified area in memory will be punched out in binary format. If more areas in memory are to be punched, return to step 2.

5. After punching the last area in memory onto your tape, hit the asterisk (*) key. (Be sure to depress SHIFT to get into upper case.) A check sum block will be generated, and the computer will halt.

6. Turn Teletype off line; punch trailer (same as leader).

7. Turn Teletype on line, turn punch off and hit CONTINUE on the console. ODT is now waiting for the next command.

If any characters other than those described in the program writeup are typed, ODT will respond with a ? and ignore the character.

E. PATCHING

ODT does not allow the programmer to <u>insert</u> any instructions between existing instructions in the program. To insert instructions, add a <u>patch</u> into an available area in memory. Usually, patches are placed near the end of the current page in memory.

To insert a patch following a given instruction, replace that instruction with a jump to an available place on the same page. The displaced instruction is the first instruction written in the patch. The desired instructions for the patch are inserted followed by a jump back to the correct location in the main program.

F. EXAMPLE

Suppose the program for adding numbers is typed in the following manner. During assembly, no error indicators were typed out, but when the assembled program was read in, the computer ran approximately 2 seconds and stopped. The correct answer did not appear in its proper location.

The symbol print from assembly is shown with a guided example for using ODT to examine, correct, and punch out the correct program. The programmer should return and make the corresponding corrections on the symbolic tape, then reassemble. This gives a permanent copy of the corrected program without wasted core space and execution time.

```
/ADD UP NUMBERS
*600
BEGN,   HLT
/LOAD 200-207 WITH NUMBERS.  HIT "CONTINUE" ON CONSOLE
/
/THE NEXT FIVE INSTRUCTIONS INITIALIZE THE ROUTINE
        CLA             /CLEAR THE ACCUMULATOR
        TAD   M10       /LOAD AC WITH -10 TO INITIALIZE TALLY
        DCA   CNTR      /DEPOSIT IN COUNTER
        CLA             /CLEAR THE ACCUMULATOR
        TAD   2HUN      /LOAD AC WITH FIRST ADDRESS IN BUFFER
        DCABUFF
/
/THE NEXT SEVEN INSTRUCTIONS ARE THE PROGRAM ITSELF
ADDR,   TAD   BUFF      /ADD NEXT NUMBER FROM BUFFER
        ISZ   BUFF      /INDEX POINTER
        ISZ   I CNTR    /INDEX COUNTER.  IS IT ZERO?
        JMP   ADDR      /NO, JUMP TO ADD NEXT NUMBER
        DCA   ANSR      /YES, DEPOSIT TOTAL IN 410
        HLT             /STOP.  HIT "CONTINUE" TO REPEAT PROGRAM
        JMP   BEGN+1
/
/THE NEXT THREE REGISTERS CONTAIN THE CONSTANTS FOR THE PROGRAM
```

```
M10,    0-10                /NEGATIVE TALLY NUMBER
2HUN,   200                 /FIRST ADDRESS IN BUFFER
ANSR,   410
/
/THE NEXT TWO REGISTERS ARE RESERVED FOR THE VARIABLES IN THE PROGRAM
CNTR,   0
BUFF,   0
/
$
BEGN    600
M10     616
CNTR    621
2HUN    617
ADDR    607
BUFF    622
ANSR    620
```

The program as shown ran for about 2 seconds and halted. The correct answer from the addition was not found in location 410. ODT was read in using the binary loader and was started in 1200. The program listing and symbol table printout were used constantly during the debugging process.

First, the initializing routine was checked to make sure the counters were initialized correctly. The breakpoint was inserted at locations 607 and ODT started the program running at location 601. Note that whatever ODT typed is underlined on the printout (the underlining was done by the author—not ODT).

```
607"
601'
0607) 0200
```

The breakpoint typeout indicated that the AC contained 200. This is incorrect because the DCA command should have cleared the accumulator. Thus, location 606 was opened.

```
606/  0000  3222
622/  0000
```

Finding it 0000, the listing was observed. The command DCA was not properly delimited. Then the instruction 3222 (DCA BUFF) was deposited and the register closed. Register 622 (BUFF) was opened and was not initialized. The command to go is given with the first breakpoint still in.

```
601'
0607) 0000
622/  0200
621/  7770
```

The registers are now correctly initialized.

A new breakpoint is inserted to trap the program just before the jump to add the next number. The programmer knew that the numbers in location 200–207 were all 0001 and expected the AC to contain 1 when the trap occurred. The results are:

```
612"
!
0612) 0200
```

The programmer examined the listing and saw why the AC was 0200 instead of 0001; he should have used the instruction TAD I BUFF instead of TAD BUFF. Thus he set bit 3 of the instruction to a 1 using ODT.

```
607/  1222 1622
```

Then the programmer checked to see if the counter indicated that one number had been added. It did not! Scrutiny of the listing indicated that he had made bit 3 to a 1 in this instruction. He had confused the use of indirect addressing. This is a common error frequently done by an inexperienced programmer. The instruction register was opened and modified as shown below.

```
621/  7770
611/  2621 2221
```

At this point, the breakpoint was cancelled and the program restarted. The computer halted immediately instead of running for two seconds. ODT was restarted at 1200 and location 410 was opened. It still contained the incorrect answer.

```
"
601'
410/  3570
```

The breakpoint was inserted again at the JMP instruction and the program restarted by ODT. The sequence of proceed commands indicated that the routine was adding the numbers correctly indicating the programmer concluded that the deposit instruction was wrong. The listing was observed and the programmer saw that DCA ANSR should have been DCA I ANSR—misuse of indirect addressing. Address ANSR was opened and, the answer was found there. The correct indirect reference and pointer were inserted and the registers closed.

```
612"
601'
0612) 0001
!
0612) 0002
!
0612) 0003
!
0612) 0004
!
0612) 0005
613/  3220  3620
620/  0010   410
```

The breakpoint was cancelled and the proceed command given. The computer halted, it was restarted at 1200 and register 410 opened. The correct answer was contained therein. The sequence of command is shown below:

```
"
!
410/  0010  0000
```

The answer location was cleared and the program rerun and checked, followed by the command sequence to punch and the contents of the corrected program. Only the low and high address observed on the printout.

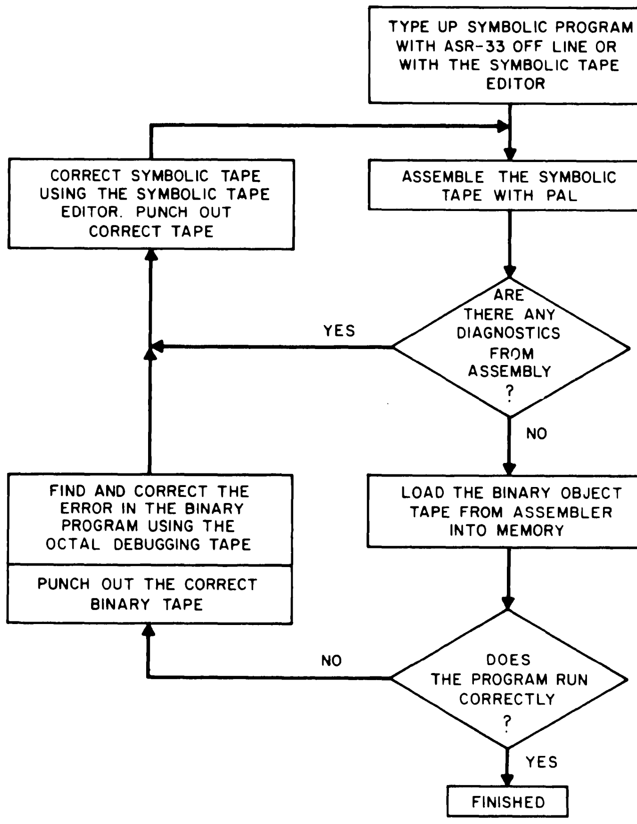
```
600;622F<:
          i
           *<*?&>
```

The binary tape was preserved, and the symbolic tape was corrected using the Symbolic Tape Editor.

G. SUMMARY OF PREPARATION AND CORRECTION OF PROGRAMS

At this point, the programmer may be confused concerning the use of each of the three main programs—PAL, Symbolic Tape Editor, and ODT.

The following flow diagram illustrates the relationship between the programs and the place of each in the preparation of correct programs.



TYPE UP SYMBOLIC PROGRAM WITH ASR-33 OFF LINE OR WITH THE SYMBOLIC TAPE EDITOR
CORRECT SYMBOLIC TAPE USING THE SYMBOLIC TAPE EDITOR. PUNCH OUT CORRECT TAPE
ASSEMBLE THE SYMBOLIC TAPE WITH PAL
ARE THERE ANY DIAGNOSTICS FROM ASSEMBLY ?
YES
NO
FIND AND CORRECT THE ERROR IN THE BINARY PROGRAM USING THE OCTAL DEBUGGING TAPE
PUNCH OUT THE CORRECT BINARY TAPE
LOAD THE BINARY OBJECT TAPE FROM ASSEMBLER INTO MEMORY
DOES THE PROGRAM RUN CORRECTLY ?
NO
YES
FINISHED

CHAPTER 8

INPUT-OUTPUT INSTRUCTIONS AND ROUTINES

A. GENERAL

An introduction to the elements of the input-output control is given in Section B, Chapter 1 of the PDP-8 Handbook. The programmer doesn't need to know the function of each of the elements in the control for elementary programming applications. Further information on the control may be found in the PDP-8 Maintenance Manual.

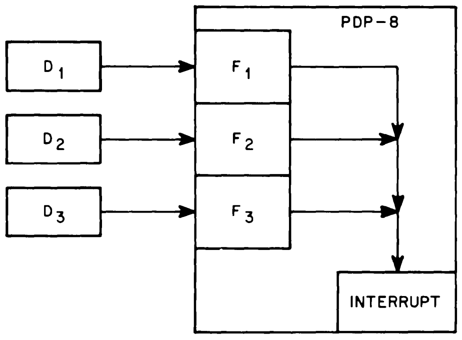
The PDP-8 is capable of servicing 64 different I/O devices, each requiring three commands. To the person watching the 64 I/O devices, the computer is handling all 64 devices simultaneously. However, to the programmer, the computer is handling each device independently and checking the status of all the others and servicing them, if necessary. If each device only requires one command to service it, the PDP-8 can handle up to 192 separate devices. The I/O instructions and suggested program sequences for each device are given in Section B of the PDP-8 Handbook. In this workbook, the printer/punch, reader/keyboard, high-speed reader, and high-speed punch will be discussed. The techniques used with I/O devices will be directly applicable to any equipment available on the PDP-8.

The specifications and commands for all devices is described in F-85. This section will deal primarily with the general I/O device and its programming techniques.

1. Device Flag

All I/O devices have a device flag. This flag indicates the state of the device. If the flag is set (a 1) the device is free; that is, it can be used if it is an output device, or has information (if it is an input device). If the flag is cleared (a 0) the device is busy; that is, currently not available as a result of issuing it a command (if it is an output device) or not having any information (if it is an input device).

The figure below illustrates the device flag.



Note that the device flags are physically located in the control processor and that each device, whether input or output has a unique flag. All the flags, however, are wired to the interrupt system. This is described in more detail in a later chapter of this workbook.

Each flag has two basic instructions associated with it.

1. Skip of the flag is set.
2. Clear the flag.

A separate device instruction is furnished to complete the action

3. Operate the device.

Thus if the programmer initially clears a device flag and operate the device, the flag associated with that device will be automatically set to a 1 when the device action is complete.

