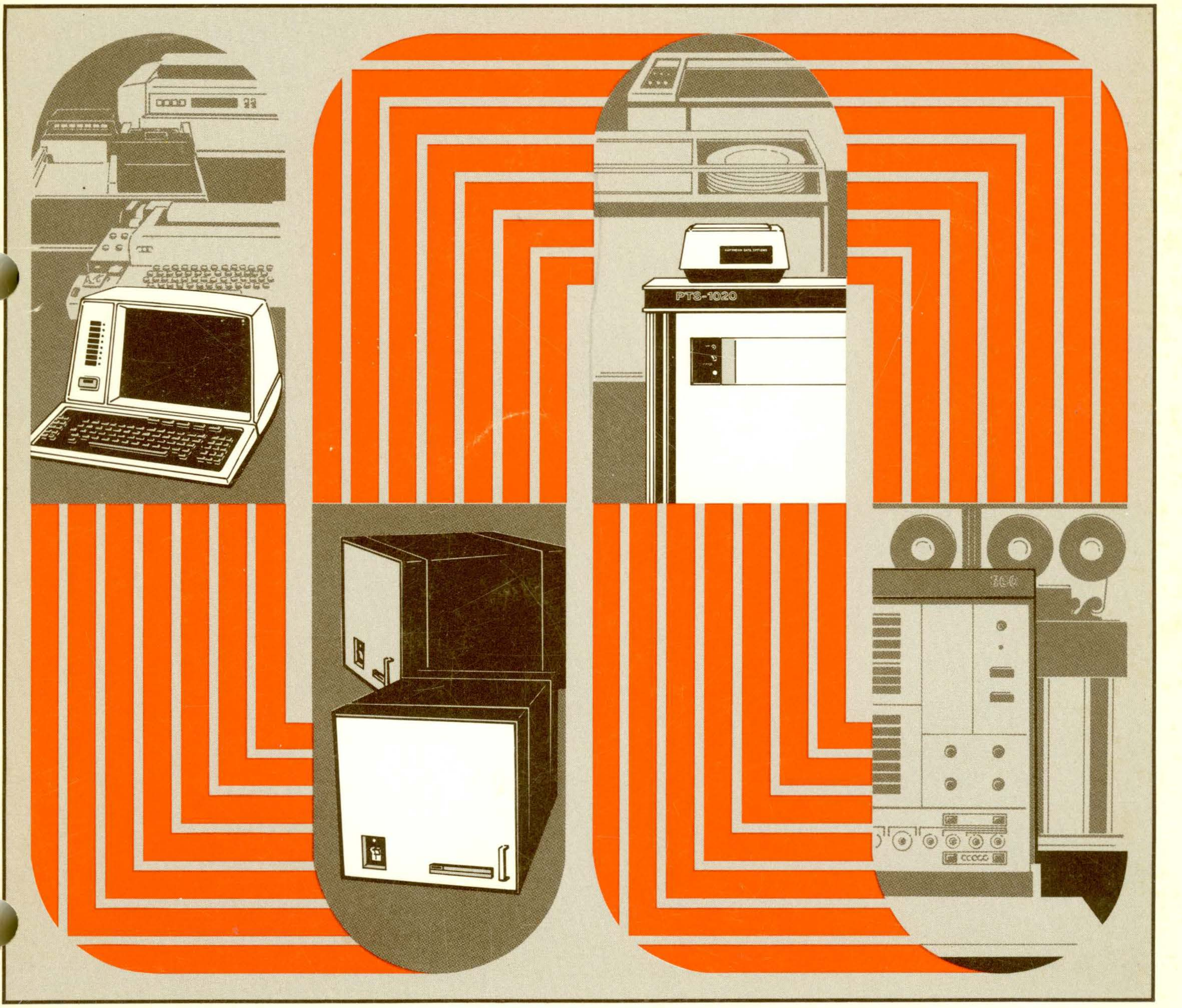
RAYTHEON

PROGRAMERS HANDBOOK PTS-100

RAYTHEON DATA SYSTEMS



44-7644

PROGRAMMABLE TERMINAL SYSTEM PTS-100 PROGRAMERS HANDBOOK

Revision 1

June 1973

PREPARED BY

RAYTHEON DATA SYSTEMS
1415 BOSTON-PROVIDENCE TURNPIKE
NORWOOD, MASSACHUSETTS 02062

PREFACE

This publication was prepared as a reference handbook for the programing personnel of PTS-100 users. It presents the information necessary to write programs to be executed on the PTS-100 and to use the software support systems provided with the PTS-100. The handbook is organized in distinct parts, as follows:

PART 1: This part of the handbook presents a programer overview of the PTS-100 operating environment and software support available to users of the PTS-100.

PART 2: This part of the handbook presents detailed descriptions of the Assembler language and programing features available to PTS-100 programers.

PART 3: Presented in this part of the handbook are "how to use" descriptions of the utility programs supplied with the PTS-100.

PART 4: This portion of the handbook describes the macro library files available to users.

The Table of Contents on the following page indicates the general coverage of information in this handbook. Each of the four parts of the handbook includes a detailed table of contents.

TABLE OF CONTENTS

PART 1

PTS-100 PROGRAMMER OVERVIEW

PART 1

PTS-100 PROGRAMER OVERVIEW

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

PART 1. PTS-100 PROGRAMER OVERVIEW

Section 1. GENERAL INTRODUCTION TO THE PTS-100

The Programable Terminal System 100 (PTS-100) contains a general purpose computer consisting of a central processor unit (CPU), a modular semiconductor (MOS) main memory expandable from 8192 to 65536 bytes, a control panel, and both low speed and high speed device controllers. A customer engineer's console is also optionally available as an expanded console for debugging programs.

The low speed controllers accommodate peripheral devices that operate at data transfer rates at or below 9600 bits per second, including CRT display devices, card readers, serial printers, teletype devices, cassette tape drives, etc. The high speed controllers provide interface for peripheral devices that operate at data transfer rates in excess of 9600 bits per second such as disc devices, magnetic tape transports, and host computer channel interface devices such as the IBM 360/370 multiplexer/selector channel interface.

The memory size and peripheral equipment configuration of a given PTS-100 installation are flexible so that individual users can select those components especially suited to their application processing needs. Certain device controllers and the peripheral devices that may be attached to them are offered to users as "standard" equipment. That is, standard hardware equipment is supported by the PTS-100 operating system--the Input/Output Control System (IOCS) monitor--which interfaces the CPU, I/O devices, and the programs to be executed on the PTS-100. The IOCS monitor, described in Section 2 following, is specially tailored for each standard PTS-100 equipment configuration via the RDS-supplied System Generator program, described in Part 3 of this handbook, and the Assembler program, described in Part 2. If nonstandard devices are to be attached to the PTS-100, the user must modify the IOCS monitor to accommodate the devices, as described in appropriate areas within this handbook.

Since the total equipment configuration of a PTS-100 installation is user-selected, no attempt has been made in this handbook to describe the operational characteristics of specific devices. For this information, the reader should consult the PTS-100 Reference Manual. Certain I/O devices are required by various software programs offered with the PTS-100, as noted in the detailed descriptions of these support systems throughout this handbook.

Regardless of the memory size and types of I/O devices attached to a given PTS-100, the CPU and the control console are standard for all installations. For convenience, information relating to programing the system and programer usage of the CPU and control console is presented in summary form on subsequent pages of this section. For detailed descriptions of the hardware characteristics of these components of the PTS-100, see the PTS-100 Reference Manual.

1.1 Central Processor Unit

The central processor unit (CPU) executes programs stored in main memory and controls data transfers between main memory and I/O devices. The CPU communicates with executing programs via registers, four of which are program addressable. CPU communication with main memory is via the 16-bit processor memory bus (PMB). Communication between the CPU and I/O devices takes place over the input output bus (IOB).

The CPU utilizes a 16-bit word, and is capable of executing one word (short) or two word (long) instructions. Each word is composed of two 8-bit bytes, or characters. Memory addressing is by word (16 bits) or byte (8 bits). The following methods of addressing may be used with or without single level indirect mode:

- Absolute addressing over the maximum memory capacity of 65,536 bytes.
- Dynamic page addressing of ±128 words relative to the program counter register in short instructions and ±32768 bytes in long instructions.
- Indexed addressing via index register 1 or index register 2 of +128 words in short instructions or ±32768 bytes in long instructions.

A 16-bit byte displacement value is used to compute the effective address of long instructions, and a 7-bit word displacement value is used to compute the effective addresses of short instructions.

CPU instructions are termed "executable." That is, they are assembly language statements that the Assembler translates to executable machine language format. Executable instructions are provided to accomplish the following:

Arithmetic operations

Branches in program execution

Loading CPU registers with data values stored in memory locations

Storing contents of CPU registers in memory locations

Comparative tests of data values

Logical testing of data values

Interrupt masking and level changes

I/O operations

For a detailed description of executable instructions and other Assembler language statements, see Part 2 of this handbook.

The CPU communicates with executing programs via registers within itself. Four of the registers are program addressable. They are the program counter, the accumulator, index register 1, and index register 2. These registers are described below.

1.1.1 Program Counter (PC)

The program counter is a 16-bit register that supplies the addresses of instructions to be fetched from main memory, and hence directs the program execution sequence. Normally, as an instruction word is fetched the PC contents are incremented by 2 to advance the byte-oriented address to that of the next program instruction word, or the second word of a double word instruction. This sequencing is disrupted only by the occurrence of a branch instruction or the CPU response to a priority interrupt. In the first case, if the branch instruction's conditions are satisfied, its effective address replaces the current PC content and initiates a sequential change in program processing. In the case of a priority interrupt, the CPU hardware automatically saves the interrupted program's current PC content, and enters the effective address of the appropriate IOCS monitor interrupt servicing routine in the PC. The interrupt servicing routine effects an interrupt return via its last executable instruction, which restores the saved content of the PC, thus restarting the interrupted program at the point of interrupt.

1.1.2 Accumulator (AC)

The 16-bit accumulator (AC) register is the principal data handling register for the CPU and is involved in the execution of most instructions. The accumulator's most significant bit (MSB), bit 0, is employed as the sign bit (value = 0 for positive, and = 1 for negative) for arithmetic operations, leaving 15 bits for fixed-point data representation of the following range of values:

$$-2^{15} \leq n \leq 2^{15} - 1$$

or, in decimal equivalents:

$$-32,768 \leq n \leq 32,767$$

1.1.3 Index Registers 1 and 2 (X1 and X2)

Index registers 1 and 2 are both 16-bit registers used, primarily, to provide address components for the computation of effective addresses. They may also be used as temporary storage registers for data and address references.

1.2 Control Panel

The control panel of the PTS-100 computer provides for primary power and initialization of computer processing. The power is controlled by the POWER ON/OFF switch on the console.

Initialization of processing is effected by depressing the IPL (Initial Program Load) push-button on the console. Whenever the IPL button is depressed, a hardwired IPL bootstrap routine is activated in the Read Only Memory (ROM). The IPL bootstrap routine then performs the following:

- Clears all main memory locations to zero values.
- Transfers a section of itself into memory, beginning at location zero.
- Activates the transferred section, which then transfers four words (i.e., 64 bits) from manually set switches to word locations 3 through 6 of main memory, and:

  Determines the address of the loading device from the first word of switch data

  Issues a read command to the loading device to cause the six-byte header record of the program to be loaded into main memory. The header record contains the following:

    Load address of the program to be loaded

    Byte count (number of characters) to be read

    Execution (starting) address of the program to be loaded

  Reads one record (the program to be loaded) into consecutive memory locations, starting at the load address, until the number of characters specified by the byte count have been loaded.

  Transfers control to the loaded program and starts its execution at the starting address specified in the header record.

The Initial Program Load bootstrap is required to load a one record binary program into PTS-100 memory. Under typical operating conditions, the one record binary program is the Piggyback Loader, which in turn loads the Absolute/Relocating Loader. The Absolute/ Relocating Loader must be used to load object programs produced by the PTS-100 Assembler. Before programs other than the Piggyback Loader can be loaded via the IPL button, they must have been assembled by the PTS-100 Assembler, loaded by the Absolute/Relocating Loader, and then dumped from main memory to a cassette tape or punched paper tape device. The procedures for loading and dumping programs are described in Part 3 of this handbook.

Section 2. PTS-100 OPERATING SYSTEM

The operating system of the PTS-100 is the Input/Output Control System (IOCS) which monitors the servicing of interrupts from the multilevel interrupt system, described in detail in Part 2 of this handbook. That is, the IOCS monitor optimizes I/O resources in the PTS-100 real time interrupt environment by interfacing executing programs, the CPU, and I/O devices. For any given PTS-100, a specially tailored IOCS is created by the System Generator and Assembler programs, as described elsewhere in this publication.

The IOCS monitor is composed of two major components: the I/O Control Nucleus and the Physical I/O Routines. The Nucleus interfaces between executing systems and applications programs and the Physical I/O Routines, which issue I/O commands to peripheral devices attached to the PTS-100 and receive and initiate processing of I/O interrupts from devices in the equipment configuration. The interface relationship of the executing object programs, the IOCS monitor, and the peripheral devices attached to the PTS-100 is illustrated in figure 1-1. The structural and operating characteristics of the Nucleus and the Physical I/O Routines are described in the following subsections.

2.1 I/O Control Nucleus

The I/O Control Nucleus contains three groups of routines:

- Level Service Routines, which perform the following functions:

  Service interrupts from I/O devices and object program calls

  Service "unknown" interrupts

  Restore interrupt levels after interrupts from other levels have been serviced.

- Monitor Service Call Routines, which perform the processing required to open, close, and initialize devices, to perform I/O operations, and to exit from the system when program processing is completed

- An optional Monitor Log Service Routine, which produces 32-character messages on the System Log device.

Within a given IOCS monitor, one set of Level Service Routines (LSRs) is generated for each of the interrupt levels 1 - 8. That is, these routines service interrupts that occur on the external (device) interrupt levels 1 - 8, to which devices have been previously assigned. Each set of LSRs contains the following functional routines:

A level service entry and save routine

A linkage to all Device Service Routines within the Physical I/O Routines

An "unknown" interrupt handling routine

A level restore and exit routine.

For each of the 11 interrupt levels, an interrupt packet, described in Part 2 of this handbook, exists in the IOCS monitor. For interrupt levels 1 - 8, the starting address of the associated LSR is stored in the interrupt packet associated with the interrupt level. Thus, when an interrupt occurs, control is turned over to the LSR associated with the level at which the interrupt occurred. The LSR then transfers control to one of the Device Service Routines (DSRs) associated with one or more devices assigned to the corresponding interrupt level. The DSR that receives control checks to see if its associated physical I/O device has an interrupt pending. If so, the DSR calls the appropriate device driver routine to service the interrupt, after which control returns to the LSR, which then returns

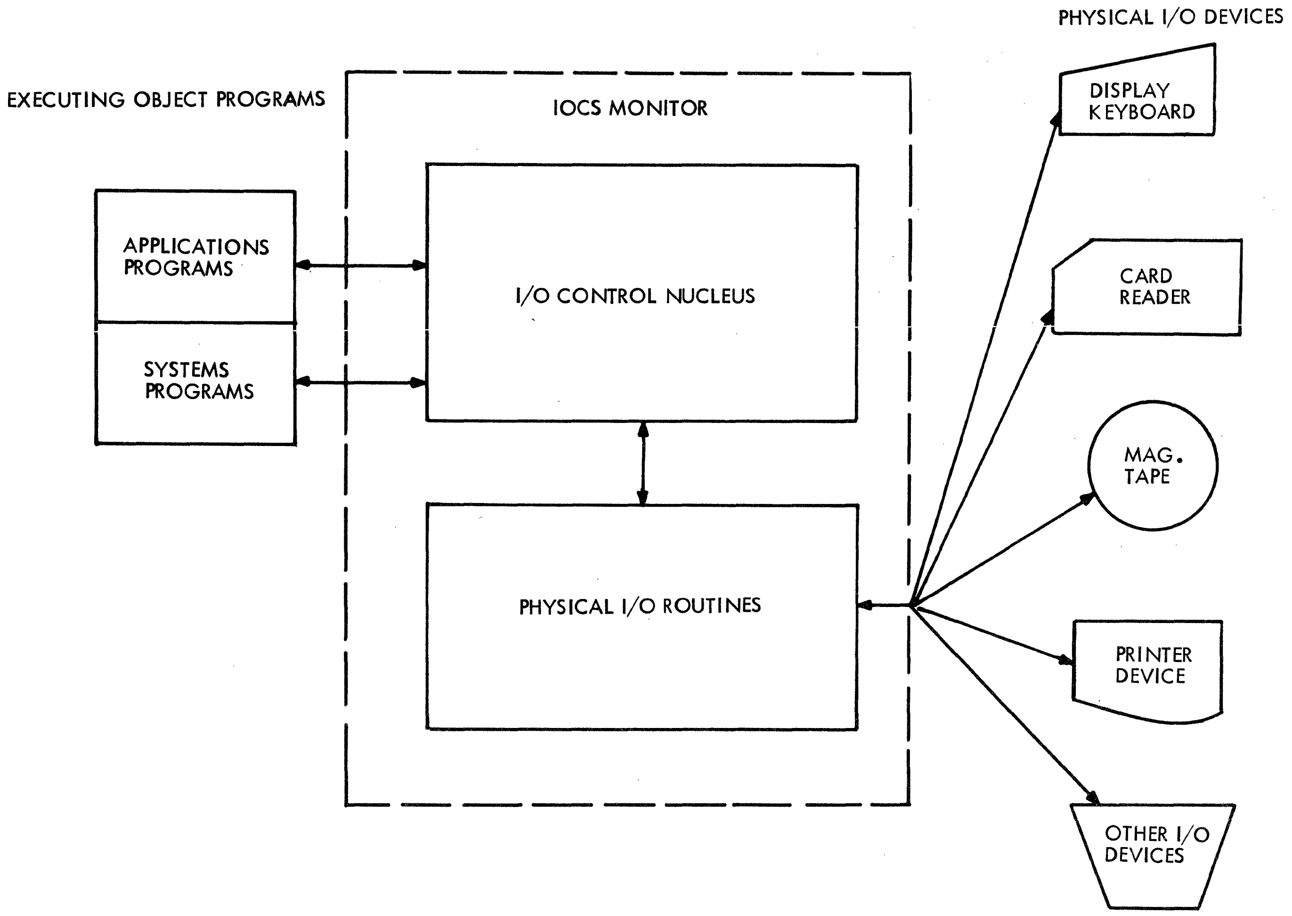


Figure 1-1. Interface Relationship of Executing Programs, IOCS Monitor, and Physical I/O Devices in the Equipment Configuration of the PTS-100

control to the previous interrupt level. If no interrupt was pending, the next DSR, if any, on the interrupt level is polled. This polling procedure continues until the interrupting device is located. If no device on this level issued an interrupt, the unknown interrupt error routine for this level is entered to log an error message on the System Log device. The level service restore routine then restores the registers of the interrupted program, and returns control to the interrupt level from which the LSR received control.

The Monitor Service Call (MSC) routines are entered by the execution of an MSC (i.e., trap) instruction within the executing program. MSC routines operate at interrupt priority level 9.

The routines that may be called by the executing program to perform I/O services are the following:

- OPEN routine, which opens a logical unit (i.e., device) by initializing the device and its related software controls so that I/O operations can subsequently be performed on the device
- IO ACTion routine, which responds to and queues requests for input/output operations on specific devices
- CLOSE routine, which immediately closes a specific logical unit (i.e., device) at the end of a processing job or to facilitate an error recovery

- INITialization routine, which resets all I/O devices on the system

- EXIT routine, which is called when an executing program exits from the system. This routine issues a message that the program has exited and waits for manual intervention to specify restart of processing.

The procedures for calling these routines from within the program are described in detail in Part 2 of this handbook.

The Log Service Routine prints monitor messages on the System Log device, if the device was assigned at system generation time. These messages can be used for error logging, operator notes, or any other short (i. e., 32-character) messages. Monitor message logging does not interfere with executing program output of messages to the System Log device. In fact, no special provisions or precautions need be made within the executing object program.

2.2 Physical I/O Routines

The Physical I/O Routines of the IOCS monitor handle the device-specific hardware/software interface. They service I/O device interrupts, control the transfer of data to and from the physical I/O devices, and initiate new I/O actions when appropriate. They also detect hardware errors and report them to executing object programs and in some cases, perform corrective actions to clear error conditions. The Physical I/O Routines include Device Driver Routines and Device Service Routines.

There is a Device Driver Routine and a Device Service Routine for each type of device in the standard PTS-100 equipment configuration. The Device Driver is called when an I/O request has been queued in the logical Input/Output Control Queue (IOCQ) table and the channel is inactive. The Driver uses the information in the appropriate entry of the IOCQ to set up the Physical I/O Control table and initiate the requested I/O action. It calls the Driver Common routine to perform any required device-independent processing.

At system generation time, a Device Service routine is generated for each device assigned to a given external interrupt level. These routines identify the cause of an interrupt, update control and status fields in the IOCQ entry, take any required actions, and then initiate action on the next I/O request that is queued as an entry in the IOCQ table (see subsection 2. 3. 1).

The level service routine for a given external interrupt level activates the appropriate Device Service routine each time an interrupt is queued for its associated device. When several devices are assigned to one interrupt level, there is a Device Service routine for each assigned device. The relative priority of several DSRs on the same interrupt level is specified at system generation time. The Device Service routines run with interrupts enabled, so that an interrupt of a higher level can always interrupt processing of a lower priority interrupt without delay.

2.3 IOCS Systems Records

There are five kinds of systems records used by the IOCS monitor:

- I/O request records, which include:

  Programer defined File Input/Output Block (FIOB), which has been assembled into the executing program

  Program defined Input/Output Control Queue (IOCQ) table entry, used by IOCS to queue I/O requests

  IOCS generated Physical Input/Output Table (PIOT)

- Two physical control records:

  Input/Output Control Table (IOCT)

  Interrupt Packets

- Two optional special function records, the Search and Translate Tables, defined by the programer within the program to be assembled and executed

- Monitor Service Call Vector Table

- Logging tables

The content and usage of each of these records are described in the following subsections.

2.3.1 I/O Request Records

There are three kinds of I/O request records: the FIOB (File I/O Block), the IOCQ (Input/Output Control Queue) table entry, and the PIOT (Physical I/O Table). The relationship of the required I/O request records for one logical unit is shown in figure 1-2.

The FIOB contains the programer defined information on the I/O request. That is, the programer defines the FIOB in the source program. At program execution time, the FIOB information is passed to IOCS when the executing program issues an IO ACTion service request. The IOCS extracts the information from the FIOB and enters it into the next entry of the IOCQ table. When the time comes for the hardware to perform the requested I/O operation, IOCS moves the information from the IOCQ entry into the PIOT, where the Physical I/O routines and hardware devices can access and use it.

While the main data flow is from the executing program to IOCS and then to the hardware, there is some status information that the hardware transmits to IOCS for the executing program. For example, when the requested I/O operation is completed, the hardware reports the logical and physical status to IOCS, which makes it available to the program via the IOCQ entry. The logical status informs the executing program that the requested service has been completed, and the physical status indicates the type of completion that occurred.

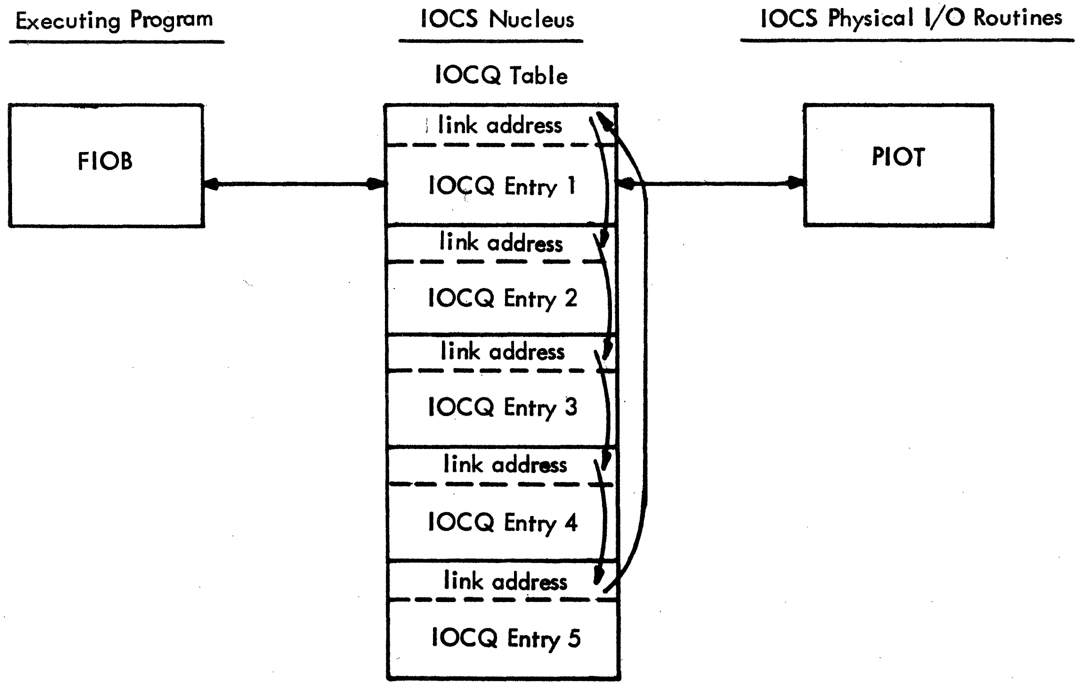
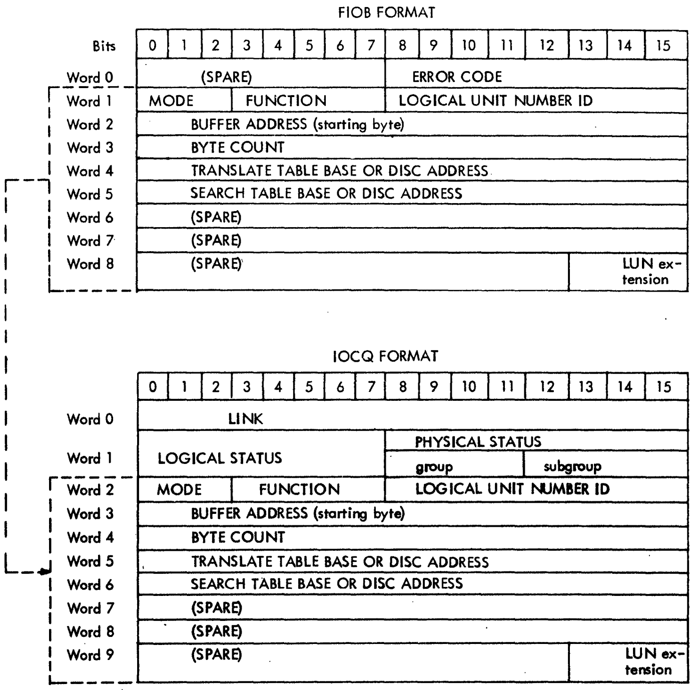


Figure 1-2. Relationship of I/O Request Records

For each IO ACTion service to be requested from IOCS, the executing program must contain a 9-word FIOB describing the parameters of the request. For each I/O device channel to be used by the executing program, the assembly language programer must set up a 10-word IOCQ table entry to be placed in the IOCQ table when the I/O request is queued by IOCS. The first word of the IOCQ entry (Word 0) contains the address of the next IOCQ entry, which is specified by the program. The second word (Word 1) is used by the hardware to report the status of the request to the program. The remaining words are filled by IOCS from Words 1 through 8 of the FIOB when the request is queued. The format and usage of the FIOB and IOCQ entry are illustrated in figure 1-3. See Part 2 of this handbook for detailed descriptions of the content of the FIOB and IOCQ entry.

When the executing program issues an IOACT request, the FIOB information is accessed by the IOCS monitor, which extracts the I/O request information in words 1-8 and enters it into the next entry of the IOCQ table. When the queued I/O request is to be serviced, the Physical I/O Routines of the monitor extract the IOCQ entry information and place it in the Physical I/O Table (PIOT) for use of the hardware device controller that performs the requested I/O operation. When the I/O request has been serviced, the device service routine returns the logical and physical status to Word 1 of the IOCQ entry.

The format of the PIOT is shown in figure 1-4. Notice that the MODE and FUNCtion information occupies the first half of Word 0 of the PIOT, and the last half of the word contains an 8-bit Interrupt Mask. This mask is set up by the device driver routine for the associated device. Each bit of the mask corresponds with a bit in the Interrupt Condition Byte (ICB) in the hardware controller for the device. When the device controller detects an ICB bit setting, indicating an interrupt condition, it compares the



FIOB FORMAT

Bits	0 1 2	3 4 5 6 7	8 9 10 11 12 13	14 15
Word 0	(SPARE)		ERROR CODE	
Word 1	MODE	FUNCTION	LOGICAL UNIT NUMBER ID	
Word 2	BUFFER ADDRESS (starting byte)			
Word 3	BYTE COUNT			
Word 4	TRANSLATE TABLE BASE OR DISC ADDRESS			
Word 5	SEARCH TABLE BASE OR DISC ADDRESS			
Word 6	(SPARE)			
Word 7	(SPARE)			
Word 8	(SPARE)			LUN extension

IOCQ FORMAT

	0 1 2	3 4 5 6 7	8 9 10 11	12 13 14 15
Word 0	LINK			
Word 1	LOGICAL STATUS		PHYSICAL STATUS group	subgroup
Word 2	MODE	FUNCTION	LOGICAL UNIT NUMBER ID	
Word 3	BUFFER ADDRESS (starting byte)			
Word 4	BYTE COUNT			
Word 5	TRANSLATE TABLE BASE OR DISC ADDRESS			
Word 6	SEARCH TABLE BASE OR DISC ADDRESS			
Word 7	(SPARE)			
Word 8	(SPARE)			
Word 9	(SPARE)			LUN extension

Figure 1-3. Format and Usage of FIOB and IOCQ Entry by IOCS

Word	0 1 2	3 4 5 6 7	8 9 10 11 12 13 14 15
0	MODE	FUNCTION	INTERRUPT MASK BYTE
1	BUFFER ADDRESS		
2	BYTE COUNT		
3	TRANSLATE TABLE BASE OR DISC ADDRESS		
4	SEARCH TABLE BASE OR DISC ADDRESS		
5	BUFFER ADDRESS - C		
6	BYTE COUNT - C		
7	(SPARE)		

Figure 1-4. Format of the PIOT Generated by IOCS

bit with the corresponding bit in the Interrupt Mask to determine whether the interrupt should be "allowed" (i.e., generated). If an interrupt is generated, the appropriate LSR receives control, calls the Device Service routine for the interrupt level, and the interrupt processing is performed. If the interrupt is not "allowed," then an invalid interrupt is logged.

IOCS queueing of I/O requests allows the executing program to operate asynchronously with I/O data transfer operations. The number of entries in a particular IOCQ table is defined by the assembly language programer when the source program is coded.

2.3.2 Physical Control Records

The IOCS monitor uses two physical records to control I/O devices and the interrupt system. These records are the I/O Control Table and the interrupt packets, both of which are described on the following pages.

2.3.2.1 I/O Control Table. The I/O Control Table (IOCT) consists of two parts: the logical-to-physical device pointers and the Physical Control Blocks (PCBs), as illustrated in figure 1-5. The IOCT is constructed from System Generator macro calls according to user-specified physical device assignment on directive cards input to the SYSGEN program, described in Part 3 of this handbook.

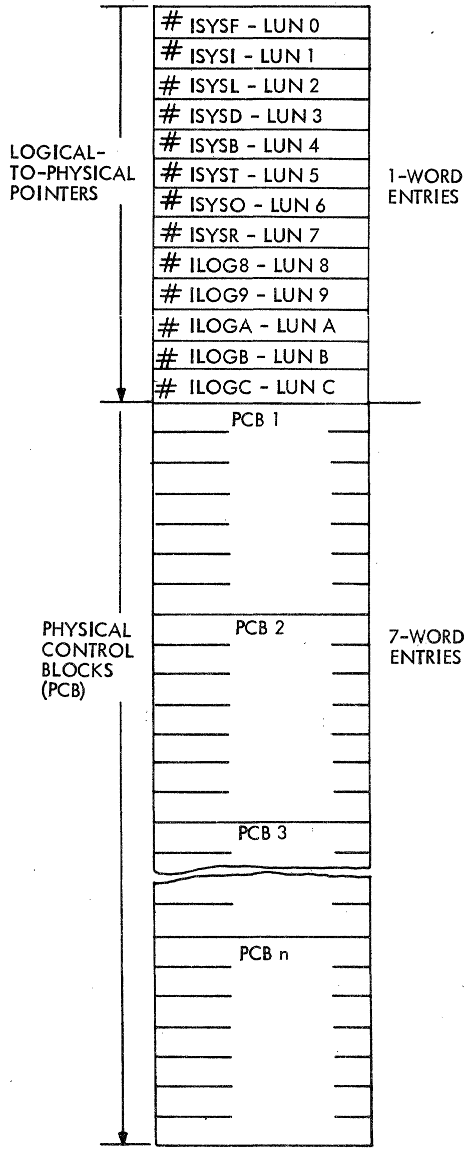


Figure 1-5. Format of the Input/Output Control Table (IOCT)

The logical-to-physical pointers portion of the IOCT contains 13 one-word entries containing the identifier of the physical unit assigned to the logical unit. That is 13 logical units may be assigned to actual physical devices. Eight logical units may be assigned for the use of system programs (e.g., the Assembler, the loaders, Batch Debug, Dump programs, etc.). Five logical units may be assigned for use of applications programs.

The names of the system-reserved logical units begin with the characters #ISYS, followed by an additional character that denotes the use of the unit by system programs. The names of logical units that may be assigned for applications programs begins with the characters #ILOG. The logical-to-physical pointer entries in the IOCT and the assigned usage of the units are shown in table 1-1.

Table 1-1. Logical-to-Physical Pointer Entries in the IOCT

LOGICAL UNIT NAME	PHYSICAL DEVICE (D_n)* LUN ID	USAGE
#ISYSF (System File)	D_1 - LUN 0	Reading and writing systems program files.
#ISYSI (System Input)	D_2 - LUN 1	Reading directive inputs to systems programs.
#ISYSL (System Log)	D_3 - LUN 2	Writing messages from systems programs.
#ISYSD (System Data)	D_4 - LUN 3	Reading data input to systems programs for processing.
#ISYSB (System Binary)	D_5 - LUN 4	Reading relocatable or absolute binary inputs to systems programs.
#ISYST (System List)	D_6 - LUN 5	Writing tabular outputs (i.e., listings) of systems programs.
#ISYSO (System Output)	D_7 - LUN 6	Systems program writing of binary text of absolute or relocatable programs.
#ISYSR (System Scratch)	D_8 - LUN 7	Systems program temporary storage of work files.
#ILOG8 (Logical Unit 8)	D_9 - LUN 8	Performing applications program I/O operations.
#ILOG9 (Logical Unit 9)	D_{10} - LUN 9	Performing applications program I/O operations.
#ILOGA (Logical Unit A)	D_{11} - LUN A	Performing applications program I/O operations.
#ILOGB (Logical Unit B)	D_{12} - LUN B	Performing applications program I/O operations.
#ILOGC (Logical Unit C)	D_{13} - LUN C	Performing applications program I/O operations.

*The D_ns are the physical device identifiers specified on the System Generator program directive that causes the IOCT to be created. Physical devices are assigned to the logical units in the order in which their identifiers appear on the directive. That is, the first device whose identifier, D_1, appears on the directive is assigned to LUN 0, etc.

The PCB portion of the IOCT contains 7-word entries that specify the necessary information to control the physical devices whose identifiers, interrupt levels, and addresses were assigned on the appropriate input directive to SYSGEN. As many as 22 devices may be assigned addresses. For each device, a PCB entry is generated in the IOCT. The format of PCB entries is illustrated in figure 1-6.

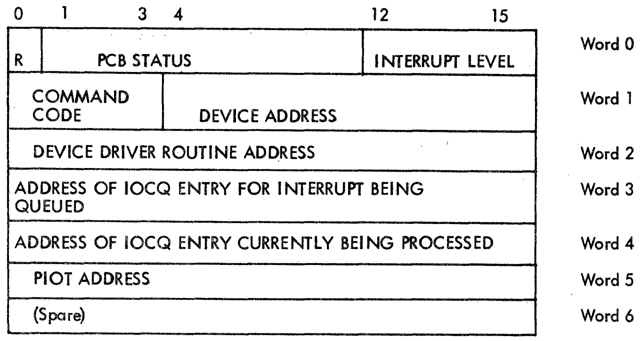

0
1
3
4
12
15
R
PCB STATUS
INTERRUPT LEVEL
Word 0
COMMAND CODE
DEVICE ADDRESS
Word 1
DEVICE DRIVER ROUTINE ADDRESS
Word 2
ADDRESS OF IOCQ ENTRY FOR INTERRUPT BEING QUEUED
Word 3
ADDRESS OF IOCQ ENTRY CURRENTLY BEING PROCESSED
Word 4
PIOT ADDRESS
Word 5
(Spare)
Word 6

Figure 1-6. Format of PCB Entries in the IOCT

2.3.2.2 Interrupt Packets. For each physical device assigned an address at system generation time, the external interrupt level to which the device is to be assigned must be specified. For each interrupt level assigned, a 4-word interrupt packet is created in the IOCS monitor being generated. The interrupt packets, described in detail in Part 2 of this handbook, are used by the Level Service Routines to record the old and new interrupt information when processing control passes from one interrupt level to another.

2.3.3 Special Function Records

The PTS-100 hardware device controllers perform two special functions: the Translate function and the Search function. These functions use byte table lookups and use the current characters passing through the controller to offset the byte table base addresses.

The Translate function enables the I/O hardware device controller to perform code translation on the I/O byte stream as it flows into or out of main memory. The Translate function therefore allows the programer to specify input/output code conversion (i.e., to specify that input/output data characters are to be converted to or from the ASCII code used internally by the PTS-100).

The Search function enables the I/O hardware device controller to test for particular control characters within the I/O byte stream as it flows into or out of main memory, and to set interrupt conditions when the control characters appear. Thus, the Search function allows the programer to specify hardware testing for the occurrence of control characters, and setting of interrupt conditions when the characters appear in the I/O data stream.

To utilize these functions, the programer must have defined and assembled the associated Search and Translate byte tables containing the control and/or conversion codes within the program to be executed. The Search and Translate functions are specified in conjunction with the IOACT service request by entering a code in the MODE field of Word 1 of the FIOB, and specifying the base address(es) of the associated table(s) in the FIOB. Detailed descriptions of the MODE code and the Search and Translate table definitions are presented in Part 2 of this handbook.

2.3.4 Master Service Call Vector Table

The Master Service Call (MSC) Vector table contains the starting addresses (i.e., entry points) of the individual MSC routines that service I/O requests from the executing programs.

2.3.5 Logging Tables

There are three logging tables within IOCS:

- Canned Messages Table (CMT), which is used for logging messages on the System Log device
- Message Locate Table (MLT), which provides IOCS with the starting address of each message in the canned message table
- LUN Conversion Table (LCT), which is used by IOCS to convert the decimal logical unit number into ASCII format.

These tables are incorporated in a given user's IOCS monitor if message logging is selected by the user. If message logging is selected, messages will be output on the System Log device. Monitor messages output to the logging device are enclosed in the special symbols < and > to differentiate between monitor output and any messages or printouts from an executing object program that may also be using the System Log device.

Following are the canned messages from the canned message table:

```
END OF JOB, 00        DATA LOST,nn
DEV NOT OPER,nn       STACK OR HOP,nn
NO LUN,nn             MOTION,nn
LUN OPEN,nn           END OF TAPE,nn
LUN NOT OPEN,nn       WRITE PROTECT,nn
QUE FULL,nn           PARITY,nn
INVALID INTR,nn       DEBUG,nn
READ CHECK,nn
```

Section 3. PTS-100 SOFTWARE SYSTEMS

In addition to the IOCS monitor, the following software systems are provided to users of the PTS-100:

- PTS-100 Assembler program, which translates source programs written in assembly language to object (executable) programs
- Utility programs to load, execute, debug, and maintain user programs.

3.1 Assembler Program

The PTS-100 Assembler program accepts source program coding as input and translates it to executable machine language instructions. The Assembler program must be used to assemble all programs to be executed on the PTS-100. There are three versions of the PTS-100 Assembler:

- PTS-100 Native Assembler
- Raytheon 704 Cross Assembler
- IBM 360/370 Cross Assembler.

Assembler program processing is accomplished in five phases:

Phase 0 determines and sets up for the output options required for the program to be assembled and calls the next processing phase.

Phase 1, the macro processor, is called when macro calls in source programs must be processed, or when an IOCS monitor is to be assembled from the macro calls generated by the System Generator program.

Phase 2 analyzes all source statements and performs the preprocessing for program assembly proper.

Phase 3 optimizes core storage requirements of object (assembled) programs.

Phase 4 completes the construction of executable machine instructions, generates any required listing of the assembled program, and produces the final object program code.

Part 2 of this handbook discusses the assembly language structure and use and Assembler program processing in detail.

3.2 Utility Programs

The PTS-100 utility programs are provided to perform such functions as:

Object program loading

Interactive debugging of object programs

Generation of specially tailored Input/Output Control System (IOCS) monitors

Dumping of the content of main memory storage areas

Dumping binary files to conventional output devices

Program file creation and maintenance

General descriptions of the functions performed by the utility programs are presented on the following pages. Detailed "how to use" descriptions of the programs are presented in Part 3 of this handbook.

3.2.1 PTS-100 Loader Programs

Two loader programs are supplied with the PTS-100: the Piggyback Loader and the Absolute/Relocating Loader. The sole function of the Piggyback Loader is to load the Absolute/Relocating Loader. The Piggyback Loader is bootstrapped into low memory by depressing the IPL button on the user console of the PTS-100. Once loaded, the Piggyback Loader loads the Absolute/Relocating Loader into high memory and starts its execution.

The Absolute/Relocating Loader must be used to load all programs assembled by the PTS-100 Assembler, which develops object coding in the format required by the Absolute/Relocating Loader. The object programs may be absolute or relocatable, and may consist of one or more segments each.

The Absolute/Relocating Loader computes effective addresses of object program instructions, sets up storage areas, loads literal values and address constants, relocates relocatable programs, establishes linkages between multiple program segments, etc. When its loading processing is completed, the Absolute/Relocating Loader terminates itself and activates the loaded program(s) at the execution address defined in the last program loaded, or entered manually into the PTS-100.

3.2.2 Interactive Debug Program

The Interactive Debug Program allows the programer to interface actively with it during object program checkout to effect the following:

Addition or subtraction of hexadecimal constants

Single or successive memory location dumps

Searches of memory locations for specific full word values, or masked searches on values less than 16-bits in length

Alterations of single memory location content to a specific value

Successive memory location loading with specific values

Breakpoint setting and clearing

Transfers of control to specific addresses and resumption of program execution

Transfers of control to specific addresses with the accumulator and/or one or both index registers set to specific values and resumption of program execution

Continuation of previously issued commands to the Interactive Debug program

Input command editing.

Thus the Interactive Debug program provides the programer with hands-on control of the execution of his program. This capability allows selective examination of memory, manipulation of memory words by accessing and altering them, selective execution of any part or all of the program, preparation of active unit tests, minor program patching, etc.

3.2.3 System Generator Program

The System Generator (SYSGEN) program provides for the generation of a specially tailored PTS-100 IOCS monitor to meet unique applications processing requirements. That is, for any given PTS-100 installation, a specialized IOCS monitor can be generated by describing its content to the SYSGEN program. The system descriptions are supplied on key word directive cards, which are input to the SYSGEN program.

SYSGEN analyzes the directives and generates specialized macro calls to the generalized IOCS monitor routines required in the described monitor. The macro calls are written onto an Assembler formated file. The Assembler processes the SYSGEN macro call file against the System Macro Library file (i.e., the generalized IOCS monitor macro routines file) to produce the specially tailored IOCS monitor the user described to SYSGEN.

3.2.4 Memory Dump Program

The Memory Dump program is a small, easily relocatable program capable of dumping the contents of contiguous locations of main memory to any sequential storage device that accepts variable length output records. The length of dumped records depends on the output device being used.

There are two versions of the Dump program:

> Version 1 dumps hard copy hexadecimal or ASCII records onto a character printing device.
>
> Version 2 dumps relcadable binary records to a magnetic tape cassette or paper tape punch device.

Either version of the program may receive dump parameters as input from an ASR keyboard device or as arguments of a subprogram assembled within the main program whose memory locations are to be dumped.

3.2.5 Peripheral Device Dump Program

The sole function of the Peripheral Device Dump (PDD) program is to produce printed listings of binary data files stored in one of the following media:

> Cassette magnetic tape files
> Punched paper tape files
> Punched card files.

The output listings of the PDD program are either in ASCII code or hexadecimal notation, as specified by the programer via a control card input to the program.

Disc files are dumped by a separate program, described in subsection 3.2.7.3.

3.2.6 File Update Program

The File Update program provides a convenient, easily used method of creating, maintaining, and updating files of both object and source programs. That is, the File Update program may be used to create a master file of object and/or source programs and subsequently to maintain and update the master file. The specific update features that can be accomplished using this program are:

> Insertion of one or more programs on the master file
>
> Correction of programs by changing their names and/or deleting, replacing, or inserting data lines
>
> Replacing one or more programs on the master file
>
> Deletion of one or more programs on the master file
>
> Creation of a file directory of the current master file.

3.2.7 Disc Support Programs

Three utility programs are available to support the use of disc files with the PTS-100.

3.2.7.1 Disc Volume Preparation. This program initializes a new disc for use in the PTS-100 system. It can also erase the information on an old disc to prepare it for reuse. A disc must be preprocessed with the Disc Volume Preparation program whether it is to accessed by physical or logical input/output.

3.2.7.2 Disc Allocator. This program must be used before any disc file can be written or read through the logical input/output. If a disc is to be accessed solely by physical input/output (not ulitizing the IOCS monitor), it is not necessary to use the Disc Allocator program.

Prior to running the Disc Allocator, the disc must have been initialized by means of the Disc Volume Preparation program. The Disc Allocator then assigns disc space to files, extends the disc area allocated to files, and deletes files. The program operates from free-form keyword type parameters read from the card reader.

3.2.7.3 Disc Dump. The Disc Dump program produces a printed listing of data on all or a selected portion of the sectors of any disc unit in use with the PTS-100. The output is listed on the serial printer in either hexadecimal or ASCII notation, as specified by the input directives. Dump parameters are input from the display or teletypewriter keyboard in response to program messages.

3.2.8 Cassette Utility Program

The Cassette Utility program provides a method of storing on, deleting, copying, positioning, and printing the contents of cassette magnetic tape files. A display keyboard is used for input directives. The output can be on any of four cassette units, the teletypewriter printer, or serial printer. The program can perform the following functions:

Copy all or parts of one cassette tape to another.

Forward or backspace one tape a specified number of records.

Position a cassette tape to a specific record located by matching a keyword.

Rewind a tape to its beginning.

Print a specified number of records from one tape.

Read cards from the card reader and write the information to a tape.

Print the input directives.

PART 2

PTS-100 ASSEMBLER LANGUAGE PROGRAMMING

PART 2

PTS-100 ASSEMBLER LANGUAGE PROGRAMING

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS (cont)

TABLE OF CONTENTS (cont)

LIST OF ILLUSTRATIONS

LIST OF TABLES

PART 2. PTS-100 ASSEMBLER LANGUAGE PROGRAMING

Section 1. INTRODUCTION TO THE PTS-100 ASSEMBLER LANGUAGE

The programing language of the PTS-100 System is the Assembler language--a symbolic, machine oriented language which is suitable for solving any application processing problem. Applications programs are coded in symbolic, or source, statements which are translated by the Assembler into object programs that can be loaded and executed on the PTS-100 System. Locations within programs can be addressed through symbolic names (i.e., tags or labels). Data constants can be defined in several different ways, either as explicit constants or as literals coded directly in the source statements.

A set of source statements constitutes a source program. The assembled program is called an object program. The object program may be either in absolute or relocatable form for execution on the PTS-100 system. The object program is output on the specific peripheral device used for loading executable programs on the specific machine used for program assembly. That is, there are three versions of the PTS-100 Assembler:

IBM 360/370 Cross Assembler
Raytheon 704 Cross Assembler
PTS-100 Native Assembler

Object programs are output in one of the following forms, depending on the available device:

Punched cards
Punched paper tape
Cassette magnetic tape.

The input/output devices for the three versions of the Assembler are specified in Section 5 of this part of the handbook, which describes the assembly process in detail. Assembler language source statements fall into four functional groupings:

Executable statements, which the Assembler translates into machine instructions (see subsection 1.1) to be executed by the computer.

Nonexecutable statements, which set up data values and storage areas for executable object program use.

Program control statements, which control Assembler output.

Input/output service statements, which effect peripheral device operations via the IOCS monitor of the PTS-100.

The format of Assembler source statements is discussed and illustrated in detail in Section 2 following. Section 3 presents a detailed description of each statement and its use.

The PTS-100 assembly language programer is provided with the capability of defining generalized sets of source statements, called macro routines, which can subsequently be specialized by the Assembler and inserted into any other source program. The definition and use of macro routines is described in detail in Section 4 of this part of the handbook.

1.1 <u>Machine Instructions</u>

As mentioned earlier, certain Assembler source statements are termed "executable" statements. That is, these statements are translated by the Assembler into machine instructions that can be executed by the central processing unit (CPU) of the computer. Machine instructions are formated as 16-bit (one word) instructions or as 32-bit (two word) instructions. One word instructions contain five fields and are said to be in short format. Two word instructions are composed of six fields and are said to be in long format. The assembly language programer may specify the long instruction format (see Section 2), or the instruction length may be left to the discretion of the Assembler, which determines whether the long or short format is required for the instruction. That is, the Assembler will optimize execution speeds and memory storage requirements by using the short format whenever possible, as described in Section 5 of this part of the handbook. The machine instruction formats are presented in figure 2-1 and described in detail in the following paragraphs.

As shown in figure 2-1, both the long and short machine instructions contain OP, R, E, and I fields. The short format contains a D field, and the long format contains a D' field. The significance of these fields is described in detail below:

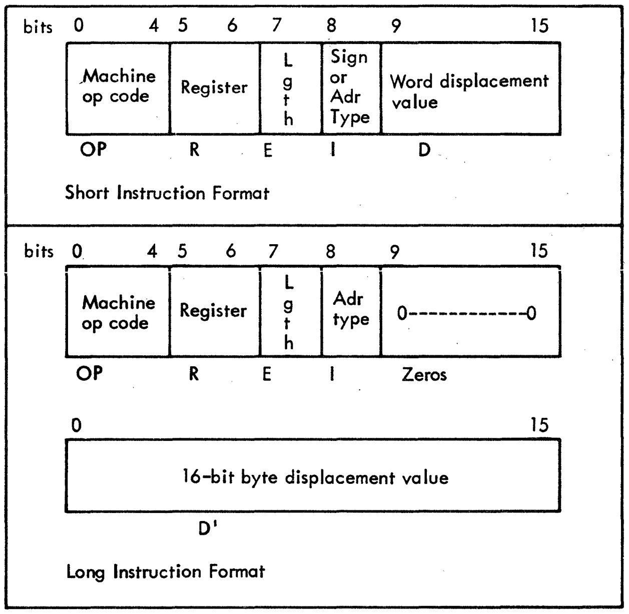


Figure 2-1. Assembler-Generated Machine Instruction Formats

OP (bits 0 - 4):	This 5-bit field contains the operation code, which identifies the specific instruction to the central processor unit as shown in table 2-1.
R (bits 5 - 6):	This 2-bit field specifies one of four address components to be used in computing the effective address, where: 0 = zero 1 = contents of the program counter (PC) 2 = contents of index register 1 (X1) 3 = contents of index register 2 (X2)
E (bit 7):	This 1-bit field specifies the instruction length, where: 0 = 16-bit (short format) 1 = 32-bit (long format)
I (bit 8):	This 1-bit field specifies direct or indirect addressing, where:

0 = direct addressing
1 = indirect addressing

except in the following cases:

When the R field = 1 (PC relative addressing) and the E field = 0 (short format), the I field specifies the sign of the 7-bit word displacement value, where:

0 = positive sign
1 = negative sign

NOTE

Indirect addressing is not available when R = 1 and E = 0.

In the Add Immediate and Load Immediate machine instructions the I field is not used. That is, the short format machine instructions contain four fields: OP (bits 0-4) R (bits 5-6), where:

0 = accumulator
1 = program counter
2 = index register 1
3 = index register 2

E (bit 7), and the OPERAND field (bits 8-15), which contains the immediate byte value to be loaded or added. The long instructions contain five fields, with the first three the same as in the short format, the fourth field (bits 8-15) containing zeros, and the fifth field (i.e., the second 16-bit word) containing the immediate word operand to be loaded or added.