Usually the skip instructions are immediately followed by jump instructions to listen or wait for a flag to be set.

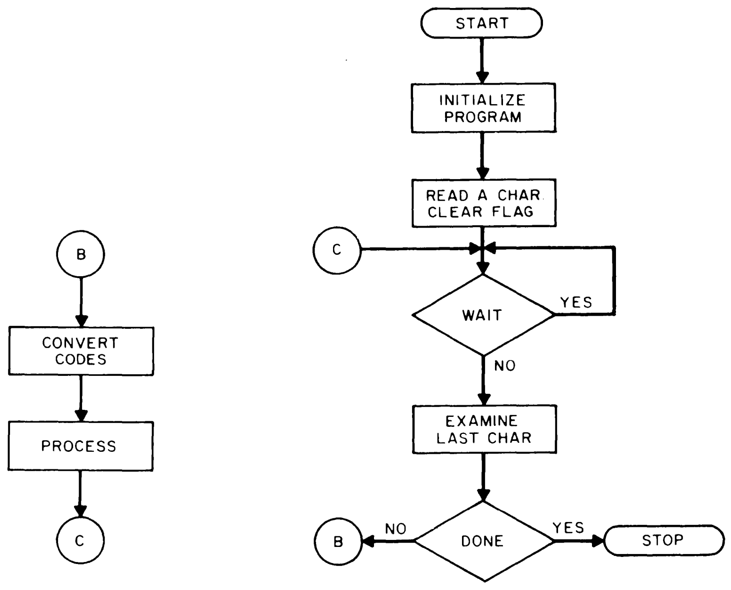
Example:

```
              CLR              /CLEAR A DEVICE FLAG
               -
               -
               -
               -
               -
              TAD
              IOT              /EXECUTE AN OUTPUT COMMAND
        A,    SKP              /SENSE DEVICE FLAG
              JMP A            /WAIT
               -               /CONTINUE
               -
```

The I/O commands can be microprogrammed to include the clearing operation in the output IOT command. When computer power is turned on, the flags can go to either a set or cleared state. Therefore, the flags should be cleared first (a must for interrupt programs).

There isn't any rule for the programmer to use to determine when to examine a flag or wait for a device; he must use his judgment. Situations may occur where computation could proceed while waiting for a device (storing or examining characters while reading paper tape), then the device waiting time can be utilized, as illustrated below in flow chart form:

Example 1:

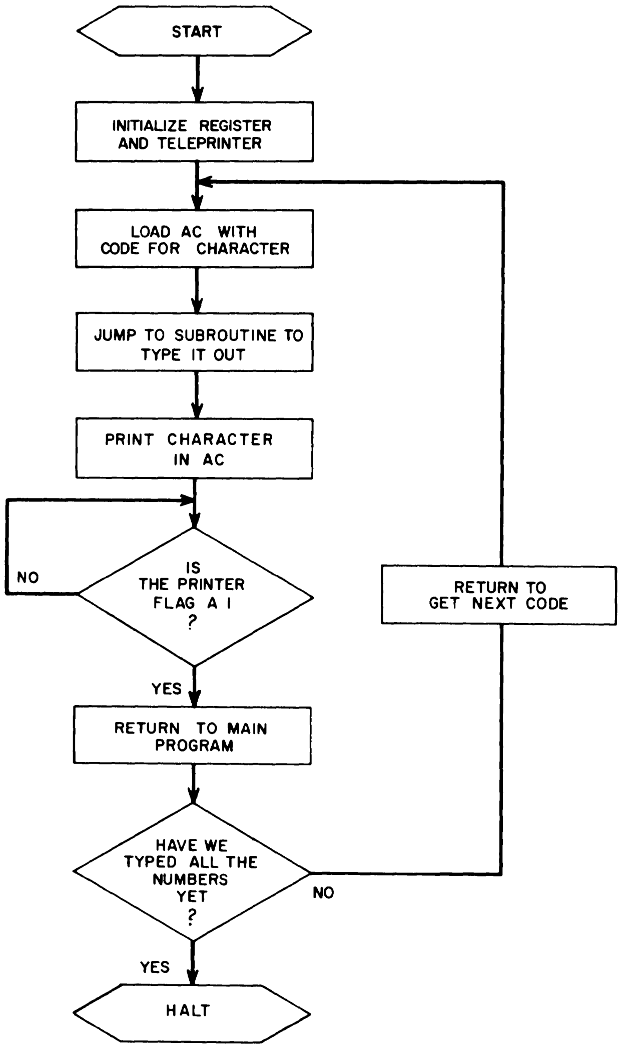


Example 2:

a. Problem

Write a program to type out the characters corresponding to the codes for 30_8 letters stored in consecutive locations starting at 1000_8. The program starts at location 600. Use a flow diagram.

b. Flow Diagram


START
INITIALIZE REGISTER AND TELEPRINTER
LOAD AC WITH CODE FOR CHARACTER
JUMP TO SUBROUTINE TO TYPE IT OUT
PRINT CHARACTER IN AC
IS THE PRINTER FLAG A 1 ?
NO
YES
RETURN TO MAIN PROGRAM
HAVE WE TYPED ALL THE NUMBERS YET ?
NO
YES
RETURN TO GET NEXT CODE
HALT

c. Program Listing

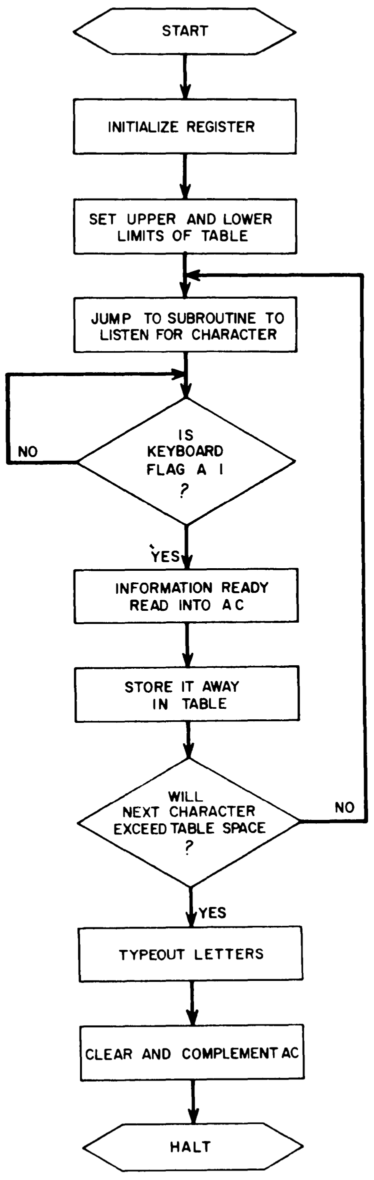
```
/ROUTINE TO TYPE 30 CHARACTERS
*600
BEGN,   HLT
/LOAD LOCATIONS IN TABLE WITH CODES FOR CHARACTERS.  HIT "CONTINUE"
/
/THE FOLLOWING 10 INSTRUCTIONS INITIALIZE PROGRAM AND PRINTER MECHANISM
        CLA             /CLEAR AC
        TAD     CR      /LOAD AC WITH CODE FOR CARRIAGE RETURN
        JMS     TYPE    /TYPE IT OUT IN SUBROUTINE
        TAD     LF      /LOAD AC WITH CODE FOR LINE FEED
        JMS     TYPE    /TYPE IT OUT
        CMA             /LOAD AC WITH "TABL-1" FOR
        TAD     TABL    /AUTO-INDEX REGISTER
        DCA     Z IR1   /STORE IN LOCATION 11
        TAD     M30     /INITIALIZE CHARACTER COUNTER
        DCA     CNTR
/
/MAIN PROGRAM
GET,    TAD     I Z IR1 /LOAD AC WITH CODE FOR CHARACTER
        JMS     TYPE    /TYPE IT OUT
        ISZ     CNTR    /INDEX COUNTER.  IS IT ZERO?
        JMP     GET     /NO, RETURN TO GET NEXT CHARACTER
        HLT             /YES, FINISHED.  HIT "CONTINUE" TO REPEAT PROGRAM
        JMP     BEGN+1
/
/TYPEOUT ROUTINE
TYPE,   0               /LINKING REGISTER TO MAIN PROGRAM
        TLS             /TYPE OUT CHARACTER IN AC
        TSF             /IS THE PRINTER FLAG A 1?
        JMP     .-1     /NO, CHECK IT AGAIN.  CHARACTER NOT YET TYPED
        CLA             /YES, CHARACTER TYPED.  CLEAR AC & RETURN TO PROGRAM
        JMP     I TYPE
/
/CONSTANTS AND VARIABLES
TABL,   1000            /INITIAL ADDRESS OF TABLE OF CHARACTERS
M30,    0-30            /2'S COMPLEMENT OF 30 FOR COUNTER
CR,     215             /CODE FOR CARRIAGE RETURN
LF,     212             /CODE FOR LINE FEED
CNTR,   0               /CHARACTER COUNTER
/
/USE AN AUTO-INDEX REGISTER WHEN SCANNING SEQUENTIAL LOCATIONS
*11
IR1,    0
/
$
```

Example 3:

a. Problem

Write a routine which will accept characters typed on the keyboard. The code for each character is to be stored consecutively in a table whose limits are set on the switch register (two settings required—upper and lower). When the codes have filled the table, the computer is to type out the word "BURP!" and halt with all ones in the AC.

b. Flow Diagram


START
INITIALIZE REGISTER
SET UPPER AND LOWER LIMITS OF TABLE
JUMP TO SUBROUTINE TO LISTEN FOR CHARACTER
IS KEYBOARD FLAG A 1 ?
NO
YES
INFORMATION READY READ INTO AC
STORE IT AWAY IN TABLE
WILL NEXT CHARACTER EXCEED TABLE SPACE ?
NO
YES
TYPEOUT LETTERS
CLEAR AND COMPLEMENT AC
HALT

c. Program Listing

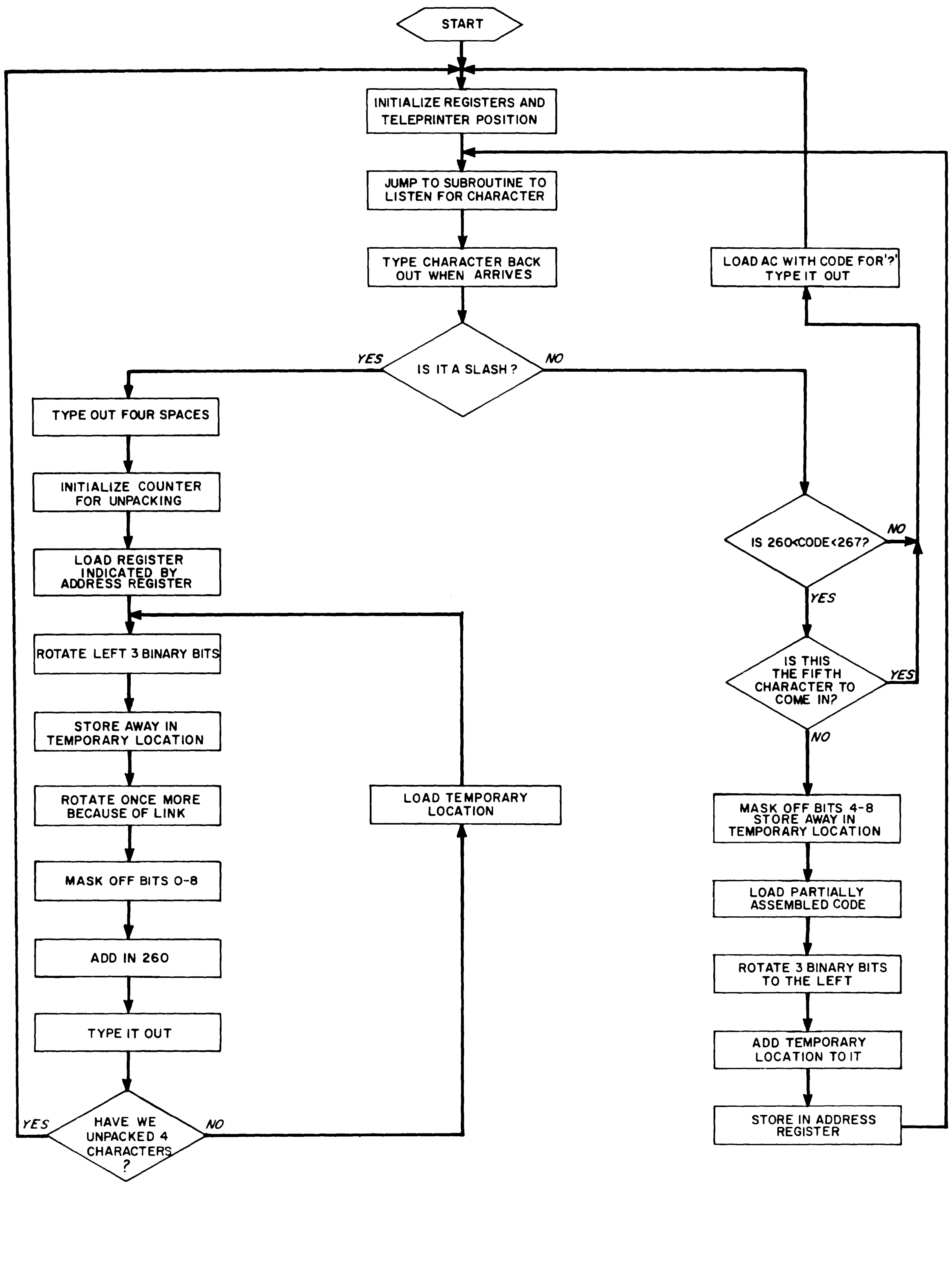
```
/TYPEIN ROUTINE
*400
/THE FOLLOWING INSTRUCTIONS INITIALIZE THE ROUTINE
STRT,   KCC             /CLEAR AC AND KEYBOARD FLAG
LOD1,   HLT             /SET INITIAL ADDRESS OF TABLE.  HIT CONTINUE
        LAS             /LOAD AC WITH SWITCH REGISTER
        DCA     TOP     /STORE INITIAL ADDRESS IN "TOP"
LOD2,   HLT             /SET FINAL ADDRESS OF TABLE.  HIT CONTINUE
        LAS             /LOAD AC WITH SWITCH REGISTER
        CIA             /COMPLEMENT AND INDEX FOR FUTURE USE
        DCA     MBOT    /STORE IN "MBOT"
INIT,   CMA             /INITIALIZE AUTO-INDEX REGISTER
        TAD     TOP     /FOR STORING CODES.
        DCA     Z IR1
        TAD     BRPT    /INITIALIZE AUTO-INDEX REGISTER FOR TYPING "BURP!"
        DCA     Z IR2   /INITIALIZE TELEPRINTER POSITION AT "CRLF"
        JMS     Z CRLF
/MAIN PROGRAM
TYPI,   KSF             /IS THE KEYBOARD FLAG A 1
        JMP     .-1     /NO, CHECK IT AGAIN
        KRB             /YES, CODE READY.  LOAD INTO AC
        TLS             /TYPE IT OUT
        DCA     I Z IR1 /STORE IT AWAY IN TABLE
CKCK,   TAD     Z IR1   /LOAD AC WITH POINTER
        TAD     MBOT    /ADD 2'S COMPLEMENT OF BOT. OF TABLE
        SPA             /IF ANSWER IS NEGATIVE, TABLE NOT FULL.  GET NEXT CHAR.
        JMP     TYPI
/
/THIS PORTION OF MAIN PROGRAM TYPES OUT INDICATION THAT TABLE IS FULL
BURP,   TAD     I Z IR2 /TABLE FULL.  LOAD AC WITH CODE FOR TYPEOUT
        JMS     X TYPO  /TYPE IT OUT
        TAD     MEXC    /ADD 2'S COMPLEMENT OF "!" TO AC
        SZA     CLA     /IS ANSWER ZERO?
        JMP     BURP    /NO, WORD NOT TYPED YET
        JMS     Z CRLF  /YES, LAST CODE WAS "!".  INITIALIZE TELEPRINTER
        CLA     CMA     /SET AC TO ALL ONES.
        HLT             /STOP.  RESTART BY HITTING "CONTINUE"
        JMP     STRT
/
/CONSTANTS AND VARIABLES
TOP,    0
MBOT,   0
BRPT,   BRPT            /BRPT = THIS LOCATION
        302             /B
        325             /U
        322             /R
        320             /P
        241             /!
MEXC,   0-241           /2'S COMPLEMENT OF "!"
/
/SMALL, OFTEN-USED SUBROUTINES PLACED ON PAGE 0
*40
CRLF,   0               /LINKING REGISTER FOR SUBROUTINE
        TAD     CR      /TYPE OUT CARRIAGE RETURN
        JMS     TYP0
        CLA
        TAD     LF      /TYPE OUT LINE FEED
        JMS     TYP0
        CLA
        JMP     I CRLF  /RETURN TO MAIN PROGRAM
TYP0,   0               /LINKING REGISTER FOR SUBROUTINE
        TSF
        JMP     .-1
        TLS
        JMP     I TYP0
CR,     215
LF,     212
/
/LOCATIONS FOR AUTO-INDEX REGISTERS
*11
IR1,    0
IR2,    0
$
```

Example 4: Rotation and Masking in Packing and Unpacking

a. Problem

Write a routine to accept octal numbers typed on the keyboard as on absolute address. Follow the address by a slash (/). The routine will then type four spaces followed by the contents of the specific address. All addresses are assumed to be right justified. If the address specified is too large or if any character other than 0-7 or / is typed, the routine will type a question mark and wait for the next address.

b. Flow Diagram


START
INITIALIZE REGISTERS AND TELEPRINTER POSITION
JUMP TO SUBROUTINE TO LISTEN FOR CHARACTER
TYPE CHARACTER BACK OUT WHEN ARRIVES
IS IT A SLASH ?
YES
NO
TYPE OUT FOUR SPACES
INITIALIZE COUNTER FOR UNPACKING
LOAD REGISTER INDICATED BY ADDRESS REGISTER
ROTATE LEFT 3 BINARY BITS
STORE AWAY IN TEMPORARY LOCATION
ROTATE ONCE MORE BECAUSE OF LINK
MASK OFF BITS 0-8
ADD IN 260
TYPE IT OUT
HAVE WE UNPACKED 4 CHARACTERS ?
YES
NO
LOAD TEMPORARY LOCATION
LOAD AC WITH CODE FOR '?' TYPE IT OUT
IS 260<CODE<267?
NO
YES
IS THIS THE FIFTH CHARACTER TO COME IN?
YES
NO
MASK OFF BITS 4-8 STORE AWAY IN TEMPORARY LOCATION
LOAD PARTIALLY ASSEMBLED CODE
ROTATE 3 BINARY BITS TO THE LEFT
ADD TEMPORARY LOCATION TO IT
STORE IN ADDRESS REGISTER

c. Program Listing

```
/OCTAL PRINT - SINGLE LOCATION
*1000
/THE FOLLOWING INSTRUCTIONS INITIALIZE THE ROUTINE
INIT,   TLS             /SET TELEPRINTER FLAG
        KCC             /CLEAR AC AND KEYBOARD FLAG
        JMS    Z CRLF   /INITIALIZE TELEPRINTER
        DCA    HOLD     /DEPOSIT ZEROS IN LOCATION TO HOLD ADDRESS
        TAD    M5       /INITIALIZE COUNTER TO COUNT NUMBERS IN THE ADDRESS
        DCA    CCTR
/
/THE FOLLOWING ROUTINE TESTS INCOMING CHARACTERS
LISN,   JMS    Z TYPI   /JUMP TO LISTEN FOR INCOMING CHARACTER
PROC,   DCA    TEM      /STORE IT AWAY FOR REFERENCE
        TAD    TEM      /LOAD IT INTO AC
        TAD    SLSH     /ADD NEGATIVE CODE FOR SLASH
        SNA    CLA      /IS THE RESULT ZERO?
        JMP    SPCR     /YES, CHARACTER WAS "/."  JUMP TO TYPE SPAC
        TAD    TEM      /NO, CHECK TO SEE IF
        TAD    M260     /CODE IS LESS THAN 260
        SPA    CLA      /IS SUM GREATER THAN ZERO OR EQUAL TO ZERO?
2LTL,   JMP    Z TYP?   /NO, CODE < 260.  TYPE "?"
        TAD    TEM      /YES, GET CODE AGAIN
        TAD    M267     /CHECK TO SEE IF CODE
        SMA    SZA CLA/IS GREATER THAN 267?
2BIT,   JMP    Z TYP?   /YES, CODE > 267
        ISZ    CCTR     /NO, INDEX CHARACTER CNTR.  HAVE 5 COME IN?
        SKP             /NO, SKIP NEXT INSTRUCTION
2MNY,   JMP    Z TYP?   /YES, TYPE "?"
        TAD    TEM      /GET CODE AGAIN
        AND    MASK     /MASK OFF BITS 9-8
        DCA    TEM      /STORE SINGLE OCTAL DIGIT IN TEM
        TAD    HOLD     /LOAD ASSEMBLED WORD
        RTL             /ROTATE THREE BITS TO THE LEFT
        RAL
        TAD    TEM      /ADD IN LATEST NUMBER
        DCA    HOLD     /STORE IN ADDRESS REGISTER.  RETURN FOR NEXT CHARACTER
        JMP    LISN
/
/ROUTINE TO TYPE OUT FOUR SPACES
SPCR,   TAD    M4       /LOAD AC WITH MINUS 4
        DCA    SCTR     /STORE IT IN SPACE COUNTER
        TAD    SPCD     /LOAD AC WITH SPACE CODE
        JMS    Z TYPE   /TYPE IT OUT
        ISZ    SCTR     /INDEX SPACE COUNTER.  IF NOT ZERO, TYPE ANOTHER SPACE
        JMP    .-3
```

```
/
/ROUTINE TO UNPACK AND TYPE OUT DESIRED REGISTER
UNPK,   TAD   M4      /INITIALIZE COUNTER TO TYPE OUT
        DCA   CCTR    /4 LETTERS
        TAD   I HOLD  /ADD CONTENTS OF EFFECTIVE ADDRESS
ROLO,   RTL           /ROTATE 3 PLACES TO THE LEFT
        RAL
        DCA   TEM     /STORE RESULTANT WORD
        TAD   TEM     /LOAD IT AGAIN
        RAL           /ROTATE ONCE MORE BECAUSE OF LINK
MSKR,   AND   MASK    /MASK OFF ALL BUT DESIRED OCTAL DIGIT
        TAD   NMCD    /ADD IN 260 TO MAKE IT CODE FOR NUM
        JMS   Z TYPE  /TYPE IT OUT
        TAD   TEM     /ADD PARTIALLY ROTATED WORD
        ISZ   CCTR    /INDEX COUNTER HAVE WE TYPED 4 LETTERS
        JMP   ROLO    /NO, JUMP BACK AND TYPE OUT NEXT CODE
        JMP   INIT+1  /JUMP BACK TO INITIALIZE ROUTINE
/CONSTANTS AND VARIABLES
NMCD,   260           /CODE FOR NUMBERS
HOLD,   0             /REGISTER TO HOLD ASSEMBLED ADDRESS
CCTR,   0             /VARIABLE FOR CHARACTER COUNTER
M5,     0-5           /CONSTANT FOR CHARACTER COUNTER
MASK,   7             /MASK TO LEAVE ONLY 1 OCTAL CODE
M267,   0-267         /NEGATIVE VALUE FOR UPPER LIMIT OF NUMBERS
M260,   0-260         /NEGATIVE VALUE FOR LOWER LIMIT OF NUMBERS
TEM,    0             /TEMPORARY LOCATION
SLSH,   0-257         /NEGATIVE CODE FOR SLASH
SCTR,   0             /VARIABLE LOCATION FOR COUNTING SPACES
M4,     0-4           /TALLY NUMBER TO COUNT 4 SPACES
SPCD,   240           /CODE FOR SPACE
/
*40
/SUBROUTINE TO TYPEOUT CHARACTER
TYPE,   0             /LINKING REGISTER TO MAIN PROGRAM
        TSF           /IS PRINTER FLAG A 1?
        JMP   .-1     /NO, CHECK IT AGAIN
        TLS           /YES, TYPE OUT NUMBER IN AC
        CLA   CLL     /CLEAR AC AND LINK, RETURN TO MAIN PROGRAM
        JMP   I TYPE
/
/SUBROUTINE TO LISTEN FOR KEYBOARD
TYPI,   0             /LINKING REGISTER TO MAIN PROGRAM
        KSF           /IS KEYBOARD FLAG A 1?
        JMP   .-1     /NO, CHECK IT AGAIN
        KRB           /YES, READ KB IN AC
        TLS           /TYPE IT BACK OUT.  RETURN TO MAIN PROGRAM
        JMP   I TYPI
/
/THIS ROUTINE TYPES A QUESTION MARK AND JUMPS TO INITIALIZE ROUTINE
```

```
TYP?,   TAD     QUMK
        JMS     Z TYPE  /TYPE "?"
        JMP     I AGIN  /JUMP BACK TO INITIALIZING ROUTINE
AGIN,   INIT+1          /POINTER TO MAIN PROGRAM
QUMK,   277             /CODE FOR QUESTION MARK
/
/THIS ROUTINE TYPES A CARRIAGE RETURN-LINE FEED COMBINATION
CRLF,   0               /LINKING REGISTER
        TAD     CR      /LOAD AC WITH CODE FOR CARRIAGE RETURN
        JMS     Z TYPE  /TYPE IT OUT
        TAD     LF      /LOAD AC WITH CODE FOR LINE FEED
        JMS     Z TYPE  /TYPE IT OUT
        JMP     I CRLF  /RETURN TO MAIN PROGRAM
CR,     215
LF,     212
$
```