D (bits 9 - 15): The D field of a short machine instruction (E = 0) contains a 7-bit positive word displacement value to be used in forming the effective address. For short machine instructions the effective address is computed as follows:

R field	I = 0	I = 1
0	2D	(2D)
1	(PC) + 2D	(PC) - 2D
2	(X1) + 2D	((X1) + 2D)
3	(X2) + 2D	((X2) + 2D)

To explain, the displacement value in the D field is multiplied by 2, and the product is added to the value specified by the R field except in the case where R = 1 and I = 1, in which case the product is subtracted from the current location of the program counter. The current location of the program counter is the next instruction.

The D field content of two machine instructions are exceptions to the above discussion. These instructions are the Add Immediate and Load Immediate instructions. In the short format of these instructions, the R field indicates the register and the D field contains the immediate value specified in the operand field of the source statement.

D' (bits 16 - 31, i. e., the second word of the two-word instruction):

The D' field of a long instruction (E = 1) contains a 16-bit byte displacement value to be used in forming the effective address. Negative displacement values are represented in two's complement form. For long machine instructions, the effective address is computed as follows:

R field	I = 0	I = 1
0	D'	(D')
1	(PC) + D'	((PC) + D')
2	(X1) + D'	((X1) + D')
3	(X2) + D'	((X2) + D')

Notice that in all cases the D' field value is added to the value specified by the R field. In the case of a long instruction, the current location of the program counter is the instruction following the second word (D' field) of the long instruction.

There are two exceptions to the above discussion: the Add Immediate and Load Immediate statements. In the long format of these machine instructions the R field indicates the register and the D' field contains the immediate value specified in the operand field of the source statement.

1.2 Machine Instruction Execution Timing

Machine instruction execution timing depends upon the length of the instruction, the number of processor cycles required to execute the instruction, and the time required for an instruction fetch. Each processor cycle requires 0.160 microsecond. All machine instructions require 0.960 microsecond for an instruction fetch. A long machine instruction (E = 1) requires 0.960 microsecond additional for execution. When an instruction specifies indirect addressing, another 0.960 microsecond is required for its execution. Hence, a long machine instruction in which indirect addressing is specified requires an additional 1.920 microsecond for execution.

Table 2-1 presents the total execution times for all short format (E = 0) machine instructions, including 0.960 microsecond for the instruction fetch.

1.3 Word and Data Formats

Internally in the PTS-100, instructions and data are stored in 16-bit word units. The words are composed of two bytes (i.e., 8-bit units). Internally, data is stored in standard ASCII code. Provision has been made, however, to accept any code up to 8 bits per character. That is, input/output controllers perform a special code Translate function to convert input/output data to or from the 7-bit ASCII code used by the PTS-100. To utilize the Translate function for data conversion, the programer must define Translate tables, as described later in Section 3.

Another special function, the Search function, is performed by the PTS-100 input/output controllers. This function allows programers to test for the occurrence of particular I/O control characters and specify interrupt conditions when the control characters appear in the data stream. See Section 3 for a description of this special function.

Table 2-1. Machine Instruction Execution Times

MACHINE OP CODE*	ASSEMBLY MNEMONIC OP CODE	INSTRUCTION	EXECUTION TIME** (in micro-seconds)
00	JMP	Jump (unconditional branch)	1.60

01	ENB
	DIN
	INR
	MSC
		R = 00 Enable Interrupts
		R = 01 Disable Interrupts
		R = 10 Interrupt Return
		R = 11 Monitor Service Call
			1.60
			1.60
			4.00
			1.60

02	BCB	Branch if Condition Bit Set	1.60
03	BRM	Branch if Accumulator Minus	1.60
04	LDI	Load Immediate	1.60
05	ADI	Add Immediate	1.60
06	SRO	Shift Right One, Arithmetic	1.60
07	DIO	Do Input/Output	2.08
08	LAX2	Load Address in Index Register 2	2.08
10	ADD	Add Memory Word to Accumulator	2.08
11	XOR	Exclusive OR	2.08
12	AND	Logical AND	2.08
14	SUB	Subtract	2.08
16	CNE	Compare for Not Equal	2.08
17	CAL	Compare Accumulator for Less than Memory Word	2.08
18	LDW	Load Word in Accumulator	2.08
19	LDB	Load Byte	2.08
20	LX1	Load Index Register 1	2.08
21	LX2	Load Index Register 2	2.08
24	STW	Store Word	2.40
26	SX1	Store Index Register 1	2.40
27	SX2	Store Index Register 2	2.40
28	STB	Store Byte	2.40
29	RIO	Read Input/Output Device Status	3.04
30	ACM	Add Accumulator to Memory Word	3.20
31	AOM	Add One to Memory Word	3.20

*The machine op codes here and throughout this handbook are expressed in decimal notation.

**These times are for short format instructions using direct addressing. They include one instruction fetch (0.960 microsecond).

Section 2. ASSEMBLER STATEMENT FORMATS

2.1 Source Statement Coding Form

All assembler source statements are written as 80-column records on the coding form shown in figure 2-2. A source statement may comprise one to five fields in the following order:

1. A label field, which is optional except for the EQUate statement.
2. An operation code field, which is required for all source statements.
3. An operand field, required as described for the individual statements in Section 3.
4. An optional comments field, which may follow the operand field and continue through column 72 to document the source statement, or which may begin with an asterisk (*) in column 1 of the coding form and continue across columns 2 through 72.
5. An optional sequence number field, which begins in column 73 and terminates in column 80.

If all fields are present in a source statement, they must appear in the sequential order shown in figure 2-2. Except for the label and sequence number fields, whose lengths are restricted as shown in figure 2-2, there is no restriction on statement field lengths. The label, operation code, and operand fields are terminated by a blank (Δ) character. The content and use of individual statement fields are described below.

2.2 Label Field

The label field is optional for all source statements except the EQUate statement described in Section 3. If a label is used, its first character must be alphabetic and must appear in column 1 of the source record. Up to five additional alphabetic, numeric, or alphanumeric characters may appear in the label field (i.e., the label must not continue beyond column 6 of the coding form). The label field is terminated by a blank character. Labels may be used as symbolic tags in the operand field to identify data locations and values on which operations are to be performed.

2.3 Operation Code Field

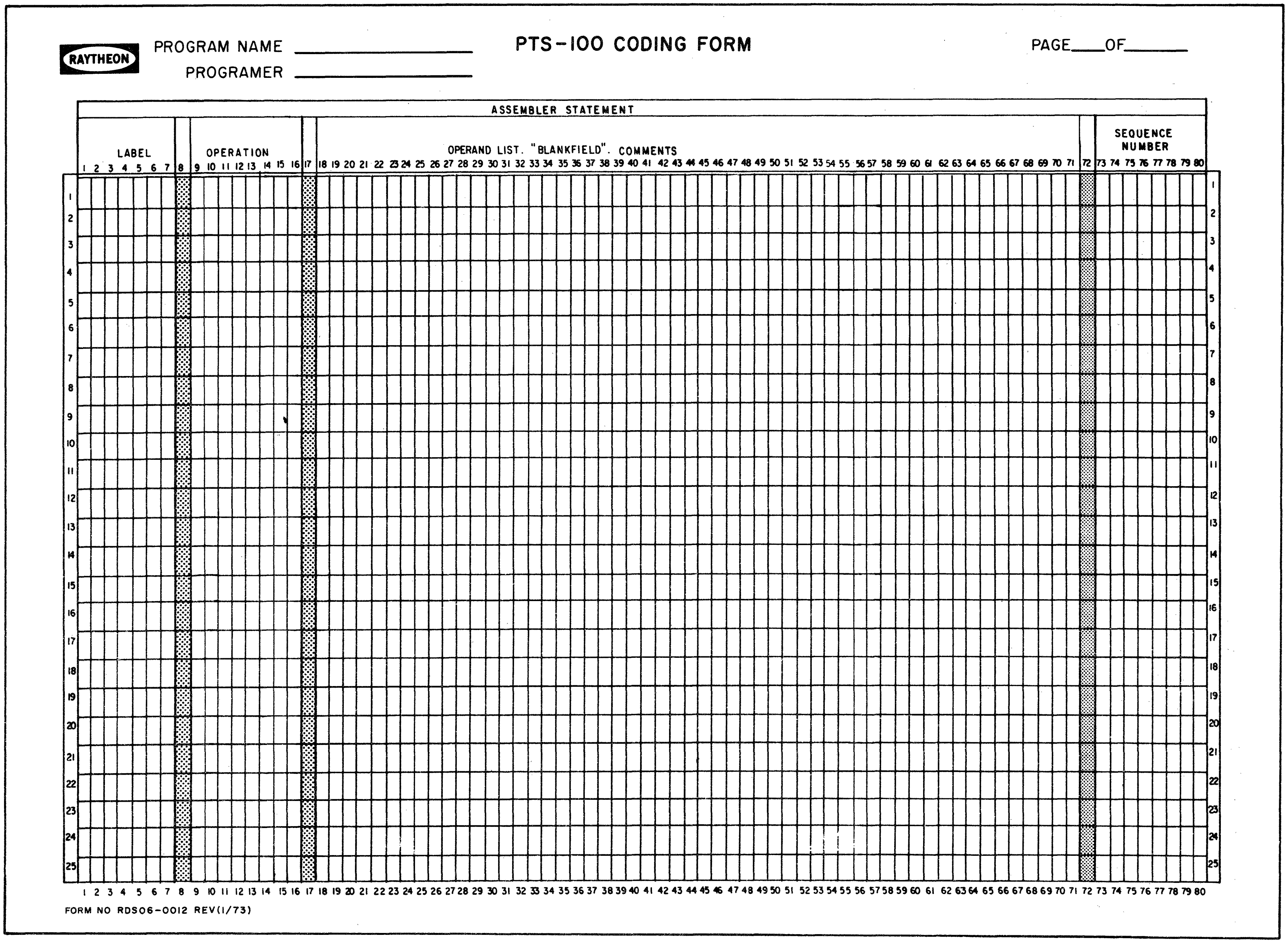
This field specifies the mnemonic operation code (op code), which identifies a unique statement specifying action to be taken by the program, the processor, or the Assembler. The op codes are of variable length.

In executable statements, op codes may be followed optionally by from one to two flags, in any order, specifying either indirect, indexed, or a combination of indirect and indexed addressing. The flags and their significance are:

N	specifies indirect addressing
X1	specifies indexed addressing using index register 1
X2	specifies indexed addressing using index register 2.

To specify both indirect and indexed addressing, the following combinations of flags are valid:

N,X1 or N,X2
X1,N or X2,N



RAYTHEON

PROGRAM NAME ____________

PROGRAMER ____________

PTS-100 CODING FORM

PAGE____OF______

ASSEMBLER STATEMENT

LABEL 1 2 3 4 5 6 7 | 8 | OPERATION 9 10 11 12 13 14 15 16 | 17 | OPERAND LIST. "BLANKFIELD". COMMENTS 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 | 72 | SEQUENCE NUMBER 73 74 75 76 77 78 79 80

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

FORM NO RDS06-0012 REV(1/73)

Figure 2-2. Sample PTS-100 Coding Form

If flags are specified they are separated from the op code and each other by commas. If a label precedes the operation code field, the op code field begins with the first non-blank character following the blank character that terminates the label field. If no label is specified, column 1 of the coding form must be blank. The op code field may begin in column 2 or any column after column 1. The operation code field is terminated by a blank character.

Each source statement is assigned a mnemonic op code that uniquely identifies it and the operation it specifies. For purposes of discussion in this manual, the Assembler source statements may be classified as follows:

- Executable statements, which result in Assembler-generated machine instructions to be executed by the CPU. Executable statements, summarized in table 2-2, specify the following:

  Arithmetic operations

  Branches in program execution

  Loading of CPU registers with data values in memory storage locations

  Storing contents of CPU registers in memory locations

  Comparative tests of data values

  Logical (true/false) testing of data values.

- Nonexecutable statements, summarized in table 2-3, which define constant data values and storage areas for executable program use.

- Program control statements, summarized in table 2-4, which direct the Assembler to perform actions regarding the end of the program and the object program listing.

- Input/output service statements, which are sets of statements defining tables and parameters for use by the IOCS monitor in servicing input/output requests, as summarized in table 2-5.

Table 2-2. Summary of Executable Assembler Statements

STATEMENT	OPERATION CODE		SPECIFIED OPERATION
	Mnemonic	Machine*	
Arithmetic Statements			
Add AC to Memory	ACM	30	Add contents of accumulator to memory word specified by operand; store results in memory word; set CB if no carry generated.
Add	ADD	10	Add contents of memory location specified by operand to accumulator value; store result in accumulator; set CB if addition overflow.
Add Immediate	ADI	5	Add immediate operand algebraically with contents of specified register and store in register. Set CB if no carry generated.
Add One to Memory	AOM	31	Increment memory word specified as operand by one; set CB if no carry.
Shift Right One	SRO	6	Shift the value in the accumulator right one bit position and retain sign bit; right-most bit is lost.

*The machine op codes are given in decimal notation.

Table 2-2. Summary of Executable Assembler Statements (cont)

STATEMENT	OPERATION CODE		SPECIFIED OPERATION
	Mnemonic	Machine*	
Subtract	SUB	14	Subtract the value in the memory location specified by the operand from the contents of the accumulator; store results in accumulator; set CB if arithmetic overflow.
Branch Statements			
Branch if AC Minus	BRM	3	Branch if value in accumulator is negative number (MSB = 1).
Branch if CB Set	BCB	2	Branch if CB set; otherwise execute next sequential instruction.
Jump (unconditional branch)	JMP	0	Jump (unconditionally branch) to execution point specified by operand.
Compare Statements			
Compare AC Less than Memory Word	CAL	17	Compare accumulator value with value of memory word specified as operand; if AC value less than operand value set CB.
Compare for Not Equal	CNE	16	Compare accumulator value with value of memory word specified as operand; set CB if values not equal.
Load Statements			
Load Address in Index Register 2	LAX2	8	Load address of the memory location specified by operand into index register 2.
Load Byte	LDB	19	Load byte from memory location specified by operand into right-hand side of accumulator and clear left-hand side.
Load Immediate	LDI	4	Load immediate operand into specified register.
Load Index Register 1	LX1	20	Load memory word specified by operand into index register 1.
Load Index Register 2	LX2	21	Load memory word specified by operand into index register 2.
Load Word	LDW	18	Load memory word specified by operand into the accumulator.
Logical Statements			
And	AND	12	And the value in the accumulator with the memory word specified by the operand and place the result in the accumulator. Set CB if result not zero.
XOR	XOR	11	Exclusive OR the value in the accumulator with the memory word specified by the operand and place the result in the accumulator.

*The machine op codes are given in decimal notation.

Table 2-2. Summary of Executable Assembler Statements (cont)

STATEMENT	OPERATION CODE		SPECIFIED OPERATION
	Mnemonic	Machine*	
Store Statements			
Store Byte	STB	28	Store the right-hand byte value in the accumulator in the memory word specified as the operand, either as left-hand or right-hand portion of word.
Store Index Register 1	SX1	26	Store content of index register 1 in memory word specified by operand.
Store Index Register 2	SX2	27	Store content of index register 2 in memory word specified by operand.
Store Word	STW	24	Store content of accumulator in the memory word specified by operand.

*The machine op codes are given in decimal notation.

Table 2-3. Summary of Constant, Address, and Storage Assignment Assembler Statements

STATEMENT	MNEMONIC OP CODE	SPECIFIED OPERATION
Address Constant Definition	ADC	Establish an address constant as specified by the expression operand.
Concatenated Integer Constant Definition	CAT	Establish a concatenated integer constant as specified by the values used as the operand. The CAT statement is not implemented in the native version of the PTS-100 Assembler.
Decimal Integer Constant Definition	DEC	Convert the decimal expression operand to a binary constant.
Hexadecimal Constant Definition	HEX	Establish a hexadecimal constant one word long as specified by the expression operand.
Octal Constant Definition	OCT	Establish an octal constant one word long as specified by the expression operand.
Text (alphanumeric constant definition)	TEXT	Establish a variable-length alphanumeric constant as specified by the operand in 8-bit code.
Text (7-bit alphanumeric constant definition)	TEX7	Establish a variable-length alphanumeric constant as specified by the operand in 7-bit code.
Equate Symbol	EQU	Assign the symbol in the label field to the value specified by the operand.
External Definition	EXDEF	Create a symbol table entry for the symbol operand and its address value to enable another program to reference the current program in which the symbol is defined.
External Reference Definition	EXREF	Create a symbol table entry for the symbol operand and its address value to enable the current program to be linked to the program in which the symbol is defined.
Literal Origin Storage Specification	LTORG	Reserve a block of sequential storage locations for literal data values.
MOD Storage Specification	MOD	Allocate the next instruction to the next location that is a multiple of _n_, a power of two value specified as the operand.

Table 2-3. Summary of Constant, Address, and Storage Assignment Assembler Statements (cont)

STATEMENT	MNEMONIC OP CODE	SPECIFIED OPERATION
Origin	ORG	Establish the origin of the object program at the absolute address specified by the decimal, hexadecimal, or octal operand (i.e., store the first statement of the object program at the location specified by the operand).
Page 0	PG0	Use the absolute address of the symbolic tag specified as operand each time it is referenced (i.e., the symbol is to be assigned an address relative to page 0).
Reserve	RESV	Reserve an area of memory for buffers, data areas, etc.
	RESV, xx	Reserve an area of memory for buffers, data areas, etc., and set each byte location to value xx, if specified.

Table 2-4. Summary of Program Control Statements

STATEMENT	MNEMONIC OP CODE	SPECIFIED ACTION
End	END	Terminate source program assembly and establish the starting address at the first statement to be executed for object program execution if an address is specified as an operand
List	LIST	Resume printing the object program listing that was suspended by the UNLIST statement
Unlist	UNLIST	Suspend the object listing until a LIST statement is processed or until the end of the program
Skip	SKIP	Skip, or space, the object listing the number of spaces specified by the operand.

Table 2-5. Summary of I/O Service Statements

I/O OPERATION	STATEMENT FORMS			EFFECT
	Label	Mnemonic Op Code	Operand	
INITialization Service Request				Initializes all I/O devices (i.e., stops any I/O operations in process and resets logical status bits to the load condition).
Issue Monitor Service Call		MSC		Transfers control to the IOCS monitor and informs it that an I/O service is required.
Identify I/O Command Code		DEC	2	Informs the monitor that the initialization request is to be serviced for all devices.
Specify Return Address		ADC	Address	Specifies the location in the object program to which the monitor is to return control.
OPEN and CLOSE Device Service Request				The OPEN device service sets up IOCQ entries and linkages, and checks the operational status of the specified device. The CLOSE device service issues a STOP I/O, which immediatelv causes a physical and logical shutdown of the specified device.
Issue Monitor Service Call		MSC		Transfers control to the IOCS monitor and informs it that an I/O service is required.
Identify I/O Command Code		DEC	6	Informs the monitor that an OPEN device request is to be serviced.
		DEC	1	Informs the monitor that a CLOSE device function is to be performed.
Specify Return Address		ADC	address	Specifies the object program location to which the monitor is to return control.
Specify the parameter address		ADC	symbolic tag	Directs the monitor to the object program location in which the logical unit number identification (LUN ID) is stored.
Establish the LUN ID	symbolic tag	HEX	LUN ID	Assigns the desired device's LUN ID.
For OPEN, specify IOCQ table starting address		ADC	IOCQ addr	Establishes the beginning location (address of first word) of IOCQ for an OPEN device request (i.e., when Command Code =6). Note that this statement is not used when a CLOSE service is specified.
Reserve an error field		RESV,ØØ	2	Establishes a storage area 2 bytes in length in which the monitor is to store a word indicating an error occurrence that prevented the successful completion of the OPEN or CLOSE service.

Table 2-5. Summary of I/O Service Statements (cont)

I/O OPERATION	STATEMENT FORMS			EFFECT
	Label	Mnemonic Op Code	Operand	
Input/Output Action (IOACT) Service Request				Transfers input and output data between memory and the specified I/O device.
Issue Monitor Service Call		MSC		Transfers control to the IOCS monitor and informs it that an I/O request is to be serviced.
Identify I/O Command Code		DEC	7	Informs the monitor that a data transfer to or from the specified device is to be serviced.
Specify Return Address		ADC	address	Specifies the location in the object program to which the monitor is to return control.
Establish the FIOB address		ADC	address	Assigns the beginning location (address of the first word) of the FIOB (File IO Block) that contains the programer-defined parameters specifying the precise I/O action requested.
EXIT Service Request				Provides a common system exit of the program when execution is completed.
Issue Monitor Service Call		MSC		Transfers control to the IOCS monitor and informs it that an I/O service is required.
Identify I/O Command Code		DEC	0	Informs the monitor that an EXIT request is to be serviced.

At assembly time, the Assembler interprets the mnemonic op code of each source statement to determine the type of operation requested, and translates the source language statement to object code format by translating executable statements to machine instruction format, resolving address computations, reserving storage locations, etc., as described in Section 5.

2.4 Operand Field

The operand field of a source statement specifies the element or elements to be used in performing the operation specified in the op code field of the statement. The specified operands that may be used in a given source statement are described in the individual statement discussions in Section 3. In general, an operand may be any of the following:

Symbolic tag (i. e., label)

Literal data value

Absolute address

Self-referencing indicator, *

Expression formed by combining two or more of the single-element operands above with plus (+) and minus (-) signs.

The operand of an executable instruction may be followed by the characters

, L

to inform the Assembler that a long machine instruction is to be generated. In all cases the operand field is terminated by a blank character.

The operand fields of the Add Immediate and Load Immediate statements are special cases, as described in Section 3.

Detailed discussions of the construction and use of each type of operand are presented in the following paragraphs.

2.4.1 Symbolic Tag Operands

A symbolic tag used as an operand may be composed of from one to six characters, the first of which must be alphabetic. When a symbolic tag is used as an operand, it must reference memory locations or data values defined elsewhere in the current program or in a program referenced by the current program. That is, a symbolic tag operand must have appeared elsewhere in the current program as one or a combination of the following:

Label of a statement in the current program.

Label (i.e., symbolic tag) of an EQUate statement, the operand of which specifies the actual value of the symbolic tag.

Operand in an EXternal REFerence statement (mnemonic op code EXREF) that specifies that the symbolic tag is defined in another program referenced by the current program.

NOTE

When a symbolic tag appears in the operand field of the EXREF statement, it may also be used as the operand in any statement in the current program where symbolic tags are permitted. The EXREF definition must precede any such use, however.

If more than one symbolic tag appears in an expression in the operand field, all but one of the tags must have been assigned absolute addresses in EQUate statements.

The use of symbolic tags as operands is illustrated in the following examples.

Example 1:

```
OP CODE        OPERAND
   .              .
   .              .
  JMP          ENDJOB
```

This statement specifies that program execution is to jump (unconditionally branch) to the current location of ENDJOB.

Example 2:

```
OP CODE        OPERAND
   .              .
   .              .
  ADD           TOTAL
```

This statement specifies that the value currently stored in the memory location associated with the symbolic tag operand TOTAL is to be added to the value in the accumulator, and the result stored in the accumulator.

2.4.2 Literal Operands

Literal operands are defined within the operand field in which they appear. A literal definition is written in one of the following formats:

=X'constant value'	which defines a hexadecimal constant value
=O'constant value'	which defines an octal constant value
=D'constant value'	which defines a decimal constant value
=constant value	which defines a decimal constant value by default (i.e., if neither of the letters X, O, or D follows the equal sign and the constant value is not enclosed in quotation marks, the constant value is assumed to be a decimal value).

The constant value must not be greater than 16 bits (a full word) in length. Leading zeros in literal constant values less than 16 bits in length are not required in the Assembler source language. That is, the Assembler stores literal constant values in 16-bit words, right-justified. Following are examples of acceptable literal operand definitions.

Example 1:

```
OP CODE          OPERAND
   .                .
   .                .
  ADD             =X'FF'
```

This statement specifies that the value currently in the accumulator is to be added to the hexadecimal constant whose value is FF, and the result stored in the accumulator.

Example 2:

```
OP CODE          OPERAND
   .                .
   .                .
  ADD             =O'377'
```

This statement specifies that the octal value of 377 is to be added to the current value in the accumulator.

Example 3:

```
OP CODE          OPERAND
  ADD             =D'10'
```

This statement specifies that the decimal value of 10 is to be added to the current value in the accumulator.

Example 4:

```
OP CODE          OPERAND
   .                .
   .                .
  SUB              =10
```

This statement specifies that the decimal (default) value of 10 is to be subtracted from the value in the accumulator.

2.4.3 Absolute Address Operands

Absolute address operands may be defined in two ways:

By identifying a symbolic tag as a reference to page 0 (see the Page 0 statement description in Section 3), and subsequently using the tag as an operand.

By specifying a decimal, hexadecimal, or octal memory address as the operand.

When absolute addresses are used, either symbolically with a PG0 statement or directly, each absolute address is assembled into the operand field of the machine instruction and the R bits of the instruction are 00. Following are examples of the use of absolute address operands.

Example 1:

```
LABEL     OP CODE     OPERAND
            PG0          A
             .
             .
             .
ONE         LDW          A
            PG0          B
             .
TWO         LDW          B
            PG0          C
             .
THREE       LDW          C
```

The Page 0 statements specify that data values associated with symbolic tags A, B, and C are to be assigned absolute addresses relative to Page 0. Statement ONE specifies that the value in the absolute address assigned to A is to be loaded into the accumulator. Statements TWO and THREE specify the same thing for the values of symbolic tags B and C. In these statements, direct addressing is specified and the absolute addresses of the respective data values will appear in the operand fields of the machine instructions generated by the Assembler.

Example 2:

```
LABEL     OP CODE     OPERAND
  .          .
             .
             .
BR1         JMP        X'F0'
             .
             .
             .
BR2         JMP        O'377'
             .
             .
             .
BR3         JMP        D'100'
             .
             .
             .
BR4         JMP         26
```

Statement BR1 specifies an unconditional jump (branch) of program control to hexadecimal F0. Statement BR2 specifies a jump to octal location 377. Statement BR3 specifies an unconditional branch to decimal location 100. BR4 specifies a branch to decimal (by default) location 26.

2.4.4 Self-Reference Operand

The self-referencing indicator (*) may be used as an operand, as illustrated below.

```
DTAG7       ADC          *
             :
DKEY        EQU          *
             :
            JMP          *
```

2.4.5 Expression Operands

Expression operands are formed by combining any of the previously described single-element operands with plus (+) and minus (-) signs. Recall, however, that when two or more symbolic tags are used as expression elements, all but one of the tags must have been assigned absolute addresses in EQUATE statements. Examples of expression operands are shown below:

Example 1:

```
OP CODE          OPERAND
  JMP              *+16
```

The JMP statement specifies a transfer of program control to a point 8 decimal locations (i.e., 16 bytes) beyond the current location of the self-referencing indicator.

Example 2:

```
OP CODE          OPERAND
  LDW             TABLE +6
```

The LDW statement specifies that the accumulator is to be loaded with the contents of the fourth word of TABLE. That is, the third word after the first (beginning) word of Table contains the value to be loaded.

Example 3:

```
OP CODE          OPERAND
  STW             X'1F0'-2
```

The STW statement specifies that the value in the accumulator is to be stored in the memory location just preceding the hexadecimal location 1F0.

2.5 Comments Field

The programer may thoroughly document his program by writing descriptive comments following the blank character that terminates the operand field and continuing through column 72. In addition, the programer may specify that an entire source record (coding line) is to be treated as a comment by writing an asterisk (*) in column 1 of the source record, and then writing the comment text in any columns from 2 through 72 of the coding form. Embedded blanks are accepted in the comments field. Comments are not processed by the Assembler, but are carried on the object listing exactly as they were specified in the source statements.

2.6 Sequence Number Field

The programer may assign a sequence number to each line of source coding of his program. If sequence numbers are specified, they must appear in columns 73 through 80. The sequence number field may contain any combination of alphanumeric characters from the PTS-100 character set. At assembly time, the programer may specify the sequence checking of his program by the Assembler, as described in Section 5. If a sequence number field is blank in a source statement, sequence checking for the associated record is bypassed by the Assembler.

Section 3. DETAILED DESCRIPTIONS OF SOURCE LANGUAGE STATEMENTS

This section presents detailed descriptions of all Assembler source statements that may be used in coding applications programs for assembly and subsequent execution on the PTS-100. For purposes of discussion, the source statements are described in the following four functional groups:

Executable statements
Non executable statements
Program control statements
Input/output service statements.

For each source statement, a format diagram is presented to graphically illustrate the statement fields that may be used and the permissible content of each field. In all cases, the mnemonic op code must be specified, as shown in upper case letters in the format diagrams. Optional fields are indicated by enclosing them in parentheses. When a required statement field permits a choice in the form of its contents, the permissible choices are shown enclosed in brackets in the format diagram. When no choice of field content is allowed, the form is shown unenclosed. The label, op code, and operand fields must be terminated by at least one blank character, indicated by the character Δ in the format diagrams.

Logical input/output for disc files is covered separately at the end of the section (subsection 3.5).

In addition to the statements presented in this section, the programer may write macro routines to be subsequently specialized and incorporated within his programs as described in Section 4.

Special statements and considerations for writing systems programs are presented in Section 7 of this part of the manual.

3.1 Executable Statements

Executable statements are those source statements the Assembler translates to machine instruction format for execution by the CPU. As described in Section 1, assembled machine instructions are either short (i.e., one word or 16 bits in length) or long (i.e., two words or 32 bits in length).

Short machine instructions contain 7-bit operands which specify either word displacement values to be used in computing the effective addresses of object code or data values, or the assigned absolute addresses of object code or data values. That is, short instructions provide addressing capability for ±128 words relative to the current program counter value, or addressing of +128 words relative to zero or the value contained in one of the index registers.

Long machine instructions contain 16-bit operands which specify byte displacement values to be used in computing the effective addresses of relocatable coding or data values, or the actual memory location of object coding or data values that have been assigned absolute addresses.

Word boundaries in memory are fixed. When a memory word is referenced in one of the machine instructions generated for a branch statement, the least significant bit (LSB) of the effective address must be zero.

In all other machine instructions that reference memory words the least significant bit (LSB) of the effective address is ignored.

When a byte address is referenced in a machine instruction, the LSB is used to select either the left-hand byte (LSB=0) or the right-hand byte (LSB=1).

The CPU provides a hardware condition bit (CB) to record status as the result of arithmetic computations, comparative testing, and the logical AND operation. A condition bit setting of one indicates the following conditions:

Arithmetic overflow
No carry
The logical AND operation result is zero
True results of comparison tests

The condition bit testing is specified in the conditional branch statement (BRANCH IF CB SET). The hardware maintains the current status of the condition bit until another instruction is executed that alters, or resets, it.

The executable statements provided in the Assembler language are discussed in functional groupings in the following paragraphs.

3.1.1 Arithmetic Statements

The source language for the PTS-100 Assembler provides six statements that perform arithmetic computations:

Add (ADD)

Add Accumulator to Memory (ACM)

Add Immediate (ADI)

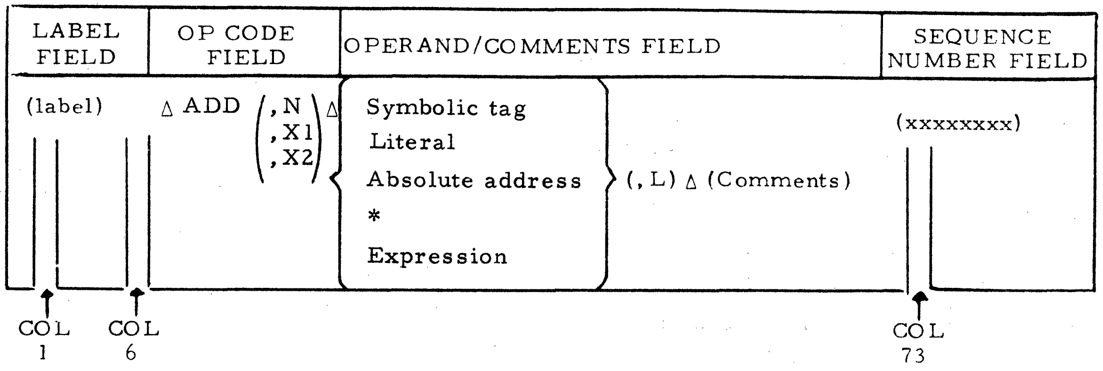
Add One to Memory (AOM)

Shift Right One, Arithmetic (SRO)

Subtract (SUB)

These statements, their permissible formats, and the effects of their use are described individually below.

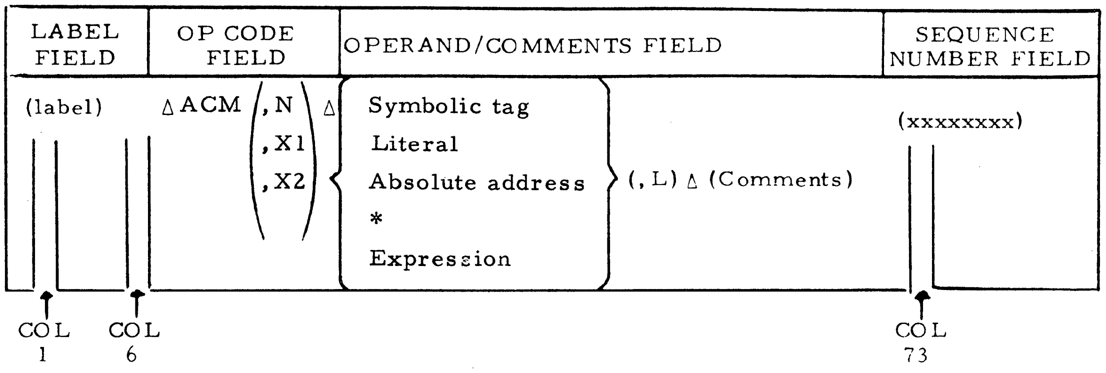
3.1.1.1 Add Statement (ADD). The ADD statement specifies that the memory word specified in the operand field is to be added to the value currently in the accumulator, and the result of the addition is to be stored in the accumulator. The acceptable formats of the ADD source statement are presented in the following diagram.



At assembly time, the Assembler generates either a short or long machine instruction with a machine command code of 10 in the op code field, and the displacement of the memory word in the operand field.

At program run time, execution of the ADD machine instruction causes the specified memory word to be accessed, its value to be added to the value in the accumulator, and the resultant value to be stored in the accumulator. If the addition operation causes an arithmetic overflow, the hardware condition bit is set to one; otherwise, the condition bit is reset to zero.

3.1.1.2 Add Accumulator to Memory Statement (ACM). This statement specifies that the current value in the accumulator is to be added to the contents of the memory word specified as the operand, and the result is to be stored in memory word. The ACM source statement format is presented in the following diagram.



At assembly time, the Assembler generates either a short or long machine instruction with the machine command code 30 in the op code field and the displacement of the memory word in the operand field.

At program run time, execution of the ACM machine instruction causes the specified memory word to be accessed, the current value of the accumulator to be added to it, and the result of the addition to be stored in the memory word. The current value of the accumulator is not modified. If no carry is generated by the addition operation, the hardware condition bit is set; otherwise, the condition bit is reset to zero.

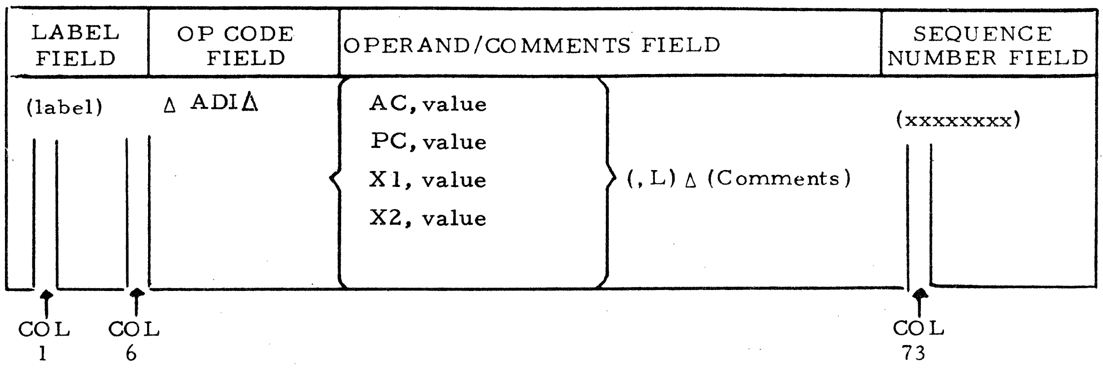
3.1.1.3 Add Immediate Statement (ADI). This statement specifies that a value is to be combined algebraically with the value in a specific register, and the resultant value is to be stored in the register. That is, the Add Immediate source statement requires a specially formated two-element operand, as follows:

- Element 1 specifies the register to be used in in the Add Immediate operation, where:

AC	= accumulator
PC	= program counter
X1	= index register 1
X2	= index register 2

- Element 2 specifies the value to be algebraically added to the specified register. The immediate value may be absolute (coded in hexadecimal, decimal, or octal notation) or may be a symbolic tag whose address becomes the immediate value.

The operand field may optionally contain the characters , L following the immediate value to specify that the long machine instruction format is to be used by the Assembler. The elements in the operand field of the source statement are separated by commas, as illustrated in the following diagram.



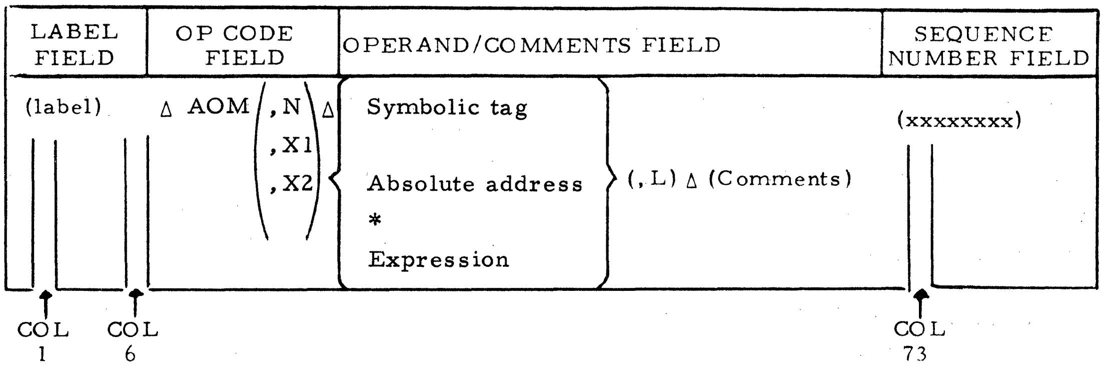
At assembly time, the Assembler generates either a long or short machine instruction with the command code 5 in the op code field and the immediate operand in the operand field.

At program run time, execution of the short form of the ADI machine instruction causes the byte operand to be treated as a sign plus 7-bit magnitude. The 7-bit field is added to the value in the register if the sign is positive (i.e., 0) or subtracted from the register if the sign is negative (i.e., 1). Note that the 7 bit field is unshifted when combined with the register data.

Execution of the long format of the ADI machine instruction causes the 16-bit operand (i.e., the second word of the machine instruction) to be added to the value in the register. Negative operands are represented in two's complement form in the long instruction.

If the first element of the ADI source operand field specified the accumulator (AC), the hardware condition bit is set when an arithmetic overflow condition occurs. If another register is specified (i.e., PC, X1, X2) as the first element of the operand field, the condition bit is set when no carry is generated as a result of the ADI instruction execution. Otherwise the condition bit is reset to zero.

3.1.1.4 Add One to Memory Statement (AOM). This statement specifies that the contents of the memory word specified as the operand is to be incremented by one (in the Least Significant Bit). The AOM source statement format is presented in the following diagram.

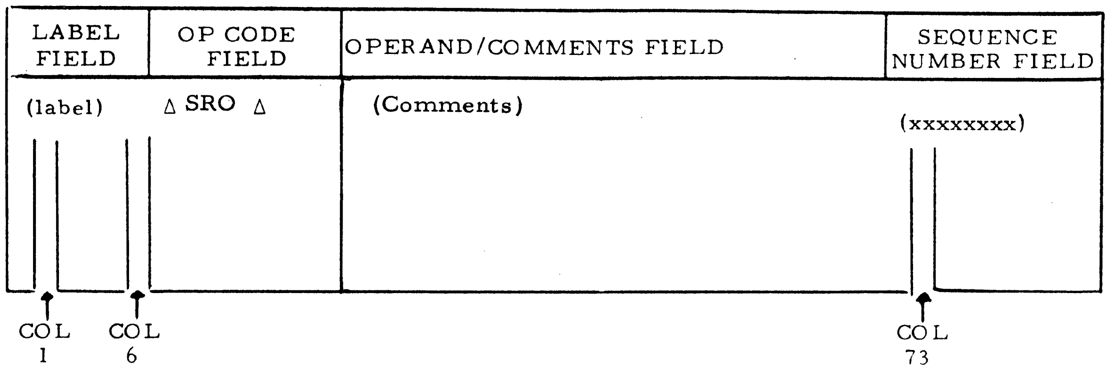


At assembly time, the Assembler generates either a short or long machine instruction with the command code 31 in the op code field and the effective address of the memory word in the operand field.

At program run time, execution of the AOM machine instruction causes the specified memory word to be accessed, and its value to be incremented by one. The hardware condition bit is set if no carry was generated by the addition operation; otherwise the condition bit is reset to zero.

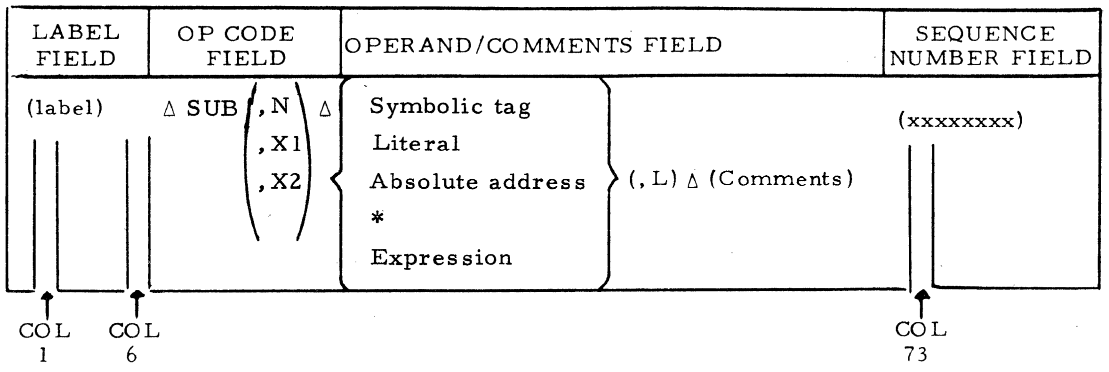
3.1.1.5 Shift Right One, Arithmetic Statement (SRO). This statement specifies that the value in the accumulator is to be shifted one bit position to the right and the sign bit is to be retained; the right-most bit is lost.

As shown in the following diagram, no operand is specified in the SRO source statement. The programer may optionally specify a label, comments, and sequence number field.



At assembly time, a 16-bit word is generated for use by the CPU when the SRO instruction is executed. At execution time, the value currently in the accumulator is shifted right one bit position.

3.1.1.6 <u>Subtract Statement (SUB).</u> The subtract statement specifies that the memory word specified in the operand field is to be subtracted from the contents of the accumulator, and the difference is to be stored in the accumulator. The Subtract statement format diagram follows.



At assembly time, the Assembler generates either a short or long machine instruction with the command code 14 in the op code field and the displacement of the memory word in the operand field.

At program run time, execution of the Subtract instruction causes the specified memory word to be accessed, its value to be subtracted from the value in the accumulator, and the resultant difference stored in the accumulator. If the subtraction operation causes an arithmetic overflow condition, the hardware condition bit is set to one; otherwise it is reset to zero.

3.1.2 Branch Statements

The source language for the PTS-100 Assembler provides two conditional and one unconditional branch statements to effect transfers of control within the executable program, as follows:

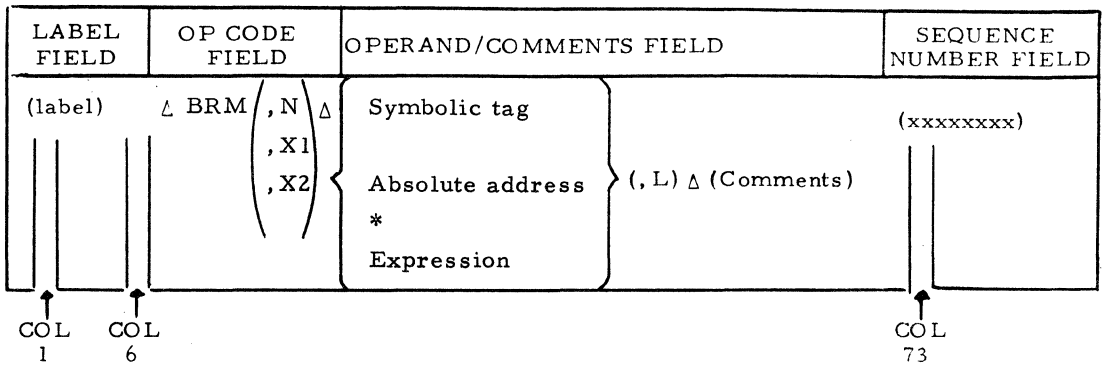
Branch if Accumulator Minus (BRM)

Branch if Condition Bit Set (BCB)

Jump (JMP)

These statements, their permissible formats, and the effects of their use are described individually below.

3.1.2.1 Branch If Accumulator Minus Statement (BRM). This statement specifies that the program execution is to branch to the location specified by the operand if the value in the accumulator is a negative number (i.e., if the Most Significant Bit = 1). The BRM source statement format is presented below.



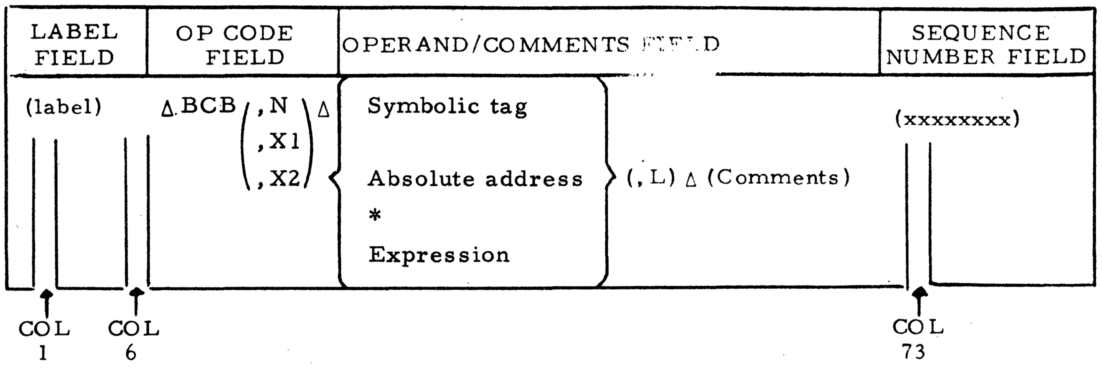
At assembly time, the Assembler generates either a long or short machine instruction with the command code 3 in the op code field and the displacement of the address to which control is to branch in the operand field.

At execution time, the most significant bit (MSB) in the accumulator value is tested. If it is 1 (minus), the branch address is placed in the program counter and the transfer of control takes place. If the MSB is 0 (positive) the next sequential program instruction is executed.

3.1.2.2 Branch If Condition Bit Set Statement (BCB). This statement tests the hardware condition bit (CB) after the execution of a machine instruction that may have set the CB to indicate one of the following:

Arithmetic overflow
No carry generated
The logical AND operation result is zero
True results of comparison tests

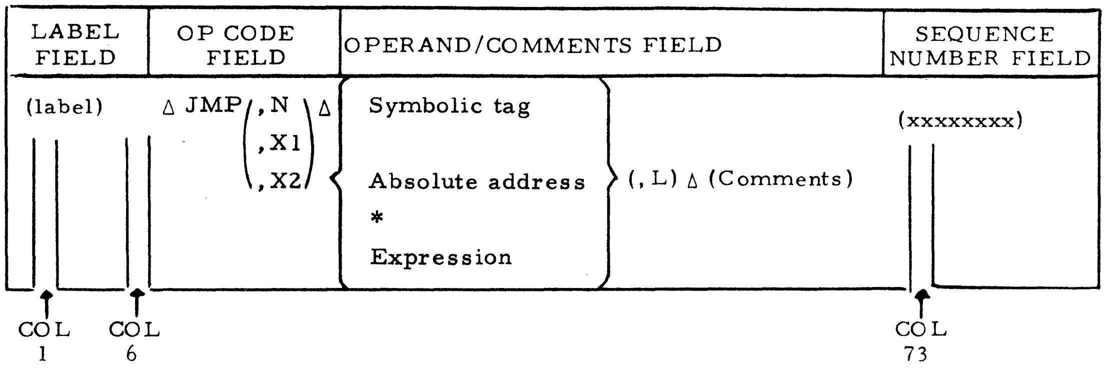
The BCB source statement format is presented below.



At assembly time, the Assembler generates either a short or long machine instruction with the command code 2 in the op code field and the effective address to which control is to transfer in the operand field.

When the BCB instruction is executed, the condition bit is tested to determine whether it is equal to one. If so, the branch address is placed in the program counter, and the transfer of control takes place. If the CB is not set, the next sequential program instruction is executed.

3.1.2.3 Jump Statement (JMP). This statement specifies an unconditional branch in program execution. The Jump statement format is diagramed below.



At assembly time, the Assembler generates either a short or long instruction with the command code 0 in the op code field and the effective address to which control is to transfer in the operand field.

When the Jump machine instruction is executed, the branch address is placed in the program counter and execution control jumps to the specified point.

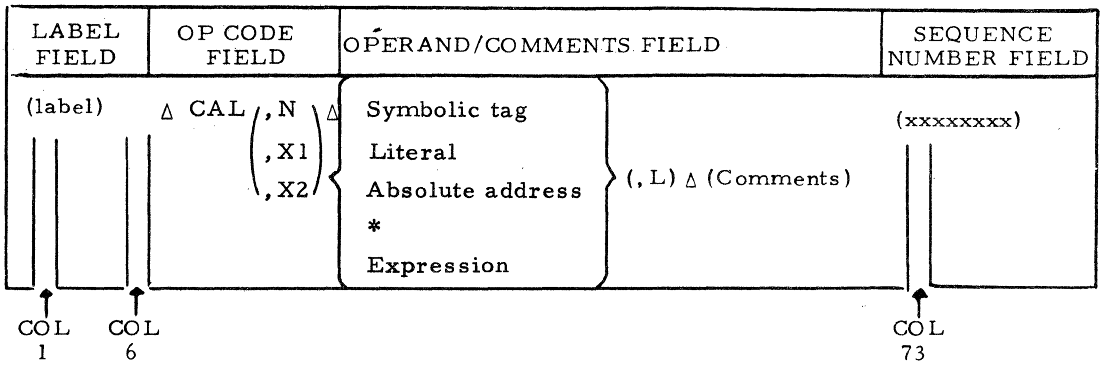
3.1.3 Compare Statements

The PTS-100 Assembler source language provides two statements to specify comparative testing of the current value of the accumulator against the value of memory words, as follows:

- Compare for Accumulator Less than Memory Word (CAL)
- Compare for Not Equal (CNE)

These statements are discussed in detail in the following paragraphs.

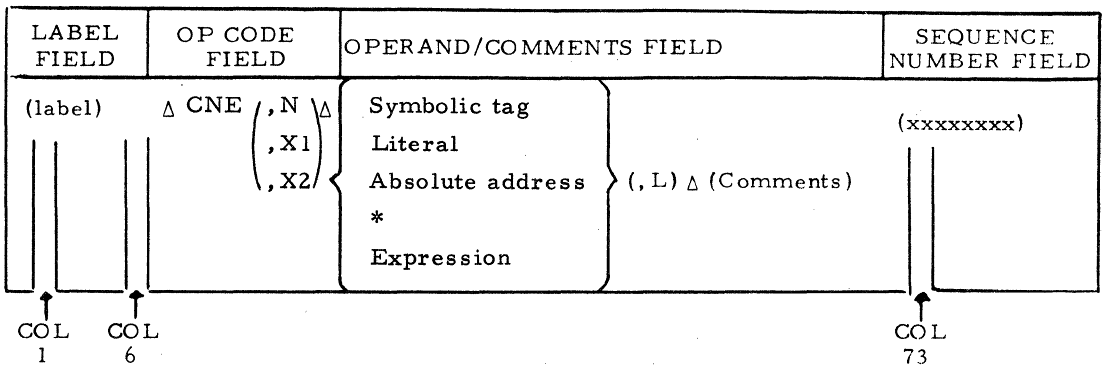
3.1.3.1 Compare Accumulator Less Than Word Statement (CAL). This statement specifies comparative testing of the current value in the accumulator with the value of the memory word specified by the operand to determine whether the magnitude of the accumulator value is less than that of the operand value. The CAL statement format is diagramed below.



At assembly time, the Assembler generates either a long or short machine instruction with the command code 17 in the op code field and the displacement of the memory word value in the operand field.

At execution time, the value currently stored in the accumulator is compared with the specified memory word value. If the magnitude of the accumulator value is less than the magnitude of the memory word, the hardware condition bit is set to one; otherwise the CB is reset.

3.1.3.2 <u>Compare For Not Equal Statement (CNE).</u> This statement specifies that a "not equal" comparison is to be made with the current value of the accumulator and the value of the memory word specified by the operand. The format of the CNE statement is diagramed below.



At assembly time, the Assembler generates either a short or long machine instruction with the command code 16 in the op code field and the displacement value of the memory word in the operand field.

At execution time, the specified memory word is accessed and compared with the value stored in the accumulator. If the two values are not equal, the hardware condition bit is set to one; otherwise it is reset.

3.1.4 Load Statements

There are six source statements that provide assembly language programers with the facility for loading data values or addresses into special registers:

Load Address in Index Register 2

Load Byte

Load Immediate

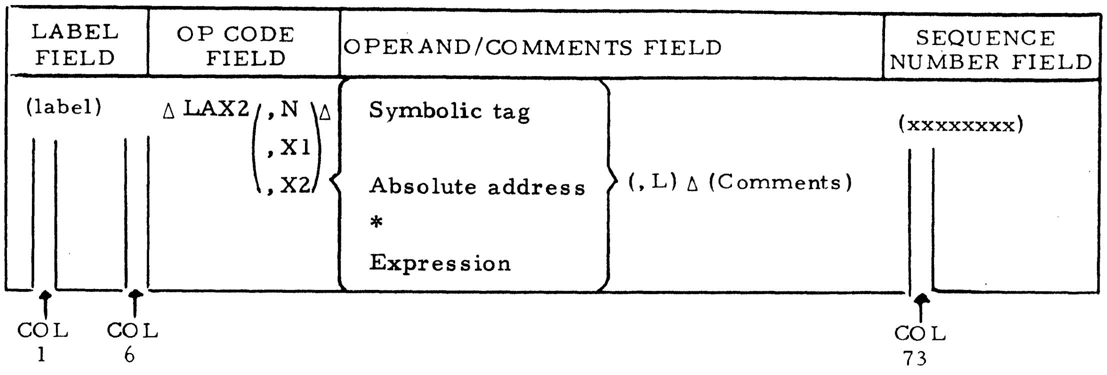
Load Index Register 1

Load Index Register 2

Load Word

These statements, their permissible formats, and the effects of their use are discussed in detail on the following pages.

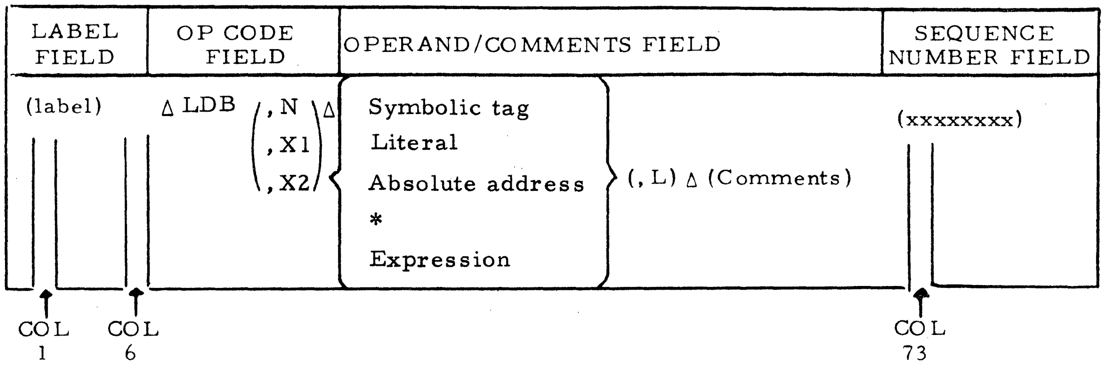
3.1.4.1 <u>Load Address In Index Register 2 Statement (LAX2).</u> This statement specifies that the address of the operand is to be placed in index register 2. The LAX2 statement format is diagramed below.



At assembly time, the Assembler generates a long or short machine instruction with the command code 8 in the op code field and the displacement value in the operand field.

At execution time, the actual address of the operand is computed and loaded into index register 2.

3.1.4.2 Load Byte Statement (LDB). This statement specifies that a data value one byte (8 bits) in length is to be retrieved from the memory location specified by the operand, stored in the right-hand half of the accumulator, and that the left-hand half of the accumulator is to be cleared. The LDB source statement format is diagramed below.



At assembly time, the Assembler generates either a short or long machine instruction with the command code 19 in the op code field and the displacement of the byte value in the operand field field.

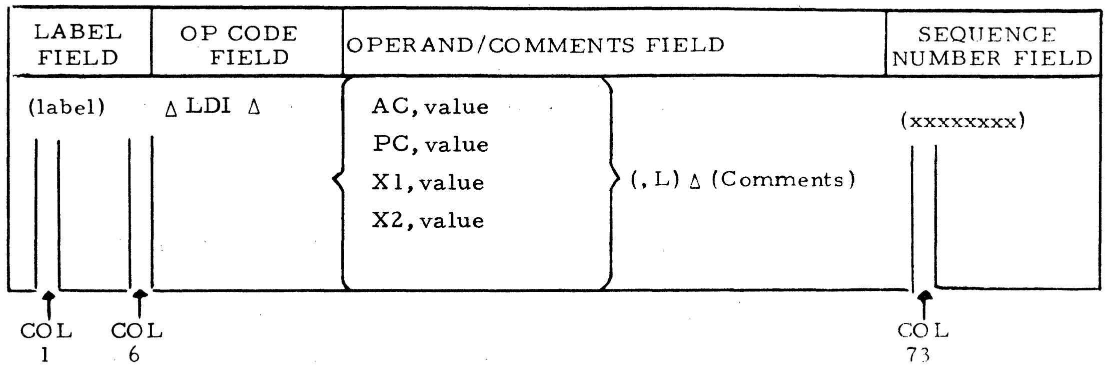
At execution time, the byte value stored at the effective address is loaded into the right-hand portion of the accumulator and the left-hand portion is zeroed.

3.1.4.3 Load Immediate Statement (LDI). This statement specifies that the value specified as the second operand is to be loaded into the register specified as the first operand. That is, the LDI source statement requires a specially formated two-element operand, as follows:

Element 1 specifies the register into which the numeric value is to be loaded.

Element 2 specifies the value to be loaded into the register specified as the first operand. The immediate value may be absolute (coded in hexadecimal, octal, or decimal notation) or may be a symbolic tag whose address becomes the immediate value.

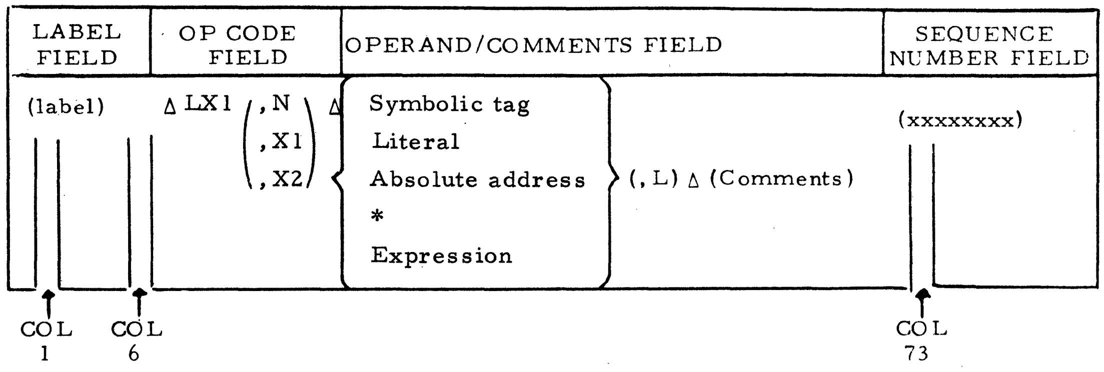
The operand field may optionally contain the characters , L following the immediate value to specify that the long machine instruction format is to be used by the Assembler. The elements in the operand field of the LDI source statement are separated by commas, as illustrated in the following diagram.



At assembly time, the Assembler generates either a long or short machine instruction with the command code 4 in the op code field. If a short instruction is generated, the immediate operand value appears in the operand field of the instruction. In a long instruction, the immediate operand value appears in the second word of the instruction.

At program run time, execution of the short form of the LDI instruction causes the byte operand to be placed in the right half of the specified register and the left half to be zeroed. Execution of the long form of the LDI instruction causes the word operand to be placed in the specified register.

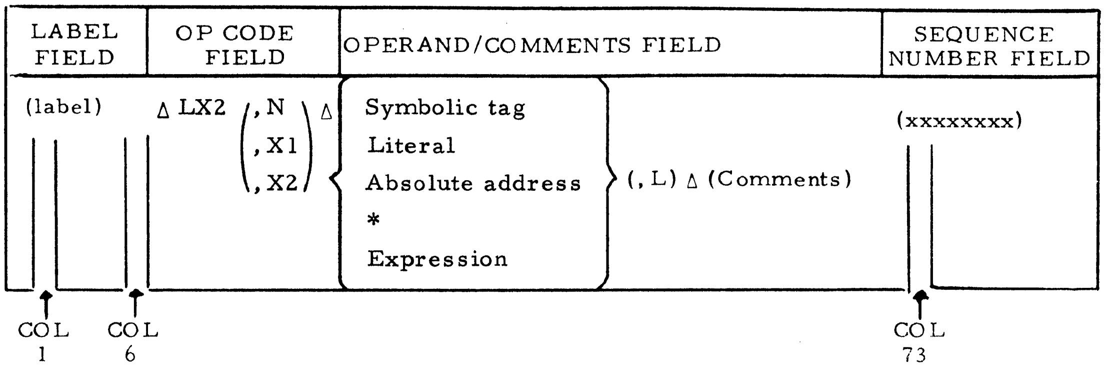
3.1.4.4 Load Index Register 1 Statement (LX1). This statement specifies that the value of the operand is to be loaded into index register 1. The format of the LX1 statement is diagramed below.



At assembly time, the Assembler generates a long or short machine instruction with the command code 20 in the op code field and the displacement of the memory word to be loaded in the operand field.

At execution time, the specified value is loaded into index register 1.

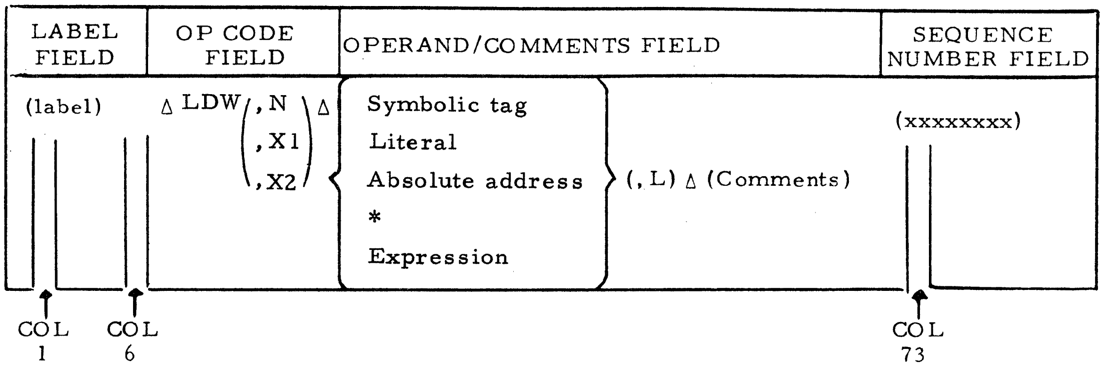
3.1.4.5 Load Index Register 2 Statement (LX2). This statement specifies that the value of the operand is to be loaded into index register 2. The format of the LX2 is presented below.



At assembly time, the Assembler generates a long or short machine instruction with the command code 21 in the op code field and the displacement of the memory word to be loaded in the operand field.

At execution time, the specified value is loaded into index register 2.

3.1.4.6 Load Word Statement (LDW). This statement specifies that the value of the operand is to be loaded into the accumulator. The format of the LDW statement is diagramed below.



At assembly time, the Assembler generates either a short or long instruction with the command code 18 in the op code field and the displacement of the value to be loaded in the operand field.

At execution time, the specified value is loaded into the accumulator.

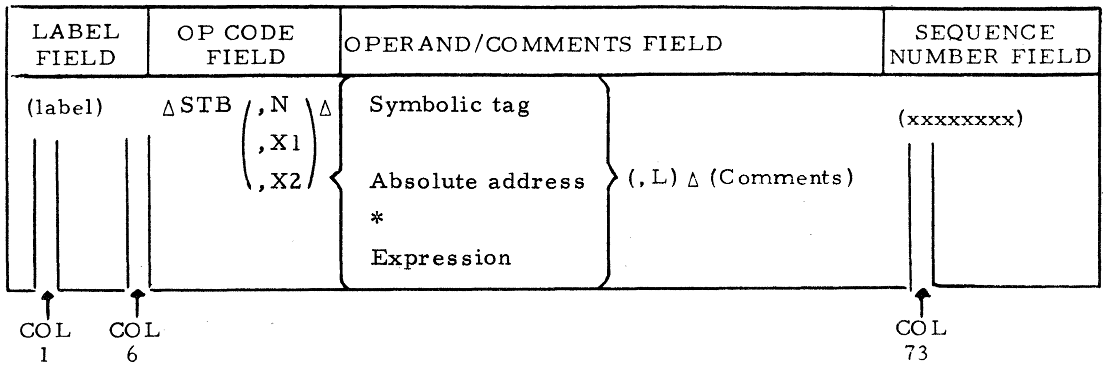
3.1.5 Store Statements

There are four source statements that provide assembly language programers with the facility for storing data values or addresses in memory locations:

Store Byte
Store Index Register 1
Store Index Register 2
Store Word

These statements, their permissible formats, and the effects of their use are discussed in detail on the following pages.

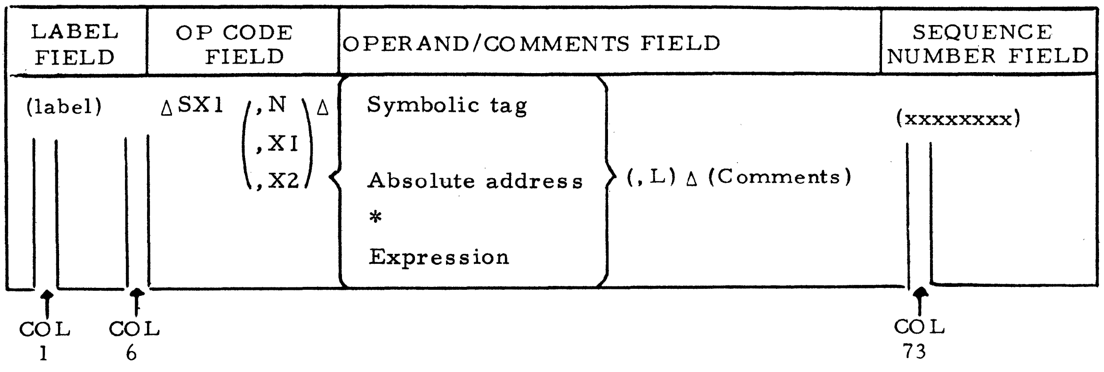
3.1.5.1 Store Byte Statement (STB). This statement specifies that the right-hand byte of the data value in the accumulator is to be stored in the byte location of the memory word specified by the operand. The format of the Store Byte statement is diagramed below.



At assembly time, the Assembler generates either a short or long machine instruction with the command code 28 in the op code field and the displacement of the value to be stored in the operand field. The least significant bit of the effective address indicates whether the byte value is to appear in the left-hand portion of the memory word (i.e., LSB = 0) or the right-hand portion (LSB = 1).

At execution time, the right-hand byte of the accumulator is stored in that portion of the memory word specified by the effective address of the machine instruction.

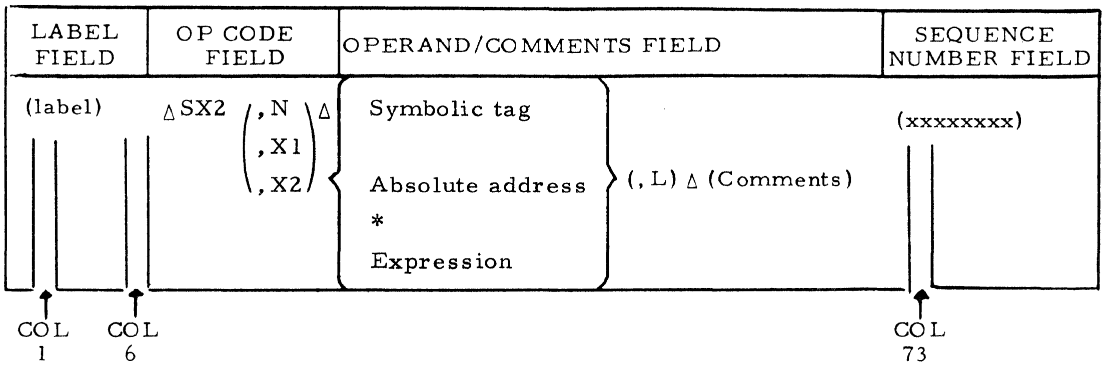
3.1.5.2 Store Index Register 1 Statement (SX1). This statement specifies that the current value in index register 1 is to be stored at the memory location specified by the operand. The format of the statement is presented below.



At assembly time, the Assembler generates a long or short machine instruction with the command code 26 in the op code field and the displacement at which the value is to be stored in the operand field.

When the SX1 instruction is executed, the value in index register 1 is stored in the memory location specified by the effective address.

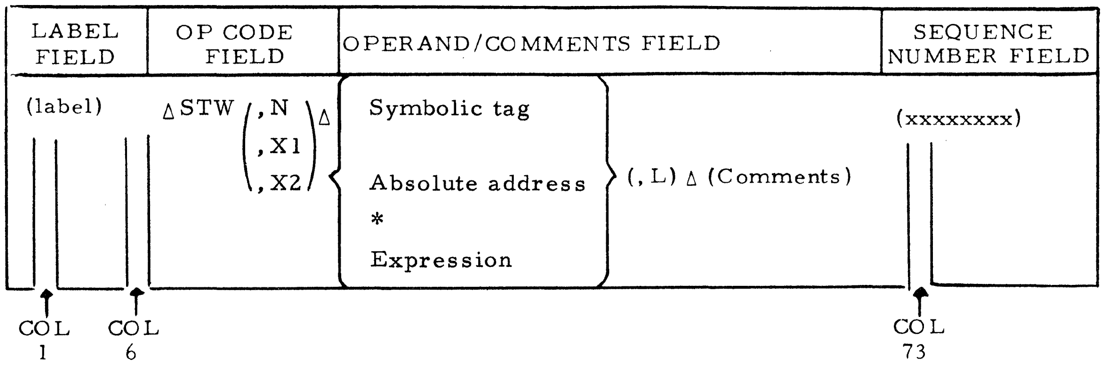
3.1.5.3 Store Index Register 2 Statement (SX2). This statement specifies that the current value in index register 2 is to be stored at the memory location specified by the operand. The permissible format of the statement is presented below.



At assembly time, the Assembler generates either a long or short machine instruction with the command code 27 in the op code field and the displacement at which the value is to be stored in the operand field.

At execution time, the current value of index register 2 is stored in the specified memory location.

3.1.5.4 Store Word Statement (STW). This statement specifies that the current value in the accumulator is to be stored in the memory word specified in the operand field. The Store Word source statement format is presented below.



At assembly time, the Assembler generates a long or short machine instruction with the command code 24 in the op code field and the displacement at which the current value of the accumulator is to be stored in the operand field.

At execution time, the current value of the accumulator is transferred to the effective address.

3.1.6 Logical Statements

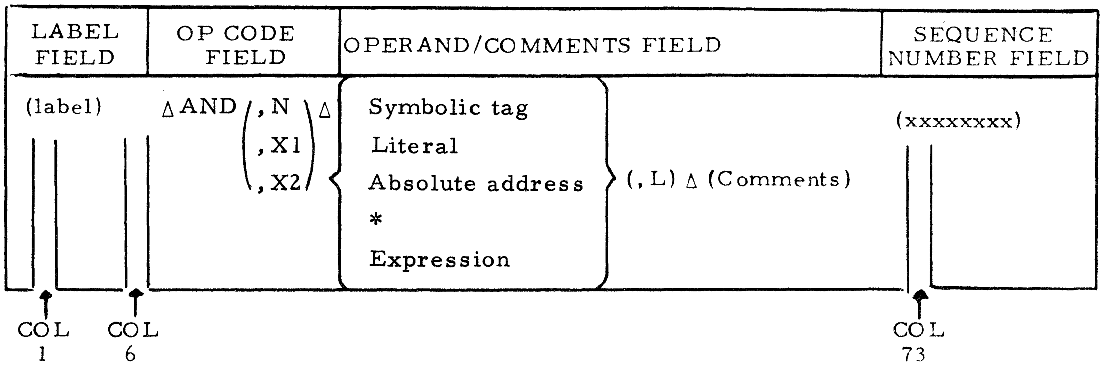
The PTS-100 Assembler provides the following two statements for logical combination of accumulator and memory word data values:

AND statement

Exclusive OR statement

These statements, their permissible formats, and the effects of their use are discussed in detail on the following pages.

3.1.6.1 AND Statement (AND). This statement specifies that the current value in the accumulator is to be ANDed with the value specified by the operand and the result is to be placed in the accumulator. The AND source statement format is diagramed below.



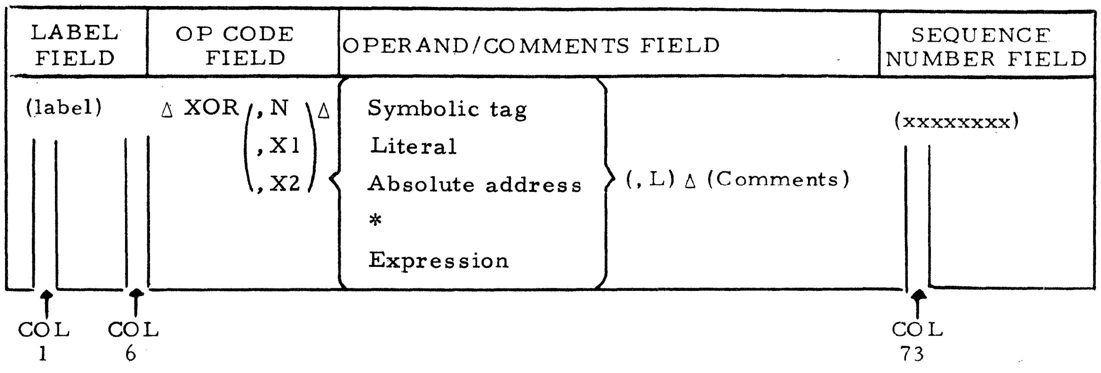
At assembly time, the Assembler generates either a long or short machine instruction with the command code 12 in the op code field and the displacement of the memory word in the operand field.

At execution time, the bits of the accumulator and of the memory word are ANDed. Both bits must equal 1 to produce a 1-bit setting in the resultant value, as illustrated below.

Current Accumulator Value:	01011101	10010011
Memory Word Value:	00100000	11101101
ANDed Resultant Value:	00000000	10000001

The resultant value is stored in the accumulator. If the resultant value of the AND operation is not equal to zero, the hardware condition bit is set to one; otherwise it is reset.

3.1.6.2 Exclusive OR Statement (XOR). This statement specifies that the current value in the accumulator is to be exclusive ORed with the value specified by the operand and the result is to be placed in the accumulator. The XOR source statement format is diagramed below.



At assembly time, the Assembler generates either a long or short machine instruction containing the command code 11 in the op code field and the displacement of the memory word value in the operand field.

At execution time, the bits of the current accumulator value and the memory word value are XORed to determine the resultant value, where one bit but not both must be one to produce a 1-bit setting in the resultant value, as illustrated below.

Current Accumulator Value:	01011101	10010011
Memory Word Value:	00100000	11101101
XORed Resultant Value:	01111101	01111110

The resultant value is stored in the accumulator.

3.2 Nonexecutable Statements

Nonexecutable statements are those that do not result in Assembler-generated machine instructions. That is, they are not executed by the CPU, but rather establish data values and reserve storage areas for use by the executable object program. For purposes of discussion, these statements are grouped as follows:

- Constant assignment statements, which are used to establish constant data values and address constants.
- Symbol defining statements, which assign values to symbols or identify symbols used or referenced by the program segment.
- Storage area assignment statements, which reserve storage areas for literal pools, absolute addresses, I/O data buffers and the executable program coding.

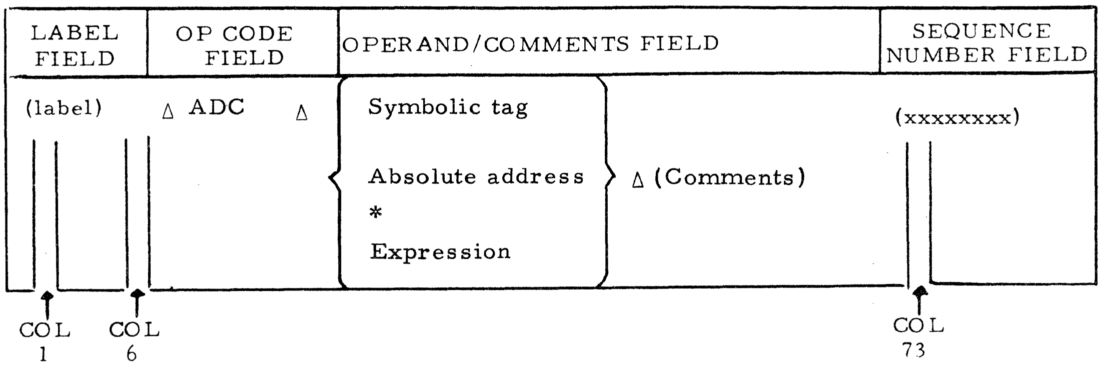
3.2.1 Constant Assignment Statements

Seven types of constants may be defined in PTS-100 Assembler source language:

Address Constants
Concatenated Integer Constants
Decimal Constants
Hexadecimal Constants
Octal Constant
Text (alphanumeric) Constants
TEX7 (7-bit alphanumeric) Constants

The constant assignment statement formats and usage are described in the following paragraphs.

3.2.1.1 Address Constant Statement (ADC). This statement defines an Address Constant. The ADC statement format is diagramed below.



There is one restriction on the use of a symbolic tag in the operand field of the ADC statement: if the operand is an external reference symbol (i.e., a symbol that is defined in a program segment other than the current one as described in the EXREF statement discussion), it must be the only element in the operand field. That is, it may not appear within an expression formed by combining other operand elements with the plus (+) or minus (-) sign.

3.2.1.2 Concatenated Integer Constant Statement (CAT). This statement is used to construct a concatenated integer constant one word (16 bits) in length, based on the values specified in the operand field. The format of the CAT source statement is presented below. However, the CAT statement is not implemented in the native version of the PTS-100 Assembler.

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS FIELD	SEQUENCE NUMBER FIELD
(label)	Δ CAT Δ	expression$_1$, expression$_n$ Δ (comments) where each expression is written in the format: absolute value:bits	(xxxxxxxx)

COL 1 COL 6 COL 73

As shown in the diagram above, the operand field may contain one or more expressions, each of which is written in the format

Absolute value:bits

where the absolute value may be expressed as one of the following:

A symbolic tag previously assigned an absolute value via the EQUate statement

An octal, hexadecimal, or decimal value

An expression formed by combining any of the above with the plus (+) or minus (-) sign

and where bits is a decimal number, from 1 - 16, specifying the number of bits the absolute value is to occupy in the 16-bit concatenated word. If a string of expressions is specified in the CAT statement operand field, the total number of bits must not be greater than 16. If fewer than 16 bits is specified for a concatenated word, the final value of the word is left-justified, and the right-most bit positions are zero filled.

Following are examples of CAT source statements, and the resulting word values they produce.

Example 1:

```
CAT  X'AB':8, 12:8
```

The expression X'AB':8 specifies that the hexadecimal value AB is to appear in an 8-bit field. The expression 12:8 specifies that the decimal value 12 is to occupy an 8-bit field in the concatenated word. The resultant integer constant constructed by this statement is shown below.

binary value	1 0 1 0	1 0 1 1	0 0 0 0	1 1 0 0
hexadecimal value	A	B	0	C

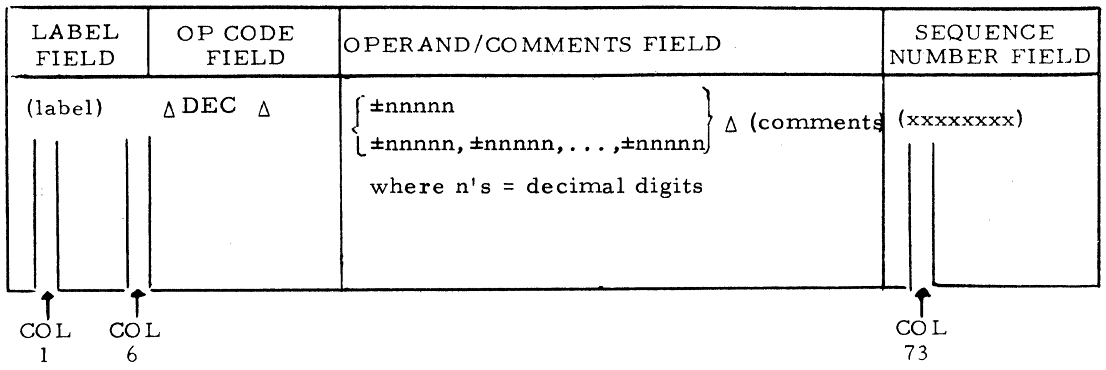
Example 2:

```
FIVE EQU 5
SIX EQU 6
     CAT  FIVE-1:3,SIX+4:7,O'77':6
```

The first two statements EQUate absolute values to the symbolic tags FIVE and SIX. The CAT statement specifies that the value FIVE-1 is to appear in a 3-bit field in the left-most portion of the concatenated constant word, the value SIX+4 is to appear in the next 7-bit field, and the octal value 77 is to appear in the right-most 6-bit portion of the word. The resultant integer constant is shown below.

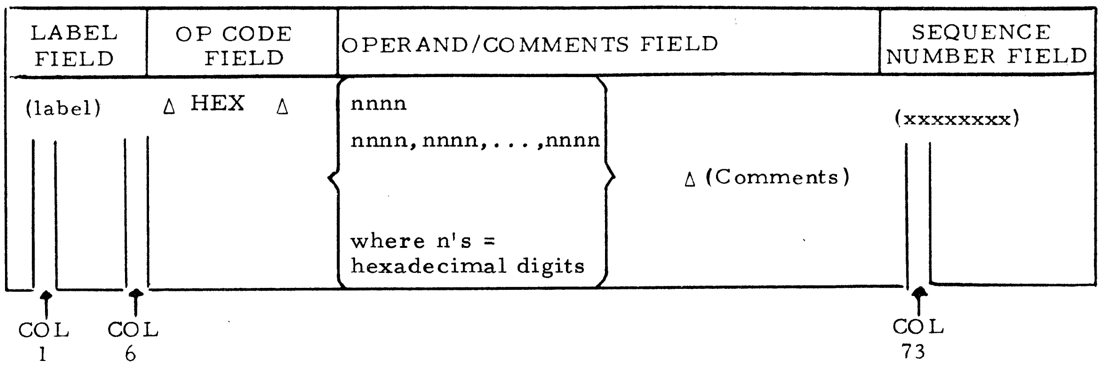
binary value	1 0 0 0	0 0 1 0	1 0 1 1	1 1 1 1
hexadecimal value	8	2	B	F

3.2.1.3 Decimal Constant Statement (DEC). This statement is used to define one or more 16-bit decimal constants. The format of the statement is shown below.



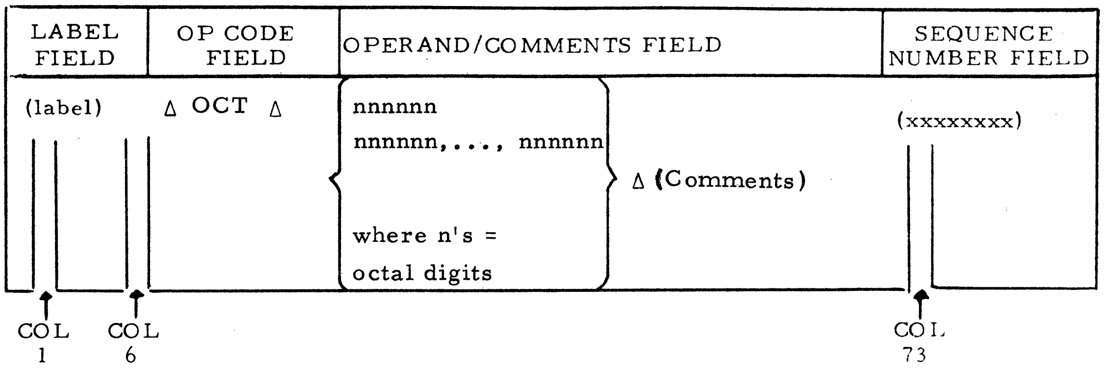
One or more decimal constant values may appear in the operand field. If two or more constant values are specified, they must be separated by commas. The magnitude of any given constant value must be less than 65535. Constant values may be preceded by the plus (+) or minus (-) signs. If a negative decimal value is specified, the two's complement of the binary representation of the value appears in the 16-bit memory word. The decimal constant value is right-justified in the memory word, with the left-most unused portion zero-filled. If a label appears in the label field of a DEC statement in which a string of constant values is specified, the label will become the symbolic tag of the first value in the operand field. That is, strings of values are assigned to consecutive storage locations, with the tag associated with the first (lowest) memory address.

3.2.1.4 Hexadecimal Constant Statement (HEX). This statement is used to define one or more 16-bit hexadecimal constant values, as shown in the following diagram.



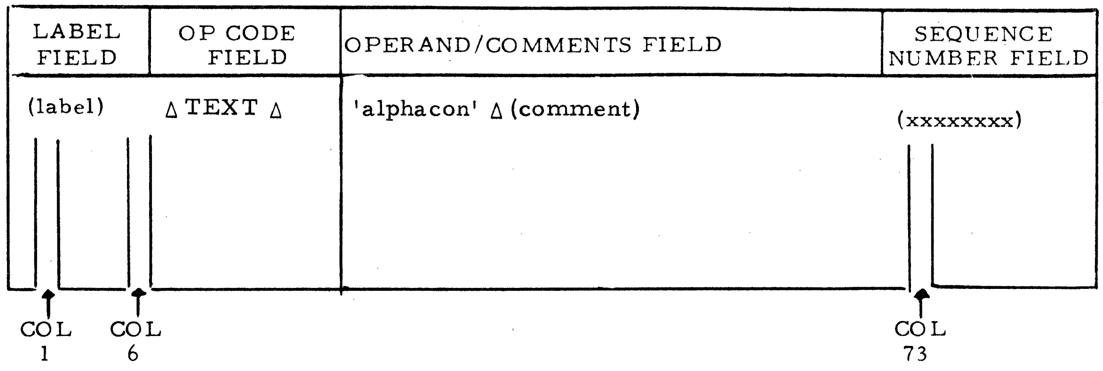
In the HEX statement, one or more hexadecimal constant values may be specified, each from one to four digits in length. If two or more constant values are specified, they are separated by commas. If a label is specified for a statement in which a string of values is specified, the label becomes the symbolic tag of the first value in the operand field. That is, strings of values are assigned to consecutive locations, with the symbolic tag associated with the first (lowest) memory address. If less than four digits are specified in a given value, the binary representation of the value is right-justified, with the left-most unused bit positions zero-filled.

3.2.1.5 Octal Constant Statement (OCT). The OCT statement is used to define one or more 16-bit octal constant values, as shown in the following diagram.



As shown in the diagram above, one or more octal constant values may be specified, each from one to six digits in length. If a label is specified for a statement in which a string of values is specified, the label becomes the symbolic tag of the first value in the operand field. If less than six digits are specified in a given octal constant value, the binary representation of the value is right-justified in the memory word, with the left-most unused bit positions zero-filled. If six digits are specified, the two high-order bits of the first digit are truncated.

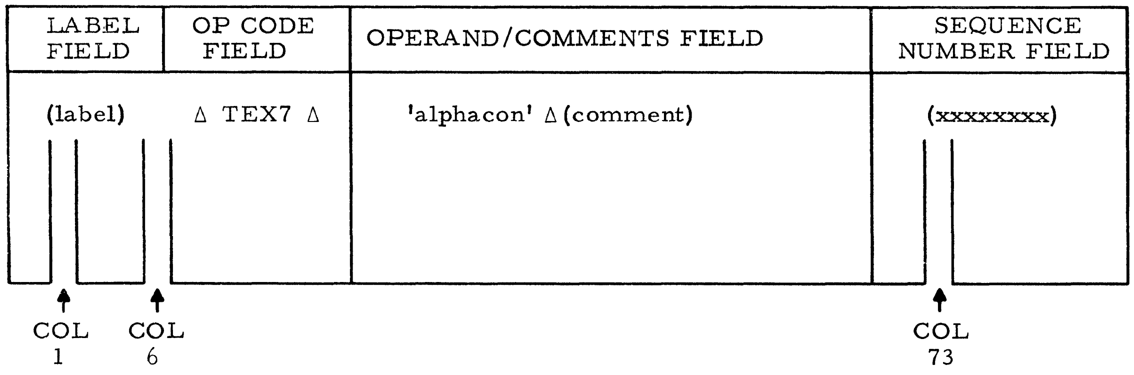
3.2.1.6 Text Constant Statement (TEXT). This statement is used to define an alphanumeric constant from one to forty characters in length. The alphanumeric constant value appears in the operand field, as shown in the format diagram below.



The alphanumeric constant value must be enclosed in single quotation marks, which are used as delimiters. The constant value may contain any characters from the PTS-100 character set (see Appendix A) except the single quotation marks. If quotation marks are to appear within the alphanumeric constant value, the programer uses double quotation marks, which will be replaced by the single quotation marks when the constant value is assembled. Alphanumeric constants must start on a word boundary.

Alphanumeric constants are stored as 8-bit ASCII characters (i.e., two characters per memory word). If the constant value contains an uneven number of characters, the last character will appear in the left-most byte of the last memory word, and the right-most byte will be blank-filled.

3.2.1.7 Text Constant (7-bit) Statement (TEX7). This statement is used to define an alphanumeric constant from one to forty characters in length. The alphanumeric constant value appears in the operand field, as shown in the format diagram below.



The alphanumeric constant value must be enclosed in single quotation marks, which are used as delimiters. The constant value may contain any characters from the PTS-100 character set (see Appendix A) except the single quotation mark. If quotation marks are to appear within the alphanumeric constant value, the programer uses double quotation marks, which will be replaced by the single quotation mark when the constant value is assembled. Alpha-numeric constants must start on a word boundary.

Each character in the alphanumeric constant is treated as a 7-bit ASCII character and stored as an 8-bit character, with the most significant bit (MSB) set to 0 (i.e., two characters per memory word). If the constant value contains an uneven number of characters, the last character will appear in the left-most byte of the last memory word, and the right-most byte will be blank-filled.

3.2.2 Symbol Defining Statements

Three source statements are used to define or identify symbols in the PTS-100 assembly language:

- The Equate statement is used to assign an absolute value to a symbol.
- The External Definition statement informs the Assembler that a defined symbol in one program segment is to be referenced in another program segment.
- The External Reference statement informs the Assembler that a referenced symbol in one program segment is to be defined in another program segment.

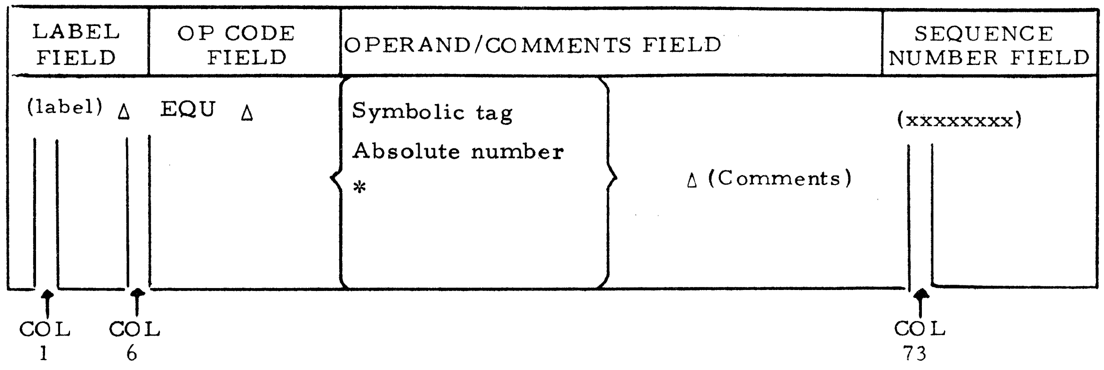
3.2.2.1 <u>Equate Statement (EQU).</u> This statement is used to assign an absolute address value to a symbol. The symbol must appear in the label field of the source statement, and the operand field must contain the absolute address value, expressed as one of the following:

The self-referencing indicator (*)

An absolute decimal, hexadecimal, or octal number

Another symbol, previously assigned an absolute number value in another EQUate statement.

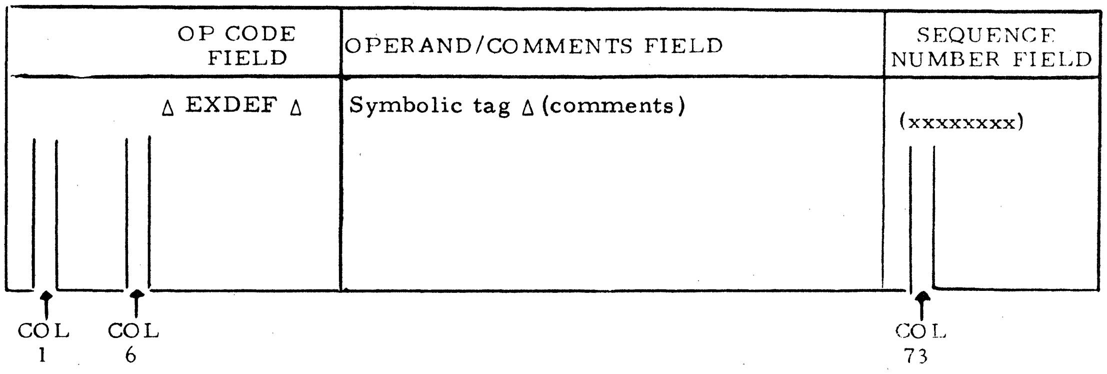
The format of the Equate statement is presented below.



At assembly time, the operand field is evaluated and the resulting absolute number is assigned as the address value of the specified symbol in the symbol table created by the Assembler.

Each time the symbol is referenced in the executable program, the address value is used to locate the associated data value.

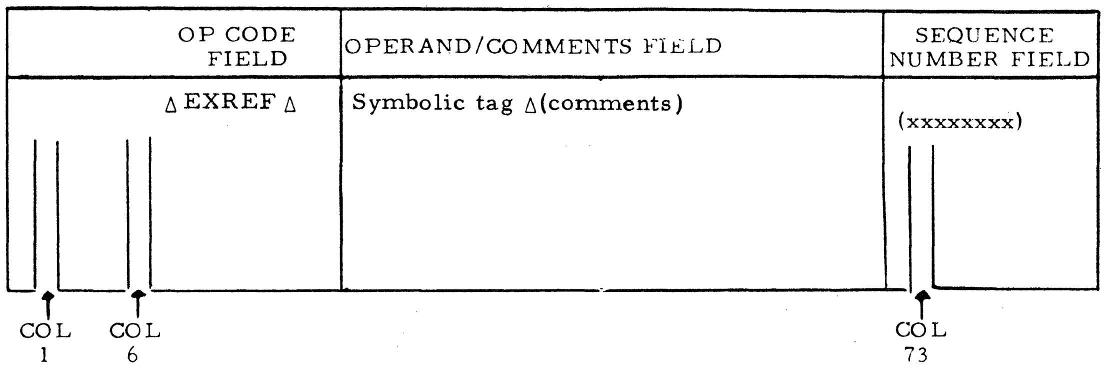
3.2.2.2 <u>External Definition Statement (EXDEF).</u> This statement is used to inform the Assembler that the symbol defined in the immediately preceding or following source statement in the program segment currently being assembled is to be referenced in some other program segment to which the current segment will be linked at load time. The EXDEF statement format is presented below.



Note that the label field in the EXDEF statement is not used. That is, a label specified for this statement will be ignored by the Assembler. The EXDEF op code may begin in any column other than column 1, which must be blank.

At assembly time, the Assembler places the named symbol and its address value in the symbol table. The symbol and its address are subsequently written on the relocatable loading file. At load time, the relocating loader resolves the address of the symbol when it is referenced in another program segment.

3.2.2.3 External Reference Statement (EXREF). This statement informs the Assembler that a symbol referenced in the program segment currently being assembled is defined in some other program segment to which the current segment will be linked at load time. The EXREF statement format is diagramed below.



Note that the label field in the EXREF statement is not used. That is, a label specified for this statement will be ignored by the Assembler. The EXREF op code may begin in any column other than column 1, which must be blank.

At assembly time, the Assembler places the symbol in the symbol table. The symbol and the addresses referencing it are written on the object file for resolution by the Absolute/ Relocating Loader. At load time, the address of the symbol is resolved by the Loader when the current program segment is linked to the program segment in which the symbol is defined.

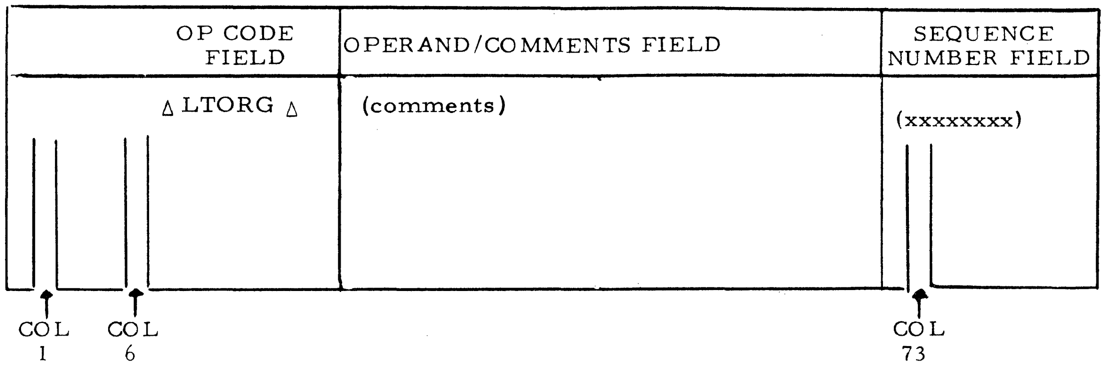
3.2.3 Storage Assignment Statements

The storage assignment statements allow the programer to establish memory storage locations for source coding, literal values, and buffer or data areas. There are five storage assignment statements:

- Literal Origin statement, which establishes the storage areas for blocks of literal data values.
- MOD statement, which causes the instruction following it to be allocated to the storage location that is the next higher multiple of a given power of two.
- Origin statement, which specifies the beginning storage location at which object program loading is to begin.
- Page 0 statement, which causes an absolute address to be assigned to a symbol.
- Reserve statement, which specifies that a memory area is to be reserved for use as a buffer or data storage area.

These statements are described in the following paragraphs.

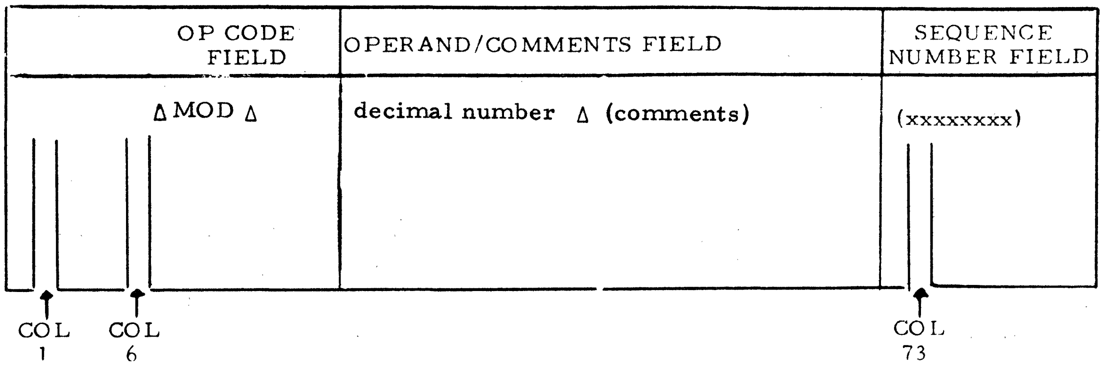
3.2.3.1 Literal Origin Statement (LTORG). It is the programer's responsibility to indicate where literal data values are to be stored within his program. The Literal Origin (LTORG) statement is used to inform the Assembler that a literal storage pool is to be set up within the program. The LTORG statement format is diagramed below.



As shown above, the label field and the operand field are not used in the LTORG statement. The op code may begin in any column after column 1, which must be blank. The op code is terminated by a blank character, after which a comment may be specified.

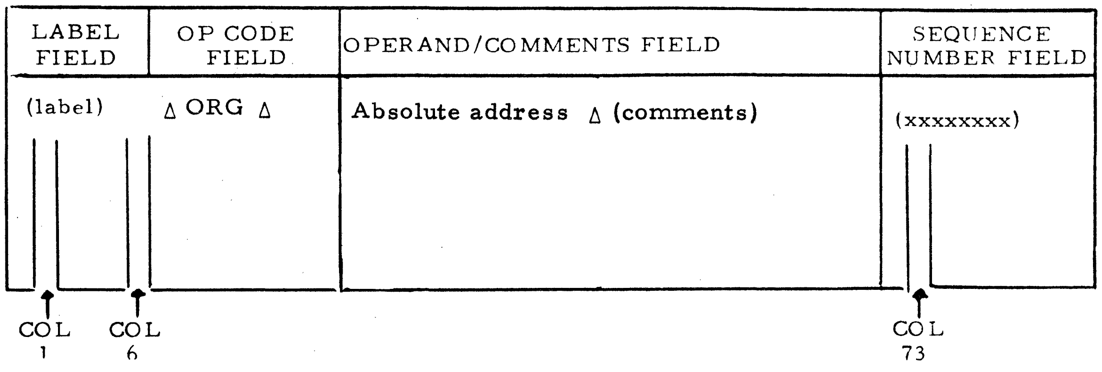
When the Assembler encounters a LTORG statement in a source program, it establishes a literal storage area, beginning with the location at which the LTORG statement was encountered and continuing through the number of sequential locations required to store all literals defined since the beginning of the program, or since the last LTORG statement was encountered. That is, all literal values defined prior to the occurrence of a given LTORG statement are assigned storage locations in the same sequential order as their appearance in the program. Duplicate literal values are eliminated only when they both appear within the block of source coding preceding a given LTORG statement. Thus, the Assembler establishes a new literal table each time a LTORG statement is encountered, and writes the previous literal table on an Intermediate Text file. Therefore, redundant entries between tables are not eliminated.

3.2.3.2 Mod Statement (MOD). This statement specifies that the statement immediately following it is to be stored in the next storage location that is a multiple of the power of two which is specified as its operand. The MOD statement format is shown in the diagram below.



As shown above, the label field in the MOD statement is not used. The op code may begin in any column after column 1, which must be blank. The op code is terminated by a blank character, after which the operand is specified as a decimal number, which must be a power of two. When the Assembler encounters this statement, it locates the object coding of the statement immediately following the MOD statement at the next full-word location that is a multiple of the value specified as the operand.

3.2.3.3 Origin Statement (ORG). One ORG statement may optionally be used as the first statement in a source program to specify the origin of the object program (i.e., the first memory location at which object program loading is to begin). The ORG statement is diagramed below.

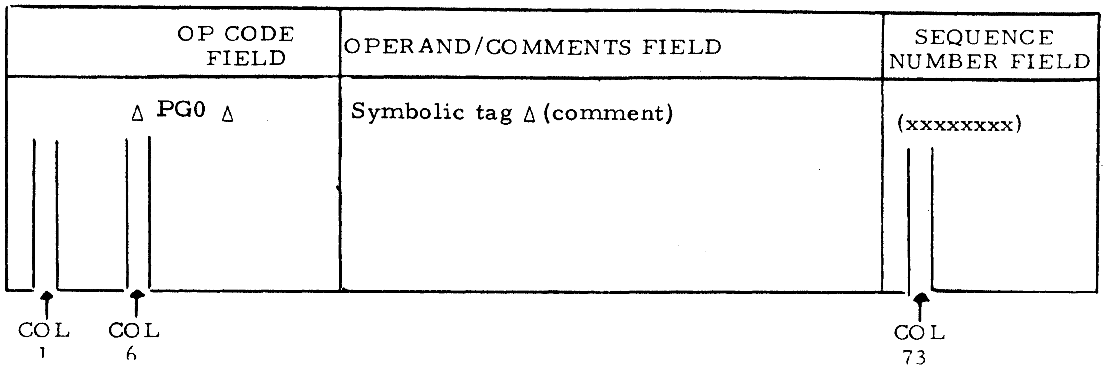


As shown in the diagram above, the ORG statement label is optional. The operand field of the statement must specify the absolute address, which must be expressed as a hexadecimal, octal, or decimal number. If a label is defined for the ORG statement, it is assigned the address value of the operand.