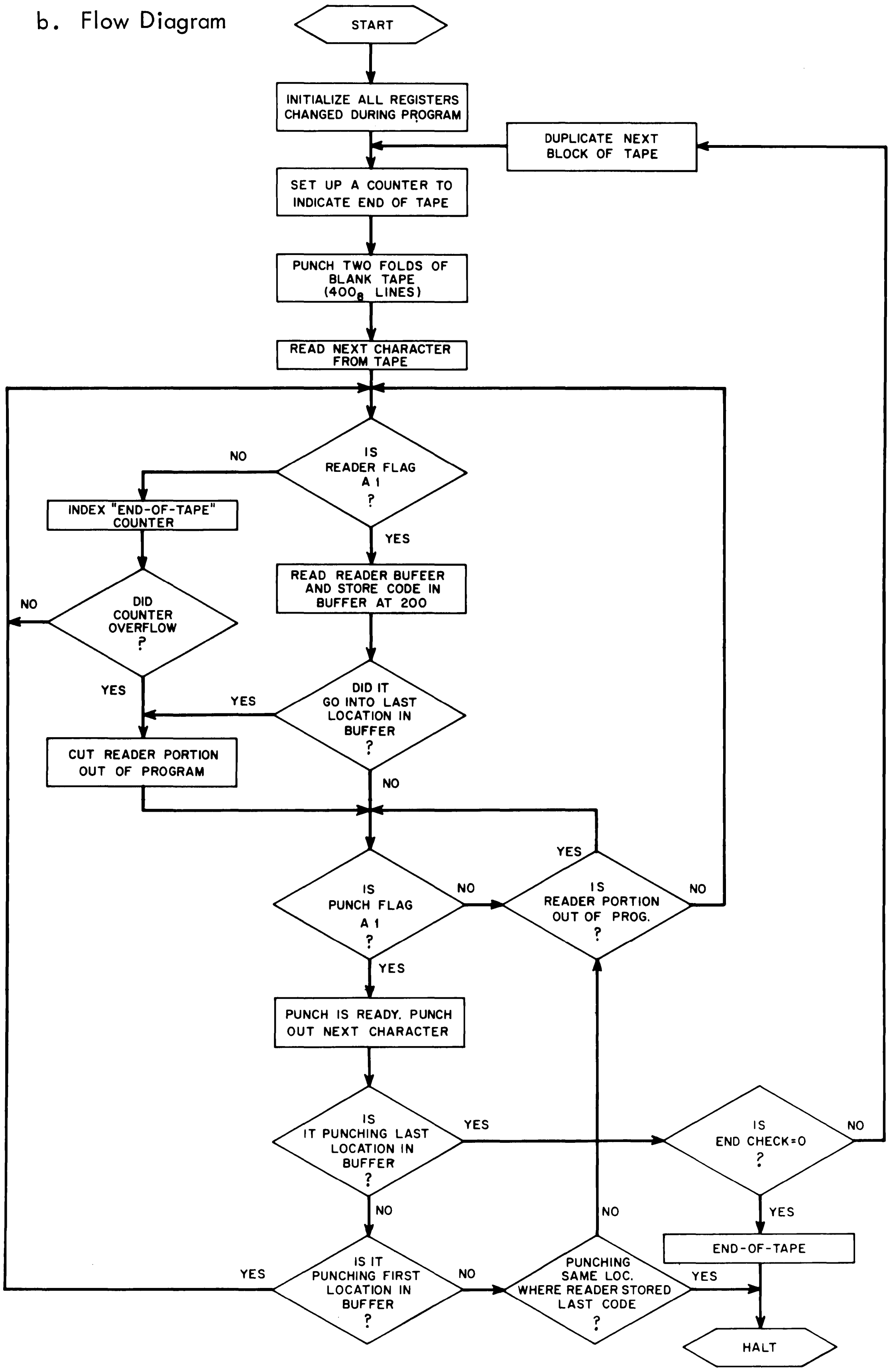
Example 5:

a. Write a tape duplicator which will read characters from a paper tape and store them in a word storage block starting at 400_8 and ending at 7000_8. At the same time, the punch should be punching the contents of the storage. Use the I/O time available while punching to read as many characters as possible; thus operating punch and reader at full speed. Devise a method of determining when the end of the tape has been reached.

The solution to this example shown on the following pages does not use the program interrupt. This type of programming simply illustrates how one can do a large number of program steps while a relatively slow I/O device is operating. In this solution, approximately 5 characters are read from paper tape and stored in memory while one character is being punched. Likewise, the routine for determining when the end of the tape has been reached is operating while information is being obtained from the reader.

The end-clock routine for determining the end of the tape is as follows. Each time the computer finds that the reader flag is a 0, it indexes a counter before returning to check the flag again. The counter is preset to overflow after looping approximately 30 milliseconds in that sequence of instructions. It usually takes only 3.3 milliseconds to read a line of tape. If a feed hole doesn't appear under the reader head, the flag is not set to a 1 and the counter indexes until overflow. The computer then assumes the end of the tape has been reached.

b. Flow Diagram


START
INITIALIZE ALL REGISTERS CHANGED DURING PROGRAM
DUPLICATE NEXT BLOCK OF TAPE
SET UP A COUNTER TO INDICATE END OF TAPE
PUNCH TWO FOLDS OF BLANK TAPE (400$_8$ LINES)
READ NEXT CHARACTER FROM TAPE
IS READER FLAG A 1 ?
NO
INDEX "END-OF-TAPE" COUNTER
YES
READ READER BUFEER AND STORE CODE IN BUFFER AT 200
DID COUNTER OVERFLOW ?
NO
YES
DID IT GO INTO LAST LOCATION IN BUFFER ?
YES
NO
CUT READER PORTION OUT OF PROGRAM
IS PUNCH FLAG A 1 ?
NO
IS READER PORTION OUT OF PROG. ?
YES
NO
YES
PUNCH IS READY. PUNCH OUT NEXT CHARACTER
IS IT PUNCHING LAST LOCATION IN BUFFER ?
YES
IS END CHECK=0 ?
NO
NO
NO
YES
END-OF-TAPE
IS IT PUNCHING FIRST LOCATION IN BUFFER ?
YES
NO
PUNCHING SAME LOC. WHERE READER STORED LAST CODE ?
YES
HALT

c. Program Listing

```
/TAPE DUPLICATOR FOR HIGH SPEED READER AND PUNCH ON PDP-5
*200
BEGN,  HLT
/LOAD TAPE TO BE DUPLICATED INTO READER.  HIT CONTINUE.
/
/THE FOLLOWING ROUTINE PUNCHES 400 LINES (2 FOLDS) OF LEADER
LEDR,   CLA             /SET UP COUNTER TO PUNCH 400 (OCTAL)
        TAD   M400      /LINES OF LEADER
        DCA   LCTR
PUNL,   PLS             /PUNCH OUT LEADER (AC=0)
        PSF             /CHECK TO SEE IF PUNCH FLAG IS 1
        JMP   .-1       /NO, CHECK IT AGAIN
        ISZ   LCTR      /YES, INDEX LEADER COUNTER.  SKIP ON OVERFLOW
        JMP   PUNL
/
/INITIALIZING ROUTINE
INIT,   TAD   BUFF      /INITIALIZE AUTO-INDEX REGISTERS
        DCA   Z IR1
        TAD   BUFF
        DCA   Z IR2
        TAD   RDIN      /INITIALIZE PROGRAM TO INCLUDE READER LOOP
        DCA   CHG1
        TAD   RDIN
        DCA   CHG2
/
/MAIN PROGRAM
READ,   TAD   MG        /SET UP COUNTER TO INDICATE END OF TAPE
        DCA   ECHK
        RFC             /FETCH FIRST CHARACTER
        RSF             /CHECK TO SEE IF READER FLAG IS A 1
        SKP             /NO, CHECK IF END OF TAPE
        JMP   .+4       /YES, JUMP TO READ READER BUFFER
TCHK,   ISZ   ECHK      /INDEX ENDCHECK.  IS IT ZERO?
        JMP   .-4       /NO, CHECK READER FLAG AGAIN.
        JMP   RSTP      /YES, CUT READER PORTION OUT OF PROGRAM. END OF TAPE
        RRB             /READ READER BUFF
        DCA   I Z IR1   /STORE IN BUFFER
BCHK,   TAD   Z IR1     /LOAD READER POINTER INTO AC
        TAD   MEND      /ADD NEGATIVE ADDRESS OF END OF BUFFER
        SNA   CLA       /ARR THE TWO NUMBERS EQUAL?
        JMP   RSTP      /YES, JUMP TO STOP READER.  BUFFER FULL
PUN,    PSF             /NO, IS THE PUNCH AVAILABLE?
CHG1,   JMP   READ      /NO, JUMP BACK TO READ OR PUNCH NEXT CHARACTER
        TAD   I Z IR2   /YES, PUNCH READY.  GET NEXT CODE
        PLS             /PUNCH IT OUT
        CLA
        TAD   Z IR2   /LOAD AC WITH PUNCH POINTER
        TAD   MEND    /COMPARE WITH LAST LOC.  IN BUFFER
        SZA   CLA     /ARE THE TWO NUMBERS EQUAL?
        JMP   .+5     /NO, BUFFER NOT PUNCHED YET.
        TAD   ECHK    /YES, LOAD AC WITH END-OF-TAPE COUNTER
        SNA   CLA     /IS IT ZERO?
        JMP   STOP    /YES, END OF TAPE REACHED.
        JMP   INIT    /NO, MORE TAPE TO DUPLICATE
        TAD   Z IR2   /LOAD AC AGAIN WITH PUNCH POINTER
        TAD   M400    /COMPARE WITH BEGINNING OF BUFFER
        SNA   CLA     /ARE THE TWO NUMBERS EQUAL?
        JMP   READ    /YES, READ NEXT CHARACTER.  ITS FIRST TIME
        TAD   Z IR2   /LOAD AC ONCE MORE WITH PUNCH POINTER
        CMA   IAC     /COMPARE WITH READER POINTER
        TAD   Z IR1
        SZA   CLA     /ARE THE TWO NUMBERS EQUAL?
CHG2,   JMP   READ    /NO, JUMP TO READER OR PUNCH LOOP
STOP,   CLA   CMA     /YES, DUPLICATION COMPLETE
        HLT           /HALT WITH ALL ONES IN THE AC
        JMP   LEDR
/BY HITTING CONTINUE, YOU MAY DUPLICATE ANOTHER TAPE
/
/FOLLOWING ROUTINE CHANGES ALL "JMP READ'S" TO "JMP PUN'S"
/TO ALLOW PUNCH TO CATCH UP WITH READER. THEY ARE REINITIALIZED EACH
/TIME PUNCH HAS PUNCHED OUT THE BUFFER
RSTP,   CLA           /READER STOP SEQUENCE
        TAD   PNIN    /LOAD INSTRUCTION "JMP PUN"
        DCA   CHG1    /DEPOSIT IN LOCATIONS WHICH CHANGE
        TAD   PNIN    /WHEN READER FILLS BUFFER
        DCA   CHG2
        JMP   PUN
/VARIABLES
ECHK,   0             /COUNTER TO DETERMINE END OF TAPE
LCTR,   0             /COUNTER TO DETERMINE END OF LEADER
/
/CONSTANTS
M400,   0-400         /CONSTANT TO INITIALIZE LEADER COUNTER
BUFF,   377           /CONSTANT INDICATING BEGINNING OF BUFFER
END,    7000          /END OF BUFFER
MEND,   1000          /TWO'S COMPLEMENT OF END OF BUFFER
PININ,  JMP   PUN     /INSTRUCTION TO CHANGE "JMP READ"
RDIN,   JMP   READ    /INSTRUCTION TO CHANGE "JMP PUN"
MG,     0-1000        /CONSTANT TO INITIALIZE END OF TAPE COUNTER
/
*11
/DEFINITION OF AUTO-INDEX REGISTERS.
IR1,    0             /READER POINTER
IR2,    .0            /PUNCH POINTER
/
$
```

REVIEW QUESTIONS - CHAPTER 8

Write routines for the following problems. All routines should be in PDP-8 Assembler language and symbolic addressing should be used.

1. Type out in octal the contents of memory, starting and ending at the locations of your choice. Type two columns, the left representing the address of the location and the right representing the contents of that location.

2. Write a routine which will accept octal instructions typed on the keyboard and store them in memory. The starting address for the sequence of instructions is determined by typing an A, the absolute octal address, and a carriage return. The following list of 4-digit instructions (each terminated by CR) will be inserted consecutively in memory until another A is hit to specify a new starting address. To make the program useful type an E, erasing all numbers typed after the preceding carriage return and enabling correction of errors in typing.

3. Write a routine which will punch out on paper tape the contents of memory starting and ending as specified by the operator. Be sure to put on the tape the starting address of the block and a suitable termination word that indicates the end of tape.

4. Write a routine to read from paper tape and store in memory the information punched by the above program (#3).

CHAPTER 9

PROGRAM INTERRUPT FEATURE OF THE PDP-8 SYSTEM

A. GENERAL

The program interrupt feature of the PDP-8 computer allows a logic line to interrupt the program that is running, to sense certain alarm conditions or event signals, or to service an I/O device. This feature speeds up processing of I/O data, because many computer instructions may be performed while waiting for an I/O device to become ready again. Also, the time may be effectively shared between two, three, four, five or more separate devices.

When an interrupt occurs, the contents of the program counter (the current address in the main program) are automatically stored in memory location 0, and an interrupt begins by forcing the computer to execute a program beginning at location 0001. The above operations occur automaticaly in 3.2 microseconds. To avoid destruction of the auto-index registers, the instruction in 0001 should be a jump to a interrupt servicing routine. The interrupt system is turned OFF automatically when the interrupt is initiated, and therefore must be turned ON again by the program immediately before returning to the main program. After the device causing the interrupt has been examined and serviced by a subroutine, the computer should return to the main routine by the execution of a JMP I Z 0000. If another interrupt request is waiting, it will be accepted immediately. If there is a second interrupt request while the computer is servicing the first request, the second request is ignored until the first request has been satisfied. However, the device flag is set and may be checked by the servicing routine before returning to the main program. It is good practice to examine the program flags for other devices before leaving the interrupt routine. This should be done prior to turning on the interrupt and executing the JMP I Z 0000.

The two instruction associated with the interrupt system are:

ION	6001	Enable the interrupt system
IOF	6002	Disable the interrupt system

B. TYPES OF INTERRUPTS

There are two kinds of interrupts: data break and program interrupt. If the computer gets a request for these two at the same time, a priority circuit will handle the data break request before the program request. If two or more program interrupts occur at the same time, the interrupt is initiated; and with program examination of the device flags, a priority of I/O devices can be assigned.

1. Data Break

The data break feature allows high-speed transfer of data from an I/O device such as a magnetic tape unit, a drum, or a microtape unit. The data break is controlled by hardware in the control unit for the specific I/O device. The programmer does not need to be concerned with the handling of the data, this is governed by the control unit. This control unit will steal one cycle per data transfer from the computer while it is executing the program and will deposit the data in or fetch data from memory, but will not disturb the arithmetic registers or the program counter.

2. Program Interrupt

In general applications, the program interrupt is used more frequently than the data break. It is particularly useful in handling data from the relatively slow I/O devices such as the high-speed reader, high-speed punch or teleprinter. It allows the main program to run continuously, and the program will be interrupted only when a data transfer is ready. If data is coming into the computer, it is stored at this time, and instructions are modified to take care of the next input of data. If data is going from the machine, the instructions are modified to select the correct data for the next output.

C. USE OF THE PROGRAM INTERRUPT

At the beginning of a program using the interrupt, the programmer should clear all the device flags connected to the computer. The state of the flags cannot be assumed, and spurious interrupts might occur if this precaution were not taken.

The following program will illustrate a typical use of the system routine, using the high-speed punch and reader. The I/O flags will be tested to see which I/O device should be handled. Each of the subroutines, that handle the equipment returns to the interrupt routine and continues to check flags to assure that another device further down in priority has not requested an interrupt during the I/O handling routine. If such a device has set a flag, it will be serviced at this time before the AC and link are restored and the interrupt system is enabled. After the interrupt system is turned on, the computer should return immediately and all flags checked again. This is not the only interrupt routine which can be used. Some programmers may prefer to jump out of the interrupt routine without checking any other flags. Either procedure may be used.

ILLUSTRATION OF INTERRUPT AND SYSTEM ROUTINE

```
*0          0
            JMP I 2
            1000
*1000
INTR        DCA TEMP        /STORE AWAY AC
            RAL             /ROTATE LEFT
            DCA LINK        /STORE AWAY LINK
            RSF             /CHECK READER FLAG
            SKP
            JMS REDR        /SERVICE READER
            PSF             /CHECK PUNCH FLAG
            SKP
            JMS PUN         /SERVICE PUNCH
EXIT,       TAD LINK        /RESTORE THE LINK
            RAR CLL
            TAD TEMP        /RESTORE AC
            ION             /ENABLE INTERRUPT SYSTEM
            JMP I Z 0       /RETURN TO MAIN PROGRAM
REDR,       0               /THIS SUBROUTINE SHOULD PROCESS
            .               /DATA FROM READER IN WHATEVER
            .               /WAY PROGRAMMER DECIDES AND
            CLA             /SHOULD CHANGE INSTRUCTIONS FOR
            JMP I REDR      /MANIPULATION OF NEXT INPUT.
PUN,        0               /THIS SUBROUTINE SHOULD GET
            .               /DATA TO BE PUNCHED OUT, AND
            CLA             /SHOULD CHANGE INSTRUCTIONS
            JMP I PUN       /FOR NEXT OUTPUT.
```

D. CRITERIA FOR DETERMINING TIMING AND PRIORITY

1. Timing

In writing an interrupt program for a real-time application, there are two important parameters which must be determined: the frequency of data transfer to or from each device and the stability of data to or from each device.

The latter consideration is seldom critical because of the buffers on each standard device. However, in some applications, (processing of radar signals) it may be important.

The first general rule may be phrased as: The time of the processing subroutine for all devices must be shorter than the time between the data transfers from the higher frequency device.

If this condition is not met, information may be lost from the device. To minimize subroutine time, the programmer may choose to use JMP instructions in servicing devices rather than JMS instructions and JMP back at end of the subroutine rather than JMP I.

2. Priority

In the standard interrupt facility of the PDP-8, the programmer has control over the sequence in which the various devices are checked. The second general rule is: Check the I/O devices in order of frequency. The faster devices should be checked first.

Again, if the stability time of the data is important, this should be considered.

CHAPTER 10

PROGRAM LIBRARY: SUBROUTINES AND PROGRAMS

A. GENERAL

It is often necessary to repeat a group of instructions during the execution of a program. Do not write out the instruction each time a function is needed. Instead, the instructions needed are written once and the main program transfers to this group of instructions each time they are required. This group of instructions is called a subroutine. These subroutines normally perform basic functions and may be used in the solution of many problems.

A great deal of the programmer's time and effort may be saved by using the subroutines which are available from the PDP-8 Library. By using these subroutines and the calling sequence as required, the programmer can write a program very quickly to accomplish a rather complex sequence of operations. The symbolic tapes of the subroutines are supplied, as well as listings of each subroutine. The tapes may be appended to the main program, and any page specifications and address assignments may be established by the programmer using the Symbolic Tape Editor.

Programs are constantly being added to the library. The subroutines and programs listed on the following pages represent only a partial list of those presently available. The requests and needs of the customer are often the criterion to write a program for the library.

B. USE OF THE SUBROUTINES

1. Assembly

Most of the subroutines in the library have an origin setting at the beginning and a $ delimiter at the end of the tape. At the time of assembly, it may be necessary for these to be removed from the final symbolic tape.

With the Symbolic Tape Editor, the operator can easily append the subroutines onto the main program and the Editor will punch out the entire program. The text storage area of the Editor holds over 64_{10} instructions (one half memory page) with comments. If the program is longer than one half memory page, the use of the form feed block on the symbolic tape will ease the job of editing.