If a program contains several segments, only one segment (the one to be loaded first) may contain an ORG statement at its beginning. If more than one ORG statement should appear in a program, the duplicate statement(s) will not be detected by the Assembler. That is, the statements will be accepted, and unpredictable results will occur at load time.

If no ORG statement appears in a program, object program loading will begin at location 0.

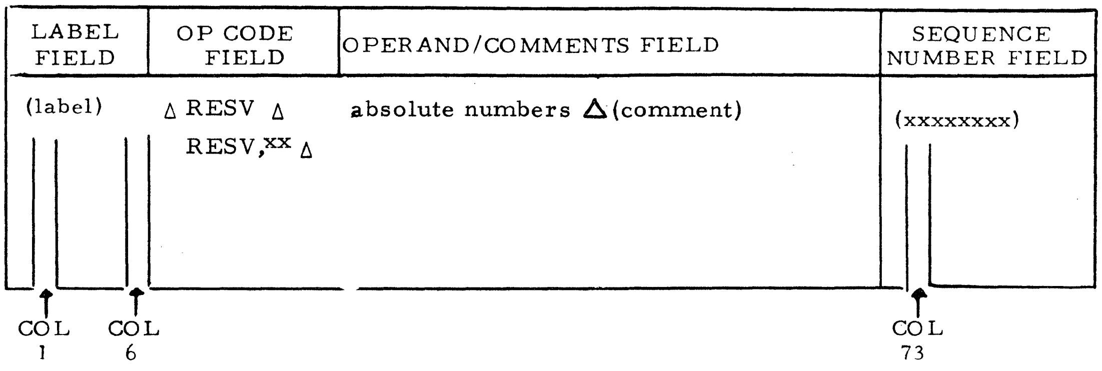
3.2.3.4 Page 0 Statement (PG0). The PG0 statement is used to specify that a symbolic tag is to be assigned an absolute address when it is used in an executable instruction. The format of this statement is shown below.



As shown above, the label field is not used in the Page 0 statement. The op code may begin in any column after column 1, which must be blank. The operand field may contain one symbolic tag of a data value defined in a statement immediately following or preceding the Page 0 statement. That is, the Page 0 source statement must physically appear either immediately before or after the statement in which the value of the symbol is defined. The symbol may be defined as the label of an EQUate statement whose operand specifies its actual value, the label of some other statement in the current program, or the operand of an EXREF statement in the current program.

The PG0 statement identifies the symbolic tag as a reference to page 0; thus when the symbolic tag is used in an executable instruction, the absolute address of the symbolic tag appears in the operand field of the Assembler-generated machine instruction.

3.2.3.5 Reserve Statement (RESV). This statement is used to inform the Assembler that an area of memory is to be reserved for use as a buffer or a data storage area. It is written in the formats shown below.



A label may optionally be specified for the Reserve statement. If it is specified, it is assigned the address of the first word location of the reserved area.

The op code may be written in either of the forms:

RESV or RESV,xx

The first form specifies that a zero-filled storage area of the byte-length specified by the operand is to be reserved on a word boundary in memory. The RESV,xx form specifies that every byte location in the reserved area is to be set to the hexadecimal value specified by xx.

The operand of the RESV statement must be an absolute number expressed in octal, hexadecimal, or decimal notation, specifying the number of bytes to be reserved. If the absolute number is an odd number, the Assembler increments it by one (i.e., makes it even) to preserve subsequent storage allocation on word boundaries. Thus, the Assembler responds to the Reserve statement request by reserving enough full storage words to accommodate the maximum number of bytes specified in the operand field.

3.3 Program Control Statements

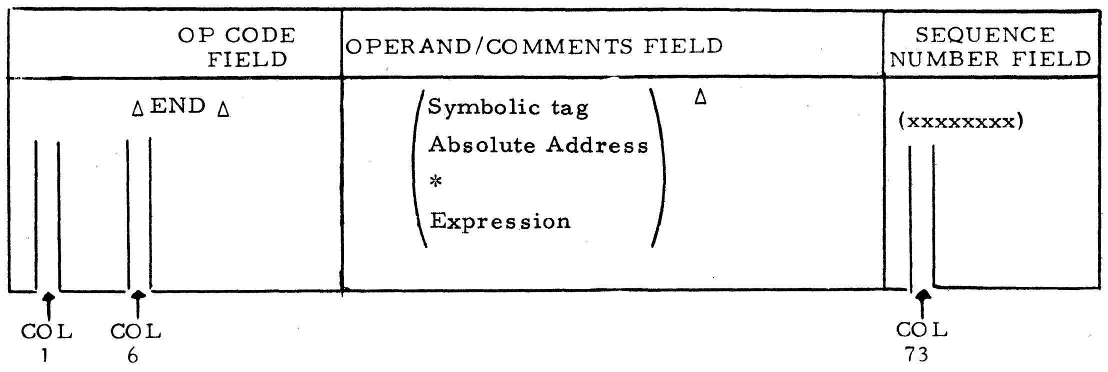
The program control statements allow the programer to control the object program listing, to specify the end of program assembly, and to specify the starting address for program execution. There are four such statements in the PTS-100 Assembler language.

- The END statement, which terminates program assembly, and optionally specifies the starting address for program execution.
- The SKIP statement, which controls the vertical spacing of the object program listing.
- The UNLIST statement, which tells the Assembler to suspend production of the object program listing.
- The LIST statement, which rescinds the UNLIST statement (i.e., resumes production of the object program listing).

These statements are individually described below.

3.3.1 End Statement (END)

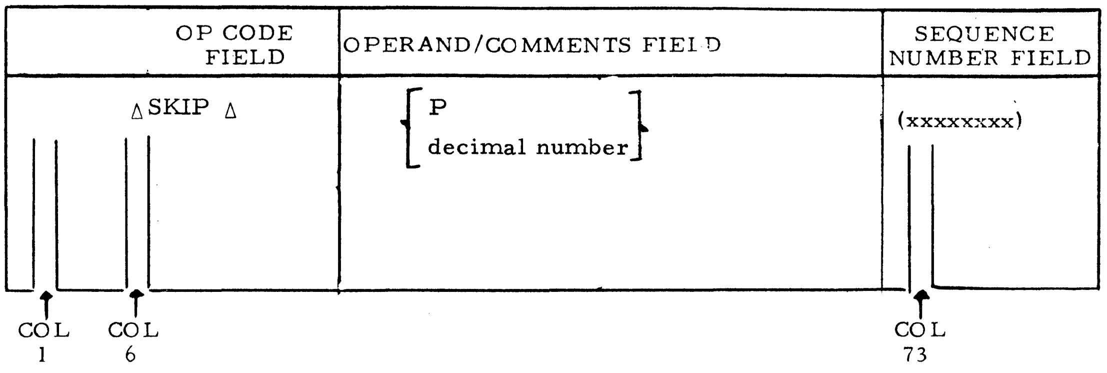
The END statement marks the end of the source program (i.e., terminates a given assembly) and may optionally specify the starting address of program execution (i.e., the address of the first instruction to be executed in object program, which is the point at which the Absolute/Relocating Loader is to turn control over to the executable program). The format of the END statement is presented below.



As shown above, the label field is not used in the END statement. The op code may begin in any column after column 1, which must be blank. The operand is optional in the END statement. If present, it may contain any of the elements shown above. When several program segments are to be individually assembled and combined into one object program, only the last segment should contain an END card with a starting address specified in the operand field. That is, when the Loader loads an END statement with a starting address, it places the address in the program counter and starts execution of the program. Hence, when the program is to be debugged, the END statement of the source program must not contain an execution starting address if the Debug program is to be loaded as the last part of the object program to be debugged, as described in Part 3 of this handbook.

3.3.2 Skip Statement (SKIP)

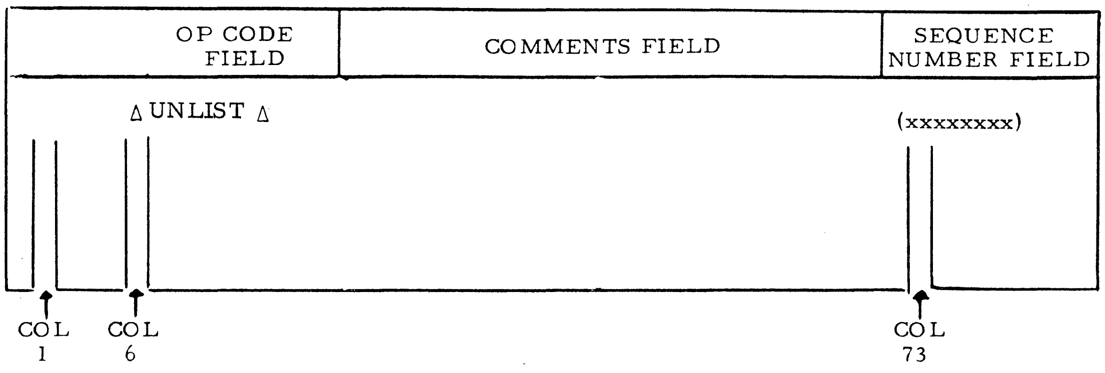
The SKIP statement causes the object listing produced by the Assembler to be vertically spaced as specified by the operand. The format of the statement is shown below.



The label field is not used in the SKIP statement. The op code may begin in any column after column 1, which must be blank. The operand may be a decimal number from 1 - 10, specifying the number of print lines to be skipped within a given page of the object listing, or the value P, specifying that the Assembler is to skip to the top of the next page to continue the object listing.

3.3.3 Unlist Statement (UNLIST)

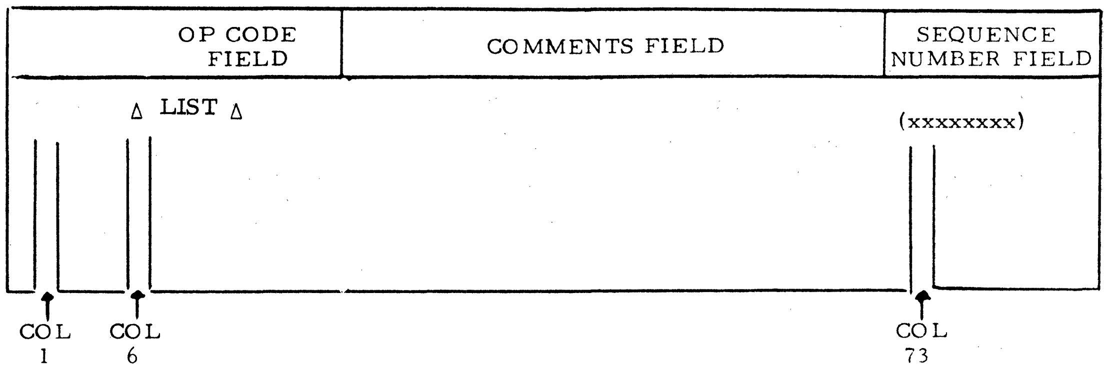
The UNLIST statement specifies that the Assembler is to temporarily suspend the output object listing at the point at which the UNLIST statement is encountered. The UNLIST statement is diagramed below.



As shown above, the label and operand fields are unused in the UNLIST statement. The op code may begin in any column after column 1, which must be blank. The object listing remains suspended from the point at which an UNLIST statement is encountered until a LIST statement appears in the program, or until the end of the source program.

3.3.4 List Statement (LIST)

The LIST statement rescinds the UNLIST statement. That is, it tells the Assembler to resume printing the object program listing. Only the op code LIST is required in the statement, beginning anywhere after column 1, which must be blank, as shown below.



3.4 Input/Output Services

With each PTS-100 System an IOCS monitor is available to perform the following functions:

Automatic device interrupt handling

Processing I/O requests from applications programs.

The device interrupt handling capability is built into the specific IOCS monitor for a given PTS-100 when the user's system is generated. Thus, device interrupt handling is performed automatically for individual programs according to the parameters and priorities defined prior to system generation. For a description of available devices, their logical unit number identifications (LUN ID's), and the interrupt priorities assigned to them, the programer should consult the system generation documentation of his specific PTS-100 System. For description of logical IOCS for disc see Section 3.5 of this part.

Applications programs may request the following basic I/O device services from the monitor:

Device initialization

Device opening and closing

Device input/output actions

A common system exit at the end of a processing job.

In all cases, a program must issue a monitor service call (i.e., a source statement with MSC in the op code field) to signal a request for monitor services. The MSC statement must be followed immediately by a Decimal Constant (DEC) statement whose operand is one of the following monitor service identification codes:

0 = request for a common system EXIT

1 = request to CLOSE a specific device

2 = request to INITialize all devices

3 = request for Watchdog Timer Service

4 = request for channel interface controller service

5 = request for device status sensing

6 = request to OPEN a specific device

7 = request to perform a specific I/O file action (IOACT) on a specific device

11 = request to change peripheral device addresses

Except for the EXIT request, the DEC statement must be followed by an Address Constant statement whose operand is an address within the program to which the monitor is to return processing control when the request has been serviced.

The three service requests concerned with specific devices are CLOSE, OPEN, and IOACT. The argument lists for these requests must specify the device to be used by passing the logical unit number identification (LUN ID) to the IOCS monitor. In the CLOSE and OPEN requests, the LUN ID is passed to the monitor via a constant statement. In the case of the IOACT request, the LUN ID is passed to the monitor in a programer-defined table called the file I/O Block (FIOB), described in subsection 3.4.1. The FIOB is referenced in an Address Constant statement which appears as the last argument of the IOACT request. Associated with the FIOB is the programer-defined Input/Output Control Queue (IOCQ) table entry, described in detail in subsection 3.4.2. The IOCQ table is used by the monitor to queue I/O requests for particular input/output device channels. An IOCQ table entry is required for each IOACT request. In the OPEN request, the starting address of the IOCQ is given as an operand in an Address Constant statement.

For CLOSE and OPEN requests, the last argument must be a Reserve statement to set up a storage word in which the monitor can store a code indicating any error that may occur during the attempt to OPEN or CLOSE the device. In the case of the IOACT request, the device error code is returned to the FIOB, as described below.

3.4.1 File Input/Output Block Definition

For each input/output service requested from the IOCS monitor, the programer describes the parameters of the request in a 9-word File Input/Output Block (FIOB), the format of which is shown in figure 2-3.

Bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Word 0	(spare)								ERROR CODE							
Word 1	MODE			FUNCTION					LOGICAL UNIT NUMBER ID							
Word 2	BUFFER ADDRESS (starting byte)															
Word 3	BYTE COUNT															
Word 4	TRANSLATE TABLE BASE OR DISC ADDRESS															
Word 5	SEARCH TABLE BASE OR DISC ADDRESS															
Word 6	(Spare)															
Word 7	(Spare)															
Word 8	(Spare												LUN extension			

Figure 2-3. Format of File Input/Output Block (FIOB)

The individual fields of the FIOB and their significance are as follows:

Word 0: Error Code Field (8-bit field). After the I/O request has been processed, the IOCS monitor returns one of the following codes to this field:

Code	Meaning
0	No error
1	Device not operational
2	No such LUN
3	LUN already open
4	LUN not open
5	Queue full

Word 1: This word contains three fields: the data transfer MODE field, the device FUNCTION field, and the LUN ID field, as described individually below.

MODE field (bits 0 - 2): the programer specifies the data transfer mode to be used by the device controller by setting these bits to the appropriate value, as follows:

Value	Transfer MODE
0	No Search or Translate function is to be used.
1	Use the Translate function with no interrupt condition when the MSB is on, using the Translate Table Base (TTB) whose address is specified in word 4 of the FIOB.
2	Not defined
3	Not defined
4	Not defined

Value	Transfer MODE
5	Use Translate function with interrupt condition when MSB is on (i.e., Search and Translate through a common table using the Translate Table Base whose address appears in word 4 of the FIOB).
6	Use Search function and set interrupt condition when the MSB is on. That is, search only and use the Search Table Base (STB) whose address is given in word 5 of the FIOB.
7	Use Search function and set interrupt condition when the MSB is on, using the STB, then translate with no interrupt condition when the MSB is on, using TTB addressed in word 4 of the FIOB.

FUNCTION Field (bits 3-7): This field specifies the code for the particular device function requested (i.e., read, write, rewind, etc.). The specific codes for various device functions are shown in table 2-6.

LUN ID Field (bits 8-15): The programer uses this field to specify the assigned LUN ID of the device controller on which the I/O request is to be performed. The IOCS monitor will translate the LUN ID into the physical address of the requested device.

Word 2: In this word the programer specifies the 16-bit starting address of the buffer to or from which input or output data is to be transferred.

Word 3: The programer specifies the number of bytes of data to be transferred in 15 bits of this word. Bit zero is not used.

Disc Word 3: The programmer specifies the byte count in bits 0 through 15 (must be an even number of bytes).

Word 4: This 16-bit field is used to specify the base address (i.e., the location of the first byte) of the Translate Table if the Translate function is specified in the MODE field of word 1.

Disc Word 4: Bits 0 and 1 are spares; bits 2 through 6 specify the track address; bits 7 through 15 specify the cylinder address.

Word 5: If the Search function is specified in the MODE field of word 1, the programer must specify the 16-bit base address (i.e., the location of the first byte) of the Search Table to be used by the monitor.

Disc Word 5: Bits 0 through 10 are spares; bits 11 through 15 specify the sector address.

Word 6: (Spare)

Word 7: (Spare)

Word 8: Bits 13 - 15 of this word are used to specify an extended identification number for a specific device on a device controller to which multiple devices may be attached. For example, four cassette tape devices may be attached to one controller. The LUN ID extension identifies the specific drive to be used, as follows:

Cassette	0 = 000
	1 = 001
	2 = 010
	3 = 011
Disc	0 = 000
	1 = 001
	2 = 010
	3 = 011
	4 = 100
	5 = 101
	6 = 110
	7 = 111

Table 2-6. Device Function Field Settings of Bits 3-7 in Word 1 of the FIOB

Device	Function	Function Field Bit Settings Bit 3 4 5 6 7
CARD READER	READ HOLLERITH	0 0 1 0 0
	READ BINARY	0 0 0 0 0
IPARS ADAPTER	START RECEIVE (look for sync)	C 0 0 1 0
	CONTINUE INPUTTING RECEIVED DATA	C 0 1 1 0
	CHECK CRC CHARACTER AND START RECEIVE	C 0 1 0 0
	START TRANSMIT1 - NO CRC TRANSMITTED AT BYTE COUNT ZERO	C 0 0 0 1
	START TRANSMIT2 - CRC TRANSMITTED AT BYTE COUNT ZERO	C 0 1 0 1
	CONTINUE TRANSMITTING DATA 1 - CRC NOT TRANSMITTED AT BYTE COUNT ZERO	C 1 0 0 1
	CONTINUE TRANSMITTING DATA 2 - CRC TRANSMITTED AT BYTE COUNT ZERO	C 0 1 1 1
	TRANSMIT IDLES	C 0 0 1 1
	SEND NEW SYNC PULSE	C 0 0 0 0
TELETYPE/ TERMINET	READ (teletype full duplex mode)	0 0 0 0 0
	WRITE (teletype full duplex mode)	0 0 0 0 1
	WRITE (terminet simplex mode)	0 0 0 0 1
CASSETTE	READ	0 0 0 0 0
	WRITE	0 0 0 0 1
	BACKSPACE	0 0 0 1 0
	REWIND	0 0 0 1 1
	ERASE	0 0 1 0 1
2848	START RECEIVE (look for sync)	C 0 0 1 0
	CONTINUE INPUTTING RECEIVED DATA	C 0 1 1 0
	RECEIVE STOP DATA	C 0 1 1 1
	START TRANSMIT	C 0 0 0 1
	TRANSMIT IDLES	C 0 0 1 1
	TRANSMIT STOP DATA	C 0 1 0 1
	SEND NEW SYNC PULSE	C 0 0 0 0

Table 2-6. Device Function Field Settings of Bits 3-7 in Word 1 of the FIOB (cont)

Device	Function	Function Field Bit Settings Bit 3 4 5 6 7
DISPLAY KEYBOARD	READ (chained)	C 0 0 0 0
DISC	WRITE	0 0 0 0 1
	READ	0 0 0 1 0
	COMPARE DATA	0 0 1 0 0
	SEEK	0 0 1 0 1
	RECALIBRATE	0 0 1 1 0

NOTE

Bit 3 is used to specify chaining of certain I/O commands, where: 0 = no chaining permitted and C = chaining permitted. To specify chaining, a one bit is set in bit 3.

When the programer issues an IOACT request, the FIOB information is accessed by the IOCS monitor, which extracts the I/O request information and enters it into the next entry of the IOCQ Table, described below. When the queued I/O request is to be performed, the monitor extracts the IOCQ entry information and places it in the internally-stored Physical I/O Table (PIOT) for use of the device controller, which performs the I/O action request.

3.4.2 Input/Output Control Queue Table Definition

For each I/O device channel to be used by the program, the programer must set up an IOCQ Table area in which entries for each I/O request can be made. The IOCQ entries are a fixed format and size, as shown in figure 2-4.

The first word of each IOCQ entry is specified via a source statement by the programer, as follows:

Word 0: Link Field (16 bits). The programer specifies the address of the next IOCQ entry in this 16-bit field.

A storage area must be reserved for the remaining nine words. The Logical and Physical Status fields in Word 1 are used by the monitor and the device drivers to report the status of I/O requests, as shown in tables 2-7 and 2-8, respectively. These status fields can be tested by the program to determine the status of each I/O request.

The remaining words are filled from the FIOB by the monitor when an IOACT request is issued by the program, and used by the specified hardware device controller when the I/O action is performed.

Bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Word 0	LINK															
Word 1	Logical Status								PHYSICAL STATUS							
									group				subgroup			
Word 2	MODE			FUNCTION					LOGICAL UNIT NUMBER ID							
Word 3	BUFFER ADDRESS (starting byte)															
Word 4	BYTE COUNT															
Word 5	TRANSLATE TABLE BASE OR DISC ADDRESS															
Word 6	SEARCH TABLE BASE OR DISC ADDRESS															
Word 7	(Spare)															
Word 8	(Spare)															
Word 9	(Spare														LUN extension	

Figure 2-4. Format of Input/Output Control Queue (IOCQ) Entries

Table 2-7. IOCQ Logical Status Codes

Code	Significance
0	Processing completed. This code is the initial value of the logical status field. When the program completes I/O request processing, it should reset the logical status code to 0.
1	I/O pending. This code is set by the IOCS monitor when the I/O request has been queued in the IOCQ table.
2	I/O initiated. This code is set by the device driver when the hardware has started the I/O action.
3	I/O completed.* The physical status field can be checked to see which kind of I/O completion has occurred.

*An exception in this case is that the display keyboard service routine sets the status to 0 on an I/O completed, rather than setting it to 3.

Table 2-8. IOCQ Physical Status Codes

GROUP STATUS		SUBGROUP STATUS	
Code (bits 8-11)	Significance	Code (bits 12-15)	Significance
0	Normal completion	1	Search requested and byte found
		2	Byte count = 0
		3	EOR
		4	EOR with attention
		5	Noncompare
		6	Seek initiated (disc)
1	Illegal operation for this device	0	None
2	Attention (hardware alert condition)	0	None
		1	Hopper check (CR)
		2	Motion check (CR)
		3	CIC tumble table entry made
		4	CIC system reset
		5	End of tape (EOT)
		6	Beginning of tape
		7	Write protect
		8	Data transmission problem
		9	Motor off
		A	Abandoned call
		B	Break
3	(spare)		
4	Error (hardware detected)	0	None
		1	a. Device not operational, or b. Present order chained, next byte count = 0
		2	Data lost
		3	Check character bad
		4	Read check (CR)/ error
		5	Illegal interrupt
		6	Format error (disc)
		7	Punch tape out
		8	Disconnected
		9	Parity error
10-15	(spares)	-	

3.4.3 Special Functions

In addition to the basic I/O services of CLOSE, OPEN, INITialize, and IOACT, the PTS-100 hardware provides two special functions:

- The Search function allows the programer to test for the occurrence of particuar control characters within the I/O character stream, and specify interrupt conditions when these characters appear.
- The Translate function allows the programer to specify input/output code conversion (i.e., to specify that input/output data characters are to be converted to or from the ASCII code used internally by the PTS-100).

These functions are specified in conjunction with the IOACT service request by entering a code in the MODE field of Word 1 in the FIOB (see subsection 3.4.1). They use numerically ordered, programer-defined byte tables containing the control and/or conversion codes. The Search and Translate Table addresses are specified in words 5 and 4, respectively, of the FIOB. The MODE field code specifies whether the hardware is to use a common table or separate Search and Translate Tables. When an IOACT service request is issued, the IOCS monitor accesses the FIOB and moves the Search and Translate Table base addresses to the IOCQ. When the IOCS is ready to start the I/O device action, it moves the base addresses into words 3 and 4 of the PIOT. Thus, by the time the hardware device controller is ready to perform the specified action, the Search and Translate Tables are accessible to the controller. The definition and usage of Search and Translate Tables are presented below.

3.4.3.1 Search Table Definition. The Search function enables the programer to specify interrupt condition settings when particular control characters appear in the data stream flowing through a device controller. The interrupt condition settings are specified by control codes stored in given byte locations within a programer-defined Search Table, whose total length is determined by the length of the I/O data code, as follows:

Code Length	Table Length
8 bit	256 bytes
7 bit	128 bytes
6 bit	64 bytes
etc........	

In the Search Table, the byte location of a given control code must correspond to the following:

search table base address
+ numeric value of the control data character

That is, the base address (the location of byte 0) of the Search Table is offset (i.e., incremented) by the value of the data character passing through the controller to determine the byte location of the corresponding control code. To effect a hardware interrupt condition setting, the left-most bit (MSB) of a control code must be set to one. Figure 2-5 illustrates the Search Table format and control code bit settings.

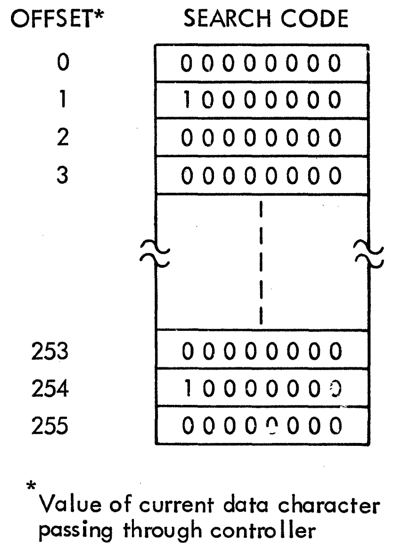


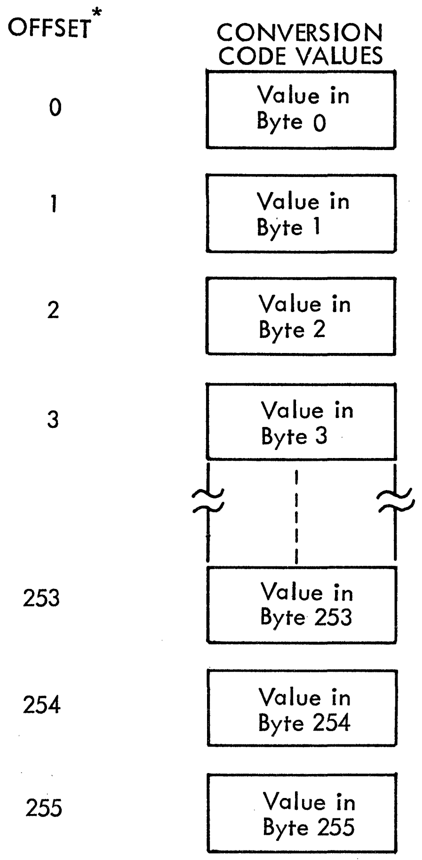
Figure 2-5. Search Table Format for 8-Bit Code

When the Search function is specified in the MODE field of Word 1 in the FIOB, each character flowing through the device controller is used to locate the control code in the Search Table. When the code is located, its MSB is tested. If it is equal to 1, an interrupt condition is set for the device. If the MSB = 0, the next character is selected and the Search function is repeated.

Notice that the byte locations corresponding to offset characters 1_{10} and 254_{10} in figure 2-5 contain MSB's equal to 1; hence, when a data character whose value is equal to 1 or 254 passes through the specified device controller an interrupt condition will be set.

The Search Table may be combined with the Translate Table when the data codes used are 7 bits or less in length. That is, if the actual translate character codes are no more than 7 bits in length, the first bit of the 8-bit field may be used as the control code setting on which to test. For 8-bit code, separate tables must be used for the Search and Translate functions.

3.4.3.2 Translate Table Definition. The Translate function allows the programer to specify conversion of data characters to or from the 7-bit ASCII code used internally in the PTS-100. When a code other than 7-bit ASCII is to be read into the main memory of the PTS-100, the programer must set up a Translate Table containing ASCII characters whose byte locations are equivalent to the value of the input data characters. That is, the base address (the location of byte 0) of the Translate Table is offset (i.e., incremented) by the value of the input character passing through the controller to determine the byte location of the conversion code value that is to replace the input character value. When output data is to be converted from ASCII to another code, a Translate Table must be set up in which the values of the ASCII characters correspond to the byte locations in which the associated values of the output conversion codes are stored. The format of a Translate Table for 8-bit code conversion is illustrated in figure 2-6.



*Value of current data character passing through controller

Figure 2-6. Translate Table Format for 8-Bit Code Conversion

When the Translate function is specified in the MODE field of Word 1 in the FIOB, the device controller matches each data character in the I/O stream with the corresponding position in the Translate Table and replaces its value with the conversion code value stored in the byte location.

If the Translate Table contains conversion codes of 7 bits or less in length, it may be used simultaneously as a Search Table. That is, the MSB of each byte location is available for use as an interrupt condition indicator, since conversion code values are right-justified in the byte fields. If 8-bit code is to be converted, a separate Search Table must be defined for Search function use.

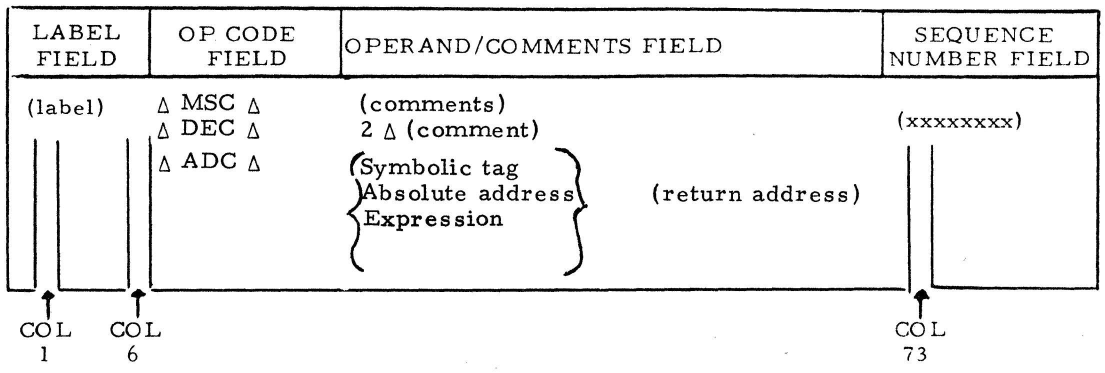
The Translate Table length is determined by the length of the code to be converted, as follows:

Code Length	Table Length
8 bit	256 bytes
7 bit	128 bytes
6 bit	64 bytes
etc.	

3.4.4 Monitor Service Calls

When the application programer wishes to initiate any of the basic I/O device services for his program he must issue a monitor service call statement, followed by the necessary number of arguments to effect the desired device service. When an MSC statement is encountered in a program, processing control is transferred to the IOCS monitor, which performs the specified service, as described in the following subsections.

3.4.4.1 <u>Device Initialization Service.</u> The initialization service requests the monitor to reset all I/O devices on the system. This service should be requested at the beginning of a program or before an interrupted program is restarted. Three statements are required in the source program to effect device initialization, as shown in the diagram below.



The MSC statement transfers control to the IOCS monitor. The DEC statement with the decimal constant 2 in the operand field specifies that the monitor is to perform the initialization service for all devices on the system. The ADC statement operand specifies the object program location to which the monitor is to return control when the service is completed.

When the monitor receives control, it initializes all I/O devices on the system as follows:

- Issues a STOP I/O command to every device, thus terminating any I/O operation in process.
- Sets the PCB's logical status bits to the initial condition for all devices.
- Enables interrupts.

3.4.4.2 <u>Device Open Service.</u> The device OPEN service requests the IOCS monitor to initialize a specific device and its associated monitor software routines so that subsequent program IOACT requests may be serviced. The statements necessary to effect an OPEN device service are shown below.

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS FIELD	SEQUENCE NUMBER FIELD
(label)	Δ MSC Δ	(comment)	(xxxxxxxx)
	Δ DEC Δ	6 Δ (comment)	
	Δ ADC Δ	Symbolic tag / Absolute address / Expression (return address)	
	Δ ADC Δ	Symbolic tag	
Symb. Tag	Δ HEX Δ	LUN ID	
	Δ ADC Δ	IOCQ Table Address	
	Δ RESV, 00Δ	2 bytes	

COL 1 COL 6 COL 73

The DEC statement with the decimal constant 6 in the operand field specifies that the monitor is to perform the OPEN service for the device specified by the LUN ID in the HEX statement, to which the monitor is directed in the preceding ADC statement. The last ADC statement specifies the starting address of the programer-defined IOCQ table for the device. The RESV statement reserves a two-byte field into which the IOCS monitor can store a code indicating that an error occurred during the attempt to service the OPEN request.

When the monitor receives control to service an OPEN request, it performs any initialization for the device, and its associated software routines. The monitor resets all relevant status fields within its internal tables and the IOCQ so that obsolete I/O requests will be eliminated.

If the specified device is a full duplex communications device, separate OPEN service requests must be issued to initialize transmit and receive actions. That is, each action is treated as though it were to be performed on a separate device, each with its unique LUN ID and associated IOCQ table entry.

3.4.4.3 I/O Action Service. The IOACT service requests the monitor to initiate specific I/O actions of which the device is capable, such as read, write, rewind, backspace, etc. Associated with each IOACT service request is a programer-defined FIOB Table (see subsection 3.4.1) that specifies the parameters to be used by the monitor in servicing the request. The source statements necessary to effect an IOACT service are shown below.

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS FIELD	SEQUENCE NUMBER FIELD
(label)	Δ MSC Δ	(comment)	(xxxxxxxx)
	Δ DEC Δ	7 Δ (comment)	
	Δ ADC Δ	Symbolic tag / Absolute address / Expression (return address)	
	Δ ADC Δ	FIOB address	

COL 1 COL 6 COL 73

The DEC statement with the decimal constant 7 in the operand field specifies that the monitor is to perform an IOACT service on the device whose associated FIOB is referenced in the operand field of the last ADC statement. The first ADC statement specifies the address in the object program to which control is to return when the IOACT request has been serviced.

When the monitor receives the IOACT service request, it accesses the specified FIOB, enters the I/O request into the IOCQ, and changes the IOCQ's logical status field setting from "Initial" (= 0) to "I/O pending" (= 1). When the device controller actually starts the specified action, the logical status field setting is changed to "I/O Initiated" (= 2). When the I/O request has been serviced the logical status setting is changed to "I/O Completed" (= 3).

Once the IOACT request has been serviced, the programer should ensure that the logical status field is cleared to zero (initial setting) to signal the monitor that the IOCQ entry is again available for use.

If an IOACT request cannot be serviced by the monitor, an error code is passed to the program via the error code field in the right-most byte in Word 0 of the FIOB. No other FIOB fields are altered by the monitor. After control is returned to the program, the error code field should be tested.

3.4.4.4. <u>Device Close Service</u>. The device CLOSE service requests the monitor to close down a device at the end of a job or to facilitate an error recovery. The statements necessary to effect a device CLOSE service are shown below.

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS FIELD		SEQUENCE NUMBER FIELD
(label)	Δ MSC Δ	(comment)		(xxxxxxxx)
	Δ DEC Δ	1 Δ (comment)		
	Δ ADC Δ	Symbolic tag Absolute Address Expression	(return address)	
	Δ ADC Δ	Symbolic tag		
Symb. Tag	Δ HEX Δ	LUN ID		
	Δ RESV, 00 Δ	2	(error code field)	

COL 1 COL 6 COL 73

The DEC statement with the constant 1 in the operand field specifies that the monitor is to perform the CLOSE service for the device specified by the LUN ID in the HEX statement, to which the monitor is directed by the preceding ADC statement. The RESV statement reserves a two-byte field into which the monitor can store a code to indicate that an error occurred during the attempt to CLOSE the device.

When the monitor receives control to service a CLOSE request, it performs all steps necessary to terminate operation on the specified device. The monitor sets all status fields in its internal tables to indicate a device "closed" condition. It does not, however, clear any of the fields in the IOCQ, since this table may be examined for error analysis by the program.

When a device has been closed in this manner, it must be initialized by an OPEN service request before subsequent IOACT service requests can be issued to it.

3.4.4.5 System Exit Service. The EXIT service requests the monitor to log the end-of-job and loop at location 0 until some manual intervention specifies a new processing step, such as the use of the Debug program, program loading, etc. The EXIT service is effected by issuing an MSC statement followed by a DEC statement with a decimal constant 0 in the operand field, as shown below.

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS FIELD	SEQUENCE NUMBER FIELD
(label)	Δ MSC Δ	(comment)	(xxxxxxxx)
	Δ DEC Δ	0 Δ (comment)	

COL 1 COL 6 COL 73

3.4.4.6 Watchdog Timer Service. The Watchdog Timer service call controls or interrogates the optional Watchdog Timer if the feature board WATCHDOG TIMER switch is set to ENABLE. The source statements necessary to effect a Watchdog Timer service call are as follows:

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS FIELD		SEQUENCE NUMBER FIELD
(label)	Δ MSC Δ	(comment)		(xxxxxxxx)
	Δ DEC Δ	3		
	Δ ADC Δ	{ Symbolic Tag / Absolute Address / Expression }	(return address)	
	Δ ADC Δ	Symbolic Tag		
Symb. Tag	Δ DEC Δ	Request Code		
	Δ HEX Δ	0	(power status field)	
	Δ RESV,00 Δ	2	(error field)	

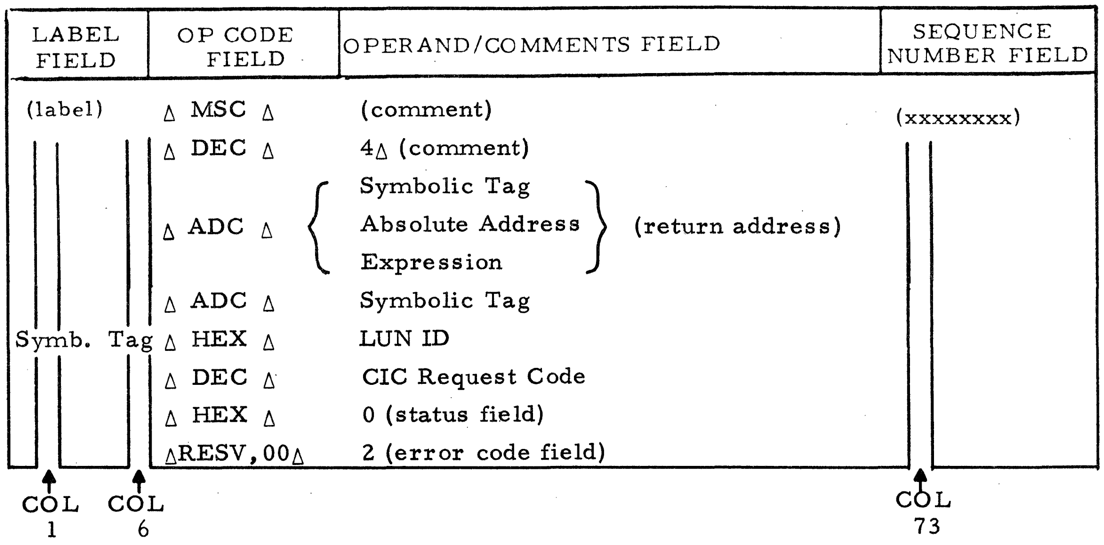
COL 1 COL 6 COL 73

The DEC statement with the constant 3 in the operand field specifies that the monitor is to control or interrogate the Watchdog Timer. The second DEC statement specifies the nature of the request as follows:

WDT Request Code	Meaning
1	Reset Watchdog Timer — must be given at least once every 34 seconds or automatic program restart will occur.
2	Start Watchdog Timer — turns Watchdog Timer on under program control and automatically initializes counter to zero.
3	Stop Watchdog Timer — turns off Watchdog Timer under program control. This should be reserved for special cases since the watchdog capability is disabled when turned off.
4	Read power status — power status is interrogated and the reading stored in the power status field.

If any other request code is specified, an error code of 1 will be placed in the error field.

3.4.4.7 Channel Interface Controller (CIC) Service. The CIC service call tests or resets busy or off-line bits of devices attached to the Channel Interface Controller. The necessary source program statements are as follows:



LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS FIELD	SEQUENCE NUMBER FIELD
(label)	Δ MSC Δ	(comment)	(xxxxxxxx)
	Δ DEC Δ	4Δ (comment)	
	Δ ADC Δ	{ Symbolic Tag / Absolute Address / Expression } (return address)	
	Δ ADC Δ	Symbolic Tag	
Symb. Tag	Δ HEX Δ	LUN ID	
	Δ DEC Δ	CIC Request Code	
	Δ HEX Δ	0 (status field)	
	ΔRESV,00Δ	2 (error code field)	

COL 1
COL 6
COL 73

The DEC statement with the constant 4 in the operand field specifies that the monitor is to test or reset the busy or off-line bits (according to the CIC request code in the second DEC statement) of the device specified by the LUN ID in the first HEX statement, to which the monitor is directed by the preceding ADC statement. The CIC request code must be one of the following:

CIC Code	Meaning
1	Test and set busy bit — the status of the device busy bit (1 = busy, 0 = not busy) will be placed in bit 0 of the status field and the remaining bits are undefined. Then the device busy bit will be set.
2	Reset busy bit — the device busy bit will be reset.
3	Test and set off-line bit — the status of the device off-line bit (1 = off-line, 0 = on line) will be placed in bit 0 of the status field and the remaining bits are undefined. Then the device off-line bit will be set.
4	Reset off-line bit — the device off-line bit will be reset.

When the routine is executed, the specified action is taken unless an unassigned LUN ID or CIC code was specified. If either error occurs, IOCS will return an error code to the error field and take no other action. The error codes are as follows:

CIC Error Code	Meaning
0	No error
1	Illegal request code
2	No such LUN in this system

3.4.4.8 Device Sensing Service. The device sensing service call requests the IOCS monitor to sense the status of a specific device. The source statements necessary to effect a device sensing service call are as follows:

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS FIELD	SEQUENCE NUMBER FIELD
(label)	Δ MSC Δ	(comment)	(xxxxxxxx)
	Δ DEC Δ	5Δ(comment)	
	Δ ADC Δ	$\left\{\begin{array}{l}\text{Symbolic tag}\\ \text{Absolute Address}\\ \text{Expression}\end{array}\right\}$ (return address)	
	Δ ADC Δ	Symbolic tag	
Symb. Tag	Δ HEX Δ	LUN ID	
	Δ HEX Δ	0 (status field)	
	Δ RESV, 00 Δ	2 (error code field)	

↑ COL 1 ↑ COL 6 ↑ COL 73

The DEC statement with the constant 5 in the operand field specifies that the monitor is to sense the status of the device specified by the LUN ID in the first HEX statement, to which the monitor is directed by the preceding ADC statement. When the routine is executed, the device's hardware status is returned to the status field, unless an unassigned LUN was specified, in which case an error code of 2 is returned to the error field. (If no error occurred the error field setting remains all zeros.)

3.4.4.9 Reconfiguration Service. The reconfiguration service call changes the addresses of one or more peripheral devices which may include serial printers, card readers, modems, magnetic tape cassettes, and display keyboards. All peripherals of the same type are handled as one group. Therefore, the programer should code one reconfiguration call per group. Reconfiguration is done only at device initialization time.

The source statements necessary to effect a reconfiguration service call are as follows:

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS FIELD	SEQUENCE NUMBER FIELD
(label)	Δ MSC Δ	(comment)	(xxxxxxxx)
	Δ DEC Δ	(comment)	
	Δ ADC Δ	Symbolic tag / Absolute Address / Expression (return address)	
	Δ ADC Δ	Symbolic tag	
Symb. tag	Δ HEX Δ	0 (error field)	
	Δ HEX Δ	LUN ID (first LUN of group)	
	Δ HEX Δ	Interrupt Level	
	Δ HEX Δ	List of Drivers	
	Δ EXREF Δ	List of Drivers	
	Δ HEX Δ	First device address*	
	Δ HEX Δ	Second device address*	
	.		
	.		
	.		
	Δ HEX Δ	Last device address*	
	Δ HEX Δ	FFFF (sentinel defining end of group)	

COL 1 COL 6 COL 73

*The devices will be placed into the PCB's in the sequence given in the parameter list.

The following drivers support reconfiguration:

#IDPM$_n$	for multiple serial printers
#IDCM$_n$	for multiple card readers
#IDMM$_n$	for multiple modems
#IDSR$_n$	for special display keyboard receive
#IDCA$_n$	for cassettes

Error codes are as follows:

0 = no error
2 = no such LUN in this system
6 = PCB overflow

3.5 Disc Logical Input/Output

The use of a disc storage device with the PTS-100 requires a specialized IOCS for disc in addition to the usual system IOCS monitor. The logical IOCS for disc has two interfaces: one with the user and one with the disc. The interface with the user is through macro calls, described in subsections 3.5.3 and 3.5.4. The interface with the disc is handled through the physical IOCS monitor.

Before a disc is used with a PTS-100 system, it must be formated with the Disc Volume Preparation program, and all files that are to be accessed must be allocated with the Disc Allocator program (see Parts 1 and 3 of this manual). Then the disc is ready to be written and read by a program incorporating the disc logical I/O macros. Any program using the disc logical I/O must contain four types of macros: the file description macros, main processing macro, action macros, and status macros.

3.5.1 User File Area

The user file area consists of the parts of the disc not occupied by the Volume Label or Volume Directory. The user file area is divided into one or more files, as indicated by the Volume Directory. Files are established, initialized, altered in extent, or deleted by the Disc Allocator utility program. A file may be one of three types: sequential, random with keys, or random without keys. All files start and end on cylinder boundaries.

3.5.1.1 Sequential Files. A sequential file consists of a series of records written and read in physical sequence. The records may be fixed or variable in length. Records are packed densely into the allocated file area, in such a way that no disc space is wasted. Thus, a record may span two or more sectors, or several records may occupy one sector.

The last record in a sequential file, the end-of-file record, is marked by a special configuration in its first word: FEDC$_{16}$. It is automatically written at the time the Close macro is executed. This record marks the end of the written portion of the file, not the end of the allocated file area, which is indicated by an address in the Volume Directory.

In files containing variable-length records, the first word of each record is the length word. The length word gives the number of bytes in the record, including the length word itself.

3.5.1.2 Random Files. A random file consists of a series of records that may be accessed either sequentially or non-sequentially. There are two types of random files: those whose records contain keys (K type) and those whose records do not contain keys (N type).

Random file records must be unblocked, and fixed in length. For K type files, the first word of every record is a banner word. (N type files do not use banner words.) The banner word is included to permit the read and write routine to determine whether a particular record position has been written, or whether it is still in the original state to which it was initialized by the Disc Allocator utility program. A value of 0000_{16} indicates that the record is unused, and a value of 0001_{16} indicates that the record is used (i.e. that it has been written).

Records may be of any length up to a track, but records always start on sector boundaries. Thus, if a record is not a multiple of the sector length (160 words), there will be unused space following each record. Records may not extend from track to track. If the track is not evenly divisible into records, there will be unused space at the end of a track.

Random files may be written only in direct access fashion, but they may be read either sequentially or directly. The macros Open, Close, Read, Write, Delete, Test, and Wait may be applied to them. Get and Put are not applicable to random files.

In files containing keys (K type files), each key value must be unique; it must be different from all other key values in the file.

3.5.2 File Description Macro

The File Description macro is called once for each file that is to be accessed. It establishes a File Control Block, which is a work area about 100 bytes long, in which all information about the file and the current state of its processing is maintained by logical I/O. The call to the file description macro has the following format:

```
        $   FCBD   a,b,c,d,e,f,g,h,i,j
```

where

a = label of File Control Block; this label is referred to in all the action macros to identify which file is to be accessed.

b = device number, a number from 0-7, identifying the disc drive on which the file resides.

c = type of buffering
  S = single buffering
  D = double buffering (may be specified only for sequencial files).

d = first buffer address.

e = second buffer address; enter 0 if single buffered.

f = address of file name; the address of a 10-byte field containing the name that was assigned to the file through the Disc Allocator program.

g = address of error word; the address of a one word field which will be set to an identifying number if an error occurs in processing this file.

h = relative position of key in record; for random files of type K, the first byte, relative to byte 0, of a field in the record that is to be used as a key in searching for a particular record.

i = length of keys; number of bytes in key field.

j = length of buffer, in sectors; the number of 320-byte sectors that can be read or written at one time, based on the length of the buffer(s) provided.

3.5.3 Main Processing Macro

There is one Main Processing macro, which is called into a program only once. It contains the coding necessary to carry out actions on the disc files requested by the action macros, which are simply branches to certain routines in the main processing macro. The call to the Main Processing macro has the following format:

```
$    LIOCSD    a,b,c,d,e,f,g,h
```

where

a = logical unit number of disc; this must correspond with the LUN ID assigned through the physical IOCS monitor.

b = SE if sequential files are used; omitted if not.

c = GT if GETD macro is used; omitted if not.

d = PT if PUTD macro is used; omitted if not.

e = RN if random files are used; omitted if not.

f = RD if READD is used; omitted if not.

g = WR if WRITED is used; omitted if not.

h = DE if DELD is used; omitted if not.

The above parameters cause the Main Processing macro to be tailored to the needs of a particular program, omitting any portions that are not to be used. For example, to generate a macro for random reads and writes only, under LUN 3, include the following call:

```
$    LIOCSD    3,,,,RN,RD,WR
```

3.5.4 Action Macros

Action macros are used as many times as necessary in a program to process the disc files. Each macro call results in a branch to the Main Processing macro, followed by a series of parameters. The first parameter of every action macro is the label of the File Control Block, which identifies the file to be accessed. This must be the same as the first parameter of the file description macro for that file.

All action macros return to the next sequential instruction after the calling sequence.

There are seven action macros: Open, Close, Get, Put, Read, Write, and Delete. Open and Close apply to all file types, Get and Put apply only to sequential files. Read, Write, and Delete apply only to random files.

3.5.4.1 Open Macro. The Open macro call has the following format:

```
$    OPEND    a,b,c
```

where

a = label of file control block

b = type of open
  I = open for input
  O = open for output

c = error exit. Location to branch to if uncorrectable error occurs on open.

The Open macro must be issued before any accessing can be done on a file. It searches for the file in the Volume Directory (established by the File Allocator program), and places in the File Control Block various pieces of information describing the file. If the file is not open, it will be opened at this time.

3.5.4.2 Close Macro. The Close macro call has the following format:

```
$    CLOSED    a,b
```

where

a = label of File Control Block

b = error exit

The Close macro is issued when all file accessing is completed. For output sequential files, it causes the last record to be written on the disc, followed by an end of file indicator. It also closes the Logical Unit if there are no other files open.

3.5.4.3 Get Macro. The Get macro call has the following format:

```
$    GETD    a,b,c,d
```

where

a = label of File Control Block

b = address of work area into which next record is to be moved

c = end-file exit. Location to branch to if end-file is found

d = error exit

The Get macro, applicable only to sequential files, causes the next logical record to be moved from the buffer to the specified work area. Buffer switching and disc reads are executed when necessary.

3.5.4.4 Put Macro. The Put macro call has the following format:

```
$    PUTD    a,b,c
```

where

a = label of File Control Block

b = address of work area from which record is to be taken

c = error exit

The Put macro, applicable only to sequential files, causes the logical record in the designated work area to be moved to an output buffer. Buffer switching and disc writes are executed when necessary.

3.5.4.5 Read Macro. The Read macro call has the following format:

```
$    READD    a,b,c,d,e,f
```

where

a = label of File Control Block

b = address of work area into which record is to be moved

c = end-file exit

d = error exit

e = type of access
  D = direct
  S = sequential

f = address of parameter list (zero if no parameter list)

The parameter list is needed only for direct type address. If present, it contains the following two fields:

address of key field in program
relative address

The Read macro is applicable only to random files (type K or N organization). Its function is to input one record from the disc and place it in the indicated work area. It is assumed that the file is unblocked, or that deblocking is to be handled by the user. It is also assumed that records are of fixed length.

For N type files (no keys), the access type given in the call is first consulted. If the access type is S (sequential), the system reads the record following the one previously read, and moves it into the work area. If the last record was at the end of the file area, the end-of-file exit is taken. If the access type is D (direct), the relative address pointed to in the call is interpreted as a record address, relative to the first record in the file. The indicated record is read and moved into the work area. For example, if relative address is 50, then logical record number 50 is read. The first record in the file is counted as number 0.