Considerable thought should be given to the organization of subroutines on the core pages. The writeup for each subroutine contains the number of locations required. The subroutines should be placed together to use as many registers on a core page as possible.

2. Requirements

Some of the subroutines in the library require other subroutines to make them work properly. Under the category labeled Needed on the official writeup, the author lists the required equipment and/or subroutines to make the particular subroutine work properly. Failure to heed the requirements will result in wasted time as the program won't work. If both subroutines are to be used without modification, they should be on the same page.

3. Intercommunication Registers

If the subroutine is not on the current page of the main program and not on Page 0, the programmer may put an intercommunication register on the current page for ease in referencing the subroutine. The address portion of the intercom register is the tag of the starting address of the subroutine. On assembly, the initial address of the subroutine is placed in the intercom register. An example of this technique will be shown using the Square Root subroutine.

a. Example Using Intercom Register

```
            .
            .                /AC CONTAINS SQUARE
      JMS I SQIN             /SUBROUTINE CALLED
            .                /AC CONTAINS SQUARE ROOT
            .
```

```
          .
          .
SQIN, SQRT                /INTERCOM TO SQRT
```

If the square root subroutine were placed on Page 0, the subroutine call would be:

```
          .
          .                 /AC CONTAINS SQUARE
    JMS Z SQRT              /SUBROUTINE CALLED
          .                 /AC CONTAINS SQUARE ROOT
          .
```

4. Tags in the Subroutine

The tags in the subroutine are specified by the author when writing it. If a tag in the main program is identical to a tag in a subroutine, PAL will type out BEGN DT and halt. Therefore, the listing of the pertinent subroutine be observed and all tags should be noted. The programmer must then use different tags.

5. Single- and Double-Precision Routines

Both single- and double-precision routines are available from the PDP-8 Program Library. Single-precision routines deal with one computer word (i.e., 12 bits). Using one word, the machine can represent 2's complement numbers in the range of +2047 to -2048. If more range is desired, the programmer should use double-precision routines (two computer words, 24 bits). By using 24 bits, the machine can represent signed 2's complement numbers in the range of +8,388,607 to -8,388,608.

The following is an example of a double-precision addition. A double-precision operand is stored in locations 20 and 21. The second operand is stored in locations 22 and 23. The high-order portions are in locations 20 and 22. Add the two numbers and store the results in locations 24 and 25 with the high-order portion in location 24.

```
CLA CLL
TAD       Z     21
TAD       Z     23
DCA       Z     25
RAL
```

```
        TAD     Z   20
        TAD     Z   22
        DCA     Z   24
```

C. SELECTED ROUTINES

1. The PDP-8 Program Library is constantly being updated and new programs added. No attempt will be made to present all the programs that are available from the library in this workbook. Your instructor will issue the latest program abstract.

   Following is a selected group of library subroutines, which are used most often. Single- and double-precision routines are similar therefore, only single-precision is presented here.

2. Arithmetic Routines

   The rules for using a 2's complement, fixed-point computer for performing arithmetic operations are referred to as scaling of binary numbers. The PDP-8 subroutines are written for general-purpose, fixed-point arithmetic. This scaling or scale factor concept must be used to obtain meaningful results. A complete discussion of the rules and techniques of scaling is available from Digital's Applied Programming Department.

   a. 2's Complement Multiply Subroutine

      The multiplicand must be present in the location following the JMS MULT instruction before the subroutine is called.

      The subroutine multiplies two 11-bit signed numbers together resulting in a 22-bit signed product. If the product is positive, bits 0 and 1 of the high-order word will be 0; if negative, bits 0 and 1 will both be 1. This provides consistency with 2's complement arithmetic.

      The subroutine returns with the high-order word of the product in the AC and the low-order bits in location MP1. If the main program is on another core page, MP1 may be referenced by an intercom register.

b. 2's Complement Divide Subroutine

The definitions of divisor, dividend, and quotient are:

$$\frac{5 \text{ (Dividend)}}{2 \text{ (Divisor)}} = 2 \text{ (quotient) with a remainder of } 1.$$

The divide subroutine is general-purpose to divide 2's complement, fixed-point, scaled binary numbers. The dividend is 24 bits and must be greater in magnitude than the divisor. When dividing scaled binary numbers, the scale factor of the quotient is the difference between the scale factor of the dividend and the scale factor of the divisor.

Example: Divide 40_8 by 20_8

Dividend	0000 0040	Scale Factor = B23
Divisor	0020	Scale Factor = B11
Quotient	0001	Scale Factor = B12

In this example, the scale factor of the quotient is outside the 12-bit word.

To preserve accuracy of the quotient, the programmer should adjust the scale factors before entering the divide subroutine. The scale factors are adjusted by shifting operations which are performed by combinations of rotates on the PDP-8.

In the above example, the scale factor of the quotient should have been B11. This could be obtained by rescaling the dividend to B22 by shifting the dividend one place to the left. When the division is performed, a binary number scaled B22 by a binary number scaled B11 will be divided. The quotient will be scaled B11 (B22 - B11).

The following is a suggested program sequence to accomplish rescaling before calling the subroutine. Assume that the high- and low-order portions of the signed, 23-bit dividend are located in HDIV and LDIV, respectively.

```
        .
        .
        .
CLA  CLL
TAD  LDIV
RAL
```

```
                DCA   CALL+1
                TAD   HDIV
                RAL
CALL,           JMS   DIV
                  .
                  .
                  .
                  .
```

Where such a technique is used, the largest acceptable dividend before manipulation is:

$$\pm 2^{22} - 1 = \pm 4,194,303_{10}$$

3. Conversion Routines

a. Single-Precision, Decimal-to-Binary Input

This subroutine will accept a signed string of decimal digits from the keyboard of the ASR-33. The number typed will be converted to its binary equivalent, which will be in the accumulator on return from the subroutine.

b. Single-Precision, Binary-to-Decimal Conversion and Output (DEC-5-32-A)

This subroutine will convert a signed 11-bit number to its decimal equivalent which will be typed out by the ASR-33 printer.

4. Teletype Output Programs

a. Teletype Output Package

This package contains several subroutines to perform different functions. The purpose of the subroutines is to provide an easy means of typing out 1, 2, or a string of characters and special characters such as carriage return/line feed combinations, tabs or spaces. This package also contains a routine to interpret a BCD code and type it out as a single decimal digit.

The codes for the characters used in the subroutine are referred to as trimmed codes. Instead of being composed of 8 binary bits, the codes contain only the last 6 binary bits of the full code. The untrimmed codes for each character are shown in Appendix 2 of the PDP-8 Handbook. The following table shows the function, the subroutine call, and the contents of the AC when the subroutine is called. Control always returns to the main program with the AC and link cleared (except for the last function).

Regular tab is equal to 8 spaces because location TTAB contains 7770 (-8 decimal). Location TTAB may be changed as desired by the programmer. For example, TTAB would contain 7772 for 6 spaces. The tab subroutine is similar to the tab on a regular typewriter. For an excellent description of the tab, refer to the program writeup. The special considerations for typing a string of characters are also contained in the writeup. The most important features are described here.

The last 2 codes in a table of packed, trimmed codes must be 0001. This signals the subroutine to discontinue typing of characters.

To type a carriage return or line feed only, the trimmed code must be preceded by two zeros. (i.e., pack the codes 0015 or 0012, respectively.)

On entering the subroutine, the AC must contain the absolute initial address of the table of trimmed codes. The following table indicates the technique for writing the program in assembler language.

```
        TB11,    2324             /TRIMMED CODES FOR STRING
                 etc.
                   .
                   .
                   .
                   .
        ENDL,    0001             /END SIGNAL
```

In order to specify the typeout of this string of characters, the calling sequence would be:

```
                   .
                   .
              TAD  TLIN           /LOAD AC WITH TABLE 1 DESIGNATOR
              JMS  TSC            /SUBROUTINE CALLED
                                  /RETURN WITH AC AND LINK
```

```
                     .          /CLEARED
                     .                 .
                     .                 .
         TLIN,    TB11          /INTERCOM REGISTER TO TABLE 1
```

Function	Call	Contents of AC
Type 1 character	JMS TY1	Trimmed codes in AC bits 6-11
Type 2 characters	JMS TY2	First code in AC bits 0-5 Second code in AC bits 6-11
Type a string of characters	JMS TSC	Initial address of stored codes
Type CR/LF	JMS TCR	
Type 1 space	JMS TSP	
Type tab	JMS TYT	
Type BCD digit	JMS TDIG	Bits 0-7-0, Bits 8-11 BCD
Convert trimmed code to 8 bit code	JMS CON	Enter with code in AC, bits 6-11 Exit with code in AC, 4-11

D. FLOATING-POINT ROUTINES

1. General

The floating-point package is provided for users who have problems involving decimal fraction arithmetic and do not want to program fixed-point arithmetic. Fixed-point programming involves a thorough knowledge of the limits of the values of the parameters both in the input/output stages and before and after each elementary calculation. To facilitate calculations of this type, a complete set of floating-point instructions are available through the use of the interpretive Floating-Point Arithmetic Package. In this program, the instructions operate on numbers in a floating-point format where the binary point (counterpart to the decimal point) is maintained automatically by the program.

The addition of two floating-point numbers in the computer corresponds to the addition of the two numbers with decimal points. Example: Add the numbers 50 and - 0.635.

$$\begin{array}{r} 50.000 \\ \underline{-0.635} \\ 49.365 \end{array}$$

The decimal points must line up before the numbers can be added correctly. The same addition could have been performed with the numbers expressed as shown.

$$\begin{array}{r} 0.50 \times 10^{2} \\ \underline{-0.635 \times 10^{0}} \\ 0.49365 \times 10^{2} \end{array}$$

Again the numbers must be arranged so that the exponent values are the same, and leading zeros must be supplied.

$$\begin{array}{r} 0.50000 \times 10^{2} \\ \underline{0.00635 \times 10^{2}} \\ 0.49365 \times 10^{2} \end{array}$$

The final fraction contains more digits than either of the two numbers involved. Under certain circumstances, there may be leading zeros which contain no significant information. The floating-point package will normalize the number (i.e., adjust the exponent part of the number to eliminate insignificant zeros).

When writing programs using the floating-point package, the programmer will need to know a conversion procedure for decimal to floating and floating to decimal. A floating-point word in the PDP-8 occupies three registers one register for the exponent and two registers for the mantissa. The following is a brief description of the conversion process.

a. Convert the number $10_{(10)}$ to floating-point. First convert the number to fixed-point binary.

$$10_{(10)} = 1010._{(2)}$$

Move the binary point to the left of the most significant bit and count the places moved. This number is the exponent. Write the mantissa at the extreme left, but not in bit 0. Bit 0 is used to indicate the sign. The number will look like this in octal notation:

$\boxed{0004}$	$\boxed{2400}$	$\boxed{0000}$
Exponent	Mantissa	

b. Convert the number $25_{(10)}$ to floating point.

$$25_{(10)} = 1100\,1._{(2)}$$

Move the binary point and count the places moved. The exponent will be 5. Write the mantissa to the left. The number will be as shown below (octal).