The Read macro operates somewhat differently for K type files (files with keys). If the access type is S (sequential), the system reads the record following the one previously read, and moves it into the work area. If a sequential Read is executed after a series of direct Reads, the first sequential Read obtains the same record as the last direct Read. If a sequential Read is executed without any previous direct Read's, the first sequential Read obtains the first record in the file.

If the access type is D (and file type is K) the relative address and key fields are used. A key is simply a field, in some fixed position, in the record, which is used in looking for a particular record. The logical I/O searches for a match between this key field in the record and a key value pointed to in the call. The key can start anywhere in a record, and be of any (even) byte length. However, its length and position are constant throughout any one given file.

The relative address is interpreted as a track address, relative to the first track in the file. The first track is relative track 0. For example, if relative address is 10, the logical I/O searches for a record with the given key value, on track number 10 of the file. If it is not found on that track, the logical I/O continues searching through the overflow area, if there is one. When a record is found whose key matches the given value, it is moved to the work area. If no such record is found either on the home track or in the overflow area, the error exit is taken.

3.5.4.6 Write Macro. The Write macro has the following format:

```
$    WRITED    a,b,c,d
```

where

a = label of File Control Block

b = address of work area

c = error exit

d = address of parameter list

The parameter list contains only the relative address.

The Write macro is applicable only to random files (type K or N organization). Its function is to output one record to the disc, taking it from the designated work area. It operates differently for type K and N files.

For type N files (no keys), the relative address is interpreted as a relative record address. The given record is written at that relative record position in the file, regardless of what was written there previously. For example, if relative address is 25, the record is written as record 25 in the file, destroying the previous record 25 if any record had already been written there.

The Write action for type K files differs in two ways. First, the relative address is interpreted as a relative track address. Second a previously written record will not be destroyed. This is possible because type K files have banner words. A banner word of 1 indicates that the record has been written by logical I/O; a banner word of 0 indicates that it is available for writing. The logical I/O searches down the designated track for an available position; if one is found, it writes the record there. If no space is available on the home track, the logical I/O then looks for space in the overflow area, if any. If no space is available there either, the error exit is taken.

3.5.4.7 Delete Macro. The Delete macro has the following format:

```
        $     DELD     a,b,c
```

where

a = label of File Control Block

b = error exit

c = parameter list address

The parameter list contains:

address of key field
relative address

The Delete macro is applicable only to type K random files. Its function is to delete a particular record from the file, making its space available for rewriting. It does so by changing the banner from 1 to 0. The record to be deleted is located in the same way as described under the Read macro for file type K and access type D (direct).

3.5.5 Status Macros

The two status macros, Test and Wait, do not result in any file accessing. One of the two must be used after each of the action macros to ensure that one action has been completed before the next one is requested. The Test or Wait macro need not follow the action macro immediately; it must, however, be issued before another call can be issued. This applies to all the action calls, including Open and Close.

3.5.5.1 Wait Macro. The Wait macro call has the following format:

```
        $     WAITD     a
```

where a is the label of the File Control Block.

The Wait macro assures completion of the last action on the designated file. Control remains in the logical I/O until the last action has been completed. Only then will an exit take place to the next sequential instruction.

3.5.5.2 Test Macro. The Test macro call has the following format:

```
        $     TESTD     a,b
```

where

a = label of File Control Block

b = address of indicator word

The test macro provides an alternate means of checking for the completion of an action macro. The logical I/O sets the designated indicator word to 0 if action is incomplete, or to 1 if the action is complete. The indicator word can then be tested by the main program to decide whether another action macro can be executed.

3.5.6 Error Indicators

Whenever an error exit occurs, the error word (item g, subsection 3.5.2) designated in the file description macro is set as follows:

Error Code	Meaning
0001	Cannot open Logical Unit specified.
0002	File name not found in Volume Directory.
0003	Previous action not completed before call for new action.
0004	File not properly opened for requested action.
0005	Attempt to write beyond end of file area.
0006	Requested key not found in random file.
0007	Random access file write overflow; requested track and overflow area, if any, are full.
0008	Incorrect file organization for requested action.
0009	Attempt to access outside of random file; relative address too large.
0010	Invalid operation.
0011	Byte count too large.
0041	Device not operational.
0043	CRC/rate error.
0046	Format error.

Section 4. MACRO ROUTINES

The PTS-100 programer may develop sets of generalized statements that may be used to create specialized sets of statements according to predefined limits and formats. Such generalized statements sets are called macro routines, which are assembly language program segments defined to perform processing for any number of other program segments into which the routines can be incorporated at assembly time. That is, a macro routine is a set of Assembler source statements that may be "called" by other program segments.

For purposes of discussion, macro routines have been classified herein as basic macro routines and extended macro routines. The structure and use of macro routines is described in detail in this section.

4.1 Basic Macro Routine Structure

Basic macro routine structure is as follows:

Statement 1: This statement must identify the program segment as a macro routine as follows:

- The question mark (?) character must appear in column 1 of the coding form.
- The unique name of the macro routine, consisting of from 1 to 8 characters, must begin in column 2. The routine name may be composed of any characters in the PTS-100 character set (see Appendix A).

The format of the first statement in the macro routine then, is:

? macronam

beginning in column 1.

Statement 2 through Statement n: The body of the macro routine begins with statement 2 and ranges through statement n. Any source language statement may appear in the body of the macro routine. In addition, these statements contain dummy arguments in the form of decimal numbers, from 1 to 99, enclosed in parentheses. The parentheses identify dummy arguments to the Assembler; the numeric value within a given set of parentheses dictates the sequential order in which an actual argument must appear in the argument list passed from the calling programs to the macro routines.

Statement n + 1: The last statement in the macro routine must contain END in the op code field. This statement marks the physical end of the macro routine for Phase 1 of the Assembler, as described in Section 5 following.

To illustrate macro routine writing, assume that a number of independent programs will require a card reading operation such as the following:

1. Read a card.
2. Test for last card and branch to a specified point if present, or branch to an IOACT service to read another card if the last card is not present.
3. Branch when a successful card read operation has been performed.

Figure 2-7 presents a generalized macro routine, named READCARD, to accomplish the desired card reading operation. In the sample routine, the operand fields of statements 2, 5, 8 and 9 indicate parenthesized dummy arguments for which actual arguments must be supplied when the macro routine is called by another

Statement Number	LABEL FIELD	OP CODE FIELD	OPERAND/ COMMENTS
1	?READCARD		
2		LDW, N	(1)+4 Get contents of first character in card.
3		CNE	EF Compare for last card.
4		BCB	RD Not last card. Go To read.
5		JMP	(2) Specifies dummy argument for branch address if last card is read during IOCS request servicing.
6	RD	MSC	Monitor service call.
7		DEC	7 Specifies that monitor is to service a device IOACT request.
8		ADC	(3) Specifies dummy argument for address to which control is to branch when read request has been serviced.
9		ADC	(1) Specifies dummy argument for FIOB address to be used by IOCS monitor.
10	EF	HEX	4546 Constant to be used for last card compare.
11		END	

Figure 2-7. Sample Macro Routine

program. The actual arguments will subsequently be inserted in place of the parenthesized dummy arguments by the Assembler's Phase 1 when it specializes the macro routine for incorporation in a given calling program.

Once a macro routine has been coded, it must be stored on the appropriate macro library file* to make it accessible to the Assembler for incorporation in any programs that call it.

4.2 Calling Macro Routines

Once a macro routine has been placed on the library file, any other source programs, including other macro routines, may call it via statements in the format shown below.

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS
$	MACRONAM	Argument $_1$, Argument $_2$, . . , Argument $_n$

As shown above, column one of the source program call statement must contain a dollar sign ($) to inform the Assembler that a macro routine is being called. The assigned macro routine name must appear as the call statement op code. The operand field of the call statement must specify the actual argument(s) to be passed to the generalized macro routine. The actual

*If the macro routine is an IOCS monitor routine, it must be stored on the System Macro Library file. If it is a user application macro routine, it should appear on the User Macro Library file.

arguments may be any valid operands, labels, or op codes permitted in the respective fields of the affected statements.

If two or more arguments are specified, they must be separated by commas and the entire list of arguments must be terminated by a blank character. Because of the way in which actual arguments are associated with dummy arguments they are to replace, the order of appearance of actual arguments in the operand field is critical. That is, when the Assembler encounters a call statement in a source program, it reads the argument list in the operand field and constructs a table in which the actual arguments are inserted in the sequential order in which they appear in the list. The Assembler then locates the called macro routine in the input library file and copies each of the routine's source statements into an intermediate file, replacing all dummy arguments in the routine with corresponding actual arguments from the Assembler-generated table. That is, the first actual argument replaces all occurrences of dummy argument (1), the second actual argument replaces all occurrences of dummy argument (2), etc. Hence, if the actual arguments are specified in improper order, they will be erroneously matched to dummy arguments and the specialized routine will produce unreliable results at execution time. At the programer's discretion, however, actual arguments may be omitted from the call statement list. Each omission must, however, be indicated by a comma in the omitted argument's position in the list.

If an argument list is too long to appear in the operand field of the call statement, it may be continued in successive statements by writing a slash (/) character in column 1, and the continued list in the operand field.

To illustrate the manner in which actual arguments replace dummy arguments, assume that the READCARD macro routine shown in figure 2-7 is called by a program containing the following call statement.

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS
$	READCARD	FIOB, TOTAL, PRINT

The assembler will replace dummy arguments (1), (2), and (3) in the generalized macro routine with the respective actual arguments FIOB, TOTAL, and PRINT to produce the specialized routine shown in figure 2-8. This routine will follow the processed call statement in the calling program.

The Assembler treats actual arguments as character strings; hence, they need not be syntactic units. For example, an actual argument value may be inserted as a character in a symbolic tag. That is, a generalized macro routine statement may contain a dummy argument such as

```
JMP          (7)TAG
```

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS
	LDW, N	FIOB+4
	CNE	EF
	BCB	RD
	JMP	TOTAL
RD	MSC	
	DEC	7
	ADC	PRINT
	ADC	FIOB
EF	HEX	4546

Figure 2-8. Specialized Macro Routine

and the seventh actual argument in the call statement list may be the character N, which would cause the JMP statement to be specialized as

```
JMP                 NTAG
```

Field content of generalized routine statements and actual arguments transmitted in call statement lists may be written in the formats permissible in source statements of a given type.

4.3 Extended Macro Routine Structure

Extended macro routines may be written to generate flexible specialized routines, depending on the needs of calling programs. That is, statements of generalized macro routines may specify the following:

- Insertion of statement labels to facilitate linkage between and within object program segments, described in subsection 4.3.1
- Conditional inclusion or deletion of generalized macro statements depending upon the presence or absence of actual arguments in the calling program's argument list as described in subsection 4.3.2.
- Deletion of generalized macro statements depending upon equality testing of actual argument values against predefined values within the macro routine as described in subsection 4.3.3.

4.3.1 Statement Label Insertion

The macro routine may contain dummy arguments in statement label fields to enable proper linkage between the calling program and the macro routine, and/or to facilitate transfers of control between coding sections within the specialized macro routine generated for insertion in the calling program. When dummy arguments have been defined in label fields, the calling program must transmit an appropriate label to the routine via its argument list.

To illustrate statement label insertion, a generalized macro routine to create an FIOB is shown in figure 2-9.

Assume that a calling program contains the following statement:

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS
$ /	FIOBMAC	RFIOB, LUNDAT, BUFAD, BCT, TTBAD, STBAD

When the specialized routine is created by the Assembler, dummy argument (1) in statement 1 will be replaced with the actual argument RFIOB. Actual arguments LUNDAT, BUFAD, BCT, TTBAD, and STBAD will replace dummy arguments (2) through (6). Thus, once the generalized macro routine is coded and filed, it can be called at any time an FIOB is needed in a source program by merely writing a call statement containing the appropriate arguments, the first of which is a label specifying the starting address of a given FIOB.

Statement Number	LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS
1	?FIOBMAC		
2	(1)	HEX	0 Spare/error code
3		HEX	(2) Dummy argument for mode, function and LUN
4		ADC	(3) Dummy argument for input buffer address
5		DEC	(4) Dummy argument for byte count
6		ADC	(5) Dummy argument for translate table base
7		ADC	(6) Dummy argument for search table base
8		RESV,0	6 6-byte spare area
9		END	

Figure 2-9. Generalized Macro Routine to Create an FIOB

4.3.2 Conditional Inclusion and Deletion of Macro Routine Statements

The PTS-100 Assembler can be directed to include or delete statements in the generalized macro routines, depending on the presence, absence, or value of actual arguments transmitted via the call statement list. That is, the programer may specify that given generalized statements are to be treated in one of the following ways:

- Included in the specialized routine only if the corresponding actual arguments appear in the call statements list.

- Omitted from the specialized routine if the corresponding actual argument is equal to, not equal to, greater than, or less than a given value.

These actions are communicated to the Assembler via the following notations in the form of dummy arguments:

- (nC) or (nN), either of which specifies that the statement is to be included in the specialized routine only if the actual argument corresponding to the nth dummy argument appears in the call statement list. The difference in the use of the dummy arguments C and N is that when the nth argument is present, it replaces the nC dummy argument in the specialized routine, whereas the nth argument does not replace the nN dummy argument in the specialized routine. For example, in the statement

  JMP (3C)

  the C in the dummy argument informs the Assembler that the JMP statement is to be inserted in the specialized routine only if an actual argument appears in the third position of the call statement list, and that if the actual argument is present, it is to be inserted in the operand field of the JMP statement. However, in the statement

  STW (3) (4N)

  the (4N) dummy argument informs the Assembler that the Store Word statement is to be included in the specialized routine only when an actual argument appears in the fourth position of the call statement list. It does not specify that the fourth argument is to be inserted in the place of the (4N) dummy argument in the specialized routine.

- (nY), which specifies that the statement is to be omitted from the specialized routine if the actual argument corresponding to the nth dummy argument does appear in the call statement argument list. For example, in the statement:

  ```
  LDI  AC,0  (4Y)
  ```

  the Y in the dummy argument informs the Assembler that the Load Immediate statement is to be omitted from the specialized routine when an actual argument appears in the fourth position of the call statement list.

NOTE

The (nY) and (nN) form of dummy arguments may be combined to specify omission of statements depending on the presence of one actual argument or the absence of another actual argument in the call list.

- (n, E or N, vv), which specifies that the statement is to be omitted from the speccialized routine if the nth actual argument is equal (E) or not equal (N) to the value of vv, a value specified as two characters. For example, in the statement:

  ```
  ADC  (6)    (5,E,01)
  ```

  the dummy argument (5, E, 01) specifies that the ADC statement is to be omitted from the specialized routine if the fifth actual argument's value is equal to 01. In the statement:

  ```
  LDW  (15)  (12,N,AA)
  ```

  the dummy argument (12, N, AA) specifies that the LDW statement is to be omitted if the value transmitted for the twelfth actual argument is not equal to AA.

- (n, G or L, vv), which specifies that the statement is to be omitted from the specialized routine if the nth actual argument is greater than (G) or less than (L) the value of vv, a value specified as two characters. For example, in the statement:

  ```
  ADC (6)   (5,G,01)
  ```

  the dummy argument (5, G, 01) specifies that the ADC statement is to be omitted from the specialized routine if the value of the fifth actual argument is greater than 01. In the statement:

  ```
  LDW  (15)   (12,L,05)
  ```

  the dummy argument (12, L, 05) specifies that the LDW statement is to be omitted if the value transmitted for the twelth actual argument is less than 05.

In all cases, omission of an actual argument from a call statement list is affected by entering a comma in the corresponding position in the list as illustrated below, where the third, fifth, sixth, and seventh actual arguments have been omitted. Trailing commas are unnecessary.

```
$ MACRONAM  Argl,Arg2,,Arg4,,,,Arg8
```

To illustrate the flexibility provided by these optional directives to the Assembler, assume that a generalized macro routine named SERREQ, shown in figure 2-10, has been coded to create specialized routines to request services from the IOCS monitor. The use of the SERREQ macro routine to generate specialized routines to request the INITialization, OPEN, IOACT, CLOSE, and EXIT services is described in the following paragraphs. In all cases, arguments 1 and 2 must appear in the SERREQ call statement list. That is, statements 1 and 2 must appear in any of the specialized routines. These two statements are the only ones required for the EXIT service request; hence, the call statement:

```
$         SERREQ         EXIT,00
```

Statement Number	LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS
	?SERREQ		
1	(1)	MSC	
2		DEC	(2) Dummy argument for the device service request code.
3		ADC	(3) (2,E,00) Dummy argument for the return address in all service requests except EXIT, whose service request code is 00.
4		ADC	(4C) Dummy argument for FIOB address in IOACT request or parameter address in OPEN or CLOSE requests.
5	(4C)	HEX	(5) (8N) Dummy argument for LUN ID assignment statement for CLOSE and OPEN requests.
6		ADC	(6) (2,N,06) Dummy argument for IOCQ address used only in OPEN service request with service request code 06.
7		RESV,(7)	(8) (5N) Dummy argument for error code field for OPEN and CLOSE requests.
8		END	

Figure 2-10. Generalized Macro Routine for Device Service Requests

will create a specialized routine as follows:

```
EXIT     MSC
         DEC          00
```

Statement 3 is required in all other service requests, and an actual argument must appear in position 3 of all the call lists. Statements 4 through 7 are not required for the INITialization request, as shown in the call statement below.

```
$     SERREQ     INIT,02,RETAD1
```

The following specialized routine will be generated as a result of the call statement.

```
INIT     MSC
         DEC       02
         ADC       RETAD1
```

Statement 4 is required for the IOACT, OPEN and CLOSE requests. However, statements 5 through 7 are not required for the IOACT request.

The call statement below

```
$    SERREQ   IOACT1,07,RETAD2,FIOB1
```

will create the service request shown below:

```
IOACT1     MSC
           DEC       07
           ADC       RETAD2
           ADC       FIOB1
```

The OPEN service request requires all statements in the generalized routine and, therefore, eight arguments must appear in its call statement list. The CLOSE service request requires all statements except statement 6, which will be omitted from the CLOSE specialized routine because of the (2,N,06) test in the statement (i.e., the CLOSE service request code is 01). Actual argument 6 in the CLOSE call statement must therefore be represented as a comma. Calls and resulting specialized routines for OPEN and CLOSE service requests are presented below:

OPEN CALL

```
$      SERREQ    OPEN1,06,RETAD3,PARAM1,
/                LUNID,IOCQ,00,2
```

OPEN SERVICE REQUEST ROUTINE

```
OPEN1        MSC
             DEC        06
             ADC        RETAD3
             ADC        PARAM1
PARAM        HEX        LUNID
             ADC        IOCQ
             RESV,00    2
```

CLOSE CALL

```
$    SERREQ      CLOSE1,01,RETAD4,PARAM2,
/                LUNID,,00,2
```

CLOSE SERVICE REQUEST ROUTINE

```
CLOSE1       MSC
             DEC        01
             ADC        RETAD4
             ADC        PARAM2
PARAM2       HEX        LUNID
             RESV,00    2
```

4.3.3 Embedded Macro Calls

Generalized macro routines may contain one or more macro call statements that specify the names of other macro routines in their op code fields. Macro routines may not call themselves recursively, however, since this would cause an endless repetition of the macro processing phase of the Assembler.

Macro call statements embedded in generalized macro routines may themselves contain dummy arguments within their argument lists. This facility allows the programer to pass arguments from one macro routine level to another.

Section 5. ASSEMBLER PROGRAM

There are three versions of the PTS-100 Assembler:

PTS-100 Native Assembler

Raytheon 704 Cross Assembler

IBM 360/370 Cross Assembler.

The applications program input requirements for the Assemblers are described in subsection 5.1 below. The input to an IOCS monitor assembly run is the Assembler-formated tape file produced by the System Generator program, described in Part 3 of this handbook. Processing and the output listing are identical for all versions of the Assembler, as described in subsections 5.2 and 5.3. Machine requirements and Assembler limitations of the two Cross Assemblers are presented in subsection 5.4. The disc version of the PTS-100 native assembler is described in subsection 5.5

5.1 Programer Inputs

The inputs to the PTS-100 Assembler are punched card decks, each of which contains both of the following:

- One assembly control card described below, which must contain the program name and may specify assembly options.
- The source program statements, the last of which must be an END statement card to terminate assembly processing.

5.1.1 Assembly Control Card Content

The assembly control card specifies two types of information:

- The program name, which must appear in columns 1 through 8.
- Assembler options, as shown in table 2-9.

File assignments for the Assembly process are default, as shown in table 2-10.

Table 2-9. Assembler Option Selection

OPTION	CONTROL CARD Content	CONTROL CARD Column
Cross reference listing	1	28
No cross reference listing (default)	No punch	
Sequence checking	1	30
No sequence checking (default)	No punch	
Macros included (default)	No punch	32
Macros not included	1	
Relocatable object text	No punch	34
Absolute object text	1	
Full listing,* macros expanded (default)	No punch	36
Full listing, macros not expanded	1	
Error listing only	2	
No listing	3	
Machine language produced (default)	No punch	38
No machine language produced	1	
Rewind - object cassette**	No punch	40
No rewind - object cassette	1	
Source program - card reader**	No punch	42
Source program - high speed paper tape	1	
Listing - serial printer**	No punch	44
Listing - ASR	1	
Object program - cassette**	No punch	46
Object program - high speed paper tape	1	
Disc scratch file 1	DriveNo. †	12
Disc scratch file 2	DriveNo. †	14
Disc macro file	DriveNo. †	16
Disc binary output	DriveNo. ‡	18

Note: If macro calls appear in the source program, the programer must ensure that the macro library file is available as input to the Assembler.

*A full listing contains diagnostic error codes, object program code, and source language statements (see subsection 5.3).

**PTS-100 native Assembler only.

†A dig t from 0-7; if no punch, drive 0 assumed.

‡A digit from 0-7; if no punch, drive 2 assumed.

Table 2-10. File (Device) Assignments for Assembly Processing

File	Default Device Assigned		
	IBM 360	PTS-100	Raytheon 704
Source Deck	Card Reader	Card Reader	Card Reader
Macro Library	Tape Unit	Cassette or Disc	Tape Unit 3
Phase 1 Work Storage	Discs	Cassette or Disc	Tape Unit 0 Tape Unit 1
Phase 2 Work Storage	Discs	Cassette or Disc	Tape Unit 0 Tape Unit 1
Object Program	Card Punch	Cassette	Card Punch or Paper Tape Punch
Listings	Line Printer	Serial Printer	Line Printer

NOTE

Device reassignment for the IBM 360 must be effected via Job Control Language cards. Specific device assignment for the PTS-100 is effected at system generation time, as is device reassignment. On the Raytheon 704, device reassignments are effected via ASR keyboard commands prior to program assembly.

5.2 Assembly Processing

Assembly processing is accomplished in four phases if no macro processing is required, or in the following five phases if macro processing is required:

- Phase 0, described in subsection 5.2.1, which reads the control card, constructs an options table for use by all subsequent Assembler phases, and transfers control to Phase 1 if macro processing is required, or to Phase 2 if no macros are present.
- Phase 1, which processes all macro calls in the source program, described in subsection 5.2.2, and transfers control to Phase 2.
- Phase 2, which analyzes the source statements and performs preprocessing for program assembly proper, as described in subsection 5.2.3.
- Phase 3, which optimizes object program memory storage requirements, as described in subsection 5.2.4.
- Phase 4, which completes the construction of executable instructions, generates the required listing, and produces the final object program code, as described in subsection 5.2.3.

Figure 2-11 presents a general flow overview of the Assembler processing steps, which are described in detail on the following pages.

5.2.1 Phase 0 Processing

Phase 0 reads the assembly control card, its only input, and constructs the options table for use by all other phases of the Assembler. The options table specifies the following:

1. The name of the program to be assembled.
2. Whether a cross reference listing is desired.*
3. Whether sequence number checking is desired.
4. Whether there are macros to be expanded.
5. Whether output is to be relocatable or absolute.
6. Listing options:
   - Full listing, with macros expanded
   - Listing of program containing macro calls, but no macro expansion

*Does not apply to PTS-100 native Assembler.

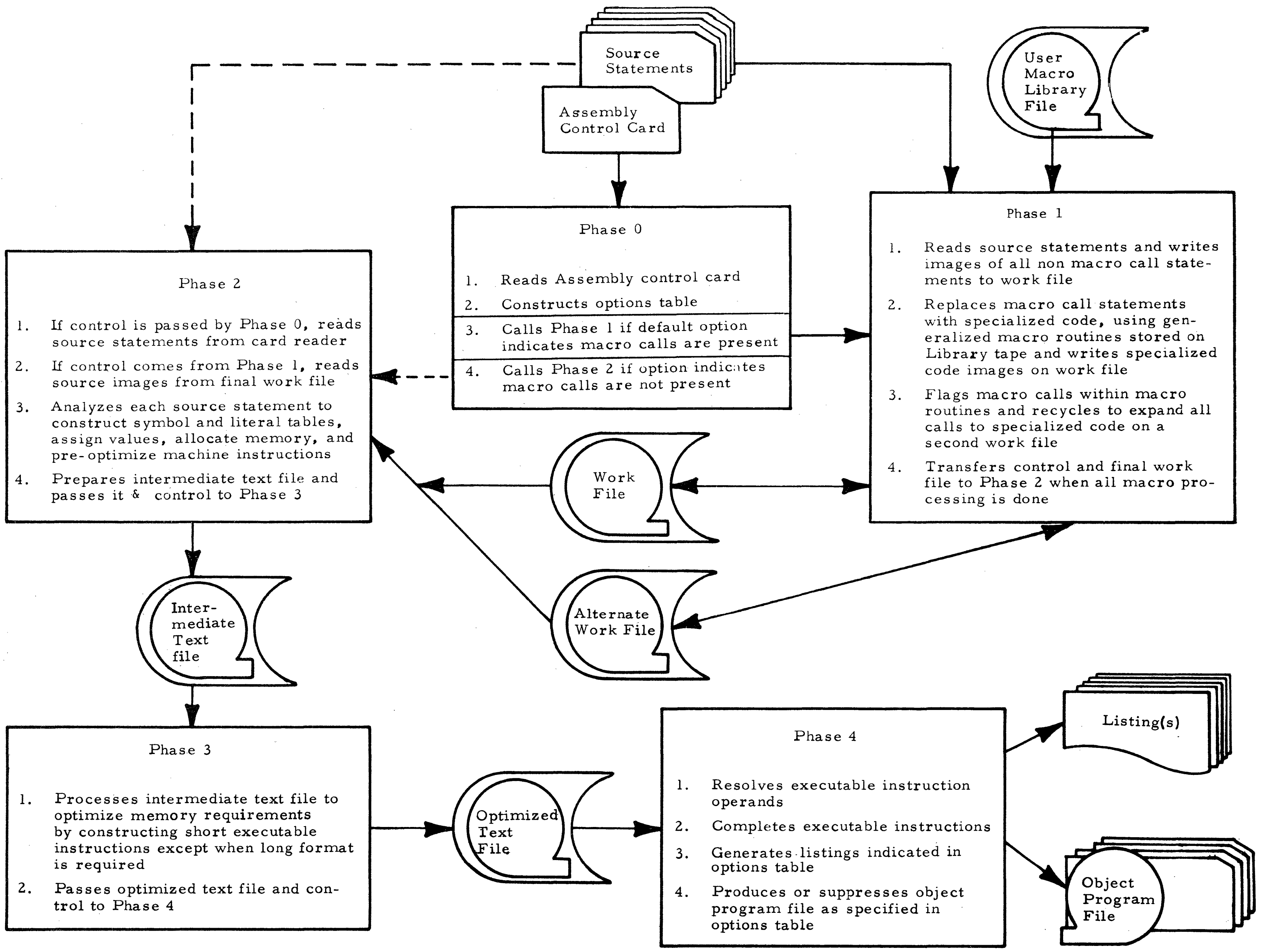

Source Statements
Assembly Control Card
User Macro Library File
Phase 0
1. Reads Assembly control card
2. Constructs options table
3. Calls Phase 1 if default option indicates macro calls are present
4. Calls Phase 2 if option indicates macro calls are not present
Phase 1
1. Reads source statements and writes images of all non macro call statements to work file
2. Replaces macro call statements with specialized code, using generalized macro routines stored on Library tape and writes specialized code images on work file
3. Flags macro calls within macro routines and recycles to expand all calls to specialized code on a second work file
4. Transfers control and final work file to Phase 2 when all macro processing is done
Phase 2
1. If control is passed by Phase 0, reads source statements from card reader
2. If control comes from Phase 1, reads source images from final work file
3. Analyzes each source statement to construct symbol and literal tables, assign values, allocate memory, and pre-optimize machine instructions
4. Prepares intermediate text file and passes it & control to Phase 3
Work File
Alternate Work File
Intermediate Text file
Phase 3
1. Processes intermediate text file to optimize memory requirements by constructing short executable instructions except when long format is required
2. Passes optimized text file and control to Phase 4
Optimized Text File
Phase 4
1. Resolves executable instruction operands
2. Completes executable instructions
3. Generates listings indicated in options table
4. Produces or suppresses object program file as specified in options table
Listing(s)
Object Program File

Figure 2-11. Flow Overview of Assembly Processing

- Listing of error lines only
- No listing.

7. Machine language output options:
   - Machine language to be produced
   - No machine language.
8. Whether object cassette is to be rewound.*
9. Whether source program is on cards or high speed paper tape.*
10. Whether listing is on the serial printer or ASR.*
11. Whether object code is to be written on cassette or high speed paper tape.*

Phase 0 transfers control to Phase 1 if macro calls are to be processed, or to Phase 2 if no macro calls were indicated by a 1 punched in control card column 32.

5.2.2 Phase 1 Processing

Phase 1 is the macro processor. Its two inputs are:

- The source program statements on punched cards.
- The user macro library file.

Phase 1 processing is performed in the following manner:

- The cards in the input deck are read one at a time.
- If column 1 of a source statement card does not contain $ or /, indicating a macro call or a call list continuation, respectively, the card image is written onto the output work file.
- If column 1 of a statement contains a $, Phase 1 constructs an argument table containing actual arguments in the order in which they appear in the call statement list. That is, the first (leftmost) actual argument in the list appears as the first entry in the argument table, followed by the second argument as the second entry, etc. Each entry in the argument table contains the following fields:

  Length of entry (1 byte)

  The variable-length argument value

  When all arguments in the call statement and any continuation cards have been entered into the table, Phase 1 searches the macro library file for the generalized routine named in the op code field of the call statement. If the named macro routine is not found in the library file, Phase 1 sets an N flag in the error code field of the macro call statement card image, writes the card image onto the work file, and reads the next source statement card. If the named routine is located in the library file, Phase 1 creates the specialized routine specified in the macro call statement list, as described in detail below.

When a call generalized routine has been located in the library file, Phase 1 scans each of the routine's source statements for dummy arguments in any of the permissible forms described in Section 4. If no dummy arguments appear in a statement, or when all dummy arguments have been scanned and processed, the source statement is written onto the work file. When a dummy argument of the form (n) is found, Phase 1 locates the nth entry in the argument table and substitutes its value for the dummy argument.

When a dummy argument of the form (nC) is found in a generalized source statement, Phase 1 locates the nth entry in the argument table, and if it contains a value, the value is substituted for the dummy argument and Phase 1 continues the statement scan or writes the specialized statement onto the work file if no more dummy arguments are

*Items 8-11 apply only to the PTS-100 native Assembler.

present. If the nth entry of the argument table does not contain a value, the source statement is not excluded from the specialized routine (i. e., it is written on the work file).

When a dummy argument in the form (nY) is located in the statement scan, Phase 1 locates the nth entry in the table. If a value appears in the entry, the statement containing the dummy argument is omitted from the specialized routine. If a value does not appear in the entry, the source statement will be written on the work file.

When a dummy argument in the form (nN) is found in the source statement scan, Phase 1 locates the nth entry in the argument table to determine whether it contains a value. If not, the source statement is omitted from the specialized routine; otherwise, it is written onto the work file.

When a statement contains a dummy argument of the form (n, E, vv) Phase 1 locates the nth entry in the argument table, compares its value to the value specified by vv, and omits the statement containing the dummy argument if the values are equal. Otherwise, the statement is written onto the work file. A dummy argument in the form (n, N, vv) causes the nth entry value and vv value to be compared for not equal, and the source statement to be omitted from the specialized routine if they are unequal; otherwise, the statement is written onto the work file.

If an embedded macro call statement (i. e., a statement calling another macro routine) appears within the generalized routine being processed, it is scanned for dummy arguments, which are processed as above. Phase 1 then sets a MORE flag within its own coding to indicate that another macro processing pass is to be performed when the current pass has been completed (i. e., when the END source statement is encountered in the input deck). That is, when a second macro routine is called within the current macro routine, Phase 1 writes the embedded call statement to the work file, finishes processing the current macro routine and the remainder of the input source statement deck, recycles to its beginning, then rewinds and reads its output work file as input to process the embedded macro calls encountered during the previous pass. The output source statements for the second pass are written onto the alternate work file. If additional embedded macro calls are found during the second pass they are flagged, the alternate work file becomes input to another pass, and Phase 1 writes processed source statements to the original work file. At the end of each processing pass, Phase 1 tests its MORE flag to determine if another pass is necessary, alternating work files and recycling if necessary. When the MORE flag is not set at the end of a pass, Phase 1 transfers the address of its final output work file and control to Phase 2, which begins the first step in assembly proper processing.

5.2.3 Phase 2 Processing

Phase 2 is the first step in the conversion of source language to object language. This phase converts the source code to an intermediate text which becomes input to Phase 3. Input to Phase 2 is either:

- The source statement deck, if control was passed from Phase 0
- The final work file from Phase 2, when macro processing was required.

Phase 2 reads input source statements, scanning each statement to identify and analyze its component parts and convert the statement to an intermediate text format. Each statement is processed according to its content and type by Phase 2, as follows:

- All hexadecimal, octal, and decimal constants are converted to binary representation.
- Memory is allocated as specified by ORIGIN, MOD, and RESERVE statements and for machine instructions.
- Executable instructions are preoptimized to long or short format where possible.
- Values are assigned to symbolic tags and placed in a symbol table for use by Phase 3 and 4.
- Literals are placed in four-byte entries in a memory table or pool. Entries in the table contain a two-byte system generated symbol, and the two-byte literal value itself. When a Literal Origin statement is processed by Phase 2, the literal table entries are written on the intermediate text file, along with their system-generated addresses. A new literal pool is then started.

The output from Phase 2 is the intermediate text file containing the processed statements and symbol and literal tables.

5.2.4 Phase 3 Processing

Phase 3 optimizes executable instructions to guarantee a minimum core requirement for the object program. That is, it determines whether the short instruction format can be used, using the long format only where necessary. Thus, Phase 3 assumes the burden of efficient core utilization for the programer and enables subsequent program changes without inducing addressing errors in existing code.

The input to Phase 3 is the intermediate text file, and the output is the optimized text file.

5.2.5 Phase 4 Processing

This phase completes assembly processing. Its input is the optimized text file. Phase 4 performs the following functions:

- Completes the construction of executable instructions by inserting memory address in operand fields.
- Generates and prints the listing as described in subsection 5.3, unless no listing is specified in the options table.
- Generates an absolute or relocatable object program file unless the options table specifies no object language file to be produced.

5.3 <u>Assembler Output Listing</u>

Depending on options specified on the assembly control card, the Assembler produces the following output listing:

- A full listing, containing the following:

  specialized macro routines
  error diagnostic codes
  object program code
  source language statements

- A full listing of the current program without specialized macro routines, containing the following:

  error diagnostic codes
  object program code
  source language code

- An error listing only

As shown in table 2-9, the programer may specify that no listing is to be produced.

Figure 2-12 illustrates a sample page of a full listing without specialized macro routines. The left-most column is titled ERRORS. If the Assembler detects coding errors in the source language statements, the appropriate error codes appear in this field, as shown below:

Error Code	Significance
A	addressing error: • attempt to reference non-word boundary with word instruction • attempt to use externally defined symbol in instruction other than ADC
B	symbol table overflow
C	constant error: • illegal constant type • illegal constant length
D	duplicate symbol
E	symbol, as used, not defined as an absolute EQU
F	format error in the operand field
G	symbol, required to be predefined, not predefined
H	too many symbols in operand field
L	label error: • in label field, either an illegal start character or label too long • in operand field, label too long
M	illegal op code modifier
O	unrecognized op code
P	macro argument error
S	sequence error
U	undefined symbol
X	symbol is both operand of EXREF and defined in current program

Column LOC of the listing specifies the byte location, expressed in hexadecimal, of the current instruction. The CONTENTS column indicates the contents of the current instruction, also expressed in hexadecimal. The columns OP, R, E, I, S, and OPERAND contain the code of executable instructions, where:

Column OP contains the machine operation code, expressed in hexadecimal.

Column R specifies the register being used, where:

0 = accumulator operation, or absolute addressing

1 = program counter relative operation

2 = index register 1 operation

3 = index register 2 operation.

Column E specifies the length of the instruction, where:

0 = short instruction

1 = long instruction.

Column I specifies the type of addressing, where:

0 = direct addressing

1 = indirect addressing.

Column S specifies the sign of the OPERAND value.

Column OPERAND specifies the displacement value used to form the effective address, where:

The OPERAND of a short instruction is a 7-bit <u>word</u> displacement value

The OPERAND of a long instruction is a 16-bit <u>byte</u> displacement value.

NOTE

See Section 1 of part 2 for a description of executable instruction format.

Column SEQ of the listing contains an Assembler-generated sequence number.

Column SOURCE contains the source statement as read by the Assembler. If the programer specified sequence number checking, the programer-assigned sequence number appears at the righthand side of the listing.

```
          MARY                                                          PAGE  ØØ14
 ERRORS   LOC CONTENTS   OP R F I S OPERAND SEQ * SOURCE

                                            Ø521  **************** ***********************************************************
                                            Ø522  *                                                                          *
                                            Ø523  *                THIS ROUTINE ATTEMPTS TO OPEN THE MODEM RECEIVE           *
                                            Ø524  *                CHANNEL AND QUEUE 3 I/O REQUESTS IN THE IOCQ. IF          *
                                            Ø525  *                INITIALISATION IS SUCCESSFUL  SNAMR IS SET TO Ø,          *
                                            Ø526  *                IF UNSUCCESSFUL  SNAMR IS SET TO 1.                       *
                                            Ø527  *                                                                          *
                                            Ø528  **************** ********************************************************* *
          13Ø2   13Ø2                       Ø529  MRSET   EQU     *
          13Ø2   DA16   18 1 Ø Ø +     16   Ø53Ø          SX2     MRRT
          13Ø4   93ØØ   12 1 1 Ø            Ø531          LDW     IOMR3
          13Ø6   FC2Ø              -  Ø3FØ
          13Ø8   C3ØØ   18 1 1 Ø            Ø532          STW     IOQMR           /POINT TO FIRST ENTRY IN IOCQ
          13ØA   FFE4              -  Ø11C
                                            Ø533  *
                                            Ø534  *****************-------- ISSUE A MSC OPEN CHANNEL REQUEST
                                            Ø535  *
          13ØC   ØEØØ                       Ø536          MSC
          13ØE   ØØØ6                       Ø537          ADC     CODE6          /OPEN CODE = 6
          131Ø   131A                       Ø538          ADC     T62            /RETURN ADDRESS
          1312   1314                       Ø539          ADC     T63            /PARAMETER LIST ADDRESS
          1314   ØØØ9                       Ø54Ø  T63     DEC     9               LUN-ID
          1316   ØFØØ                       Ø541          ADC     IOMR1          /IOCQ ADDRESS
          1318   ØØØØ                       Ø542  MRERR   RESV,Ø 2
                                            Ø543  *
                                            Ø544  *****************-------- CHECK FOR ERROR RETURN
                                            Ø545  *
          131A   2ØØØ   Ø4 Ø Ø Ø +     ØØ   Ø546  T62     LDI     AC,Ø
          131C   8283   1Ø 1 Ø Ø -     Ø3   Ø547          CNE     MRERR          /CHECK FOR ERROR RETURN
          131E   12Ø1   Ø2 1 Ø Ø +     Ø1   Ø548          BCR     ERRMR          /YES
          132Ø   Ø2ØB   ØØ 1 Ø Ø +     ØB   Ø549          JMP     OPNCH          /NO.
                                            Ø55Ø  *
                                            Ø551  **********---------TURN ON SNA LIGHTS
                                            Ø552  *
          1322   2ØØ1   Ø4 Ø Ø Ø +     Ø1   Ø553  ERRMR   LDI     AC,1
          1324   C3ØØ   18 1 1 Ø            Ø554  T2ØØ    STW     SNAMR          /SET  SNAMT = 1 OR Ø
          1326   FECE              -  Ø132
          1328   42Ø1   Ø8 1 Ø Ø +     Ø1   Ø555          LAX2    T2
          132A   Ø233   ØØ 1 Ø Ø +     33   Ø556          JMP     SNALT
          132C   132C                       Ø557  T2      EQU     *
          132C   AAØ1   15 1 Ø Ø +     Ø1   Ø558  MRRTN   LX2     MRRT
          132E   Ø6ØØ   ØØ 3 Ø Ø +     ØØ   Ø559          JMP,X2  Ø              /RETURN
          133Ø   133Ø                       Ø56Ø  MRRT    ADC     *
                                            Ø561  *
          1332   1332                       Ø562  OMBTP   ADC     *
          1334   1334                       Ø563  MRIDX   ADC     *
          1336   1ED6                       Ø564  OMBA    ADC     OMB1
                                            Ø565  **********-------- QUEUE 3 I/O REQUESTS IN IOCQ
                                            Ø566  *
          1338   1338                       Ø567  OPNCH    EQU    *
          1338   2ØØØ   Ø4 Ø Ø Ø +     ØØ   Ø568          LDI     AC,Ø
          133A   C284   18 1 Ø Ø -     Ø4   Ø569          STW     MRIDX
          133C   9284   12 1 Ø Ø -     Ø4   Ø57Ø          LDW     OMBA
          133E   C287   18 1 Ø Ø -     Ø7   Ø571          STW     OMBTP
          134Ø   A3ØØ   14 1 1 Ø            Ø572          LX1     FIBMR           /LOAD ADDRESS OF MODEM RECEIVE FIDB
          1342   FEC2              -  Ø13E
```

Figure 2-12.   Sample Assembler Output Listing

5.4 Assembler Limitations and Machine Requirements

Following are the recommended machine requirements and pertinent limitations of the Raytheon 704 and IBM 360 Cross Assemblers.

5.4.1 Raytheon 704 Cross Assembler

The recommended minimum Raytheon 704 equipment configuration for the PTS-100 Assembler is as follows:

1	card reader
1	card punch
3	magnetic tape drives if the source program contains macro calls; otherwise only 2 tape drives are required
1	line printer
1	ASR 33/35
16K	words of core storage

Device reassignment on the Raytheon 704 may be effected through the use of the I/O device reassignment facility of the Series 700 operating system.

5.4.2 IBM 360/370 Cross Assembler

The recommended minimum IBM 360/370 configuration for the PTS-100 Assembler is as follows:

1	card reader
1	card punch
1	magnetic tape drive
1	disc drive
1	line printer
32K	bytes of core

The IBM 360/370 version of the PTS-100 Assembler is designed to run under both the DOS and OS systems. With the exception of the system dependent code to produce object code in column binary, no system dependent macros are used, facilitating compatibility between systems. All input/output is performed through the use of a Cobol subroutine.

The IBM 360/370 version of the PTS-100 Assembler ignores all device reassignment facilities of the assembly control card. Device reassignment is performed through the Job Control Language.

5.4.3 PTS-100 Native Assembler

The recommended minimum PTS-100 configuration for the PTS-100 Native Assembler is as follows:

1	card reader
4	cassette drives
1	serial printer
16K	bytes of core

Optionally, the source program may be read from a high speed paper tape reader, eliminating the necessity of the card reader. Also, the object program may output onto a high speed paper tape punch.

A disc may be used for intermediate text storage, eliminating the necessity of two of the cassettes.

5.5 Disc Assembler

For the Disc Assembler, the scratch files and the macro file may be on the same or different disc drives. To designate the drive number locations for each file, the following fields have been added to the control card:

SCRATCH FILE 1	-	COL. 12
SCRATCH FILE 2	-	COL. 14
MACRO FILE	-	COL. 16
BINARY OUTPUT	-	COL. 18

Columns 12, 14, and 16 should contain a number from 0 through 7, designating the drive on which the corresponding file is mounted; if there is no punch, drive 0 is assumed. Column 18 should contain the drive number for the binary output; if this column is left blank, drive 2 is assumed.

The scratch and macro files must be allocated previous to the assembly execution, using the Disc Allocator utility program. (If macros are not used, only two files need be allocated.) The following parameters should be used for allocation:

Scratch File 1:

FILENAME	=	ASSEMBSCR1
DRIVENO	=	same as punched in column 12 of the assembly control card.
FUNCTION	=	NEW
FIRSTCYL	=	(see Note)
LASTCYL	=	(see Note)
FILEORG	=	S
RECSIZE	=	164

Scratch File 2:

FILENAME	=	ASSEMBSCR2
DRIVENO	=	same as punched in column 14 of assembly control card.
FUNCTION	=	NEW
FIRSTCYL	=	(see Note)
LASTCYL	=	(see Note)
FILEORG	=	S
RECSIZE	=	164

Macro File:

FILENAME	=	DMACROFILE
DRIVENO	=	same as punched in column 16 of assembly control card.
FUNCTION	=	NEW
FIRSTCYL	=	(see Note below)
LASTCYL	=	(see Note below)
FILEORG	=	S
RECSIZE	=	80

NOTE

The parameters FIRSTCYL and LASTCYL are not given above, because the sizes of the files are variable.

The sizes of scratch files 1 and 2, which should be the same size, depend on the size size of the program to be assembled. Allow one cylinder for each 78 statements in the program to be assembled. For example, if a source program contains 500 statements, a minimum of 7 cylinders should be allocated to each of the scratch files.

The size of the macro file depends on the total number of statements in the macros that are to be put into the file. Allow one cylinder for every 160 macro statements or fraction thereof.

Section 6. PROGRAMING TECHNIQUES

Presented below are some special techniques that the PTS-100 application programer may find useful.

6.1 Shifting Techniques

The Shift Right One, Arithmetic statement is the only shift statement provided in the PTS-100 Assembler language. There are some techniques, however, that may be used to effect shifting, as follows:

1. To shift left one position, add the value to be shifted to itself. For example,

   X'10' + X'10' = X'20'
   X'60' + X'60' = X'CO'

   This technique can be used for each multiple of two in a multiplier. For example,

   $5_{10} \times 8_{10} = 5_{10} \times 2^3_{10}$

   $= 101_2$ shifted left three

   $= 101000_2 = 40_{10}$

2. To shift left or right eight positions, execute a Load Byte instruction, and then a Store Byte instruction.

6.2 Setting Addresses

The programer should use the Load Address In Index Register 2 statement to effect the following:

1. Set a return address and/or an argument list address when calling a subroutine.

2. Obtain the address of a value instead of defining an ADC for that value, if possible.

6.3 Defining Message Content

The Text constant statement should be used to define the content of message buffers.

6.4 Label Definition

The Equate statement may be used to define labels, which facilitates program changes or corrections. For example, the statements

```
START   EQU   *
        LDW   KOUNT
```

may be used instead of

```
START   LDW   KOUNT
```

6.5 Constant Definitions

In working with constants, the Load Immediate statement saves core storage requirements. For example,

```
LDI     AC,1
```

may be used instead of

```
        LDW   One
ONE     HEX   1
```

6.6 Comparison Bit Setting

It is possible and sometimes helpful to set a compare bit before the actual branch, as shown below:

```
LDI     AC,1
CNE     TWO
LDW     CONSTANT
BCB     SUBR
```

When this technique is used it is essential for the programer to remember all instructions that use the comparison bit.

Section 7. SYSTEM PROGRAMING CONSIDERATIONS

The PTS-100 accommodates a wide range of external input/output device types. Input/output operations for application programs are managed by the Input/Output Control System (IOCS) monitor, which is a resident modular software system composed of two major components:

- The I/O Control Nucleus, which handles monitor service calls from applications programs and services interrupts from the I/O devices.
- The physical I/O routines, which handle requests to the specific I/O devices supported by the system.

The I/O Control Nucleus provides two kinds of service: device interrupt handling and processing of service calls from application programs. The interrupt handling service is provided by the level service routines, which provide entry and exit control for all interrupts, save and restore registers, and link to the appropriate device service routines.

The physical I/O routines handle all requests for each physical I/O device in the system. There is a set of physical I/O routines for each type of I/O device in the equipment configuration. A set of routines includes the device driver routine and the device service routine. The device driver routine is called when there is an I/O request in the logical IOCQ table and the channel for the device is inactive. The device driver routine uses the information in the IOCQ entry to set up the physical I/O control table and to initiate the I/O action.

The device service routines are assigned to one of eight external interrupt levels when a particular IOCS monitor is generated for a specific installation. These routines determine the reason for an interrupt, update control and status fields, take any required action, and then initiate action on the next I/O request in the IOCQ table.

The application program service calls are processed as described in Section 3 of this part of the handbook.

For any given PTS-100 installation, a hardware specialized IOCS is created by the System Generation program, described in Part 3 of this handbook. If the PTS-100 user wishes, he may alter the IOCS monitor by adding special physical I/O and control routines to accommodate nonstandard devices that are not supported by the IOCS monitor, or he may develop his own IOCS monitor. In either case, an understanding of the interrupt system of the PTS-100 and the systems programing I/O and interrupt statements is required, as described in the remainder of this section.

7.1 Interrupt System

A multilevel interruption system provides eight external (device) interrupt levels and three internal (CPU) levels. The CPU operates at a given level and may be interrupted when an enabled higher priority interrupt condition is detected. Instructions are provided to enable and disable interrupts, trap to a higher priority internal level, and return to prior levels after servicing interrupts. The priority of interruption is shown in figure 2-13, with the highest numbered level having the highest priority.

The Parity Interrupt is optional. It occurs when the processor hardware detects invalid parity on the data returned from memory. At the completion of the current instruction the interrupt is serviced and level 10 is entered.

10	PARITY
9	TRAP
8	EXTERNAL 8
7	EXTERNAL 7
6	EXTERNAL 6
5	EXTERNAL 5
4	EXTERNAL 4
3	EXTERNAL 3
2	EXTERNAL 2
1	EXTERNAL 1
0	PROCESSOR/INTERVAL TIMER

Figure 2-13. Interrupt Priority Levels in the PTS-100

The Trap Interrupt is a synchronous interrupt that occurs when a Monitor Service Call (MSC) instruction is encountered in an executing program. This interrupt may be issued at any level. The interrupt is not maskable by the Disable Interrupts instruction. Execution of the MSC instruction consists of storing the present status and loading the program counter from the level 9 interrupt packet. Instruction execution then resumes at level 9.

There are eight External Interrupt signals in levels 1 through 8. These may be assigned to any configuration of input/output devices. Interrogation and resetting of interrupt conditions are accomplished by executing the Read Device Status instruction, described in subsection 7.3.2.

The Processor Interrupt level 0 does not have the ability to interrupt execution at another level. Level 0 may only be entered via the Interrupt Return instruction with no higher level interrupts outstanding. This is the level at which object programs execute.

The Interval Timer Interrupt is an optional external interrupt condition that occurs once every 67 milliseconds. The interrupt may be taken only when the CPU is already operating at level 0 with external interrupts enabled. The interrupt causes present status to be stored and the program counter to be loaded from the level 0 packet. Processing continues at level 0.

The central processor may be operating at any of the 11 interrupt levels and is normally enabled for external interrupts that occur at a higher priority level than the present level. The trap and parity interrupts are always enabled. Interrupts of the same or lower priority than the present operating level remain pending. All external interrupts may be disabled (held pending) by executing the Disable Interrupts instruction. The processor returns to the enabled state when the Enable Interrupts instruction is executed.

For each assigned interrupt level, an associated four-word interrupt packet must be set up in the format shown in figure 2-14.

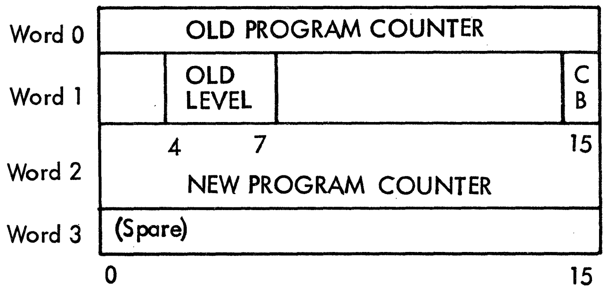


Figure 2-14. Interrupt Packet Format and Content

When the processor is operating at one level and an interrupt of higher priority is enabled, the processor completes the execution of the current instruction and then enters the following fixed sequence.

1. The value of the program counter (pointing to the next sequential instruction to be executed) is stored in the first word in the interrupt packet.

2. The old interrupt level and the condition bit (CB) are then stored in the next sequential word of the packet, in bits 4-7 and 15, respectively.

3. A new value for the program counter is then loaded and the new interrupt level is entered.

This sequence of events cannot be interrupted. If a higher priority interrupt occurs during the sequence, servicing is deferred until completion of one CPU instruction at the new level.

An Interrupt Return instruction should be issued immediately following the completion of interrupt servicing. This causes the processor status to be restored to the point prior to the interruption. The old PC, old interrupt level, and old condition bit are restored by the hardware from the save area at the departing level.

7.2 Interrupt Statements

There are three statements provided for changing the external interrupt level at which the central processor is currently operating, as shown in table 2-11.

Table 2-11. Interrupt Statements

LABEL FIELD	OP CODE FIELD	COMMENTS
(label)	DIN	(disable interrupts)
(label)	ENB	(enable interrupts)
(label)	INR	(interrupt return)

The DIN (disable interrupts) statement specifies that all interrupts at levels 0 through 8 are to be disabled. That is, they are to be held pending so that current instruction execution cannot be interrupted.

The ENB (enable interrupts) statement specifies that all external interrupts at levels 1 through 8 are to be enabled. That is, external interrupts of a higher priority than the current CPU operating level are to be serviced when they occur.

The INR (interrupt return) statement specifies that the CPU is to return to the interrupt level that was current just prior to the most recent interrupt.

At assembly time these statements are translated to short machine instruction format. The machine op code in all cases is 01. The R field of a given machine instruction serves as an extended op code to identify the specific interrupt statement involved, where:

R = 00	Enable Interrupts
R = 01	Disable Interrupts
R = 10	Interrupt Return

In all cases, the operand field of the machine instruction should be zero.

7.3 System Programing of I/O Operations

Input/output operations occur via direct memory access channels. Data, addresses, and status information are exchanged between the CPU and device controllers across a 16-bit bidirectional bus. The I/O controllers may initiate data transfers between devices and memory and may also initiate limited arithmetic operations to be performed in the CPU. These actions are overlapped with CPU instruction execution.

For system programs that are to run independently of the IOCS monitor, two statements are provided to perform I/O operations and interrogate status indicators in the I/O controllers and devices. These statements are:

- Do IO statement, which is used to perform all I/O operations.

- Read IO Status statement, which is used to test the operational status of devices and device controllers.

7.3.1 Performing I/O Operations

When I/O operations are to be performed independently of the IOCS monitor, the Do IO statement must be used in conjunction with a Load Word statement as shown below.

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS
(label)	LDW	X'0 + Device Address' (start I/O on DA)
(label)	DIO	I/O Packet Address
(label)	LDW	X'1 +Device Address' (stop I/O on DA)
(label)	DIO	*

The operand of the LDW statement must have been previously stored by the programer. It specifies the start or stop command and the physical address of the device on which the operation is to be performed. The format of the LDW operand is shown below.

0 1 2 3	4 5 6 7 8 9 10 11 12 13 14 15
CMD	DEVICE ADDRESS

where: bits 4 - 15 specify the physical address of the device, and

bits 0 - 3 specify one of the following:

CMD = 0_{16} specifies that an IO operation is to start

CMD = 1_{16} specifies that the IO operation is to stop

If CMD = 1_{16} when the Do IO instruction is executed, the addressed device will be stopped as soon as possible and all pending or active memory requests from the device will be cleared. The IO controller for the device will be left in the not busy state.

NOTE

When a stop I/O command code is specified, the Do IO statement operand must be a symbolic tag or a self-referencing indicator.

If CMD = 0_{16}, the Do IO instruction operand must be the starting address of an I/O packet specifying all information necessary for executing the I/O operation, as described in subsection 7.3.1.1 below.

When the Do IO instruction to start an operation is executed, the value in the accumulator is placed on the input/output data bus for the specified device. The I/O packet address is then transferred to the selected controller. The controller uses the address to locate the packet and perform the specified operation. The controller subsequently interrupts the CPU to signal significant device events.

7.3.1.1 I/O Packet. When I/O operations are being performed independently of the IOCS monitor, the I/O packet performs the functions of the FIOB and PIOT for operations under control of the IOCS monitor. That is, it specifies the input/output function to be performed on the device, the data storage area to or from which data is to be transferred, the total number of bytes of data involved in the transfer, and the base addresses of any Search or Translate tables to be used in the operation or disc address. The I/O packet must start an eight-word boundary, in the format shown in figure 2-15. I/O packet field content is discussed in detail below.

Bits	0 1 2	3 4 5 6 7	8 9 10 11 12 13 14 15
Word 0	ORDER COC	BYTE DOC	INTERRUPT MASK BYTE
Word 1	BYTE ADDRESS		
Word 2	BYTE COUNT		
Word 3	TRANSLATE TABLE BASE (TTB) OR DISC ADDRESS		
Word 4	SEARCH TABLE BASE (STB) OR DISC ADDRESS		
Word 5	ALTERNATE BYTE ADDRESS		
Word 6	ALTERNATE BYTE COUNT		
Word 7	SPARE		

Figure 2-15. I/O Packet

The Order Byte field of the I/O packet contains the device Controller Order Code (COC) in bits 0 - 2 and the Device Order Code (DOC) in bits 3- 7.

The Controller Order Code specifies the data transfer mode (i.e., whether Search and Translate functions are to be performed by the I/O controller). See Section 3 for a detailed description of these special functions.

The Device Order Code (DOC) in bits 3-7 of the Order Byte specifies the desired I/O function to be performed on the specific device, as shown in the right-most column of table 2-6.

The interrupt mask in the right-hand byte of Word 0 of the I/O packet is used to allow or inhibit interrupts. That is, the bits of the Mask Byte correspond one-for-one with the bits in the Interrupt Condition Byte (ICB) in the device controller. Hence, the programer may set a one bit in each position of the Interrupt Mask Byte where the corresponding interrupt is to be allowed and a zero bit in each bit position of the Interrupt Mask where the corresponding interrupt is to be inhibited. When an interrupt condition occurs in the device controller, the Interrupt Mask is ANDed with the ICB to determine whether an interrupt should be generated.

NOTE

Mask bits do not reset ICB bits. They merely specify whether interrupts are to be enabled or diabled for a given I/O activity.

The possible bit settings of the Interrupt Mask and IC bytes are as follows:

Bit 0	=	Search requested and MSB = 1
Bit 1	=	Byte count incremented to zero
Bit 2	=	Start command issued when the device is in a NOT READY state
Bit 3	=	Device "END OF RECORD" (EOR)
Bit 4	=	Attention *
Bit 5	=	Error * (Data overrun, data error, or unit check generated by the device)

Byte Address. Word 1 of the I/O control packet specifies the address of the first byte of the memory storage area into or from which input/output data is to be transferred.

Byte Count. Word 2 of the I/O packet specifies the two's complement of the total number of bytes of I/O data to be transferred. The byte count is incremented each time a byte of data is transferred by the I/O controller. When the byte count reaches zero, the data transfer is complete.

Words 5 and 6 of the I/O packet are used to specify the alternate data storage address and byte count when I/O commands are chained. Command chaining is specified by a one in bit 3 of the device order code, as shown in table 2-6. When command chaining is specified, the I/O

*The attention and error bits are summary bits indicating a broad classification of the type of interrupt that was generated, depending on the variable device controller conditions which may be indicated by individual bits in the device status byte, described in subsection 7.3.2. The programer should consult the PTS-100 Reference Manual for detailed information about status indicators of specific devices and controllers.

controller executes the first order specified by the DOC and uses the byte address and count located in words 1 and 2 of the I/O packet. Data transfer is halted when the byte count in word 2 reaches (is incremented to) zero. When the next I/O command is executed, the I/O controller uses the alternate address and byte count specified in words 5 and 6 of the packet. Prior to issuing another chained set of commands against the packet, the programer must reset the byte addresses and counts in the packet. As chained commands are subsequently received, the controller again alternates between the byte addresses and counts. Odd numbered orders utilize words 1 and 2 of the packet, and even numbered orders utilize words 5 and 6 of the packet. Command chaining continues until a device order with bit 3 set to zero is executed or until a Stop I/O command is issued.

Disc Address. Bits 2 through 6 of word 3 contain the track address, and bits 7 through 15 contain the cylinder address. Bits 11 through 15 of word 4 contain the sector address.

7.3.2 Testing Device Operational Status

Before and after issuing a Do IO command, the programer should test the operational status of the addressed device. Status testing is specified via the Read IO Status statement, preceded by a Load Word statement as shown below.

LABEL FIELD	OP CODE FIELD	OPERAND/COMMENTS
(label)	LDW	X'0 + Device Address' (reads and resets status)
(label)	RIO	Memory Address
(label)	LDW	X'1 + Device Address' (reads device status)
(label)	RIO	Memory Address

The operand of the LDW statement must have been previously stored by the programer. It specifies the command code and the physical address of the device whose status is to be checked. The format of the LDW operand is shown below.

0 1 2 3	4 5 6 7 8 9 10 11 12 13 14 15
CMD	DEVICE ADDRESS

where: bits 4 - 15 specify the physical address of the device and bits 0 - 3 specify one of the following:

CMD = 0_{16}	specifies that the device status is to be read and interrupts are to be reset.
CMD = 1_{16}	specifies that the device status is to be read, but no interrupt conditions are to be altered.

In both cases above, the RIO statement operand specifies the memory address to which the device status is to be transferred.

When the accumulator has been loaded with a command code of zero and the device address, execution of the RIO statement causes the device's status to be read and any pending interrupts to be cleared (i.e., the device status is reset). That is, if interrupts are pending the following will appear in the memory word specified in the RIO operand field:

0 7	8 15
0---------------0	ICB

If no interrupts were pending, the memory word will contain all zeros after the read and reset interrupts operation. It should be noted in this case that the ICB in the controller may not be zero because the interrupt mask may have inhibited the generation of an interrupt.

When the accumulator has been loaded with a command code of one and a device address, execution of the RIO statement causes the device status and the ICB to be stored in the memory, as follows:

0 7	8 15
Device Status Byte	ICB

Device/controller status in every case consists of a minimum of two bits. The two bits are defined universally for all controllers and are bits 0 and 1 of the status byte as shown below.

READY	BUSY	
0	0	Device Not Operational
0	1	Order In Process (i. e., Busy)
1	0	Device and Controller Available for New Order
1	1	(Undefined)

BUSY — Bit 1 of device controller status byte

READY — Bit 0 of device controller status byte

Therefore, only bit 0 of the status byte must be tested to determine if a new Do IO instruction may be issued.

INDEX TO PART 2

PART 3

PTS-100 UTILITY PROGRAMS

PART 3

PTS-100 UTILITY PROGRAMS

TABLE OF CONTENTS

TABLE OF CONTENTS (cont)

TABLE OF CONTENTS (cont)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (cont)

LIST OF TABLES

PART 3. PTS-100 UTILITY PROGRAMS

Section 1. GENERAL INTRODUCTION

Optionally available to PTS-100 users are a number of utility programs to aid in the development, checkout, execution, and maintenance of systems and applications processing programs. The following types of programs are available.

- Two loader programs, the first of which is used to initialize the computer and to load the main loader program, which in turn must be used to load all assembled programs to be executed on the PTS-100.
- The Interactive Debug program, which allows the programer to interface actively with it during object program checkout and testing.
- The System Generator program, which performs the initial processing to produce a specially-tailored PTS-100 IOCS monitor to meet the unique applications program I/O requirements of any given installation.
- The Memory Dump program, available in two versions, which dumps main memory contents onto peripheral devices.
- The Peripheral Device Dump program, which dumps serial binary data file records onto a character printing device.
- The File Update program, which provides a convenient, easily used method of creating, maintaining, and updating files of both object and source programs.
- Three disc support programs to initialize new discs for use with a PTS-100, allocate disc file space, and dump disc files onto a printing device.
- The Cassette utility program, which provides a method of storing on, deleting, copying, positioning, and printing the contents of cassette magnetic tape files.

Detailed, "how to use" descriptions of these utility programs are presented in this part of the Programers Handbook.

Section 2. PTS-100 LOADER PROGRAMS

There are two loader programs supplied with the PTS-100:

- The Piggyback Loader, the sole function of which is to load the Absolute/Relocating Loader.
- The Absolute/Relocating Loader, which must be used to load all other object programs, including systems programs, to be executed on the PTS-100.

The inputs to the loading process are:

- The binary code of the Piggyback Loader.
- The assembled, relocatable code of the Absolute/Relocating Loader program.
- The assembled absolute or relocatable object code of the programs or program segments to be loaded.

The inputs to the total loading process for the PTS-100 are illustrated in figure 3-1. The Piggyback Loader is bootstrapped into low memory by depressing the Initial Program Load (IPL) button on the user console of the PTS-100. Once loaded, the Piggyback Loader initializes its tables and storage addresses and reads the object code of the Absolute/Relocating Loader, loads it into high memory, and then activates it. The Absolute/Relocating Loader then reads and loads the object program(s) from the input device.

The processing, inputs, and outputs of the two loaders are described in the remainder of this section.

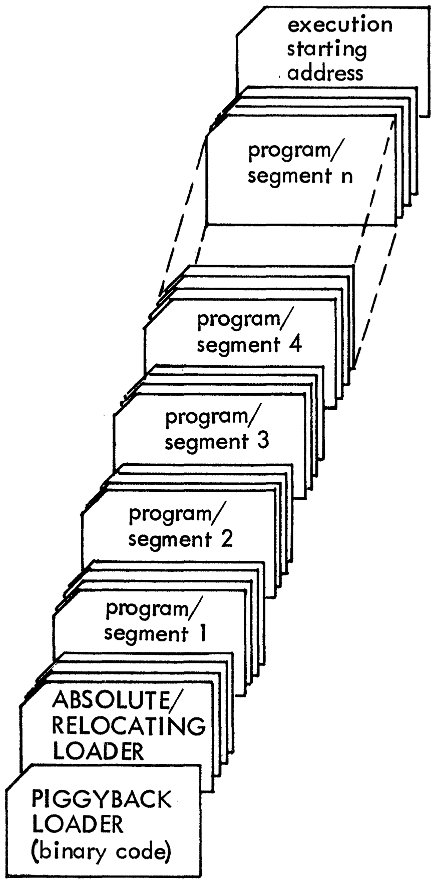


Figure 3-1. Inputs to the PTS-100 Loading Process, Assuming the Card Reader as Input Device

2.1 Piggyback Loader

The Piggyback Loader is used to load only one program: the Absolute/Relocating Loader. As mentioned earlier, the Piggyback Loader is itself loaded via IPL bootstrap. Once loaded, the Piggyback Loader performs the following:

- Initializes its own tables and storage addresses.

- Determines the highest available memory location in the computer into which it has been loaded.

- Reads the object coding of the Absolute/Relocating Loader from the input device whose physical address has been assembled into the Piggyback Loader.*

- Validates input records as they are read, and loads the Absolute/Relocating Loader into the highest memory area.

- Activates the Absolute/Relocating Loader when it has been completely loaded.

2.1.1 Piggyback Loader Input

The input to the Piggyback Loader is the object code of the Absolute/Relocating Loader, which has been assembled as a relocatable program by the PTS-100 Assembler. The device from which the Piggyback Loader is to read the object code must be the same device whose address* has been assembled into the Piggyback Loader. The programer must ensure that the input device is operational and ready with the input object code before the Piggyback Loader is activated.

2.1.2 Piggyback Loader Output

The output of the Piggyback Loader is the activated Absolute/Relocating Loader, residing in high memory.

2.2 Absolute/Relocating Loader

The Absolute/Relocating Loader must be used to load all assembled systems and applications programs to be executed on the PTS-100. The programs to be loaded must have been assembled by the PTS-100 Assembler, which develops object coding in the format required by the Absolute/Relocating Loader. The object programs may be absolute or relocatable, and may consist of one or more segments each. If an execution address is specified in a given program or program segment, it must appear at the end of the last segment or program loaded, since the Loader will immediately activate the given program at the specified address as soon as it is detected. That is, the Loader will turn control over to the loaded program and start its execution at the specified address. If no execution address is specified, the Loader will wait for additional input to read or for a starting address to be specified manually.

If additional programs or program segments are to be loaded after the Loader has started execution of a loaded program, the Loader must be reinitialized in one of the following ways:

- The object program issues a call or transfers control (via an EXREF #LOADR statement) to the starting address of the Loader.

- The programer halts the current object program execution via the customer engineer's console or some other direct memory access device, enters the starting address of the Loader (#LOADR, as specified in the previous load map) as the new program counter setting, and restarts execution.

The Absolute/Relocating Loader performs the following processing for a given object program that is being loaded:

*On systems with changeable Read Only Memories (ROMs) the input device address will be determined by the first changeable word of the ROM.

- Loads all address constants and absolute values.

- Computes the effective addresses of all object program instructions.

- Relocates a relocatable program.

- Resolves addresses and prints a map of symbols named in the External Definitior (EXDEF) and External Reference (EXREF) statements to establish linkages between multiple programs or program segments.

- Sets blocks of memory locations to values specified by object coding.

- Performs validity checks of input data records and reports loading errors via an output listing.

- Stores the execution starting address, if specified, in the level zero interrupt packet, and starts execution of the program via an interrupt return instruction.

2.2.1 Absolute/Relocating Loader Input

The input to the Absolute/Relocating Loader is one or more object program files produced by the PTS-100 Assembler, described in Part 2, Section 5, of this handbook. The object program file(s) may be in one of the following forms:

punched cards

magnetic tape records

punched paper tape records

Hence, the input device must be an appropriate device to read the object code. The address of the device must have been assembled into the version of the Absolute/Relocating Loader being used.*

The order of applications object programs or program segments is program-determined. Certain systems programs, if present in the input device, must be loaded in prescribed orders. For example, the Interactive Debug program code should be loaded as the last program or program segment if it is to be initialized prior to starting execution of the programer's object program. On the other hand, if the object program(s) to be executed require input/output services from the IOCS monitor, the object code of the monitor should be loaded before any other programs. That is, the monitor is an absolute program, which is always loaded in low memory. Hence, if a relocatable program is loaded first, and the monitor is subsequently loaded, the monitor will be loaded over the first part of the earlier program.

2.2.2 Absolute/Relocating Loader Output

There are three types of output produced by the Absolute/Relocating Loader:

- The loaded executable program, residing in main memory.
- A symbol map.
- A listing of diagnostic messages signaling load errors detected during the attempt to load the object program.

The output device address must have been assembled directly into the version of the Absolute/Relocating Loader that is being used.

2.2.2.1 Symbol Map. The Absolute/Relocating Loader produces a listing of its own starting address, the program name(s), and all externally defined and referenced symbols in the program segment(s) it has loaded. A sample Symbol Map listing is shown in figure 3-2. Each symbol is listed with the memory address to which it was assigned by the Loader. That is,

*On systems with changeable ROM's the input device address will be determined by the first changeable word of the ROM.

Program/ Symbol Name	Location	Program/ Symbol Name	Location
# LOADR	3888	PROGN01	1020
USERLC1	0000 UD	USERLC2	1050
USERLC3	1055	USERLC4	1060
USERLC3	1055 DD		

Figure 3-2. Sample Symbol Map Produced by the Absolute/Relocating Loader

the listing indicates the location of program elements after they have been relocated. Any undefined symbols will be shown by zero characters in the location field, to which the characters UD (i.e., undefined) will be appended. Duplicate symbols will be shown with the address of the first definition of that symbol, and the characters DD (i.e., duplicate definition) will be appended to the location field.

If the symbol table created by the Loader is too large, a symbol overflow condition occurs, and the attempt to load the program is aborted. In this case, the symbol map is produced unconditionally to indicate the last symbol processed prior to the overflow condition. The symbol table overflow condition occurs because the Loader starts the symbol table in high memory, immediately preceding the first location used by the Loader itself, and builds it downward, toward the executable object code, which is built upward from lower memory. If the symbol table storage area reaches the program code area before the program is completely loaded, the Loader cannot complete the load process, which is aborted. If the program code area reaches the symbol table storage area during program load, a memory overflow error condition is declared and the load is aborted in the same way as for the symbol overflow condition.

2.2.2.2 Error Diagnostic Listing. The Absolute/Relocating Loader produces an error coded listing of any load errors that occurred during the attempt to load the object program. The error codes, their causes, and the required program actions are presented in table 3-1.

Table 3-1. Error Codes Output by the Absolute/Relocating Loader

Error Code	Cause	Programer Action
CK	A checksum (i.e., record read) error has occurred during the Loader's input record reading.	Effect a re-read of the input record by repositioning input tape to beginning, or refeeding object deck, and restarting load process from beginning.
RD	A read error has occurred during the Card Version Loader's attempt to read the last input record.	
SO	A symbol table overflow has occurred and the Loader process has been aborted.	Reduce number of symbols in the total program, reassemble, and reload.
MO	A memory overflow condition has occurred (i.e., the object program is too large for available memory) and the Loader process is aborted.	Reduce total program size, reassemble, and reload.
SQ	A record sequence number is out of order (i.e., not in order produced by the Assembler).	Correct sequence order of object code and restart the load process from the beginning.

Section 3. INTERACTIVE DEBUG PROGRAM

The Interactive Debug Program for the PTS-100 System allows the programer to actively interface with it during object program checkout to effect the following:

- Addition or subtraction of hexadecimal constants.
- Single or successive memory location dumps.
- Searches of memory locations for specific full word values or masked searches on values of less than 16-bits in length.
- Alterations of single memory location content to a specific value.
- Successive memory location loading with specific values.
- Breakpoint setting and clearing.
- Transfers of control to specific addresses and resumption of program execution.
- Transfers of control to specific addresses with the accumulator and/or one or both index registers set to specific values and resumption of program execution.
- Continuation of previously issued Debug commands.
- Input command editing.

Thus the programer is provided hands-on control of the execution of his program. This capability allows selective examination of memory, manipulation of memory words by accessing and altering them, selective execution of any part or all of the program, preparation of active unit tests, minor program patching, etc.

The Interactive Debug program requires the ASR device for input and output. If some other device is to be used for I/O, the RDS-supplied Interactive Debug program must be modified by replacing its ASR driver routines with the appropriate nonstandard driver routines.

The Debug program interrelates with the programer and the executing object program as to the functions it is to perform. Since it is a slave type program it waits for input once it is initialized. No timers are used and there are no restrictions placed on the length of time between commands or between parameter entries within commands. The types of functions performed and program interface with the Debug program are described in detail in the remainder of this section.

3.1 Inputs to the Interactive Debug Program

There are two basic inputs required to initialize the Interactive Debug process:

- The object code of both the object program and the Interactive Debug program, which are entered into the computer via the Absolute/Relocating Loader (see Section 2) from whatever input device is required to read the object code (i.e., card reader, cassette tape device, etc.).
- The interactive debug input commands, which are entered one at a time via the ASR device keyboard.

When the Absolute/Relocating Loader completes loading the Interactive Debug program, it activates Debug, which then performs its own initialization and indicates that it is ready to receive input commands by printing the word DEBUG at the ASR device. That is, the only programer action required to initialize Debug is to ensure that its object code is loaded. Depending on the equipment resources available in a particular PTS-100 configuration, there are a number of ways in which the Debug program may be loaded, as follows:

- On a standard PTS-100, the most efficient way to load the Debug program is to load its object code immediately following the object code of the program to be debugged. That is, the Debug program is treated as though it were the last segment of the programer's object program. In this case, the last statement of the Debug object code would specify the address at which Debug is to start executing. When the Debug program has been loaded and initialized, it will then notify the programer (via the Debug printout) that it is ready to receive input commands from the ASR device.

- If a customer engineer's console is available on the PTS-100 being used, the programer may load and start execution of his object program, then subsequently interrupt execution and initialize the Absolute/Relocating Loader to load and activate the Debug program.

- If the console is not available on the PTS-100 being used, the programer may program a transfer of control to the Absolute/Relocating Loader at the point at which Debug is to be loaded. That is, the programer may write a source program branch statement to cause a transfer of control to the starting address of the Absolute/Relocating Loader, thus setting up the mechanism to cause the Loader to read the Debug object code at the desired point in the object program.

In any case, once Debug has indicated that it is ready to receive commands, the programer may enter the desired Debug input commands described in subsection 3.1.1.

The outputs from Interactive Debug are in the form of hexadecimal printouts indicating responses to the input commands, as described for those commands that ellicit a Debug keyboard printout, and in the form of error messages, described in subsection 3.2 at the end of this section.

3.1.1 Interactive Debug Input Commands

There are six kinds of input commands:

- Keyboard editing commands, which provide the programer with the facility to correct typographical errors or edit input commands before transmitting them to the Debug program, as described in subsection 3.1.1.1.

- Memory value access commands, described in subsection 3.1.1.2.

- The program execution control command, Go To, which returns control to the executing program, as described in subsection 3.1.1.3.

- The address or location computation statements, Addition and Subtract, described in subsection 3.1.1.4.

- The Proceed command, described in subsection 3.1.1.5, which causes Debug to create a new command of the type just previous to its occurrence.

- The breakpoint control commands, Set Breakpoint and Clear Breakpoint, described in subsection 3.1.1.6.

In all cases input commands are terminated, and therefore transmitted to Debug, by a carriage return on the ASR device, or by the entry of 72 consecutive input characters.

All Debug program input commands must be specified in hexadecimal notation (i. e., 16-bit unsigned quantities).

The generalized format of all input commands is shown below:

```
PARAM1  FUNCTION CODE  PARAM2, PARAM3, PARAM4
```

where the command field significance is as follows:

Field 1: PARAM1 is the effective address to be used by Debug except in the special case of the Addition and Subtract commands.

Field 2: FUNCTION CODE is a single character indicating the operation Debug is to perform.

Fields 3-5: PARAM2, PARAM3, and PARAM4 are unique parameters requesting Debug special actions.

In all cases, input command elements are written without intervening spaces. The three parameters to the right of the function code must be separated by commas if they are all present; or if the first or second parameter is omitted, their omission must be indicated by an extra comma. The formats of individual commands are diagramed and illustrated in the specific detailed descriptions of each command in the remainder of this section.

3.1.1.1 <u>Keyboard Editing Commands</u>. There are two editing commands provided: the Cancel Record command, and the Logical Backspace command.

3.1.1.1.1 <u>Cancel Record command (/)</u>. The Cancel Record command is used to terminate a partially completed input command. The command is specified by typing a slash (/) character, which is AF in ASCII code. When the slash character is typed, Debug expects the first character of a new command to be entered. Examples of the cancel record command are shown below.

<u>Example 1</u>:

Programer input command: 1000D5/1010D

The programer specified that the entire command to the left of the / character was to be replaced by the command following the cancel record character. The printout below indicates Debug's response to the command:

Debug response: 1010*0000

<u>Example 2</u>:

Programer input command: 12/0000D
Debug response: 0000*0A00

The incorrect command 12 was cancelled and replaced by the command to dump location 0000, which was effective, as shown by Debug's response.

3.1.1.1.2 <u>Logical Backspace command (←)</u>. The Logical Backspace command is used to replace the preceding character with the following character. The backspace is specified by typing the ← character, which is the DF in ASCII code. When the backspace character is typed, the Debug program replaces the character just preceding it with the character immediately following it. Contiguous preceding characters may be replaced by typing contiguous backspace characters followed by the replacement characters. There are two restrictions on the use of this editing command:

1. Backspacing is limited to the current field being entered.

2. The backspace command cannot be used to override the slash (/) character (i. e., the cancel record command).

Examples of the use of the backspace command are presented below.

Example 1:

Programer input command: 1000D2←3

Changes the count field 2 to 3. Hence the Debug program dumps three locations, starting at location 1000, as shown below:

Debug response: 1000*D900 0826 4283

Example 2:

Programer input command: 1000G←D

Asks that the G function code be changed to D. Debug responds by dumping the value in location 1000, as shown below:

Debug response: 1000*D900

Example 3:

Programer input command:
1111←← ←←0000D

Changes the entire ADDRESS field from 1's to 0's, and adds the function code D.

Debug response: 0000*0A00

Example 4:

Programer input command:
1000A← B←C←D←F←G←AFFF

Specifies multiple corrections of the function code A, with the last correction specifying a function code of A and the value 0FFF to which location 1000 is to be altered, as shown in the printout

Debug response: 1000*000B 0FFF

3.1.1.2 Memory Value Access Commands. These commands direct the Debug program to perform the following operations on the values stored in the memory locations used by the executing object program:

- Dump the content of one or more locations.

- Alter the content of a location with a specific value.

- Fill one or more locations with a specific value.

- Search values in memory locations to find a specific value.

Detailed descriptions of the statements to effect these operations are presented in the following subsections.

3.1.1.2.1 Dump command. The Dump command is used to specify that a single memory word value is to be dumped on the output device, or to specify that successive word values are to be dumped, beginning at a specific location, as shown in the format diagram below.

FORMAT			SIGNIFICANCE
PARAM1	FC	PARAM2	
ADDRESS	D	Δ or 1	Specifies that the content of the memory word located at ADDRESS is to be printed on the output device.
ADDRESS	D	(count > 1)	Specifies that the content of a block of memory words is to be printed on the output device, starting with the value stored in ADDRESS.

If the single word Dump format is used (i.e., if no count or a count of 1 is specified), Debug prints the specified address and its content in hexadecimal notation on the output device, as illustrated in the examples below.

Example 1:

Programer input command: 0100D

Debug response: 0100*C204

Example 2:

Programer input command: 1000D1

Debug response: 1000*0281

The programer may specify that the single-word Dump command is to be repeated, as described under the Proceed command discussion in subsection 3.1.1.5.1.

If the multiword Dump format is used (i.e., if the count value is greater than 1) Debug prints the specified hexadecimal address of the first value, followed by a maximum of eight hexadecimal values per line on the output device. If more than eight values were specified, the address and the content of the ninth location are printed on the second line, followed by a maximum of seven additional memory word values. If other lines are required to output the specified number of values, the address of the first value appears at the beginning of each line, followed by successive values from the memory block locations. Examples of multiword Dump commands are presented below:

Example 1:

Programer input command: 1000D5

Debug response:

```
1000*C204 AA02 0300 0806 0108
```

Example 2:

Programer input command: 0100D20

Debug response:

```
0100*C204 AA02 0300 0806 0108 1010 03F3 20A0
0110*E300 02E3 AA02 0300 07F4 011A 042E 03F8
0120*9300 0300 C300 02D4
```

The following three exceptions occur in the Debug program's responses to multiword Dump commands.

Exception 1

If the starting address specified as PARAM1 in the Dump command does not end in zero, the location of the next lowest MOD 8 word is taken as the starting address of the memory block dump, in order to maintain column integrity. For example, assume the following Dump command, which specifies that four word values are to be dumped, beginning at location 0106:

```
0106D4
```

The response of the Debug program is:

```
0100*C204 AA02 0300 0806 0108 1010 03F3
```

where the fourth value from the right (0806) is from the programer-specified starting address 0106. Hence, the first three values were printed as a result of Debug's adjusting the starting address back to the next lowest MOD 8 word location.

Exception 2

If all the memory values to be printed on a line are identical, only the address and value of the starting location are printed. For example, assume the Dump command

```
1000D6
```

which specifies that the values from six locations are to be dumped, starting at location 1000. If all six locations contain the same value, Debug's response will be:

```
1000*0281
```

Exception 3

If all memory values to be printed on several successive lines are identical, the output lines are suppressed until a line containing an unequal value is detected by Debug, at which point the location and value are printed at the beginning of the line, followed by subsequent values to complete the line or the dump request. For example, the command

```
1000D40
```

asks that the values stored in 40 consecutive locations are to be dumped, starting at location 1000. The Debug response below

```
1000*0281
1020*0281 0281 1002 1002 0281 0281 0281 0281
1030*AAAA AAAA 0281 0281 0281 0281 0281 0281
```

indicates that the first 20 locations contained the identical value 0281. Beginning at locations 1020 and 1030 in the memory block, unequal values were detected in the output lines.

3.1.1.2.2 Search command. The Search command directs the Debug program to search the values in memory locations, compare them to a programer-specified value, and print the addresses and values when equal conditions result from the comparisons. The Search command may specify a search on a single memory location, a series of successive locations, or a series of successive locations whose stored values are masked before the comparison with the specified value. The permissible formats of the Search command are diagrammed below:

FORMAT			SIGNIFICANCE
PARAM1	FC	PARAM2, PARAM3, PARAM4	
ADDRESS	S	VALUE Δ or VALUE, 1	Specifies that the memory word located at ADDRESS is to be compared to VALUE and dumped on the output device if the values are equal.
ADDRESS	S	VALUE, COUNT	Specifies that memory will be searched starting at ADDRESS, and each word will be compared to VALUE and dumped if the comparison results are equal. The search will terminate after the number of words specified by COUNT.
ADDRESS	S	VALUE, COUNT, MASK	Specifies that memory will be searched starting at ADDRESS, and each word logically ANDed with MASK, the result compared to VALUE, and equal values and their addresses dumped. The search will terminate after the number of words specified by COUNT.

If the single word Search format is used (i. e., if no count or a count of 1 is specified), Debug compares the programer-specified VALUE with the content of the memory address, and if the values are equal, prints the address and the memory value in hexadecimal notation on the output device.

The programer may specify that the single-word search command is to be repeated as described under the Proceed command discussion in subsection 3. 1. 1. 5. 2.

When the multiword Search command is issued, Debug searches each location from the starting ADDRESS through the specified COUNT, compares each stored value with VALUE, and reports each match via a hexadecimal printout of the memory address and value on the output device.

When the MASK parameter is specified in the multiword Search command, Debug searches each location from the starting address through the specified COUNT, logically ANDs each stored value with MASK, compares the result with VALUE, and reports each match via a hexadecimal printout of the memory address and its value on the output device.

Example 1:

Programer input command: 1000SF000, 8

This command specifies that Debug is to search eight locations, beginning at location 1000, for a stored value of F000 and report a match if it is found.

Debug response: 1008*F000

The response indicates that Debug found a stored value matching VALUE at location 1008.

Example 2:

Programer input command: 1000SFFFF, 8

The command specifies that eight locations are to be searched, beginning at location 1000, to determine if the value FFFF is stored anywhere within the memory block.

Debug response: 100E*FFFF

indicates that the value was stored at location 100E.

Example 3:

Programer input command: 1000S3, 8

Asks Debug to search eight locations, starting with location 1000, for the value 3. A line feed without an accompanying printout indicates that the value 3 was not stored within the eight locations specified for searching.

Example 4:

Programer input command:
1000S8000, 8, 8000

Asks Debug to search eight memory locations, starting at location 1000, mask their contents, compare the results of the AND mask with 8000 to determine if the most significant bit (MSB) is set. Assume the following response from the Debug program:

```
1008*F000 100A*FF00 100C*FFF0 100E*FFFF
```

A match is found at locations 1008, 100A, 100C, and 100E.

Example 5:

Programer input command: 1000S1, 8, 000F

Asks Debug to search eight memory locations, starting at location 1000, mask their stored values with 000F, and report any matched values when compared to 1. The search operation is to test for addresses whose contents have their LSB and not bits 12, 13, 14 set. The Debug printout

```
1006*0001
```

indicates that a match was found at location 1006.

3.1.1.2.3 Alter command. The Alter command directs the Debug program to alter, or replace, the memory word value in a specific location with the value specified in the command, the format of which is shown below.

FORMAT			SIGNIFICANCE
PARAM 1	FC	PARAM 2	
ADDRESS	A	VALUE	Specifies that the content of the memory location indicated by ADDRESS is to be replaced by the VALUE in parameter_2 of the command.

When the command is processed by the Debug program, the specified VALUE replaces the original content of the ADDRESSed location. The ADDRESS, the original value stored there, and the new value are printed in hexadecimal notation on the output device.

The programer may specify that the Alter command is to be repeated for the next consecutive location(s), as described under the Proceed command discussion in subsection 3.1.1.5.1. To alter or fill blocks of memory locations with new values, the programer should use the Fill command described below.

3.1.1.2.4 Fill command. The Fill command directs the Debug program to fill one or more memory locations with a specific value, as shown in the format diagram below:

If the single word Fill command format is used (i.e., if no count or a count of 1 is specified) the Debug program fills the ADDRESSed location with the specified VALUE.

Under Interactive Debug, the programer may specify that the single word Fill command is to be repeated for the next consecutive location(s), as described under the Proceed command discussion in subsection 3.1.1.5.1. If the multiword Fill format is used, the Debug program fills each location from the starting ADDRESS through the COUNT with the specified VALUE.

Neither format of the Fill command causes printed output at the interactive device. If the programer wishes to verify that the Fill operation is successful, he should dump the pertinent memory locations before and after issuing the Fill command.

3.1.1.3 Go To Command. This command directs the Debug program to transfer control to a specific address in the object program, and start object program execution at that address. The permissible formats of the Go To command are presented in Table 3-2.

As shown in table 3-2, the Go To command may optionally specify that new values are to be loaded into any one, two or all three of the following registers: accumulator, index register 1, and index register 2. If values are to be loaded into the respective registers, they must be specified in the appropriate order: AC value, X1 value, and X2 value. If either the AC value

FORMAT			SIGNIFICANCE
PARAM1	FC	PARAM2, PARAM3	
ADDRESS	F	VALUE Δ or VALUE, 1	Specifies that the ADDRESSed location is to be filled with VALUE
ADDRESS	F	VALUE, COUNT	Specifies that successive memory locations, specified by COUNT, are to be filled with VALUE. ADDRESS specifies the first, or starting, location of the memory block.

Table 3-2. Permissible Formats for Go To Command

FORMAT			SIGNIFICANCE
PARAM1	FC	PARAM2, PARAM3, PARAM4	
ADDRESS	G		Specifies that Debug is to transfer control to ADDRESS in the object program and start its execution.
ADDRESS	G	ACVAL, X1VAL, X2VAL	Directs Debug to load specified VALS in the AC, X1, and X2 and transfer control to and start execution of the object program at ADDRESS.
		ACVAL	Directs Debug to load specified VAL in AC, leave X1 and X2 unmodified, and start object program execution at ADDRESS.
		ACVAL, X1VAL	Directs Debug to load specified VALs in AC & X1, leave X2 unmodified, and start execution at ADDRESS.
		ACVAL,, X2VAL	Directs Debug to load specified VALs in AC & X2, leave X1 unmodified, and start execution at ADDRESS.
		, X1VAL, X2VAL	Directs Debug to load specified VALs in X1 & X2, leave AC unmodified, and start execution at ADDRESS.
		, X1VAL	Directs Debug to load specified VAL in X1, leave AC and X2 unmodified, and start execution at ADDRESS.
		,, X2VAL	Directs Debug to load X2 with specified VAL, leave AC & X1 unmodified, and start execution at ADDRESS.

or the X1 value or both values are unspecified, their respective omission must be indicated by a comma. That is, Debug assumes that the first value following the G function code is to be loaded into the accumulator, the second in index register 1, and the third in index register 2. Two or more values must be separated by commas. Trailing commas are not required for unspecified values. Examples of Go To commands are presented below.

Example 1:

Programer input command: 1000G1,,1

Debug reponse: loads AC with the value 1, leaves X1 unmodified (specified by the second comma), loads X2 with the value 1, transfers control to location 1000 in the object program, and starts its execution.

Example 2:

Programer input command: 1010G, 1, 2

Debug response: loads X1 with the value 1, X2 with the value 2, leaves AC unmodified, and starts execution of the object program at location 1010.

Example 3:

Programer input command: 1000G, , 2

Debug response: loads X2 with the value 2, leaves both AC and X1 unmodified, and starts object program execution at location 1000.

Example 4:

Programer input command: 1000G, 3

Debug response: loads X1 with the value 3, leaves AC and X2 unmodified, and starts object program exeuction at location 1000.

Example 5:

Programer input command: 1000G6

Debug response: loads AC with the value 6, leaves X1 and X2 unmodified, and starts program execution at location 1000.

Example 6:

Programer input command: 1000G

Debug response: transfers control and starts execution of the object program with AC, X1, and X2 containing their original values.

3.1.1.4 Address Computation Commands. To aid in on-line testing and checkout of object programs, the capability to add or subtract two hexadecimal constants and output the result has been provided via the Addition and Subtract commands. These statements assist the programer in computing absolute locations in relocatable programs, or in computing the absolute location of a data word based on a program counter relative instruction referencing that data word. The following pages present detailed descriptions of the use of the Addition and Subtract commands.

3.1.1.4.1 Addition command. The Addition command specifies that the Interactive Debug program is to add one hexadecimal constant to another, as shown in the format diagram below. The constants may be from one to four digits in length. Leading zeros need not be written in constants less than four digits long.

FORMAT			SIGNIFICANCE
PARAM1	FC	PARAM 2	
CON1	+	CON2	Directs Debug to add the left constant to the right constant.

NOTE

The Addition code is the plus (+) sign, which is the shifted semicolon on the ASR device.

When the Addition command is issued, the Debug program adds the first constant (CON1) to the second constant (CON2) and reports the results preceded by an equal (=) sign, as illustrated below.

Example 1:

Programer input command: 1000+1

Debug response: =1001

Example 2:

Programer input command: 1000+1234

Debug response: =2234

Example 3:

Programer input command: 1+1

Debug response: =0002

Example 4:

Programer input command: 8000+1

Debug response: =8001

Example 5:

Programer input command: FFFF+1

Debug response: =0000

Example 6:

Programer input command: 7FFF+1

Debug response: =8000

Example 7:

Programer input command: 100+1000

Debug response: =1100

Example 8:

Programer input command: ABCD+F

Debug response: =ABDC

Example 9:

Programer input command: FFFF+FFFF

Debug response: =FFFE

The Proceed command (subsection 3.1.1.5.3) may be used following an Addition command to supply different CON2's to be added to the original CON1, and thus perform other Addition computations.

3.1.1.4.2 Subtract command. The Subtract command specifies that the Interactive Debug program is to subtract one hexadecimal constant from another, as shown in the format diagram below. The constants may be from one to four digits in length. Leading zeros need not be written in constants less than four digits long.

FORMAT			
PARAM1	FC	PARAM2	SIGNIFICANCE
CON1	-	CON2	Directs Debug to subtract the right constant from the left constant.

When the Subtract command is issued, the Debug program subtracts the second constant (CON2) from the first constant (CON1) and reports the results, preceded by an equal (=) sign, as illustrated below.

Example 1:

Programer input command: 0-1

Debug response: =FFFF

Example 2:

Programer input command: 1-1

Debug response: =0000

Example 3:

Programer input command: 2-1

Debug response: =0001

Example 4:

Programer input command: FFFF-1

Debug response: =FFFE

Example 5:

Programer input command: FFFF-FFF

Debug response: =F000

Example 6:

Programer input command: 8000-1

Debug response: =7FFF (overflow condition)

Example 7:

Programer input command: FFFF-7FFF

Debug response: =8000

Example 8:

Programer input command: 0-CA

Debug response: =FF36

Example 9:

Programer input command: ABCD-AAAA

Debug response: =0123

The Proceed command, described below, may be used following a Subtract command to supply different CON2's to be subtracted from the original CON1 and thus perform other Subtract computations.

3.1.1.5 Proceed Command. The Proceed command specifies that the Debug program is to refer to the input command just preceding it (i.e., the most recent command specifying a command code other than P) and create a new command of the same type with different parameters to the right of the function code. That is, the previous effective address and function code are used with the new parameters. The format of the Proceed command depends, of course, on the particular Debug input command it immediately follows, as shown in the following subsections. The Proceed command is not effective after a Set Breakpoint or Clear Breakpoint command.

3.1.1.5.1 Proceed command use after Alter, Dump, and Fill commands. After execution of a single word Alter, Dump, or Fill command, the Debug program increments the ADDRESS specified in the command by 2 and saves it. If the next sequential command is a Proceed (i.e., begins with function code P), the Debug program performs an Alter, Dump, or Fill operation, as specified by the new parameter(s) following the P function code, as shown in table 3-3 and illustrated in the examples at the end of this subsection.

Example 1:

Assume that the most recent Debug input command is an Alter command: 1000AF0F0

The Proceed command below

PFFFF

tells Debug to fill location 1002 (i.e., the saved Alter ADDRESS+2) with the value FFFF.

Example 2:

Assume that a second Proceed command follows the one in Example 1, which specifies the following: PAAAA

The command specifies that Debug is to fill the location following 1002+2 (i.e., 1004) with the value AAAA.

Example 3:

Assume that the most recent Debug input command is the Dump command

0100D

after which the Proceed command

P

causes location 0102 (Dump ADDRESS+2) to be dumped, as follows: 0102*AA02

Table 3-3. Effect of Proceed Command After Alter, Dump or Fill Commands

PREVIOUS COMMAND	PROCEED COMMAND FORMAT		SIGNIFICANCE
	FC	PARAM1, PARAM2	
ALTER	P	VALUE1	First Proceed command causes VALUE1 to be stored in ADDRESS+2, which was saved after preceding ALTER command execution, and a printout of ADDRESS+2, the original value and the new value to be printed out.
	P	VALUE2	Second Proceed command causes ADDRESS+2 to be incremented by 2, VALUE2 to be stored in ADDRESS+4, and a printout of ADDRESS+4 and its original and new value.
DUMP	P	Δ or 1	The first Proceed following DUMP causes a dump of the stored value in ADDRESS+2, which was saved after the Dump command execution.
	P	Δ or 1	Second Proceed causes ADDRESS+2 to be incremented by 2 and the stored value of ADDRESS+4 to be dumped.
	P	(count > 1)	Causes the ADDRESS+2 saved after Dump execution to be used as the starting address and to be sequentially incremented to dump the number of consecutive locations specified by count. The starting location and stored values in the block of locations are dumped.
FILL	P	VALUE 1Δor VALUE1, 1	The first Proceed following FILL causes VALUE1 to be stored in ADDRESS+2, which was saved after Fill command execution.
	P	VALUE2Δ or VALUE2, 1	Second Proceed command causes ADDRESS+2 to be incremented and VALUE2 to be stored in ADDRESS+4.
	P	VALUE, (count>1)	Causes the ADDRESS+2 saved after FILL execution to be used as the starting address and to be sequentially incremented to fill the number of consecutive locations specified by count with VALUE.

NOTE
All addressing is in bytes.

Assume that the programer now wants to dump the value in the next location, which can be effected by the P command

P

which causes the Debug response

```
0104*0300
```

In the same manner, the programer can dump the next two locations by issuing the Proceed command

P2

which causes Debug to begin the dump at the original Dump address 0100, as shown by the printout

```
0100*C204 AA02 0300 0806 0108
```

Example 4:

Assuming that the most recent Debug input command is the Dump command

```
0100D
```

the ten locations starting at 0100 can be dumped by issuing the Proceed command

```
PA
```

The dump is as follows:

```
0100*C204 AA02 0300 0806 0108 1010 03F3 20A0
0110*E300 02E3 AA02
```

Example 5:

Assuming that the most recent input command is the Fill command

```
1000F9999
```

The following Proceed commands illustrate a method of program patching:

1002F1234,2	fills locations 1002 and 1004 with the value 1234
P4321	fills location 1004 with the value 4321
PABCD	fills location 1006 with the value ABCD
PFFFF,8	fills location 1008 through 1016 with FFFF

3.1.1.5.2 Proceed command use after Go To and Search commands. When a Go To or Search command is executed by Debug, the execution ADDRESS is saved. If the next sequential command is a Proceed (i.e., begins with a P function code) the Debug program uses the saved address to create a new command with the new values specified to the right of the P function code, as shown in table 3-4.

The Proceed command may be used to create a new Go To command only when both of the following conditions exist:

- The object program has received control from Debug as the result of a programer-issued Go To command.
- Control has subsequently been returned to Debug because the object program encountered a breakpoint address.

If the above conditions are met, the programer may issue a Proceed command to cause control to be returned to the ADDRESS specified in the original Go To command just preceding the return of control to Debug. The Proceed command may optionally specify one or more new values to be loaded into the accumulator, X1, and/or X2. If no new values are specified, the transfer from Debug to the object program takes place without modification of the registers.

If a Proceed command follows a Search command (function code S) the starting address of the search operation will be the same as in the original Search command, but new values must be entered following the P code.

3.1.1.5.3 Proceed command use after Addition and Subtract commands. When the Interactive Debug program is being used, the Proceed command may be used following the Addition or Subtract command by merely entering the P function code followed immediately by a new CON2 value. That is, when an Addition or Subtract command is executed by Interactive Debug, the CON1 value to the left of the + or - function code is saved. Hence, if the next sequential command is a P command, Debug performs the specified operation using the original CON1 value and the new CON2 value specified in the Proceed command. The format, then, for the Proceed command following an Addition or Subtract command is:

```
PCON2
```

Table 3-4. Effect of Proceed Command After Go To or Search Commands

PREVIOUS COMMAND	PROCEED COMMAND FORMAT		SIGNIFICANCE
	FC	PARAM1, PARAM2, PARAM3	
GO TO	P		Specifies that Debug is to transfer control to the original Go To ADDRESS and start program execution without modifying any registers.
	P	ACVAL, X1VAL, X2VAL	Specifies that Debug is to load new values in the AC, X1, and X2 and start object program execution at the ADDRESS specified in original Go To command.
	P	ACVAL	Directs Debug to modify the current value in the AC with the new value and return control to ADDRESS specified in preceding Go To command.
	P	ACVAL, X1VAL	Directs Debug to load specified VALues in the AC and X1, leave X2 unmodified, and start program execution at original Go To ADDRESS.
	P	ACVAL, , X2VAL	Directs Debug to modify AC and X2 with new VALues, and return control to ADDRESS specified in preceding Go To command.
	P	, X1VAL, X2VAL	Directs Debug to load X1 & X2 with new VALues, leave AC unmodified, and return control to preceding Go To ADDRESS.
	P	, X1VAL	Directs Debug to load X1VAL in X1, leave AC and X2 unmodified, and return control to ADDRESS specified in original Go To.
	P	, , X2VAL	Directs Debug to load X2 with the specified X2VAL, leave AC and X1 unmodified, and return control to original ADDRESS specified in preceding Go To command.
SEARCH	P	VALUE△ or VALUE1	Specifies that Debug is to search the memory word located at the ADDRESS originally specified in preceding Search command, compare its content with VALUE, and print the ADDRESS and value if an equal condition is met.
	P	VALUE, COUNT	Specifies that the memory locations specified by COUNT are to be searched, starting at ADDRESS specified in preceding Search command, compared to VALUE, and any equal values are to be reported on the output device.
	P	VALUE, COUNT, MASK	Specifies that Debug is to search the memory locations specified by COUNT, starting at the ADDRESS specified in the preceding Search command, logically AND each stored value with MASK, compare the results with VALUE, and report equal values and their address via a printout on the output device.

Following are examples of Proceed command use and the results obtained.

Example 1:

Proceed following Addition command

```
Programer input command:  1000+1
Debug response:  =1001
Programer input command:  P2
Debug response:  =1002
Programer input command:  PFF
Debug response:  =10FF
```

Example 2:

Proceed following Subtract command

```
Programer input command:  0-1
Debug response:  =FFFF
Programer input command:  P2
Debug response:  =FFFE
Programer input command:  PF
Debug response:  =FFF1
```

3.1.1.6 Breakpoint Control Commands. The breakpoint control commands are Set Breakpoint and Clear Breakpoint.

3.1.1.6.1 Set Breakpoint command. This command is used to set from one to four breakpoint addresses within the object program. The Set Breakpoint command format is shown below.

FORMAT		
PARAM1	FC	SIGNIFICANCE
ADDRESS	B	Specifies the address at which a breakpoint is to be set in the object program.

There are two restrictions on the use of the Set Breakpoint command:

- No more than four breakpoints may be in effect at a given time in the executing program.
- Breakpoint addresses must not specify the second word of a two word (long format) executable instruction.

When a Set Breakpoint command is issued, the Debug program performs all of the following:

- Saves a 5-word block of code in the object program, beginning at the specified ADDRESS.
- Overlays the saved object program code area with a 5-word Breakpoint Transfer Block (BTB) to effect a transfer of control to Debug when the breakpoint address is encountered in the object program.
- Waits for the next programer input command.

When a breakpoint address is encountered in the executing program, control transfers to the Debug program which performs the following:

- Prints the breakpoint address, and the current values of the accumulator, X1, and X2.
- Waits for the next programer input command.

In all cases, to return to the object program, the programer must issue a Go To command.

3.1.1.6.2 Clear Breakpoint command. The Clear Breakpoint command, formated below, clears a breakpoint previously set by the Set Breakpoint command.

FORMAT		
PARAM1	FC	SIGNIFICANCE
ADDRESS	C	Specifies that the breakpoint previously set at ADDRESS is to be cleared.

When a Clear Breakpoint command is issued, the Debug program clears the breakpoint set for the specified address, restores the saved object program's code in the area currently occupied by the BTB associated with the breakpoint, and waits for the next input command. To return to the object program, the programer must issue a Go To command.

3.2 Debug Output Error Messages

When an error in processing or data entry is detected by Debug a two character error message is logged on the list device. Debug's recovery procedure is to terminate processing the current input command, log the error code on the ASR device, issue a line feed and carriage return, and initiate input of another input command. No data from erroneous input commands is saved. The user may reenter that command correctly, or enter any other valid input command. The error codes and their signficance are shown in table 3-5.