$\boxed{0005}$	$\boxed{3100}$	$\boxed{0000}$
Exponent	Mantissa	

c. Convert $-10_{(10)}$ to floating-point. Convert to binary, working only with the absolute magnitude of the number. Move and count the binary point. Put the exponent and mantissa in as before. Take the 2's complement of the mantissa. The floating-point number looks like this:

$\boxed{0004}$	$\boxed{5400}$	$\boxed{0000}$
Exponent	Mantissa	

d. Convert the number -.0625 to floating-point.

$$.0625_{(10)} = .0001$$

Move and count the binary point to the right. Move the point to the left of the first binary one. The exponent is - 3. Expressing - 3 in 2's complement is $7775_{(8)}$. Put the mantissa at the left, but to the right of bit 0, take the 2's complement of the mantissa yields $\boxed{6000}$ $\boxed{0000}$

The complete floating-point number will look like this.

$\boxed{7775}$ $\boxed{6000}$ $\boxed{0000}$

Exponent Mantissa

2. Interpretive Floating-Point Arithmetic Package

a. Tape Format

This routine is long and assembly in each program is needless. Thus, the program is available only in binary format and may be loaded with the regular binary loader.

b. Internal Format of Floating-Point Words

The heart of the floating-point package is a 3-register block ($44_8 - 46_8$) called the floating accumulator (FA). All data calculations and transfers are made through the floating accumulator, similar to the 12-bit accumulator (AC).

Floating-point data in the FA and in storage registers is composed of two parts:

Mantissa. This portion occupies two registers (45 and 46 of the FA) and contains the normalized data word.

Exponent. This register (44) indicates the magnitude of the exponent. The magnitude limits are $\pm 2047_{10}$.

Unless the programmer is doing special manipulations, he does not need to be concerned with the details of this format. The commands and the program keep consistent control over the data.

NOTE: A floating-point word occupies <u>three</u> registers. If a floating-point storage register is set aside, three (12-bit) registers must also be set aside.

3. Floating-Point I/O Package

This routine performs the same routines that the input-output conversion subroutines perform. When the input routine is called, it will accept a signed decimal number typed on the ASR-33, convert it to floating-point format, and store the number in the floating accumulator. On output it will print out in decimal format the floating-point number contained in the floating accumulator. This routine is only available in binary format.

a. Floating-Point Input Conversion (FINK)

The floating-point input conversion routine is of the form:

±ddd.dddE±dd

The d's represent decimal digits. The E must be typed preceding the digits which represent the decimal exponent of the number. Either of the signs, the decimal point, or the entire exponent may be omitted. If the decimal point is omitted, it is assumed to be to the right of the last significant digit of the mantissa. Any character which is not legally a part of the above format terminates input of the number. The input will allow up to 6 digits in the data word and 2 digits in the exponent. All rubout codes are ignored.

b. Floating-Point Output Conversion (FOUT)

The floating-point output routine converts the number in the floating accumulator to decimal and types it out in the following format:

±.dddddd±dd

The decimal point is always in front of the leftmost digit. The last two digits (i.e., those after the second sign) indicates the exponent of the 6-digit decimal fraction. If the magnitude of the exponent exceeds 99, XX will be printed.

c. Sample Problem to Demonstrate the Use of the Floating-Point Package

Four decimal numbers are typed on the ASR-33 representing the coefficients A, B, C, and D of a third-order nonlinear equation. The floating-point routine calculates the value of the equation and prints the answer on the ASR-33. The values of X and Y are stored on the current page.

The equation is:

$$AX^3 \quad + \quad BX^2Y - CXY^2 \quad + \quad (D/A)Y^3$$

The portion of the program, which will accept the four decimal parameters from the keyboard, calculate the expression, type out the answer, and leave it in the floating accumulator, is shown below.

```
                        .
                        .
                        .
                        .
TRM1,       JMS   I   Z FINK  /READ A INTO F.A.
            JMS   I   Z FPNT  /TRANSFER CONTROL TO FLOATING POINT
            FPUT    A         /STORE IT FOR FUTURE USE
            FMPY    X         /CALCULATE FIRST TERM
            FMPY    X
            FMPY    X
            FPUT    STR1      /STORE F.A. IN STORAGE AREA 1
            FEXT              /EXIT FROM FLOATING POINT MODE
TRM2,       JMS   I   Z FINK  /READ B INTO F.A.
            JMS   I   Z FPNT  /TRANSFER CONTROL TO FLOATING POINT
            FMPY    X         /CALCULATE SECOND TERM
            FMPY    X
            FMPY    X
            FPUT    STR2      /STORE F.A. IN STORAGE AREA 2
            FEXT              /EXIT FROM FLOATING POINT MODE
TRM3,       JMS   I   Z FINK  /READ C INTO F.A.
            JMS   I   Z FPNT  /TRANSFER CONTROL TO FLOATING POINT
            FMPY    X         /CALCULATE THIRD TERM
            FMPY    Y
            FMPY    Y
            FPUT    STR3      /STORE F.A. IN STORAGE AREA 3
            FEXT              /EXIT FROM FLOATING POINT MODE
TRM4,       JMS   I   Z FINK  /READ D INTO F.A.
            JMS   I   Z FPNT  /TRANSFER CONTROL TO FLOATING POINT
            FDIV    A         /CALCULATE FOURTH TERM
            FMPY    Y
            FMPY    Y
            FMPY    Y
ADUP,       FSUB    STR3      /ADD UP THE TERMS
            FADD    STR2
            FADD    STR1
```

```
                FEXT                /EXIT FROM FLOATING POINT MODE
OUTP,           JMS   I   Z FOUT    /PRINT OUT ANSWER
                          .
                          .
                          .
                          .
X,              0                   /X PARAMETER STORED HERE
                0
                0
Y,              0                   /Y PARAMETER STORED HERE
                0
                0
STR1,           0                   /STORAGE REGISTERS
                0
                0
STR2,           0
                0
                0
STR3,           0
                0
                0
A,              0
                0
                0
```

Example 6: Expensive Adding Machine

a. Problem

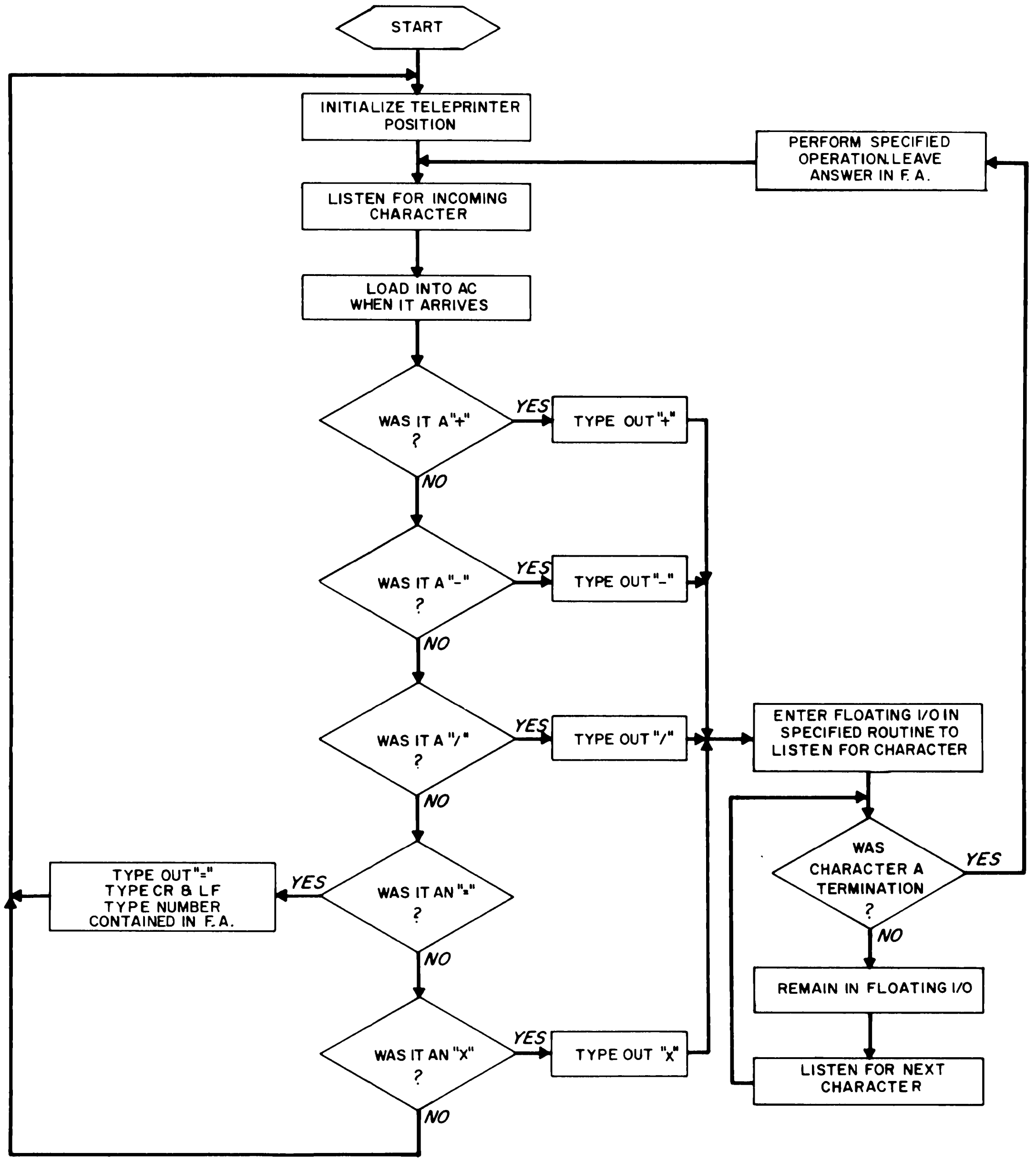
Write a program which will accept decimal digits from the ASR-33 preceded by an operator (+, -, X, or /) and terminated by a nondigit character (a space is suggested). The indicated operation will be performed in floating-point arithmetic. When an equal sign is typed, the routine will generate a carriage return/line feed combination and print the value of the expression. If any other character is typed as input, the routine will ignore it and return to listen for a meaningful character.

The program that is shown as a solution does not inherently contain an error correction facility. However, if an error is made, simply type a space followed by an equal sign. The incorrect answer will be typed out and the operator may return to type in the correct sequence of digits.

The value of the expression up to that point is calculated after each termination character. The ordinary rules of algebra concerning order of operations do not hold. This routine is a calculating routine not a compiling routine.