Table 3-5. Interactive Debug Error Messages

Message Code	Meaning
BF	Input buffer full; more than 72 characters in input record.
BP	Clear Breakpoint command given for ADDRESS not currently in breakpoint stack.
FC	Function Code or Field Delimiter omitted from input record.
HX	Non-Hex number found in Hex Data Field.
P1	Parameter 1 omitted from a command requiring it be supplied.
P2	Parameter 2 omitted from a command requiring it be supplied.
P3	Parameter 3 omitted from a command requiring it be supplied.
S1	System Error - reload or maintenance required.
S2	System Error - reload or maintenance required.
WA	Least significant bit set in parameter 1, indicating effective address on a non-word boundary.
4B	Set Breakpoint command issued after the maximum number have been stored in the breakpoint save stack.

Section 4. SYSTEM GENERATOR PROGRAM

The System Generator (SYSGEN) program provides for the generation of a specially-tailored PTS-100 IOCS monitor to meet unique applications processing requirements. That is, for any given PTS-100 installation, a specialized IOCS monitor can be generated by describing its content to the SYSGEN program. The system descriptions are supplied on key word directive cards, which are input to the SYSGEN program. There are six key word directives: TITLE, ACIC, ASGL, ASGP, CALL, and END. These directives are described individually in the subsections immediately following.

4.1 Command Directives

The inputs to SYSGEN are command directives containing key words and the parameters necessary to describe the system to be generated. Input on punched cards, the six directives are:

- One TITLE directive, which is used to assign a name to the system being created, and to specify PTS-100 Assembler file assignments and assembly options.

- One optional ACIC directive, which is used only when a channel interface controller (CIC) is attached to the object PTS-100. The ACIC directive specifies the number of devices to be attached to the CIC, the tumble table to be used, the base device address, and the channel address of each group of 16 devices attached to the CIC.

- One ASGL directive, which is used to assign physical I/O devices to logical units whose names appear in IOCS tables.

- One or more ASGP directives, which are used to assign hardware addresses to physical I/O devices, and to specify the interrupt levels to which the devices are to be assigned.

- One or more CALL* directives, which are used to insert macro calls to effect specialized IOCS monitor routine creation by the PTS-100 Assembler when the generated system is assembled.

- One END directive, which terminates the input directive processing.

Within the input card deck, the directives must appear in the order in which they are listed above. Except for the ASGP and CALL directives, only one directive of a given type may appear in the input deck. The specifications for writing the individual directives and their parameters are described and illustrated in the following subsections.

4.1.1 TITLE Directive

One TITLE directive is used to establish the information the PTS-100 Assembler requires for its options record when a given generated system is assembled. The directive specifies three types of information:

- The system name, which must be specified. The name is eight characters in length.

- The file (i.e., device) assignments for those devices to be used by the Assembler, if other than default device assignments are to be used (see Section 5 of Part 2 of this handbook).

- The assembly options desired.

The TITLE directive format is illustrated in figure 3-3.

---

*The CALL directive permits users to call their own generalized IOCS monitor macro routines, defined to satisfy applications-unique requirements such as nonstandard device driving and servicing routines. The source code of called routines must have been stored on the System Macro Library file to be used as input to the Assembler.

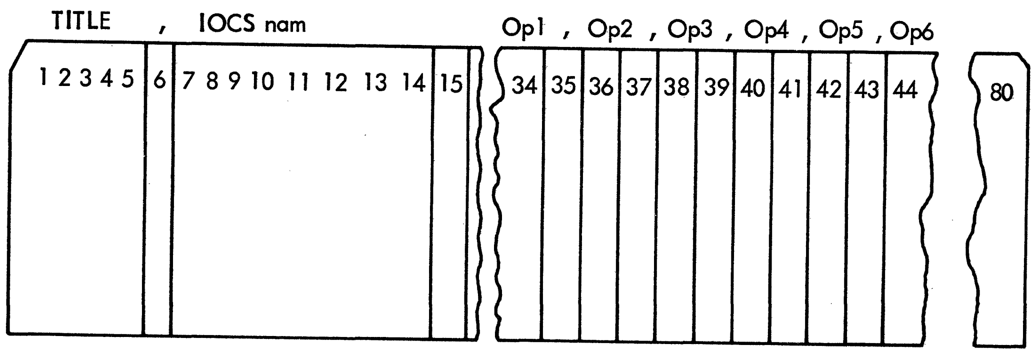


Figure 3-3. Format of the TITLE Directive

The content of the individual fields of the TITLE directive is presented below.

Field 1: TITLE must appear in columns 1-5, followed by a comma in column 6

Field 2: The system name must appear in columns 7 - 14, followed by a comma in column 15.

Fields 3 - 8: Assembler options (Op1 through Op6) may be specified as shown below.

4.1.2 ACIC Directive

One ACIC directive is used when, and only when, the channel interface controller (CIC) is to be used on the PTS-100 for which a given IOCS monitor is being generated. When used, the ACIC directive must follow the TITLE directive in the SYSGEN input deck. It specifies the information needed by the IOCS monitor to support the CIC and its attached devices. The ACIC directive is written in the format

$$\text{ACIC, nn, T, BDA, CAD}_1\text{, CAD}_2\text{, CAD}_3\text{, CAD}_4$$

OPTION	SELECTION	TITLE CARD	
		Content	Column
Op1	Cross reference listing No cross reference listing (default)	1 Δ	34
Op2	Sequence checking No sequence checking (default)	1 Δ	36
Op3	Macros included No macros included (default)	Δ 1	38
Op4	Relocatable object text (default) Absolute object text	Δ 1	40
Op5	Full listing, macros expanded (default) Full listing, macros not expanded Error listing only No listing	Δ 1 2 3	42
Op6	Machine language produced (default) No machine language produced	Δ 1	44

where:

nn specifies the total number of devices attached to the CIC, and must be one of the following: 16, 32, 48, or 64.

T specifies the tumble table to be used.

BDA specifies the base device address to be used.

CAD_1 - CAD_4 specifies channel addresses, one of which must be specified for each group of 16 devices attached to the CIC.

A minimum of one and a maximum of four groups of devices may be attached to the CIC. That is, the number of groups attached and the CADs required relates to the nn specification as shown below:

nn	Number of Groups	Channel Address(es) (CADs) Required
16	1	CAD_1
32	2	CAD_1, CAD_2
48	3	CAD_1, CAD_2, CAD_3
64	4	$CAD_1, CAD_2, CAD_3, CAD_4$

As shown above, unnecessary CADs in the ACIC directive may be left blank.

If, however, more than one group of devices is to be attached to a given channel, the channel address must be specified once for each group attached. For example, the directive

```
ACIC,64,1,F40,F80,F80,F80,F80
```

specifies that all 64 devices (i.e., four groups) are to be attached to channel address F80.

4.1.3 ASGL Directive

For a given system, one ASGL directive is used to specify the assignment of physical I/O devices to the logical units whose pointers appear in the IOCS input/output control table (IOCT). The directive contains ASGL in columns 1 - 4 of the input card, followed by the physical device notations to be used as assignments to logical units in the IOCT. The device notations are separated by commas, and are written in the format illustrated below:

```
ASGL,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13
```

where ASGL must appear in columns 1 - 4 followed by a comma in column 5, and the Ds are device notations which have the following significance in the order in which they appear:

D1	device assigned for SYSF (System File)
D2	device assigned for SYSI (System Input Device)
D3	device assigned for SYSL (System Logging Device)
D4	device assigned for SYSD (System Data Device)
D5	device assigned for SYSB (Binary Output)
D6	device assigned for SYST (System Listing Device)
D7	device assigned for SYSO (System Output Device)
D8	device assigned for SYSR (System Scratch Device)
D9	device assigned for LOG8 (Logical Unit 8)
D10	device assigned for LOG9 (Logical Unit 9)
D11	device assigned for LOGA (Logical Unit A)
D12	device assigned for LOGB (Logical Unit B)
D13	device assigned for LOGC (Logical Unit C)

NOTE

If a device is not to be assigned, the omission of its notation must be indicated by a series of three zeros.

The device notations that are permissible for various device assignments are shown in table 3-6.

Table 3-6. Device Notations for Use as SYSGEN Directives

Notation	Associated Device
AMn	Asynchronous Modem
CAn	Cassette Unit
CIn	Channel Interface Controller
CMn	Card Reader Multiplexed
CPn	Card Punch
CRn	Card Reader
DKn	Disc Unit
DR	Display Keyboard Receive
FPn	High Speed Paper Tape Punch
FRn	High Speed Paper Tape Reader
KRn	ASR Keyboard Receive
KSn	ASR Keyboard Send
LP	Line Printer
MHn	Modem Half Duplex (synchronous)
MMn	Modem Multiplexed
MRn	Modem Receive (PARS)
MSn	Modem Send (PARS)
PMn	Printer Multiplexed
SRn	Special Display Keyboard Receive
SPn	Serial Printer
TPn	Magnetic Tape Unit

NOTE

n is a decimal integer to specify a particular device where multiple devices of the same type appear in the equipment configuration.

4.1.4 ASGP Directive

This directive is used to specify the identification, hardware address, interrupt level, and macro processor special code for each physical I/O device to be used by the system being generated. The parameters specified on ASGP directive cards are used to complete the IOCT, started with the ASGL directive parameters, to build other tables (e.g., the PIOT), in the IOCS monitor, and to specialize the interrupt packets, level service routines, and the driver and service routines for each assigned device. The ASGP directive format is as follows:

$$\text{ASGP}, D_1\text{ID}, D_1\text{AD}, D_1\text{IL}, D_1\text{SC}, D_1\text{IM}, \text{SEN}, \ldots,$$

$$D_n\text{ID}, D_n\text{AD}, D_n\text{IL}, D_n\text{SC}, D_n\text{IM}, \text{SEN}$$

The ASGP key word must appear in card columns 1 - 4, followed by a comma in column 5, which is followed by a string of parameters, separated by commas. The parameter strings are composed of sets of parameters, with six parameters in a precise order required in any given set, as follows:

DID, DAD, DIL, DSC, DIM, SEN

where:

DID is one of the following:

the device identifier, specified as the appropriate notation from table 3-6, when only one device of this type is attached to a multiplex controller, or

a comma, indicating that multiple devices of this type are attached to the same multiplex controller, as specified by the DIM parameter in position 5 of the parameter list.

DAD is the device hardware address assigned to the device identified by DID or DIM.

DIL is the external interrupt priority level (level 1 through 8, as shown in table 3-7) to which the device is to be assigned.

DSC is a two character code which may be used to effect further specialization of the particular device's interrupt handling, service, and driver routines by the macro processor of the PTS-100 Assembler.

Table 3-7. Interrupt Priority Levels in the PTS-100

Interrupt Level	Interrupt Type	
10	Parity	
9	Trap	
8	External 8	Device Interrupt Levels
7	External 7	
6	External 6	
5	External 5	
4	External 4	
3	External 3	
2	External 2	
1	External 1	
0	Processor/Interval Timer	

DIM is one of the following:

the device identifier, specified as the appropriate multiplexed identifier notation from table 3-6, when more than one device of the same type is attached to the same multiplex controller, or

a comma if only one device of this type is attached to a given multiplex controller.

SEN is a sentinel, which must be one of the following:

an L to indicate the last device type that is attached to the same multiplex controller, or

a comma if DID is specified in the parameter list or if this is not the last device of an identical type attached to the same multiplex controller.

A special code is used in the generalized #IMLSR macro routine in the #IMPIT macro routine that creates PIOTs for each I/O device. That is, the parameters within each set in the ASGP directive become actual arguments in the call to the #IMLSR and other macro routines. The special code may be any two character value the programer wishes to use, but must be different in each parameter set. It must be specified as the fourth parameter in each set of parameters for a specific device.

The DID and DIM parameters should not both appear in the same parameter set. That is, DID specifies that only one device of this type is attached to a given multiplex controller, and will cause a unique set of device handling routines to be created in the IOCS monitor. The DIM parameter specifies that more than one device of this type is attached to a given multiplex controller, and will cause a common set of device handling routines for this type of device to be created in the IOCS monitor.

Any number of ASGP directives may appear in succession in the input deck. Each directive begins with ASGP, followed by a set of parameters.

To illustrate the use of the ASGP directive, assume the following two cards:

```
ASGP, CR1, 01A, 3, 00,,,
ASGP, CA1, 025, 1, 0X,,,
```

The first set of parameters specifies the following:

CR1 is the identifier for card reader device 1

01A is the hardware address to be assigned to card reader 1

3 is the external interrupt priority level to which card reader 1 is to be assigned

00 is the default special code

NOTE

Since the device identifier is specified in position 1 of the parameter set, the DIM and SEN parameters in positions 5 and 6 are omitted, as indicated by commas in these positions.

The second set of parameters specifies the following:

CA1 is the identifier for cassette unit 1

025 is the hardware address to be assigned to cassette unit 1

1 is the external interrupt priority level to which cassette unit 1 is to be assigned

0X is a special code to be used by the Assembler in specializing the IOCS routines

The omission of the DIM and SEN parameters is indicated by commas in their positions in the parameter set.

The following example illustrates the four ASGP directive cards necessary to specify four multiplexed serial printers:

```
ASGP,PM1,018,2,00,,,
ASGP,,0C0,2,00,PM1,,
ASGP,,010,2,00,PM1,,
ASGP,,014,2,00,PM1,L
```

The parameters on the first card specify the following:

PM1 is the identifier for the first serial printer to be multiplexed.

018 is the hardware address to be assigned to the first serial printer.

2 is the external interrupt priority level to which the first serial printer is to be assigned.

00 is the default special code.

Positions 5 and 6 are omitted (see Note on previous page).

The parameters on the second card specify:

The comma in position 1 indicates that multiple printers are attached to the same printer driver and service routine as specified in position 1 of card 1.

The next three positions are the same as on card 1, except for changes in the parameter values.

PM1 is the identifier for the device that is being multiplexed.

Card 3 is similar to card 2, with the parameter values changed. Card 4 is also similar except that position 6 contains an L to signify that this is the last device of this type.

As many as 22 sets of parameters may be specified for a given system generation. That is, a maximum of 22 devices may be entered into a system to be generated by SYSGEN.

4.1.5 CALL Directive

CALL directives effect the generation of macro routine call statements to cause the PTS-100 Assembler to create specialized macro routines from user-defined generalized macro routines, according to the arguments specified in the CALL directives. The format of the CALL directive is shown below.

CALL,Δ\$$\Delta$macronam$\Delta$Argument$_1$,... Argument$_n$

The key word CALL must appear in card columns 1 - 4, followed by a comma in column 5, a space in column 6, a \$ in column 7, and a space (i.e., blank character) in column 8. The macro routine name, up to 8 characters in length, starts in column 9 and may range through column 16. The name is terminated by a blank character. The name must be that of a generalized macro routine on the System Macro Library file that is input to the Assembler run.

Following the blank character terminating the macro routine name is the actual argument list (i.e., one or more actual arguments, separated by commas) that the Assembler is to use to specialize a routine for the monitor being generated. For a detailed description of macro routine definition, see Section 4 of Part 2 of this handbook.

Any number of CALL directives may appear in the deck. Each card is written in the format shown above.

4.1.6 END Directive

The END directive terminates the input directive file processing by SYSGEN. It must, therefore, be the last card in the input deck. The END directive key word appears in columns 1 - 3 of the last input card. No other columns are used on this card.

4.2 SYSGEN Processing

SYSGEN generates the necessary call statements to effect the creation of a specially-tailored IOCS monitor according to the key word directives and their parameters as described above. The output of SYSGEN is a file, in PTS-100 Assembler format, composed of an Assembler options record and the macro calls to the generalized System Macro Library routines required to specialize and order the described IOCS monitor. Some of the macro calls generated by SYSGEN are listed below:

$△#IMSCP	Global Area Initialization
$△#IMIP	Interrupt Packet Initialization
$△#IMPIT	PIOT Table Interfacing
$△#IMCTL	Logical Control Table
$△#IMCTP	Physical Control Blocks
$△#IMLSR	Level Service/Restore Routines and Necessary Device Drivers and Device Service Routines
$△#IMMSC	MSC Service Routine
$△#IMOPL	OPEN LUN Routine
$△#IMCLL	CLOSE LUN Routine
$△#IMACT	I/O Action Routine
$△#IMCLK	Clock Routine
$△#IMPAR	Parity Routine
$△#IMLOG	Error Logging Routine
$△#IMEXT	EXIT Routine
$△#IMPCB	PCB Computer Routine
$△#IMINT	IOCS Initialization

The output file from SYSGEN must be processed by the PTS-100 Assembler before the specialized IOCS monitor will be executable on the PTS-100. The total process of generating a system is illustrated in figure 3-4.

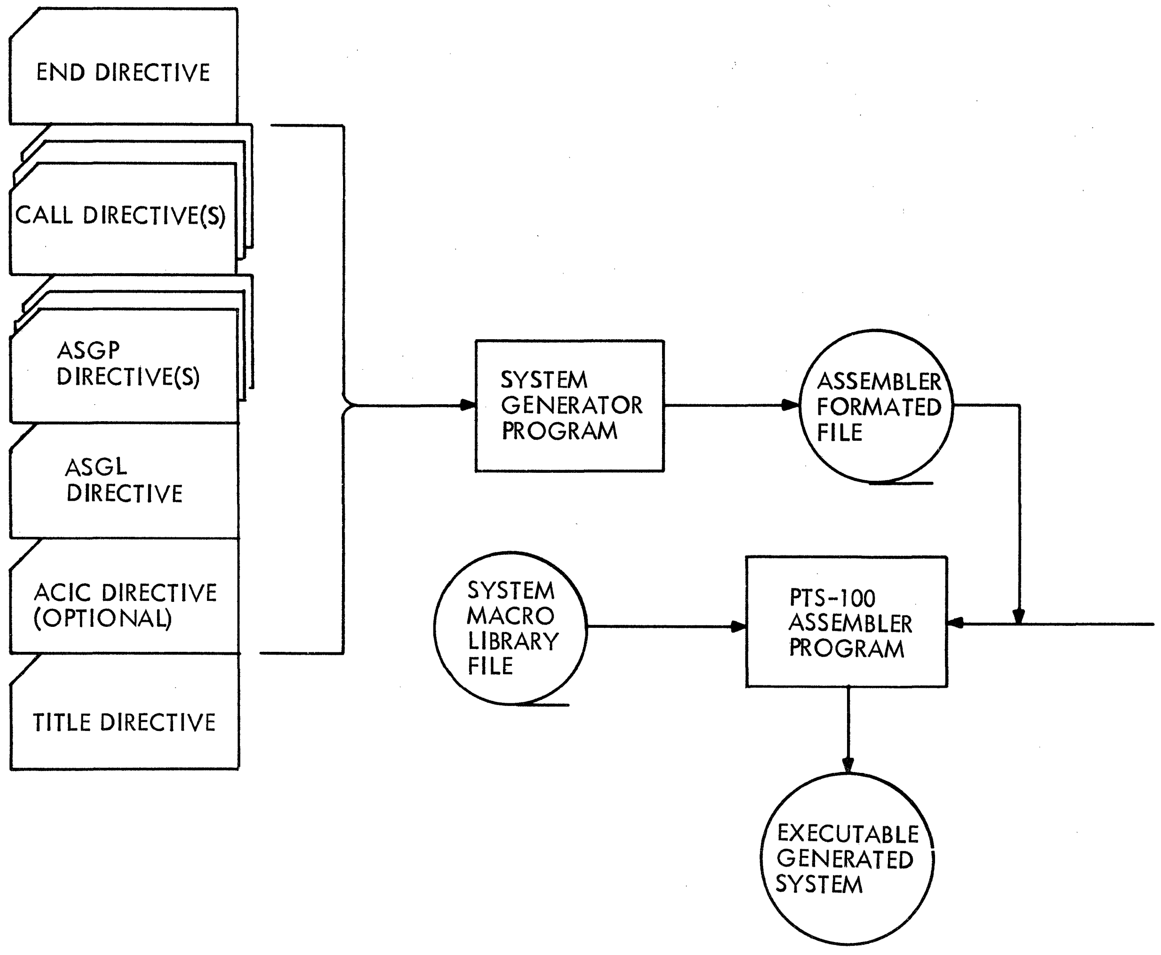


Figure 3-4. Processing Flow of Specialized System Generation

To produce the Assembler formated file, the SYSGEN program reads, interprets, and processes the key word directives that compose its input. The processing performed depends on the particular key word directive that has been input, as described in the following subsections.

4.2.1 TITLE Directive Processing

When the first four characters of an input record are TITL, the SYSGEN program reads the parameters from the record and composes an Assembler options record containing the name of the system to be generated and any specified assembly options. The first six characters of the input directive record (i.e., TITLE,) are discarded, and the output record begins with the name of the system, followed by the specified options. The record is written in Assembler formated file.

4.2.2 ACIC Directive Processing

When an ACIC directive is input to SYSGEN, the parameters are read and SYSGEN generates a series of macro calls to effect the creation of specialized macro routines to support the channel interface controller and its attached devices. The first macro call is

$△#IMCDP△ nn, T

which causes a specialized macro routine to be created from the generalized system macro routine #IMCDP, with the actual arguments nn (the number of devices specified in the ACIC directive) and T (the tumble table to be used) to be inserted in place of dummy arguments in the generalized routine. An additional macro call is generated for each device attached to the CIC. The macro calls take the form

$△ #IMCCB△PTAG, CAD, PDAD

where:

PTAG is the tag of the Physical I/O Table (PIOT) to be used by IOCS.

CAD is the channel address.

PDAD is the physical address of a specific device within a group of devices attached to a channel.

All macro calls are output to the Assembler formated file.

4.2.3 ASGL Directive Processing

When an ASGL directive is input to SYSGEN, the physical device notations are read, and a system logical macro call is generated in the format:

$\$\triangle$ #IMCTL$\triangle$Argument$_1$, Argument$_2$, . . . Argument$_n$

where:

$ is the special symbol to signal the Assembler that a macro routine is being called.

#IMCTL is the name of the generalized system logical macro routine to be specialized with the arguments in the call.

Arguments 1 - n are the physical device notations in the precise order in which they were specified on the ASGL directive.

The macro call is output to the Assembler formated file.

During Assembler processing, when the Assembler reads the macro call from the formated file, it locates the generalized #IMCTL macro routine on the System Macro Library file and creates a specialized macro routine by replacing the corresponding dummy arguments in the generalized routine with the actual arguments in the SYSGEN macro call.

4.2.4 ASGP Directive Processing

When an ASGP directive is input to SYSGEN, the program issues the necessary macro calls to effect Assembler specialization of interrupt packets for all external interrupt levels to which devices have been assigned, creation of interrupt level servicing routines for each external interrupt level assigned, creation of a Physical Control Block for each device assigned via the ASGP directive, and specialization of device driving and servicing routines.

At assembly time, the Assembler creates specialized IOCS monitor routines to process interrupts from all assigned levels and accommodate all devices within the described equipment configuration.

4.2.5 CALL Directive Processing

When SYSGEN reads an input CALL directive, it outputs the dollar sign ($) and the macro routine name specified in the CALL directive, followed by the argument string that the Assembler is to use in creating the specialized macro routine for the system being generated.

4.2.6 END Directive Processing

When SYSGEN reads the END directive in the input deck, it writes END as the last entry on the Assembler formated output file and terminates processing.

At assembly time, the END statement terminates the assembly of the generated object IOCS monitor.

Section 5. PTS-100 DUMP PROGRAMS

There are two dump programs provided for use on the PTS-100:

- The Memory Dump Program, which allows contiguous locations of main memory to be dumped to a character printer, a magnetic tape cassette device, or to a paper tape punch device.

- The Peripheral Device Dump program, which provides for dumping serial binary data file records to a character printer device.

These two programs and their utilization on the PTS-100 are described in detail in this section.

5.1 Memory Dump Program

The Memory Dump program is a small, easily relocatable program capable of dumping the contents of contiguous locations of main memory to any sequential storage device that accepts variable length output records. The length of dump records depends on the output device being used.

There are two versions of the Dump program:

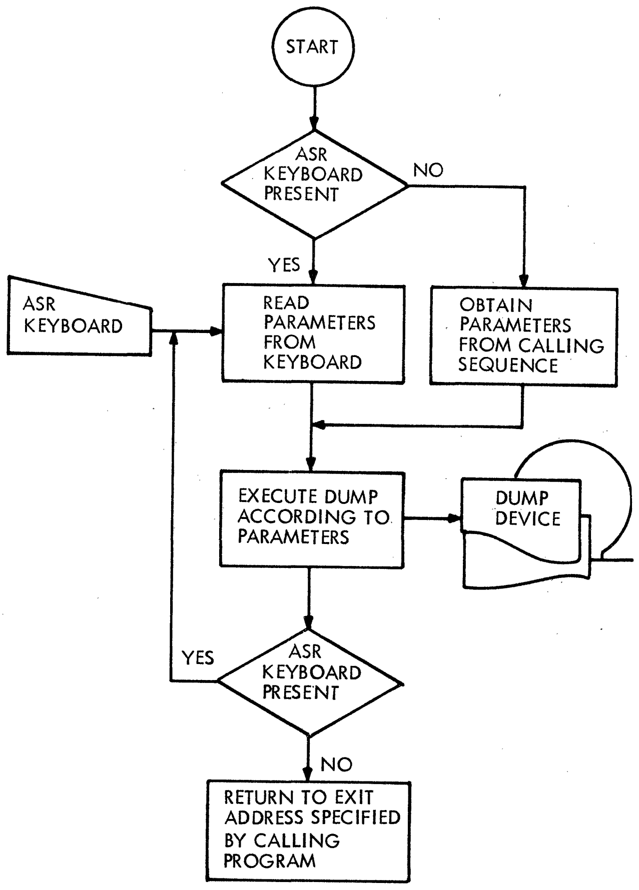
- Version 1 dumps hard copy hexadecimal or ASCII records onto a character printing device.

- Version 2 dumps reloadable binary records to a magnetic tape cassette or paper tape punch device.

Either version of the program may receive dump parameters as input from a keyboard device or as arguments of a subprogram assembled within the main program whose memory locations are to be dumped. The Dump program follows one or two paths, depending on the presence of an ASR keyboard, as shown in figure 3-5. When input is from a keyboard, the Dump program reads the dump parameters, executes the specified dump, and recycles to indicate its ability to accept a new set of parameters via the printout

RDY

at the keyboard device.



START
ASR KEYBOARD PRESENT
NO
YES
ASR KEYBOARD
READ PARAMETERS FROM KEYBOARD
OBTAIN PARAMETERS FROM CALLING SEQUENCE
EXECUTE DUMP ACCORDING TO PARAMETERS
DUMP DEVICE
YES
ASR KEYBOARD PRESENT
NO
RETURN TO EXIT ADDRESS SPECIFIED BY CALLING PROGRAM

Figure 3-5. Alternate Flow Paths of the Memory Dump Program

When dump parameters are passed from a subprogram within a main program, the Dump program executes the specified dump and returns control to the exit address transmitted from the calling program.

In all cases, the Dump program is located and activated by the Absolute/Relocating Loader.

The inputs, processing, and outputs of Versions 1 and 2 of the Dump program are described in subsection 5.1.1 and 5.1.2 on subsequent pages of this section.

5.1.1 Version 1 of the Memory Dump Program

Version 1 of the Dump program dumps the contents of contiguous memory locations on a character printing device. The hard copy dump printouts are in either hexadecimal or ASCII representation of a maximum of 72 characters (i.e., one ASR print line) per record. The dump parameters may be input via a keyboard or via the object code of a subprogram assembled within a main (i.e., calling) program. The procedures for specifying dump parameters and the resulting output are described below.

5.1.1.1 Version 1 Keyboard Input Format. Each set of keyboard input parameters to the Version 1 Dump program must be specified as a 15-character line with no intervening spaces and in the precise order shown in the following format diagram:

Character Position						
1 - 2	3 - 6	7	8 - 11	12	13	14 - 15
DU	STAR	,	STOP	,	X A	crlf

where:

DU appears in character positions 1 and 2 to specify the dump.

The STARting address of the dump appears as four hexadecimal digits in character positions 3 through 6.

A comma must appear in character position 7 to separate the starting and stopping address of the dump.

The STOPping address of the dump appears as four hexadecimal digits in character position 8 through 11.

A comma must appear in character position 12 to separate the stopping address and the output format designator.

The output format designator in position 13 must be an X if hexadecimal output is desired, or an A if ASCII format is to be used for the dump output.

Character positions 14 and 15 contain carriage return and line feed characters for the keyboard device being used.

If an error is made in entering keyboard input parameters, the slash character may be used to negate all previously typed characters on a line, and the correct entry may be typed immediately following.

When a set of input parameters has been terminated by a carriage return/line feed, the Version 1 Dump program performs the following:

- Stores the dump parameters in the symbolic memory locations #DFORM, #DSTAR, and #DSTOP.

- Validates the dump parameters, and if they are valid performs the following:

  dumps the contents of the accumulator, index register 1, and index register 2,

  dumps the specified contiguous memory locations in the specified hexadecimal or ASCII notation.

- Signals its ability to accept another set of dump parameters by printing RDY on the character printer device.

5.1.1.2 Version 1 Calling Sequence Parameters. When Version 1 dump parameters are specified via a subprogram within a main program, they are transmitted to the Dump program in the following calling sequence:

```
      LX2   TAG
      JMP   #DUMP
TAG   ADC   *
TAG1  HEX   0000   starting dump address
TAG2  HEX   0000   stopping dump address
TAG3  HEX   0000   negative zero if hexa-
                   decimal output desired,
                   or negative one if ASCII
                   output desired
TAG4  HEX   0000   own code exit address
```

When the dump parameters are transmitted to the Version 1 Dump program, it performs the following:

- Stores the dump parameters in the symbolic memory locations #DFORM, #DSTAR, and #DSTOP.

- Validates the dump parameters, and if they are valid performs the following:

  dumps the contents of the accumulator and index register 1. Index register 2 is not dumped since it is used to handle the calling sequence between the main program and the Dump program

  dumps the specified contiguous memory locations in the specified hexadecimal or ASCII notation

  returns control to the calling program at the specified exit address.

NOTE

The Dump program <u>does not</u> restore registers prior to returning control to the calling program.

5.1.1.3 Version 1 Dump Output. Regardless of the specified format of Version 1 Dump program output, a given dump line begins with a 4-digit hexadecimal integer, which is the byte address (modulo 16) of the first byte or word in that line, followed by an asterisk (*), and the formated content from the storage locations, as follows:

- If hexadecimal format was specified, eight data values are represented on a given line, each represented by four hexadecimal digits, followed by a space.

- If ASCII format was specified, 16 contiguous bytes of stored data are represented in 8-bit ASCII code per line.

If all data values for a given output line are identical, only the first value is printed, prefaced by the word ALL, as shown below:

```
0000*ALL△FFFF
```

If succeeding lines also contain the same value, line printing will be suppressed until some new data value occurs in the Dump data stream. For example, if locations 0 through 77 contain all ones the following would appear on the printout:

```
0000*ALL△FFFF
```

As mentioned above, if keyboard input is used, the value of the accumulator, X1, and X2 are printed prior to the specified dump. If calling sequence input is used, the value of X2 is not printed.

Formats of Version 1 Dump output are indicated below.

- Register dump line, Keyboard Input Response:

```
AC:0000ΔX1:Δ 0000ΔX2:Δ 0000
```

- Register dump line, Calling Sequence Input Response:

```
AC:0000ΔX1:Δ0000
```

- Hexadecimal format dump line:

```
0000*1111Δ2222Δ3333Δ4444Δ5555Δ6666Δ7777Δ8888
```

- ASCII format dump line:

```
0000*ABCDEFGHIJKLMNOP
```

- All hexadecimal or ASCII data values identical:

```
0000*ALLΔ FFFF
```

- Ready for Keyboard Input:

```
RDY
```

- Parameter error encountered:

```
?
```

5.1.2 Version 2 of the Memory Dump Program

Version 2 of the Dump program dumps reloadable binary data records onto a magnetic tape or paper tape device. The length of a given dump record is within the maximum allowable on the output device being used. The dump parameters may be specified via a keyboard or via a calling sequence specified in object code assembled within a main program. The procedures for specifying dump parameters and the resulting output for Version 2 are described below.

5.1.2.1 Version 2 Keyboard Input Format.

Each set of keyboard input parameters to the Version 2 Dump program must be specified as an 18-character line with no intervening spaces and in the precise order shown in the following diagram:

Character Position						
1 - 2	3 - 6	7	8 - 11	12	13 - 16	17 - 18
DU	STAR	,	STOP	,	EXEC	crlf

where:

DU appears in character positions 1 and 2 to specify the dump.

The STARting address of the dump appears as four hexadecimal digits in character positions 3 through 6.

A comma (,) must appear in character position 7 to separate the starting and stopping address of the dump.

The STOPping address of the dump appears as four hexadecimal digits in character positions 8 through 11.

A comma (,) must appear in character position 12 to separate the stopping address and the execution address.

The dumped program's execution address must appear as four hexadecimal digits in character positions 13 through 16.

Character positions 17 and 18 contain the carriage return and line feed characters for the keyboard device being used.

If an error is made in entering keyboard parameters, the slash character may be used to negate all previously typed characters on a line, and the correct entry may be typed immediately following it.

When a set of input parameters has been terminated by a carriage return/ line feed, the Version 2 Dump program performs the following:

- Stores the dump parameters in the symbolic memory locations #DSTAR, #DSTOP, and #DEXEC.

- Validates the dump parameters, and if they are valid dumps binary output to magnetic or punched paper tape.

- Notifies the programer that it is ready to receive another set of parameters by printing

  RDY

  on the keyboard device.

5.1.2.2 Version 2 Calling Sequence Parameters. When Version 2 dump parameters are specified via a subprogram within a main program, they are transmitted to the Dump program in the following call sequence:

```
      LX2   TAG
      JMP   #DUMP
TAG   ADC   *
      HEX   0000     starting dump address
      HEX   0000     stopping dump address
      HEX   0000     execution address
      HEX   0000     own code exit address
```

When the dump parameters are transmitted from the main program to the Version 2 Dump program, it performs the following:

- Stores the dump parameters at symbolic locations #DSTAR, #DSTOP, and #DEXEC.

- Validates the dump parameters, and if they are valid dumps binary output to magnetic or punched paper tape.

- Returns control to the calling program at the specified exit address.

NOTE

The Dump program does not restore registers prior to returning control to the calling program.

5.1.2.3 Version 2 Dump Output. The Version 2 Dump program produces variable length dump records within the maximum size allowable on the device to which output is dumped. Each dumped record is in binary format and has a 6-byte header containing the starting dump address, the number of bytes dumped, and the dumped program's execution address.

5.2 The Peripheral Device Dump Program

The sole function of the Peripheral Device Dump (PDD) program is to produce printed listings of binary data files stored in one of the following mediums:

cassette magnetic tape files
punched paper tape files
punched card files.

The printed output produced by the PDD program is either in hexadecimal notation or ASCII code, as specified by the programer, or in hexadecimal (default) notation if the programer does not specify the type of output listing to be produced. The PDD program operates under control of and in conjunction with the IOCS monitor, which performs all I/O operations on the PTS-100. That is, since the function of the PDD program is to produce printed listings of data files, it must call the IOCS monitor to perform the read operations required to bring its inputs into memory and to perform all write operations to produce the printed listings of data files. There are three logical file devices used by the IOCS monitor in servicing PDD program calls:

- The system input device (SYSI), from which the monitor reads a PDD control director specifying the format of the output listing to be produced.

- The system data device (SYSD), from which the monitor reads the binary data file to be listed.

- The system output device (SYST), to which the monitor writes the content of the specified data file.

The actual physical addresses of these devices must have been assigned when the IOCS monitor was generated by the System Generator program, described in Section 4 of this part of the handbook. The File Input/Output Blocks (FIOBs) and Input/Output Control Queue tables (IOCQs) required for each device by the IOCS monitor must have been created within the PDD program at system generation time. I/O data buffer areas must have also been reserved in the PDD program when it was generated. The buffer size selection should be the size required to store any physical records to be read from the SYSD device.

5.2.1 Inputs to the Device Dumping Process

There are three inputs to the device dumping process:

- The object code of the IOCS monitor and the Peripheral Device Dump (PDD) program, which must be loaded in that order by the Absolute/Relocating Loader.

- A control director record specifying the desired format of the output listing.

- The binary data file to be listed.

5.2.1.1 <u>IOCS Monitor and PDD Program Object Code</u>. The object code of the IOCS monitor and the Peripheral Device Dump program must have been assembled by the PTS-100 Assembler. The actual physical device addresses of the SYSI, SYSD, and SYST logical file devices must have been assembled in the IOCS monitor. The necessary FIOBs and IOCQs for monitor use with these devices must have been assembled in the PDD program. The monitor and PDD program object code, illustrated as punch card Loader input in figure 3-6, must be loaded by the Absolute/Relocating Loader.

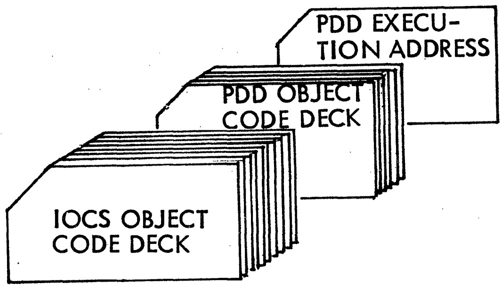


Figure 3-6. IOCS Monitor and PDD Program Object Code Input to the Absolute/Relocating Loader

The IOCS monitor is an absolute program and must be loaded first. Furthermore, the execution address of the PDD program must be specified, since the PDD program must initialize itself and subsequently call the monitor to read its inputs from SYSI and SYSD. Hence, the PDD program must appear at the end of the input to the Loader.

5.2.1.2 PDD Control Director Record. The PDD control director record is an 80 character record formated in the medium required by the System Input Device (SYSI). The character positions and usage on the control director record are as follows:

Character positions 1-3	These character positions may contain the characters ASC to specify that the output listing is to be in ASCII code; the characters HEX specify that the listing is to be in hexadecimal notation; any other characters produce a default listing in hexadecimal notation.
Character position 4	This character position is not used and must therefore be blank.
Character position 5	This character position is not used except when the data file to be listed is to be read from a cassette magnetic tape, in which case the particular cassette transport to be used must be specified.
Character positions 30 - 80	These character positions may optionally be used to enter a comment to be printed on the header line of the output listing. There is no restriction on the starting position of the first character of the comment. That is, it may begin in any position after 29 and range through position 80. The comment is printed exactly as it was specified on the director record. The comment field is useful for supplying dump identification, destination, etc.

The control director record, illustrated in punch card format in figure 3-7, must be ready in the System Input Device (SYSI) when the PDD program is initialized.

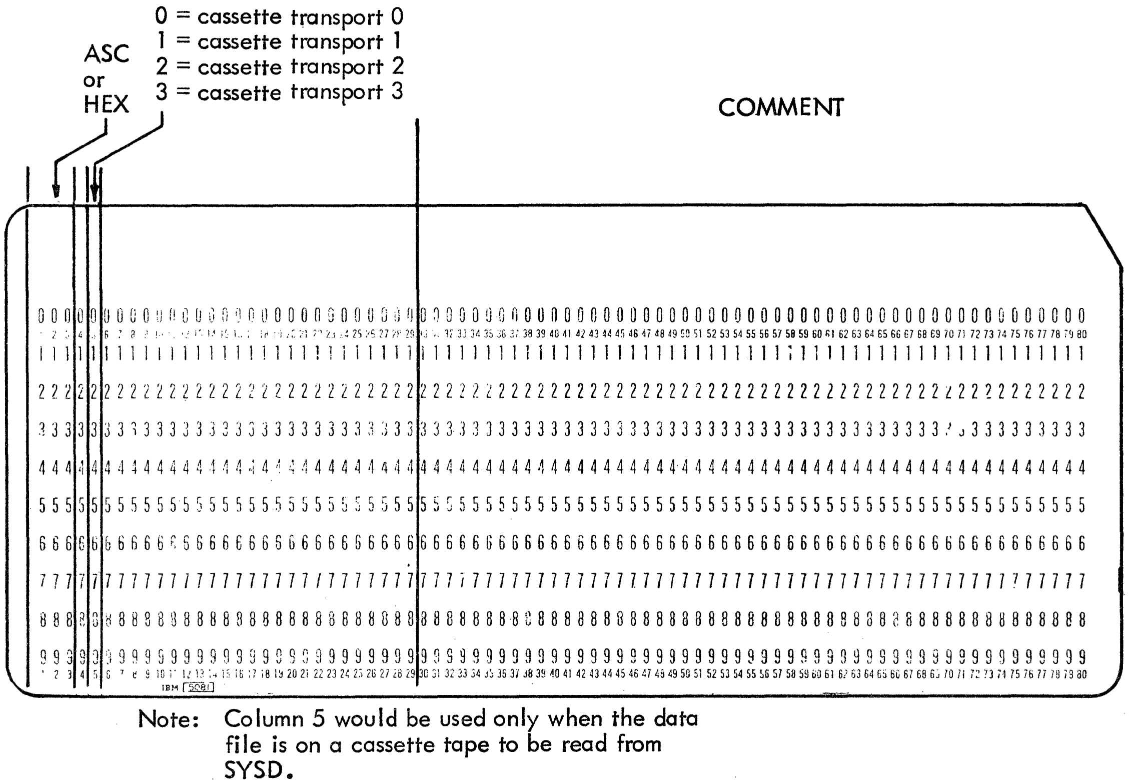


Figure 3-7. PDD Control Director Record Format, Assuming the Card Reader as the System Input Device (SYSI)

5.2.1.3 Data File. The data file to be listed must be in binary format in a serial storage medium, and must be ready in the System Data Device (SYSD) prior to initialization of the PDD program.

The size of records in the data file must be no greater than the size of the input buffer area reserved in the PDD program at system generation time. The programer should consult the documentation of his particular version of the PDD program if in doubt about the maximum record size that can be dumped. Typically, a record size of 200_{10} words is used.

5.2.2 Peripheral Device Dump Processing

To initialize the Peripheral Device Dump program the programer must perform the following:

1. Load the object code of the IOCS monitor and the PDD program via the loading procedure described in Section 2 of this part of this handbook.

2. Prepare a control director record, place it in the System Input Device, and ensure that the device is ready.

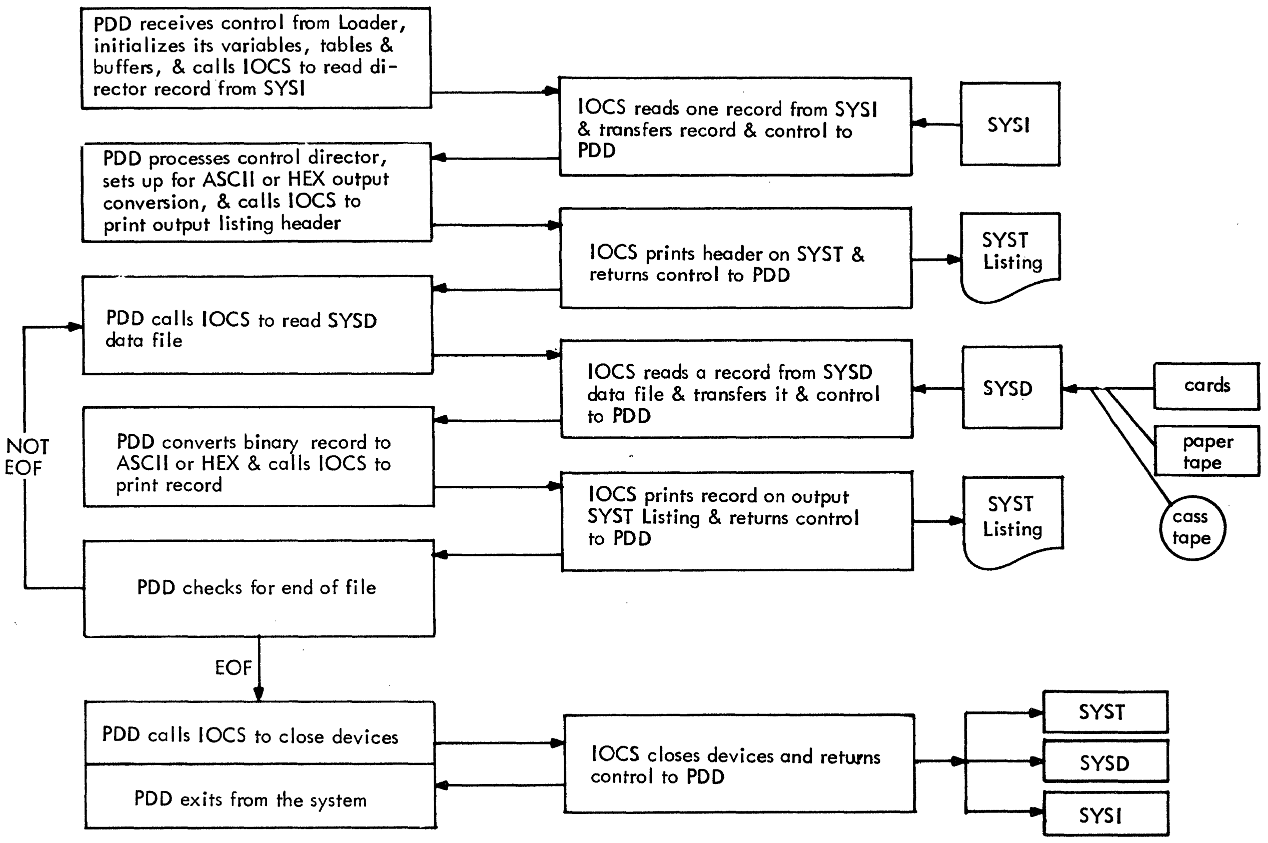
3. Place the data file to be dumped in the System Data Device and ensure that the device is operational.

When the Absolute/Relocating Loader has finished loading the PDD program, it activates the program at its starting address. The PDD program initializes its buffers, tables, variables, and constants, and then issues a call to the IOCS monitor to read the control director record from the SYSI device. The monitor reads the record into the reserved storage area within the PDD program.

The PDD program reads and interprets the control director and performs the necessary actions to set up for the conversion from binary input data format to either ASCII code or hexadecimal output format, as specified by the control director. PDD then calls the monitor to print the output listing header either as

```
        ASCII DUMP    (comment)
or
        HEX DUMP      (comment)
```

When the header has been printed, IOCS returns control to PDD, which then calls the monitor to read a record from the data file on the SYSD device. IOCS reads a binary input data record into PDD's input buffer area. PDD then reads the record, converts it to the required format, moves the converted record to the output buffer, and calls IOCS to print the record on the SYST listing. IOCS prints the record header (i.e., the word RECORD, followed by the record number), spaces to the next line, and prints the converted data values of the record. This process continues until the last record from the data file has been read, converted, and printed. At this point, the PDD program calls the IOCS monitor to close the SYSI, SYSD, and SYST devices, after which PDD exits from the system. The functional processing flow between the PDD program and the IOCS monitor is illustrated in figure 3-8.



PDD receives control from Loader, initializes its variables, tables & buffers, & calls IOCS to read director record from SYSI
IOCS reads one record from SYSI & transfers record & control to PDD
SYSI
PDD processes control director, sets up for ASCII or HEX output conversion, & calls IOCS to print output listing header
IOCS prints header on SYST & returns control to PDD
SYST Listing
PDD calls IOCS to read SYSD data file
IOCS reads a record from SYSD data file & transfers it & control to PDD
SYSD
cards
paper tape
cass tape
NOT EOF
PDD converts binary record to ASCII or HEX & calls IOCS to print record
IOCS prints record on output SYST Listing & returns control to PDD
SYST Listing
PDD checks for end of file
EOF
PDD calls IOCS to close devices
PDD exits from the system
IOCS closes devices and returns control to PDD
SYST
SYSD
SYSI

Figure 3-8. Functional Flow of Peripheral Device Dump/IOCS Monitor Processing

5.2.3 Peripheral Device Dump Program Output

The output from the PDD program is a hard copy listing of the input data file. The listing will be in ASCII code or hexadecimal notation, depending upon whether the programer specified ASC in columns 1 - 3 of the control director record. That is, the programer may specify ASC, HEX, or leave these columns blank. If the characters HEX or blank characters appeared in these columns the output listing will be in hexadecimal notation.

The output listing will contain a header line indicating the type of dump output, followed by the comment, if any, specified in columns 30 through 80 of the control director. Each record is identified on the listing, and the record identification is immediately followed by the data in the record. ASCII and hexadecimal output listing are illustrated in figure 3-9.

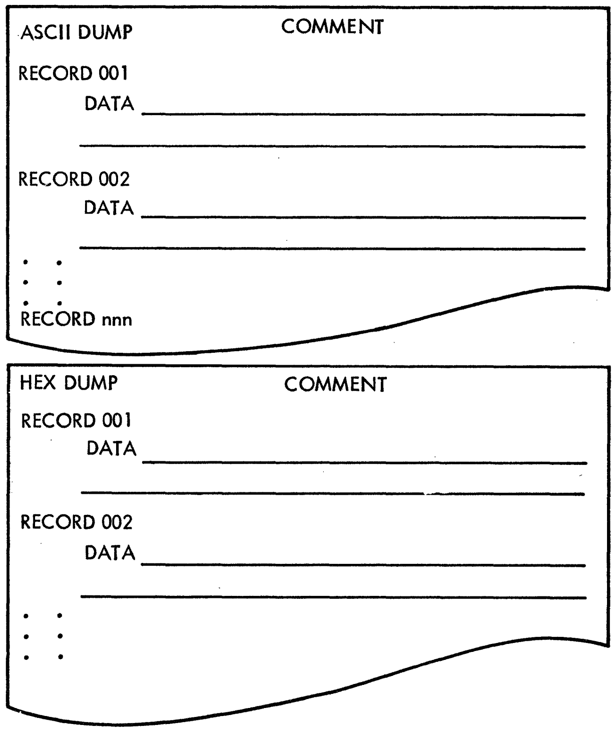


Figure 3-9. ASCII and Hexadecimal PDD Output Listing

Section 6. FILE UPDATE PROGRAM

This program provides a convenient, easily used method of filing, maintaining, and updating source and object programs. That is, the File Update program may be used to create a master file containing either object or source programs, or both object and source programs, and subsequently to maintain and update the master file. Object programs may be maintained on a program basis only.

The File Update program may be directed to perform the following functions:

- Insert one or more programs on the master file.
- Delete one or more programs on the master file.
- Correct programs on the master file by changing their names and/or deleting, replacing, or inserting lines.
- Replace one or more programs on the master file.
- Produce a master file directory.
- Block primary files (320 bytes/record), if specified in option field.

The File Update program utilizes the IOCS monitor to perform all I/O operations. Therefore, the monitor must be loaded and operational prior to initialization of the File Update program.

File Update program processing is specified via four types of input directors, described in subsection 6.1 below.

6.1 Input Directors

Four types of directors are input to the File Update program:

- Program directors, which specify actions to be taken on an entire program on the master file.
- Data line directors, which specify actions to be taken for one or more specific lines within a given program on the master file.
- The END director, which terminates the inputs to the File Update program.
- The EOF director, which writes an end of file record (1EOF) on the master file and terminates the File Update program immediately.

The input directors are read from the System Input (SYSI) device. If the File Update program is to produce the first version of a master file, only one type of Program director, the Insert director, the associated object or source program code, and an EOF director are required as input from the System Input device. No other input is required. If, however, the File Update program is to update an existing master file of programs, the current master file must be input via the System Data (SYSD) device, and the appropriate directors must be input via the SYSI device.

The File Update program processes the input directors to produce the following outputs:

- The original or updated version of the master file, which is written to the Logical Unit A (LOGA) device.
- A listing of changes specified by the directors and any errors detected in the input directors and, optionally, a file directory on the System List (SYST) device.

- Card image format (if specified) on the System List (SYST) device.

The four types of input directors and the File Update processing of the directors are described in the subsections 6.1.1 through 6.1.3. The outputs from the program are described in subsection 6.2.

6.1.1 Program Directors

The program directors specify actions to be taken for an entire program. They may be composed of five fields, as shown in the following generalized format diagram.

Field 1	Field 2	Field 3	Field 4	Field 5
$xxxP	current program name	new program name	options	comments

where:

Field 1 contains one of the 5-character identifiers for the four directors, each of which begins with a $ in record position 1, and ends with the letter P in record position 5. The specific action to be performed is indicated by the three intervening alphabetic characters (xxx), which must be one of the following:

INS	specifies that a new program is to be INSerted on the updated master file.
DEL	specifies that one or more programs are to be DELeted from the master file.
COR	specifies that a program on the master file is to be CORrected.
REP	specifies that a new program is to REPlace a program currently on the master file.

Field 2 contains the name, up to eight characters in length, of a program on the master file. The name begins in position 6 and may range through position 13 of the input director. The named program is to be processed as specified in Field 1.

Field 3 contains the name of a new program to be assigned as specified by the identifiers $INSP, $REPP, or $CORP, or the name of the last program in a string of programs to be deleted from the master file. The name begins in position 14 and may range through position 21 of the input director.

Field 4 may contain option designators to effect specific types of output by the File Update program, as follows:

B	specifies that the program about to be inserted is in binary format.
N or F	specify, respectively, that no file directory is to be produced (i.e., N), or that a full file director (F) is to be produced.* A full file directory will consist of all header records being listed on the SYST device for those programs remaining on the master input device. For this reason, no other processing is allowed if the F option is specified.
L	specifies that the program being corrected will be listed in card image format on the SYST device. (This option is valid only for the $CORP command.)
C	specifies primary files will be blocked (320 bytes/record).