NOTE: Use a 4K version of the PAL Assembler marked PAL 4K33 (EOT) to assemble a symbolic program using the floating-point commands.

b. Flow Diagram


START
INITIALIZE TELEPRINTER POSITION
LISTEN FOR INCOMING CHARACTER
LOAD INTO AC WHEN IT ARRIVES
WAS IT A "+" ?
YES
TYPE OUT "+"
NO
WAS IT A "-" ?
TYPE OUT "-"
WAS IT A "/" ?
TYPE OUT "/"
WAS IT AN "=" ?
TYPE OUT "=" TYPE CR & LF TYPE NUMBER CONTAINED IN F.A.
WAS IT AN "X" ?
TYPE OUT "X"
ENTER FLOATING I/O IN SPECIFIED ROUTINE TO LISTEN FOR CHARACTER
WAS CHARACTER A TERMINATION ?
REMAIN IN FLOATING I/O
LISTEN FOR NEXT CHARACTER
PERFORM SPECIFIED OPERATION. LEAVE ANSWER IN F.A.

c. Program Listing

```
/EXPENSIVE ADDING MACHINE - FLOATING POINT
/SET UP THE INTER-COM REGISTERS FOR FLOATING POINT
*5
FINK,           7400
FOUT,           7200
FPNT,           5600
/
/TYPEOUT SUBROUTINE FOR PAGE ZERO
*173
TYPE,           0                         /LINKING REGISTER TO MAIN PROGRAM
                TSF
                JMP      .-1
                TLS
                JMP    I  TYPE
/
/THE STARTING ADDRESS IS 200
/INITIALIZATION ROUTINE
BEG,            KCC                       /CLEAR KEYBOARD FLAG AND AC
                TLS                       /GIVE TYPEOUT COMMAND TO SET FLAG
                TAD      C215             /INITIALIZE TELEPRINTER POSITION
                JMS      Z  TYPE
                CLA
                TAD      C212
                JMS      Z  TYPE
                CLA
                DCA      ACC              /CLEAR OUT SPACE FOR
                DCA      ACC+1            /FLOATING NUMBER
                DCA      ACC+2
/
/SECTION TO LISTEN FOR INCOMING CHARACTER
GOPR,           KSF                       /GET OPERATOR
                JMP      .-1
                KRB
/
/DISPATCH ROUTINE
                TAD      M253             /ADD TO CODE MINUS 253
                SNA                       /WAS IT A PLUS SIGN, CODE 253?
                JMP      ADD              /YES, JUMP TO ADD
                TAD      M2               /NO, ADD -2 TO RESULT
                SNA                       /WAS IT A MINUS SIGN, CODE 255?
                JMP      SUB              /YES, JUMP TO SUBTRACT
                TAD      M2               /NO, ADD -2 TO RESULT
                SNA                       /WAS IT A SLASH, CODE 257?
                JMP      DVD              /YES, JUMP TO DIVIDE
                TAD      M16              /WAS IT AN =, CODE 275?
                JMP      EQL              /YES, JUMP TO EQUATE
```

```
               TAD     M33        /ADD - 33 TO RESULT
               SNA                /WAS IT AN "X", CODE 330?
               JMP     MLT        /YES, JUMP TO MULTIPLY
               JMP     GOPR       /ILLEGAL CHARACTER, TRY AGAIN
/ADD ROUTINE
ADD,           TAD     C253
               JMS     Z TYPE     /TYPE A "+" SIGN
               CLA
               TAD     C240
               JMS     Z TYPE     /TYPE A SPACE AFTER OPERATOR
               JMS     I Z FINK   /ENTER FLOATING I/O TO GET NUMBER
               JMS     I Z FPNT   /ENTER INTERPRETIVE MODE
               FADD    ACC        /ADD CONTENTS OF "ACC" TO CONTENTS OF
                                  /FLOATING AC
               FPUT    ACC        /STORE IN ACC
               FEXT               /LEAVE INTERPRETIVE MODE
               JMP     GOPR
/END OF ADDITION.  GET NEXT OPERATOR
/
/SUBTRACT ROUTINE
/THE FOLLOWING ROUTINES ARE ESSENTIALLY THE SAME AS THE
/"ADD" ROUTINE SO WILL NOT BE COMMENTED
SUB,           TAD     C255
               JMS     Z TYPE     /TYPE MINUS SIGN
               CLA
               TAD     C240
               JMS     Z TYPE
               JMS     I Z FINK
               JMS     I Z FPNT
               FPUT    ACC+3
               FGET    ACC
               FSUB    ACC+3
               FPUT    ACC
               FEXT
               JMP     GOPR
/END OF SUBTRACTION.  GET NEXT OPERATOR
/
/DIVIDE ROUTINE
DVD,           TAD     C257       /TYPE SLASH FOR DIVIDE
               JMS     Z TYPE
               CLA
               TAD     C240
               JMS     Z TYPE
               JMS     I Z FINK
               JMS     I Z FPNT
               FPUT    ACC+3
               FGET    ACC
               FDIV    ACC+3
```

```
                FPUT    ACC
                FEXT
                JMP     GOPR
/END  OF DIVISION.  GET NEXT CHARACTER
/
/MULTIPLY ROUTINE
MLT,            TAD     C330
                JMS     Z TYPE      /TYPE MULTIPLY OPERATOR, "X"
                CLA
                TAD     C240
                JMS     Z TYPE
                JMS     I Z FINK
                JMS     I Z FPNT
                FMPY    ACC
                FPUT    ACC
                FEXT
                JMP     GOPR
/END OF MULTIPLICATION.  GET NEXT OPERATOR
/
/EQUALS ROUTINE
EQL,            TAD     C275
                JMS     Z TYPE      /TYPE EQUALS OPERATOR
                CLA
                TAD     C215
                JMS     Z TYPE      /TYPE CR
                CLA
                TAD     C212
                JMS     Z TYPE      /TYPE LF
                JMS     I Z FPNT
                FGET    ACC         /GET ANSWER IN F.A.
                FEXT                /LEAVE INTERPRETIVE MODE
                JMS     I Z FOUT    /TYPE OUT ANSWER
                CLA
                JMP     BEG+2
/JUMP TO INITIALIZE PROGRAM FOR NEXT CALCULATION
/CONSTANTS
C215,           215
C212,           212
M253,           0-253
M2,             0-2
M16,            0-16
M33,            0-33
C253,           253
C240,           240
C255,           255
C257,           257
```

```
C330,          330
C275,          275
ACC,           0
               0
               0
$
```

CHAPTER II

INTRODUCTION TO THE PDP-8 DEC DEBUGGING TAPE (DDT-8)

A. GENERAL

DDT-8 is a debugging program for the PDP-8 computer which allows the programmer to correct a binary program which contains errors. DDT-8 occupies locations 5240 to 7600 and location 4. Its internal symbol table extends down to 5000 and allows an input of 200 (decimal) symbols. Its starting address is 5400. DDT-8 is operated from the ASR-33 Keyboard, and the output may be on the ASR-33 Printer-Punch or the High-Speed Punch, depending on the operator's choice and available equipment.

B. DEFINITIONS

1. Symbol - A string of letters or letters and numbers, the first character of which must be a letter. Up to six characters may be used.

2. Number - A string of up to four octal digits.

3. Expression - A string of symbols and numbers separated by a plus (+), minus (-), or space.

4. Open Register - When a register is opened, its contents are printed out, and the register becomes available for modification.

5. Closed Register - When a register is closed, any modifications requested are made and further access to the register is denied until it is opened again.

DDT will respond to operator errors by typing a question mark (?) and will ignore the error.

C. LOADING PROCEDURE

1. Load binary object tape into memory using the Binary Loader.

2. Load DDT-8 into memory using the Binary Loader.

3. Set switch register to 5400, depress LOAD ADDRESS, depress START.

4. DDT-8 is now running and awaiting the first command.

D. CONTROL CHARACTERS

NOTE: Use of a left bracket ([) indicates depression of ALT MODE key which is echoed as a left bracket ([). The switch register is positive when the leftmost switch (bit) is down (0) and negative when it is up (1).

1. Arithmetic

a. Plus (+) - Separation character meaning arithmetic plus.

b. Minus (-) - Separation character meaning arithmetic minus.

c. Space - Separation character indicating that the following expression is to be taken as an address.

d. Period (.) - Has the numerical value of the current location.

e. Equal (=) - Convert the last expression printed into its octal equivalent.

2. Program Examination and Modification

a. Slash (/) - Open the register whose address precedes the slash. The operation causes the contents of the register to be printed out and makes it available for modification.

b. Carriage Return (↲) - Closes the register opened by the slash. Any expression typed before carriage return will become the contents of the register.

c. Line Feed - Has the same effect as carriage return, however, it opens the next consecutive register.

d. Up Arrow (↑) - Opens the register specified in the address portion of the instruction contained in the open register. Use of this character does <u>not</u> follow an indirect chain.

e. [O - Convert to octal mode. Causes contents of registers examined to be typed out in octal.

f. [S - Convert to symbolic mode. Causes contents of registers examined to be typed out in symbolic.

3. <u>Search</u>

a. N[W - Causes a word search for the expression or number N. The search will take place between the limits in locations [L and [U of DDT and the contents of the registers will be masked (logical ANDed) by the contents of [M.

b. [L - Permits access to the register which contains the lower limit of the search (operates in the same way as the slash), originally 0001.

c. [U - Permits access to the register which contains the upper limit of the search, initially 5000.

d. [M - Allows access to the register which contains the mask of the search, originally 7777.

4. <u>Breakpoints</u>

a. Y[B - Inserts a breakpoint before location Y. The breakpoint is inserted only when a G or C command is given. When a breakpoint is encountered, control goes to DDT and the AC and link are saved.

The location of the breakpoint is typed out followed by a right parenthesis and the contents of the AC at that point. Breakpoints may be removed by typing [B alone with no address indicated.

NOTE: Only one breakpoint may exist at any time. The last specified breakpoint is the one which DDT establishes when the GO or CONTINUE command is given.

b. Y[G - Go to address Y and begin execution of the user's program at that point. Control goes to the user's program and its execution begins. The computer will stay in the user's program until a breakpoint is reached at which time control will return to DDT. If no trap occurs (no breakpoint established or reached due to branching) the computer will stay in the user's program until a programmed halt occurs, at which time the computer will halt. DDT will have to be restarted by putting 5400 in the switch register, depressing LOAD ADDRESS, and then depressing START.

c. [C - Continue from trap. This command will cause control to go from DDT to the user's program with the AC and link returned to their original condition before the breakpoint was encountered. Control will stay in the user's program until a breakpoint is reached, at which time control goes back to DDT. If the breakpoint is in a loop and it is desired to go around the loop a specific number of times, the command N[C may be typed, at which time DDT will allow the program to loop N times before the breakpoint is encountered.

d. [A - Opens the register which contains the contents of the AC at a breakpoint.

e. [Y - Opens the register which contains the contents of the link at a breakpoint.

5. Symbol Definition

a. [R - Read symbol table. If it is desired to define symbols to use in the symbolic mode and these symbols are on the binary tape produced from the Assembler, place this portion of the tape in the reader, type [R, and then turn on the reader. If the symbols on the tape are less than 200 (decimal), they will be read into the buffer area of DDT which is reserved for the definitions of tags used in a program. Then hit CONTINUE. If a new symbol table is being created, the contents of the switch register should be negative; however, if it is desired that the symbols coming in append the table currently in the buffer, the contents of the switch register should be positive. To define symbols not on a binary tape, use the following procedure:

1. Set switch register to positive or negative.
2. Type [R.
3. Type CARRIAGE RETURN—LINE FEED.
4. Type symbol to be defined, at least one space, the address of the symbol, CARRIAGE RETURN—LINE FEED.
5. Repeat step 4 for each symbol to be defined.
6. When all symbols have been defined, type EOT and hit CONTINUE. After the symbol table has been read in by DDT, the bottom address of the external symbol table will be typed out. The operator's program must not exceed this address.

6. Punch

a. Y;Z[P - Punch contents of locations Y through Z inclusive in binary format on tape.

b. [T - Punch leader-trailer.

c. [E - Punch checksum and trailer.

d. Procedure to punch binary tape on ASR-33.

1. Turn the Teletype to on line; turn punch off.
2. Set switch register to negative.
3. Type [T, turn punch on, depress CONTINUE. Leader will be generated.
4. Turn punch off, type in low address of area to be punched followed by a semicolon (;), then the high address of the area to be punched, then type [P.
5. Turn punch on, depress CONTINUE.
6. Repeat steps 4 and 5 for more blocks to be punched.
7. With Teletype on line and punch off, punch checksum and trailer by typing [E, turn punch on and hit CONTINUE.

8. Turn punch off after tape is punched. DDT is now awaiting the next command.

To use the high-speed punch (75A), use the same procedure as with ASR-33 except the switch register should be set to positive and the punch on ASR-33 is left off.

d i g i t a l

5233 PRINTED IN U.S.A. 5-3/65