*The N and F options are mutually exclusive. That is, only one of the options should be specified, since the first one will be accepted and remain in effect to be acted upon by the File Update program.

The options start in position 22 and may range through position 24. They may appear in any order. They are terminated by the first blank character in Field 4.

Field 5 may contain a comment to be carried in the header record when a program is to be inserted, corrected, or replaced. The comment appears in positions 30 through 80 of a given input program director.

The program directors cause a search forward and copy function to be performed by the File Update program. That is, the input master file is searched and copied onto the output master file until the program named in Field 2 of the program director is found, at which point the File Update program performs the action specified in Field 1. It is for this reason that actions on programs must be specified in the exact order in which the programs appear on the master file, as indicated on the file directory, which may be obtained from the File Update program.

The individual program directors, their formats and content, and the processing specified by them are described in detail below.

6.1.1.1 Insert Program Director ($INSP). The INSert program director is used to add one or more new programs to an existing master file, or to create the master file initially. The format of the director is shown below:

Field 1	Field 2	Field 3	Field 4	Field 5
$INSP	—	new program name	option(s)	comment
	current program name	new program name	option(s)	comment

The $INSP identifier must appear in Field 1 of the input record. If an existing master file is being updated, the name of the program after which the new program is to be inserted may appear in Field 2. If no name is specified in Field 2, the new program will be inserted at the current location at which the master file is positioned when the $INSP director is read. That is, the File Update program will insert the new program immediately following the last program for which it performed processing as specified by the previous program director, or at the beginning of the file if the $INSP director is the first director read by the File Update program.

If the master file is being created initially, Field 2 would not be used in the $INSP director. That is, the File Update program would merely write the object or source programs onto the output master file in the sequential order in which they and their $INSP directors were read from the SYSI device. A name must be assigned to the new program by entering it in Field 3. If this field is all blanks, an error message will be generated and the director will be ignored. No blank headers are allowed. If a name is specified or assigned in this field, a header record will be created and written preceding the new program's code on the output master file. The header will contain the name and the information, if any, specified as a comment in Field 5 of the director.

The source or object records of the program to be inserted must immediately follow the $INSP director in the input file being read by the File Update program. If object records are to be read, the B option must be specified and the last object record per program must be a multipunched 2-7-8-9 record (standard object end of file mark).

6.1.1.2 Delete Program Director ($DELP). The DELete program director is used to specify that a single program or a string of consecutive programs is to be deleted from the master file. The format of the director is diagramed below.

Field 1	Field 2	Field 3
$DELP	program name	—
	program name 1	program name n

The $DELP identifier must appear in Field 1. If only one program is to be deleted, its name must appear in Field 2. If a string of consecutive programs is to be deleted, the name of the first program to be deleted must appear in Field 2, and the name of the last program in the string must appear in Field 3. No other fields are used on this director.

The single name director causes the File Update program to delete the named program. That is, the header record and the program code associated with it are not written onto the output master file. If the delete director specifies that a string of programs is to be deleted (i. e., if Field 2 and Field 3 both contain a program name), the File Update program skips all programs and their header records, beginning with the first program and going through program n, when it reads the input master file. That is, none of these programs is copied onto the output master file. Any header records associated with the deleted programs are also deleted.

6.1.1.3 Correct Program Director ($CORP). The CORrect program director specifies that a program on the master input file is to be corrected. The permissible formats of the director are shown below:

Field 1	Field 2	Field 3	Field 4	Field 5
$CORP	progname			
	old program name		option(s)	
	old program name	new program name	option(s)	comment

The $CORP identifier must appear in Field 1, and the name of the program to be corrected must appear in Field 2. If the program is to be assigned a new name, it must appear in Field 3. Optional outputs may be specified in Field 4, and any information to be carried in a header record for a program assigned a new name may appear in Field 5.

The $CORP director must be followed with the necessary data line directors (see subsection 6.1.2) to effect the desired program corrections, such as deleting or replacing lines in the original program, or inserting new lines in the program.

When the $CORP director specifies a new program name in Field 3, the File Update program creates a new header record containing the new name and the comment, if any, in Field 5. The header record is written onto the master output file, and the File Update program then processes the data line director to create and write the corrected program code following its associated header record on the output master file.

If Field 3 does not contain a new program name, the original header record is transferred to the output master file, followed by the corrected program.

When all data line directors following the $CORP director have been processed, (i. e., when File Update reads a new program director), the correction process for this program is completed. Any remaining lines in the original program are copied to the output master file.

The $CORP director may be used to obtain a file directory printout without actually performing an update run. This is effected by using only the $CORP director, and $END director, and the input master file as inputs to the File Update program. In this case, the $CORP director would contain the following:

$CORPprogname

where "progname" is the name of any program known to be stored on the input master file. No other information (except for an F in the option field for a full directory) is specified on the director, which is followed immediately by the $END director. This director causes the File Update program to read the input master file and produce the directory on the SYST device.

6.1.1.4 Replace Program Director ($REPP). The REPlace program director specifies that a program on the master file is to be replaced by the program whose source or object code immediately follows the director. The permissible formats of the director are diagramed below:

Field 1	Field 2	Field 3	Field 4	Field 5
$REPP	old program name		option(s)	
	old program name	new program name	option(s)	comment

The $REPP identifier must appear in Field 1, and the name of the program to be replaced must appear in Field 2. Field 3 may optionally specify a name to be assigned to the new program. If a name is specified in Field 3, Field 5 may optionally specify a comment to be carried in the header record that the File Update program will create for the new program. If Field 3 does not contain a name, no new header record will be created.

The source or object records of the new program must immediately follow the $REPP director. If object records are to be read, the B option must be specified and the last object record must be a multi-punched 2-7-8-9 record (standard object end of file mark).

6.1.2 Data Line Directors

These directors specify actions to be taken for one or more lines within a given program named on an immediately preceding $CORP program director. There are three data line directors: the INSert director, the DELete director, and the REPlace director. The generalized format of the data line directors is shown below:

Field 1	Field 2	Field 3
$xxx∆	line number	line number

where:

Field 1 contains one of the 4-character identifiers for the three directors, each of which begins with a $ in record position 1 and ends with a blank character in position 5. The specific action to be performed is indicated by the three intervening alphabetic characters (xxx), which must be one of the following:

INS	specifies that one or more data lines are to be inserted in the program.
DEL	specifies that one or more data lines are to be deleted from the program
REP	specifies that a data line is to be replaced by one or more data lines.

Field 2 contains the logical line number (positions 6-9) after which new data lines are to be inserted, or the first line to be replaced or deleted.

Field 3 contains the logical line number (positions 14-17) of the last data line to be deleted.

The logical line numbers are kept within the File Update program. That is, the program associates a number with each line in any given source or object program on the input master file. For source programs, the line number will be maintained according to the order of appearance of source statements. That is, the File Update program will associate numbers from 1 to n with each source statement read from the file.

6.1.2.1 <u>Insert Data Line Director ($INS)</u>. The INSert data line director specifies that the data line(s) immediately following it are to be inserted in the existing program following the specified logical line number. The format of the director is shown below:

Field 1	Field 2
$INSΔ	line number

The $INS identifier must appear in Field 1, and be followed by a blank.

Field 2 must contain the correct line number following which the data line(s) are to be inserted. The data lines must immediately follow the INSert director.

6.1.2.2 <u>Delete Data Line Director ($DEL)</u>. This director specifies that one or more existing lines in the subject program are to be deleted. The format of the director is shown below:

Field 1	Field 2	Field 3
$DELΔ	line number	
	line number 1	line number n

The $DEL identifier must appear in Field 1, and be followed by a blank. Field 2 must contain the logical line number of the single line to be deleted or the first logical line number of a series of lines that is to be deleted. If only one line is to be deleted, Field 3 is not used. If a series of consecutive lines is to be deleted, Field 3 must contain the logical line number of the last line to be deleted.

When a $DEL data line director is read by the File Update program, the specified lines are skipped when the program named on a $CORP program director is written on the output master file.

6.1.2.3 <u>Replace Data Line Director ($REP)</u>. The format of this director is shown below:

Field 1	Field 2	Field 3
$REPΔ	line number	

The $REP identifier must appear in Field 1, followed by a blank. Field 2 must contain the logical line number of the single line to be replaced. The replacement lines must follow the $REP director.

6.1.3 END Director ($END)

The END director signals the end of the input data stream to the File Update program. It is written in record positions 1-4 in Field 1 simply as

$END

When the program reaches the END director, it copies the remaining portion of the master input file to the master output file. It then produces a file directory (unless it has been suppressed by the N option on a program director record) and terminates processing.

6.1.4 EOF Director ($EOF)

The EOF director causes an end of file record (1EOF) to be written on the master output file. This will be followed by the production of a file directory (unless it has been suppressed by the N option on a program director record) and the processing terminates. This director is intended to be used when initially putting programs on the master output file using the $INSP director. If used, it takes the place of the $END director, which also copies the remainder of the master input file.

6.2 File Update Program Outputs

There are three standard outputs from the File Update Program:

- The updated master file on the Logic Unit A (LOGA) device, containing all correctly submitted corrections and programs.
- A listing on the SYST device of all updates submitted to the File Update program, with embedded error printouts indicating any illegal input directors and the cause of the error conditions.
- A printout on the SYST device of the file directory, unless it was suppressed by the N option on a program director.

The file directory contains the program name for each program on the master file. This information is printed in the exact order in which the associated program appears on the updated master file. Nonstandard outputs may be requested via the option designator L in Field 4 of program directors as described earlier in this section.

Section 7. DISC SUPPORT PROGRAMS

7.1 Disc Volume Preparation Program

This program initializes a new disc for use in the PTS-100 system. It may also be used to erase the information on an old disc to prepare it for reuse. A disc must be preprocessed with the Disc Volume Preparation program whether it is to be accessed by physical or logical input/output.

The Disc Volume Preparation program performs three functions:

- Accepts from the card reader a set of parameters describing the desired format of the disc.
- Formats and checks the disc surface. The program executes the Write Address operation on each track of the disc surface, checks whether each track can be read without error, and issues messages indicating any tracks that cannot be read.
- Writes out the Volume Label and makes an initial entry in the Volume Directory.

This program assumes that the disc is to have 320 data bytes per sector, and 20 sectors per track. The number of tracks per cylinder and number of cylinders per disc are variables, which are accepted as parameters.

Disc Volume Preparation is a standalone program; it does not use the IOCS monitor. All input and output for the disc, the card reader, and the serial printer are handled directly at the physical level.

7.1.1 Input to the Disc Volume Preparation Program

Input consists of a set of six free form parameters read by the card reader. Each parameter value is preceded by an identifying keyword. All six of the parameters listed in table 3-8 must be specified; they may be punched on any number of cards. If several parameters are punched on one card, they must be separated by commas. The parameter cards must be followed immediately by a card with /* punched in columns 1 and 2.

Table 3-8. Disc Volume Preparation Program Parameters

Parameter	Keyword and Value	Description
Disc drive number	DRIVENO=n	n is a number from 0 to 7, designating the drive address to be formated.
Volume serial number	VOLSER=aaaaaa	aaaaaa is any six characters used to identify the disc.
Address of start of Volume Directory	VOLDIRSTRT= nnn/nn/nn	The parameter value is a disc address in the form cylinder/track/sector. The Volume Directory will start at this address.
Last cylinder in Volume Directory	VOLDIRLAST= nnn	nnn is the last cylinder number to be assigned to the Volume Directory.
Number of tracks per cylinder	TRACKS=nn	nn is the number of tracks per cylinder on the disc to be processed.
Number of cylinders per disc	CYLINDERS=nnn	nnn is the total number of cylinders on the disc to be processed.

7.1.2 Disc Volume Preparation Program Output

The major output is a disc on which all tracks have had correct addresses written and with all tracks checked to see that they can be read and written. For each track that cannot be read and written correctly, the following message is printed on the serial printer:

BAD TRACK nnn/nn

where nnn/nn is cylinder/track.

As part of the output, the Volume Label is written on the disc at location cylinder 0, track 0, sector 0. It has the following format:

Volume serial number	6 bytes
Address of start of Volume Directory	4 bytes
Last cylinder number in Volume Directory	2 bytes
Highest track number in cylinder (track number is in bits 2-4)	2 bytes
Highest cylinder number in disc	2 bytes
Unused	remainder of sector

7.1.3 Processing

The processing done by the Disc Volume Preparation program can be separated functionally into three segments: parameter input, formating and checking, and volume initialization.

7.1.3.1 Parameter Input. Input cards are read until a card is found with */ in columns 1 and 2. As each card is scanned, the six keywords are isolated and identified.

If the keyword is not recognized, the serial printer prints out the message UNRECOGNIZED KEYWORD, followed by the word as read from the card.

If the parameter keyword is recognized, a routine is entered for converting (if necessary), checking, and storing the value. If the value is outside permissible limits, the serial printer prints out the message PARAMETER ERROR, followed by the keyword.

7.1.3.2 Formating and Checking the Disc Surface. Using the parameters disc drive number, tracks per cylinder, and cylinders per pack, the entire disc surface is formated by means of the Write Address operation. If this is not successfully completed, it is retried three times before a BAD TRACK message is written on the serial printer.

After the Write Address operation is performed for each track, an attempt is made to read each sector in the track, to see that it is all zeros. If not, the read is retried three times before a BAD TRACK message is written.

7.1.3.3 Volume Initialization. Using the parameters provided on the input cards, the Volume Label is written. Also, an end-of-directory record is written at the beginning of the Volume Directory area.

7.2 Disc Allocator Program

The Disc Allocator assigns disc space to files, extends the disc area allocated to files, and deletes files. It must be used in connection with the logical IOCS for disc. Before any file can be written or read through the logical input/output, it must be allocated on a disc by the Disc Allocator program. If a disc is to be accessed by means of physical input/output only, it is not necessary to process it with the Disc Allocator program.

Prior to running the Disc Allocator the disc must be initialized by means of the Disc Volume Preparation program. Disc Volume Preparation writes the Volume Label, from which the Disc Allocator obtains the Volume Directory limits, the number of tracks per cylinder, and the number of cylinders per disc.

The program operates from free form keyword type parameters read from the card reader.

7.2.1 Input to Disc Allocator Program

Input consists of free form parameters read by the card reader. Each parameter value is preceded by an identifying keyword. Different combinations of parameters are required, depending on the function being requested and the file organization involved.

Several files may be allocated in one program run. A new file is indicated by the appearance of a FILENAME parameter. All parameters following that one, up to the END card or another FILENAME parameter, are assumed to apply to the same file.

Except for the FILENAME parameter, the parameters may appear in any order. If several parameters are punched on one card, the parameters must be separated by commas. The last parameter card must be followed by a card with /* punched in columns 1 and 2.

Table 3-9 lists the parameters, their keywords, their permissible values, and rules governing their use.

7.2.2 Disc Allocator Program Output

The Disc Allocator has two outputs: one to the disc and one to the serial printer. The output to the disc consists of an entry in the Volume Directory (in the format shown in subsection 7.2.2.1) and of the file area itself, which is initialized to zeros. The output to the serial printer consists of the messages listed in subsection 7.2.2.2.

7.2.2.1 <u>Disc Allocator Entries in Volume Directory.</u> Each entry in the Volume Directory contains the following ten fields:

(1) Banner word (2 bytes):

0000_{16} if unused
0001_{16} if active
$FEDC_{16}$ if end of directory

(2) File name (10 bytes). This is the name by which the file is always accessed in programs that manipulate it. It is assigned by the user, and can contain any characters.

(3) First cylinder number in file (2 bytes).

(4) Last cylinder number in file (2 bytes).

(5) File organization (2 bytes):
K = random organization with keys
N = random organization without keys
S = sequential organization.

(6) Number of sectors per block (2 bytes); applies to random access only.

(7) Record size (2 bytes). This is the number of bytes per record; it must be an even number.

(8) Whether record is fixed or variable in length (2 bytes):
F = fixed
V = variable

(9) Address of start of overflow area (2 bytes); applies to random files only; 0 signifies no overflow area.

(10) Reserved for expansion (26 bytes).

Table 3-9. Disc Allocator Program Parameters

Parameter	Keyword and Value	Use
File name	FILENAME = aaaaaaaaaa	The file name must be the first parameter in a group of parameters applying to a particular file. The parameter value can consist of any characters, and any number of characters up to ten.
Drive number	DRIVENO = n	n is a number from 0 to 7, identifying the device number on which the disc is mounted. If this parameter is omitted, device number 0 is assumed.
Function	FUNCTION = $\left\{\begin{matrix}\text{NEW}\\\text{EXT}\\\text{DEL}\end{matrix}\right\}$	The function parameter indicates whether a file is being allocated (NEW), extended (EXT), or deleted (DEL). This parameter must always be present.
First cylinder in file	FIRSTCYL = nnn	This parameter identifies the first (lowest) cylinder number to be allocated to the file. It must be present whenever FUNCTION = NEW.
Last cylinder in file	LASTCYL = nnn	This parameter identifies the last (highest) cylinder number to be allocated to the file. For random files, it includes the overflow area, if any. This parameter must be present whenever FUNCTION = NEW or EXT.
File organization	FILEORG = $\left\{\begin{matrix}\text{K}\\\text{N}\\\text{S}\end{matrix}\right\}$	This parameter indicates whether a file is random organization with keys (K), random organization without keys (N), or sequential organization (S). This parameter must be present whenever FUNCTION = NEW.
Record type	RECTYPE = $\left\{\begin{matrix}\text{F}\\\text{V}\end{matrix}\right\}$	This parameter specifies whether the records in a file are fixed (F) or variable (V) inlength. Only F may be specified for random files. If this parameter is omitted, F is assumed.
Record size	RECSIZE = nnn	This parameter gives the number of bytes per record and must be an even number. It must be present whenever FUNCTION = NEW <u>and</u> RECTYPE = F.
Overflow cylinder address	OVERFLOW = nnn	This parameter indicates the lowest cylinder number of the overflow area for a type K file. The overflow area extends from this cylinder to the last cylinder in the file (LASTCYL). This parameter should be included only when FUNCTION = NEW <u>and</u> FILEORG = K. If the overflow parameter is omitted, it is assumed that there is no overflow area.

7.2.2.2 Disc Allocator Output to Serial Printer. The Disc Allocator program outputs the following messages to the serial printer:

FILENAME = aaaaaaaaaa

This message is printed whenever a new FILENAME parameter is read. The file name identifies the file to which the following message applies.

UNRECOGNIZED KEYWORD keyword
PARAMETER ERROR keyword
PARAMETER OMITTED keyword
VOLUME DIRECTORY OVERFLOW
FILE NAME ALREADY USED
FILE AREA UNAVAILABLE
FILE NOT FOUND
BAD TRACK nnn/nn (cylinder/track)

7.2.3 Processing

The processing done by the Disc Allocator program can be separated functionally into four segments: parameter input, check of Volume Directory against parameters, entry of parameters into Volume Directory, and initialization of new file area to zeros (with a read check).

The first program segment reads and checks the parameters. Each card is scanned, and in the process each keyword is updated and identified. If the keyword is not recognized, or if the value is outside its permissible limits, an error message is printed. When all the parameters for one file have been read, the resulting parameter table is examined to see if all required parameters have been specified. If not, an error message is printed.

The second segment checks the Volume Directory against the parameter list to see if the requested action can be performed. It checks to see that a new name is unique, that specified file limits do not conflict with any existing file, and that there is room in the Volume Directory for a new entry.

The third segment makes the necessary entry in the Volume Directory. This may be a deletion, a new entry, or an alteration to an old entry, depending on what function was requested.

The fourth segment initializes the new file area to zeros and checks that it can be read back, issuing an error message if it cannot be.

7.3 Disc Dump Program

The sole function of the Disc Dump program is to produce a printed listing of data on all or a selected portion of any disc unit is use with the PTS-100. The output is listed on the serial printer in either hexadecimal or ASCII notation, as specified by the input directives. All dump parameters are input from a display or teletypewriter keyboard in response to program messages. As indicated in Figure 3-10, the PTS-100 Disc Dump program is a standalone program; it does not use the IOCS monitor.

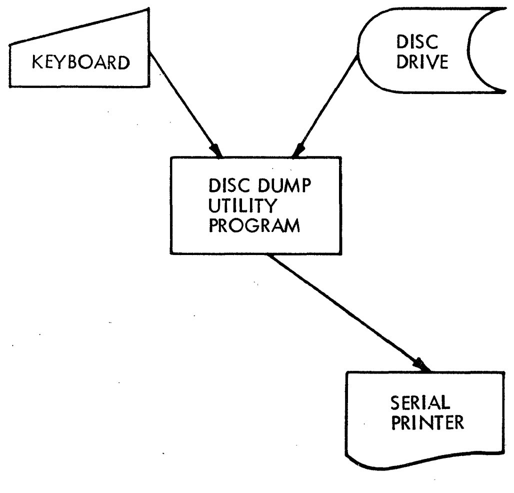


Figure 3-10. Disc Dump Flowchart

```
PTS-100 DISK DUMP         HEX EXAMPLE


BYTE                               CYLINDER  0050  TRACK  0  SECTOR  0000

0000  AAD4C8C9  D3A0C9D3  A0C1A0D3  C1CDD0CC  C5A0CFC6  A0D4C8C5  A0CFD5D4  D0D5D4A0  CFC6A0D4  C8C5A0D0
0040  D4D3ADB1  B0B0A0C4  C9D3C3A0  C4D5CDD0  AEA0A0C9  D4A0C3C1  CEA0C2C5  A0CCC9D3  D4C5C4A0  A0A0A0A0
0080  AAC5C9D4  C8C5D2A0  C9CEA0C8  C5D8C1C4  C5C3C9CD  C1CCA0CF  D2A0C9CE  A0C1D3C3  C9C9ACA0  CFCEA0D4
0120  C8C5A0D3  C5D2C9C1  CCA0D0D2  C9CED4C5  D2AEA0A0  D0C1D2D4  C9C1CCA0  D3C5C3D4  CFD2D3A0  A0A0A0A0
0160  AACFD2A0  C5CED4C9  D2C5A0D3  C5C3D4CF  D2D3A0C3  C1CEA0C2  C5A0C4D5  CDD0C5C4  AEA0A0D4  C8C5A0C2
0200  C5C7C9CE  CEC9CEC7  A0C1CEC4  A0C5CEC4  C9CEC7A0  C1C4C4D2  C5D3D3C5  D3A0CFC6  A0D4C8C5  A0A0A0A0
0240  AAC1D2C5  C1A0D4CF  A0C2C5A0  C4D5CDD0  C5C4A0C1  D2C5A0C5  CED4C5D2  C5C4A0CF  CEA0D4C8  C5A0D4C5
0280  CCC5D4D9  D0C5ACA0  C9CEA0D4  C8C5A0C6  CFD2CDA0  C3D9CCC9  CEC4C5D2  AFD4D2C1  C3CBAFD3  C5C3D4CF
```

Figure 3-11. Disc Dump Listing in Hexadecimal Notation

```
PTS-100 DISK DUMP         ASCII EXAMPLE


BYTE                               CYLINDER  0050  TRACK  0  SECTOR  0000

0C00  *THIS IS   A SAMPL  E OF THE   OUTPUT   OF THE P  TS-100 D  ISC DUMP  .  IT CA  N BE LIS  TED
0080  *EITHER   IN HEXAD  ECIMAL O  R IN ASC  II, ON T  HE SERIA  L PRINTE  R.  PART  IAL SECT  ORS
0160  *OR ENTI  RE SECTO  RS CAN B  E DUMPED  .  THE B  EGINNING   AND END  ING ADDR  ESSES OF   THE
0240  *AREA TO   BE DUMP  ED ARE E  NTERED O  N THE TE  LETYPE,   IN THE F  ORM CYLI  NDER/TRA  CK/SECTO
```

Figure 3-12. Disc Dump Listing in ASCII Code

7.3.1 Disc Dump Program Assumptions

The PTS-100 Disc Dump program assumes:

- Discs with two tracks per cylinder.
- The "dump to" address is greater than or equal to the "dump from" address. If this is not true, the program will dump to the end of the disc.
- The user always responds to the IDENTIFICATION request by typing any 20 characters on the display or teletype keyboard.

7.3.2 Input to the Disc Dumping Process

After loading the Disc Dump program, if the display keyboard is going to be used to input directives the following sequence must take place:

1. User strikes any key on the display keyboard to be used to input the directives.
2. The display reads:
   CHARS PER LINE/NO OF LINES 99/99=
3. User responds by typing a two-digit decimal number representing the number of characters on one line of the display, followed by another two-digit decimal number representing the number of lines on the display.

When the teletypewriter is used for input the process begins with step 4.

4. The Disc Dump program issues seven messages to the display or teletype, each expecting a response. The sequence of messages and permissible response directives are as follows:

Program Message	User Types Reply
DEVICE ADDRESS =	one-digit drive address (0-7)
FROM CYLINDER TRACK SECTOR 999/99/99 =	cylinder, track, and sector data (in format indicated) of dump start location; leading zeros required
TO CYLINDER TRACK SECTOR 999/99/99 =	cylinder, track, and sector data (in format indicated) of dump end location; leading zeros required
HEX/ASCII =	HEX for hexadecimal output, or ASC for ASCII output
PARTIAL SECTOR 999 OR ALL =	three-digit byte count for partial sector dump; or ALL for full record dump
IDENTIFICATION =	20 characters to appear in the heading of the dump for identification purposes; any character is allowed

After the user has typed 20 characters following the IDENTIFICATION message, the disc will be dumped as directed. At the completion of the dump, the following message will be written on the teletype or display:

Program Message	User Types Reply
END OF DUMP ? =	Y will terminate the Disc Dump program. N will call up the first message listed above and reenter the dump cycle.

If the user types an invalid reply to any program message, the message is reissued. The display version of the disc dump can be terminated at any point by depressing the CANCEL key.

7.3.3 Disc Dump Output

The Disc Dump program generates three types of print lines on the listing. As shown in Figures 3-11 and 3-12, the first print line is the header, which appears once; this is the 20 characters that the user typed in reply to the IDENTIFICATION message. The second type of print line has the format:

```
BYTE  CYLINDER___  TRACK  ___  SECTOR___
```

and appears for each disc sector dumped, indicating the cylinder, track, and sector numbers. BYTE is the heading for a byte number, which will be the left-most entry on each following detail print line.

The third type of print line is the detail line, containing the dumped data. Four or eight detail lines will be printed per sector, depending on whether hexadecimal or ASCII representation was specified. Eight lines will be printed for hexadecimal notation, and four lines for ASCII code. Each line will begin with the byte number of the first byte to be printed on the line, followed by 80 printed characters. (For hexadecimal notation the byte numbers on the eight lines will be 0, 40, 80, 120, 160, 200, 240, and 280; for ASCII code the byte numbers will be 0, 80, 160, and 240.) Format of the detail lines is as follows:

```
9999* XXXXXXXX XXXXXXXX XXXXXXXX...
```

Padding characters (X'00') within a sector will be suppressed from the printout when full sectors are being dumped.

*Byte number of the first byte of the following printed characters.

Section 8. CASSETTE UTILITY PROGRAM

This program provides a method of storing on, deleting, copying, positioning, and printing the contents of cassette magnetic tape files.

The Cassette Utility program may be directed to perform the following functions:

- Copy all or parts of one cassette magnetic tape to another.
- Forward or backspace one tape a specified number of records.
- Position a cassette tape to a specific record located by matching a keyword.
- Rewind a tape to its beginning.
- Print a specified number of records from one tape.
- Read cards from the card reader and write information to a tape.
- Print the input directives.

The input directives can be keyed on any 2260 display keyboard, but all inputs must be keyed from the same one.

The Cassette Utility program requires the IOCS monitor for all input and output data and device assignments. Peripheral devices that may be called by the program include:

Logical Name	Description
SYSD (LUN3)	Card reader. Used to read in cards as requested by a Write directive (see subsection 8.2.6).
SYST (LUN5)	Serial printer. Used optionally to print directives and for Edit function printing (see subsection 8.2.2).
SYS0 (LUN6)	Display keyboard. Used to enter directives.
LOG8 (LUN8)	Magnetic tape cassette for performing the requested operation.
LOG9 (LUN9)	Teletypewriter printer. Used optionally to print directives and for Edit function printing (see subsection 8.2.2).

The system flow is shown in Figure 3-13.

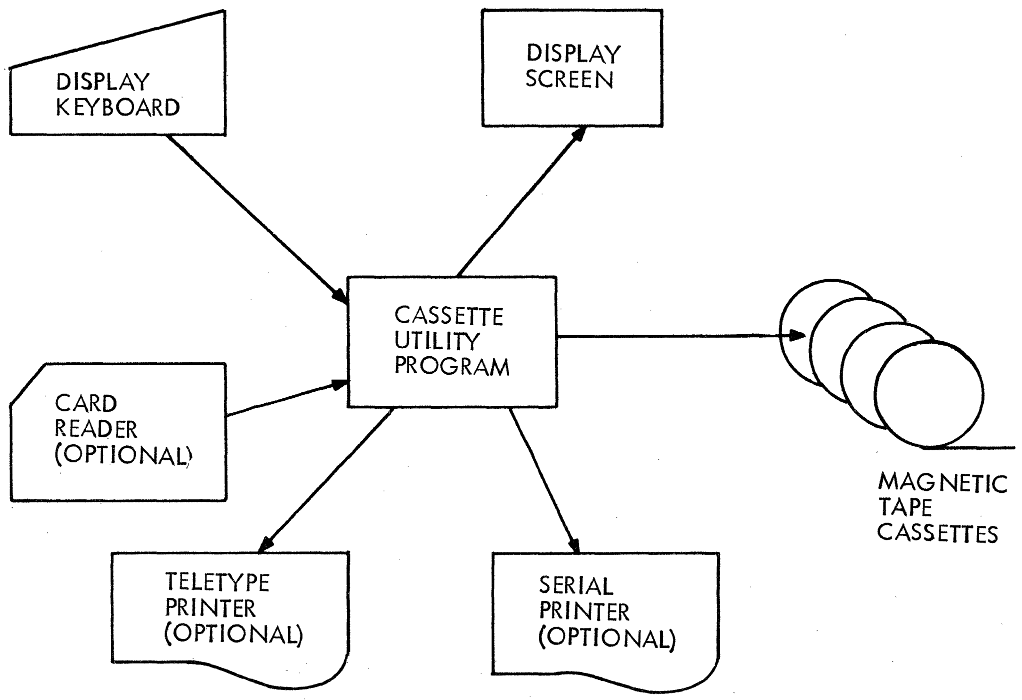


Figure 3-13. Cassette Utility Program Flowchart

8.1 Input and Output Devices

This program can have three types of inputs: input directives keyed by the operator on the disply keyboard, one of four cassette files, and optionally the card reader (for the read-to-tape function). Output will be to one of four cassette files and optionally to the teletypewriter and/or serial printers if printout is requested of the directives or files.

8.2 Operator Input

The Cassette Utility program is loaded using the Absolute/Relocating Loader program. After loading, the operator activates the program by striking any key on any of the display keyboards connected to the PTS-100 system. The message NC/CPL/DO/LO will then appear on the display associated with the keyboard in use.

The operator should respond on the same keyboard by keying the following hexadecimal information (using the symbol / as a separator):

Message	Decimal	Hexadecimal
Number of characters on screen (NC)	480	IE0
	960	3C0
	1920	780
Characters per line (CPL)	40	28
	64	40
	80	50
Device on which keyed directives are to be listed (DO):		
no listing		00
serial printer		05
teletype		09
Device on which file records are to be printed as output of Edit function (LO)		
no listing		00
serial printer		05
teletype		09

These must be followed by the ENTER key, signifying end of message. A typical operator response to NC/CPL/DO/LO might be:

3C0/40/09/05 ENTER

signifying a 960 character screen, 40 characters per line, printout of keyed directives on the teletypewriter, and printout of the file records on the serial printer.

The program then displays on the screen the following listing of functions, their command formats, and the request for input.

FUNCTION	COMMAND
COPY	C/F/T/NNNN
EDIT	E/#/NNNN/M
FORWARD SPACE	F/#/NNNN
BACKSPACE	B/#/NNNN
REWIND	R/#
WRITE	W/#/M
SEARCH	S/#/SSSS/VVVV
REQUEST?	

The operator responses are described in detail in the following subsections. He must depress the ENTER key to indicate end of message. He may also use the backspace and forwardspace keys to edit his input before entering it. Depressing the CANCEL key will terminate the function at the end of the next cassette Read operation.

8.2.1 Copy Function

The Copy function is used to copy one cassette tape to another. No rewind of either tape is performed. Therefore the tapes must be positioned (using Forward Space, Backspace, Rewind, or Search functions) previous to the Copy function. The keyboard entries necessary for this function are:

C/F/T/NNNN

where

C	=	Copy function
F	=	"from" cassette drive number (0, 1, 2 or 3)
T	=	"to" cassette drive number (0, 1, 2 or 3)
NNNN	=	number of records.

The Copy function will stop when it has copied NNNN records or it encounters 1EOF in positions 1 through 4 of a record, on the "from" tape; the 1EOF will be copied to the "to" tape.

8.2.2 Edit Function

The Edit function is used to print a specified number of records from any one of four cassette tape units on a serial printer or teletypewriter printer. The printout may be hexadecimal or in ASCII code. The keyboard entries necessary for this function are:

E/#/NNNN/M

where

E	=	Edit function
#	=	cassette drive number (0, 1, 2, or 3)
NNNN	=	number of records
M	=	mode (H = hexadecimal, A = ASCII)

The printed output will include the tape record number of the record being printed.

8.2.3 Forward Space Function

The Forward Space function is used to position any cassette tape forward a specified number of records. The keyboard entries necessary for this function are:

F/#/NNNN

where

F	=	Forward Space function
#	=	cassette drive number (0, 1, 2, or 3)
NNNN	=	number of records

8.2.4 Backspace Function

The Backspace function is used to position any cassette tape backward a specified number of records. The keyboard entries necessary for this function are:

B/#/NNNN

where

B	=	Backspace function
#	=	cassette drive number (0, 1, 2, or 3)
NNNN	=	number of records

8.2.5 Rewind Function

The Rewind function is used to position any cassette tape to its beginning. The keyboard entries necessary for this function are:

R/#

where

R	=	Rewind function
#	=	cassette drive number (0, 1, 2, or 3)

8.2.6 Write Function

The Write function is used to read either Hollerith or binary information from punched cards and to write the information unblocked onto any magnetic tape cassette unit. The keyboard entries necessary for this function are:

W/#/M

where

W	=	Write function
#	=	cassette drive number (0, 1, 2, or 3)
M	=	mode (H = Hollerith, B = binary)

The Write operation begins with the first card read. The end of file card for Hollerith has 1EOF punched in columns 1 through 4; 1EOF is copied onto the tape. The end of file card for binary has a 2-7-8-9 punch in column 1; this is not written to the tape.

8.2.7 Search Function

The Search function is used to position a cassette tape at a certain record. The program reads the tape until it encounters a specified key. The keyboard entries necessary for this function are:

$/#/SSSS/VVVV

where

S	=	Search function
#	=	cassette drive number (0, 1, 2, or 3)
SSSS	=	starting position of the key in hex (0000 is the first position in the record)
VVVV	=	value of the key in ASCII (up to 15 characters long)

If 1EOF is read on the tape before the key is found, the message NOT FOUND will be displayed on the screen.

8.3 Error Messages

If the keyed function cannot be completed, one of the following error messages will be displayed:

Error Message	Meaning
READ ERROR	Unable to read cassette record.
CASSETTE NOT OPERATIONAL	Cassette tape not in drive, or cassette not working.
KS NOT OPERATIONAL	Teletypewriter printer not working.
SP NOT OPERATIONAL	Serial printer not working.
CR NOT OPERATIONAL	Card reader not working.
END OF TAPE	Have reached end of tape.
KEYBOARD ERROR	Keyboard not working.
NOT FOUND	On search, value was not found.

INDEX TO PART 3

PART 4

PTS-100 MACRO LIBRARY FILES

PART 4

PTS-100 MACRO LIBRARY FILES

TABLE OF CONTENTS

TABLE OF CONTENTS (cont)

PART 4. PTS-100 MACRO LIBRARY FILES

Section 1. SYSTEM MACRO LIBRARY FILE

The System Macro Library file is a collection of all the generalized, source-coded routines used to create a user-specialized Input/Output Control System (IOCS) monitor. That is, the System Macro Library file contains all of the optional and standard routines that may be incorporated into an IOCS monitor to be created for any given user of a PTS-100. Any routine on the System Macro Library file may be called by the System Generator (SYSGEN) program, which generates macro call statements according to user-specified system descriptions. The SYSGEN macro call statements are written onto an Assembler-formatted file, which is input, along with the System Macro Library file, to a PTS-100 Assembler run, and processed by the macro processor phase to produce specialized IOCS monitor routines.

Presented on the following pages are detailed descriptions of the macro routines comprising the IOCS monitor, and the manner in which they are specialized to create a unique version of the monitor. The purpose of these descriptions is to indicate to the user the intricate relationship and structure of the various parts of the monitor and the technical requirements that must be met if additions or alterations are to be made to the IOCS monitor supplied by Raytheon Data Systems.

1.1 System Cells Macro Routine (#IMSCP)

This macro routine causes the IOCS monitor's global communications (system cell) area to be initialized in decimal locations 0-127. That is, it sets up the cross-reference area via which routines within the monitor can be accessed by other monitor routines. Specifically, this routine establishes the origin of the monitor at location 0, reserves communications areas, identifies symbols used in monitor routines other than itself, and assigns the addresses of all monitor service routines, some of which are OPEN LUN, CLOSE LUN, EXIT, INITialization, and IOACT. The system cells routine is always called by the SYSGEN program, which generates the call

$Δ #IMSCP

Since there is no actual argument list supplied in this macro call, and no dummy arguments in the routine, the entire #IMSCP routine is incorporated in the IOCS monitor being generated.

1.2 Interrupt Packet Initialization Macro Routine (#IMIP)

This macro routine is called by the SYSGEN program to effect the creation of the interrupt packets of the IOCS monitor. The routine contains specialized coding for interrupt packets for interrupt levels 0 and 9, and generalized coding for the packets of the external interrupt levels 1 - 8 and the parity interrupt level 10, as shown in Figure 4-1.

Notice that the generalized coding for the interrupt packets contain dummy arguments in the form

$$n, \overset{E}{\underset{N}{}}, 00$$

This type of dummy argument specifies that the macro routine statement in which it appears is to be omitted from the specialized routine if the nth actual argument in the associated macro call statement is equal (E) or not equal (N) to the value 00. The associated macro call generated by the SYSGEN program is shown below:

```
?#IMIP
*********************************************************************
*********************************************************************
*
*                INTERRUPT PACKETS
*
*********************************************************************
*********************************************************************
*
*
#ITIP0  RESV,0 2           OLD PC FOR LEVEL 0
        RESV,0 2           OLD LEVEL AND CB
        ADC    #IYCLK      CLOCK   SERVICE ROUTINE
        INR                INTERRUPT RETURN
*
#ITIP1  RESV,0 2           OLD PC FOR LEVEL 1
        RESV,0 2           OLD LEVEL AND CB
        ADC    #IXSR1      LEVEL 1 SERVICE ROUTINE        (1,E,00)
        ADC    *+2         IGNORE INTERRUPT               (1,N,00)
        INR                INTERRUPT RETURN
*
#ITIP2  RESV,0 2           OLD PC FOR LEVEL 2
        RESV,0 2           OLD LEVEL AND CB
        ADC    #IXSR2      LEVEL 2 SERVICE ROUTINE        (2,E,00)
        ADC    *+2         IGNORE INTERRUPT               (2,N,00)
        INR                INTERRUPT RETURN
*
#ITIP3  RESV,0 2           OLD PC FOR LEVEL 3
        RESV,0 2           OLD LEVEL AND CB
        ADC    #IXSR3      LEVEL 3 SERVICE ROUTINE        (3,E,00)
        ADC    *+2         IGNORE INTERRUPT               (3,N,00)
        INR                INTERRUPT RETURN
*
#ITIP4  RESV,0 2           OLD PC FOR LEVEL 4
        RESV,0 2           OLD LEVEL AND CB
        ADC    #IXSR4      LEVEL 4 SERVICE ROUTINE        (4,E,00)
        ADC    *+2         IGNORE INTERRUPT               (4,N,00)
        INR                INTERRUPT RETURN
*
#ITIP5  RESV,0 2           OLD PC FOR LEVEL 5
        RESV,0 2           OLD LEVEL AND CB
        ADC    #IXSR5      LEVEL 5 SERVICE ROUTINE        (5,E,00)
        ADC    *+2         IGNORE INTERRUPT               (5,N,00)
        INR                INTERRUPT RETURN
*
#ITIP6  RESV,0 2           OLD PC FOR LEVEL 6
        RESV,0 2           OLD LEVEL AND CB
        ADC    #IXSR6      LEVEL 6 SERVICE ROUTINE        (6,E,00)
        ADC    *+2         IGNORE INTERRUPT               (6,N,00)
        INR                INTERRUPT RETURN
*
#ITIP7  RESV,0 2           OLD PC FOR LEVEL 7
        RESV,0 2           OLD LEVEL AND CB
        ADC    #IXSR7      LEVEL 7 SERVICE ROUTINE        (7,E,00)
        ADC    *+2         IGNORE INTERRUPT               (7,N,00)
        INR                INTERRUPT RETURN
*
#ITIP8  RESV,0 2           OLD PC FOR LEVEL 8
        RESV,0 2           OLD LEVEL AND CB
        ADC    #IXSR8      LEVEL 8 SERVICE ROUTINE        (8,E,00)
        ADC    *+2         IGNORE INTERRUPT               (8,N,00)
        INR                INTERRUPT RETURN
*
#ITIP9  RESV,0 2           OLD PC FOR LEVEL 9
        RESV,0 2           OLD LEVEL AND CB
        ADC    #IUMSC      MONITOR SERVICE CALL ROUTINE
        RESV,0 2           SPARE
*
#ITIPA  RESV,0 2           OLD PC FOR LEVEL 10
        RESV,0 2           OLD LEVEL AND CB
        ADC    #IVPAR      PARITY ERROR RETURN            (10,E,00)
        ADC    *+2         IGNORE INTERRUPT               (10,N,00)
        INR                INTERRUPT RETURN
        SKIP   P
         END
```

Figure 4-1. Generalized Coding of the Interrupt Packet Initialization Routine

$\$\Delta$ #IMIP Δ $Arg_1, Arg_2, Arg_3, Arg_4, Arg_5, Arg_6, Arg_7, Arg_8, 00, Arg_{10}$

The actual arguments, Arg_1 through Arg_{10}, correspond to the interrupt levels 1 through 10. That is, the first argument in the list pertains to external interrupt level 1, the second to interrupt level 2, etc. The 9th argument is a default argument to cause an interrupt packet to be generated for level 9, the interrupt level at which the monitor runs. The level 10 packet is generated only if an argument other than 00 appears in position 10 of the #IMIP call. The argument list is generated by SYSGEN to indicate whether or not one or more devices have been assigned to the interrupt levels, as specified on the ASGP directive. Thus, if a device is assigned to interrupt level 1, the first argument's value is 01. In this manner, the remaining arguments indicate both the level and assignment of devices to the level. If no device assignment has been specified for a given interrupt level, SYSGEN generates an actual argument of 00 for the corresponding position of the argument list.

When the $\$\Delta$#IMIP macro call is processed by the Assembler's macro phase, the arguments are read and inserted in a table in the order of their appearance in the list. The #IMIP macro routine is then located in the System Macro Library file and the specialization of the routine is performed. The first step in processing is to write the coding for the level 0 interrupt packet onto the IOCS output file. The coding for packet 1 is then read. The first two statements are written onto the output file. The third statement contains the dummy argument (1, E, 00). To specialize this statement, the processor compares argument 1 in the argument table with the specified value 00. If the two values are equal, the ADC statement is omitted from the specialized interrupt packet. That is, if the argument value is 00, no device has been assigned to this interrupt level, hence no service routine will be generated for the interrupt level. Statement 4 in the packet coding contains the dummy argument (1, N, 00). When the statement is selected for processing, the actual argument 1 is compared to 00, and if they are not equal the statement is omitted from the interrupt packet coding that is written onto the output file for the IOCS monitor being created. Thus, the coding of interrupt packets 1 through 8 specifies that either statement 3 or 4, but not both, is to be included in the specialized coding, depending on whether or not a device assignment has been specified for the corresponding interrupt level.

The interrupt packet for level 9 contains no dummy arguments, and is therefore written onto the output file of the monitor.

The interrupt packet for level 10 is specialized with the address constant statement assigning the address of the parity routine if the value of actual argument 10 in the macro call statement is other than 00. The value of actual argument 10 is assigned in the SYSGEN CALL directive which specifies the optional Parity Error Routine, as described in subsection 1.20.

1.3 Logical IO Control Table Macro Routine (#IMCTL)

The IO Control Table (IOCT) of the IOCS monitor consists of two parts: the logical-to-physical device pointers and the Physical Control Blocks (PCBs). The logical-to-physical pointers portion of the table is created by the Assembler via specialization of this macro routine. The logical-to-physical pointers portion of the IOCT will contain 13 one-word entries containing the logical unit name, the identifier of the physical unit assigned to the logical unit, and the logical unit number (LUN) of that particular device. That is, for any given IOCS monitor, 13 logical units may be assigned to actual physical devices. Eight logical units may be assigned for the use of systems programs (e.g., the Assembler, the loaders, the debug and dump programs, etc.). Five logical units may be assigned for use of applications programs. The names of the system-reserved logical units begin with the characters #ISYS, followed by an additional character which denotes the use of the unit by systems programs. The names of logical units that may be assigned for applications programs begin with the characters #ILOG. The logical-to-physical pointers are developed from user-specified physical device identifiers on the ASGL directive in the SYSGEN input deck. That is, the ASGL directive may assign from one to 13 devices to logical units by entering the device identifiers in the order in which they are to be assigned to logical units in the IOCT. If no physical device is to be assigned to a logical unit, the user indicates the omission by three zeros in the corresponding position on the ASGL directive. When the SYSGEN program analyzes the ASGL directive, a macro call is generated in the format:

$$\$\Delta\,\#\text{IMCTL}\,\Delta\,D_1, D_2, D_3, \ldots, D_{13}$$

where the Ds are device identifiers, described in the detailed discussion of the PCBs in the following subsection.

When the Assembler macro processor encounters the $\$\Delta$ #IMCLT macro call, it constructs an argument table, enters the D arguments in the table in the order in which they appear in the call statement list, locates the #IMCLT macro routine on the System Macro Library file, and specializes the IOCT logical-to-physical pointers by inserting the table entries in the corresponding dummy argument positions in the generalized #IMCTL routine, shown in Figure 4-2.

The second portion of the IOCT, the Physical Control Blocks, is generated from the generalized #IMCTP macro routine, described in the following subsection. Notice that the specialized #IT(n) address constant becomes the label of the corresponding PCB for the assigned device.

```
?#IMCTL
**************************************************************************
**************************************************************************
*
*                       IO CONTROL TABLE - LOGICAL
*
**************************************************************************
**************************************************************************
*
*
        PGØ     #ITØØØ
#ITØØØ  EQU      #IØABS
#ISCTL  EQU     *
#ISYSF  ADC     #IT(1)      SYSTEM FILE
#ISYSI  ADC     #IT(2)      SYSTEM INPUT
#ISYSL  ADC     #IT(3)      SYSTEM LOG
#ISYSD  ADC     #IT(4)      SYSTEM DATA
#ISYSB  ADC     #IT(5)      BINARY INPUT
#ISYST  ADC     #IT(6)      LISTING
#ISYSO  ADC     #IT(7)      SYSTEM OUTPUT
#ISYSR  ADC     #IT(8)      SCRATCH
#ILOG8  ADC     #IT(9)      LOGICAL UNIT 8
#ILOG9  ADC     #IT(1Ø)     LOGICAL UNIT 9
#ILOGA  ADC     #IT(11)     LOGICAL UNIT A
#ILOGB  ADC     #IT(12)     LOGICAL UNIT B
#ILOGC  ADC     #IT(13)     LOGICAL UNIT C
        SKIP    P
         END
```

Figure 4-2. Generalized Coding of the #IMCTL Routine

1.4 Physical Control Block Macro Routine (#IMCTP)

The Physical Control Blocks (PCBs) form the second portion of the Input/Output Control Table in the IOCS monitor. The PCBs are seven-word tables that specify the necessary information to control the physical devices within the equipment configuration. That is, a PCB is created for each device to be serviced by the monitor.

The information for each PCB is specified in sets of parameters on the ASGP directive that is input to SYSGEN. Each set of parameters is specified as follows:

DID, DAD, DIL, DSC, DIM, SEN

where:

DID is one of the following:

- the device identifier if this is the only device of this type to be attached to a multiplex controller.
- the first of a group of identical devices.
- a comma if more than one device of this type is attached to a given multiplex controller, as specified by the DIM parameter in position 5 of the parameter set.

DAD is the device hardware address of the assigned device.

DIL is the external interrupt level 1 through 8 to which the device is being assigned.

DSC is a two-character code to be used to effect the unique identification of the particular device's Physical Input/Output Table (PIOT).

DIM is one of the following:

a comma if the DID parameter specifies a device identifier

the multiplexed device identifier if DID is a comma, indicating that two or more devices of the same type are attached to a given multiplex controller

SEN is a sentinel specified as one of the following:

a comma if the DIM parameter is a comma or if this is not the last device of the same type to be attached to a given multiplex controller

an L to indicate that the device identified by the DIM parameter just preceding is the last device of its type attached to a given multiplex controller.

For each parameter set specified on the ASGP directive, the SYSGEN program generates a macro call with the parameter set as its argument list, as shown below:

Δ#IMCTPΔ DID, DAD, DIL, DSC, DIM, SEN

At assembly time, the macro processor constructs an argument table for the Δ#IMCTP argument list, locates the #IMCTP macro routine on the System Macro Library file, and inserts the arguments for each device in place of dummy arguments in the PCB generalized macro routine, shown below. That is, a PCB is created for each device assigned on the ASGP directive.

```
?#IMCTP
*
*-----PHYSICAL CONTROL BLOCKS
*
#IT(1)  HEX      (3)                                          (1N)
        HEX      (3)                                          (1Y)
        HEX      1(2)        DEVICE ADDRESS
        ADC      #ID(1)      DEVICE DRIVER                    (1N)
        ADC      #ID(5)      DEVICE DRIVER                    (1Y)
        RESV,0   4           IOCQ CONTROL
        ADC      #ITP(4)     PIOT ADDRESS
        RESV,0   2           SPARE                            (6Y)
        HEX      FFFF        MPX SENTINEL                     (6N)
         END
```

Notice that if the DID parameter is specified, the first HEX statement of the generalized routine is specialized with the DID completing the label and the interrupt level (DIL) completing the operand field.

If DID is not specified, the first HEX statement is omitted from the routine and the interrupt level is entered as the operand field of the second HEX statement. In all cases, the DAD parameter must appear and is inserted to complete the operand field of the third HEX statement.

The first ADC statement operand is completed with the DID value, if specified. If it is not specified, the statement is omitted from the specialized routine. The second ADC statement is specialized if the first ADC statement is omitted. That is, if the DID parameter is not specified, the DIM parameter must be specified and is used to complete the multiplexed Device Driver Routine address.

The PIOT address in the last ADC statement is specialized with the DSC parameter. If the SEN parameter is specified, the last RESV statement is omitted from the specialized routine and the last HEX statement is included. If the SEN parameter is not specified, the last RESV statement is included and the following HEX statement is omitted when the specialized coding is written by the Assembler.

1.5 Channel Interface Control Block Macro Routine (#IMCCB)

The #IMCCB macro routine is called once for each device assigned to the Channel Interface Controller via the ACIC directive input to SYSGEN. Each macro call is in the format:

$\$\Delta$#IMCCB$\Delta$TAG, CAD, PDAD

where:

TAB is the symbolic tag to be assigned to the starting location of the CCB for each assigned device.

CAD is the channel address of the device.

PDAD is the physical address of a specific device within a group of devices attached to a channel.

At assembly time, the #IMCCB routine is called and a channel control block (CCB) is specialized for each device by replacing the dummy arguments with the actual arguments in the SYSGEN macro call. The generalized coding of the #IMCCB macro routine is as follows:

```
?#IMCCB
*
*-----CIC DEVICE
*
#ITC(1)    HEX      0
           HEX      1(2)       CHANNEL ADDRESS
           ADC      #IDCT0     CIC DEVICE DRIVER
           RESV,0   4          IOCB CONTROL
           ADC      #ITD(1)    PACKET ADDRESS
           HEX      0(3)       DEVICE ADDRESS
           END
```

Notice that the first actual argument is used to specialize the label of the CCB and also to specialize the address constant #ITD(1) for the associated device packet. That is, a device packet is created for each assigned device. Device packets are created by specialization of the #IMCDP macro routine for which SYSGEN generates the call

$ Δ #IMCDP Δ nn, T

where:

nn is the total number of devices attached to the CIC (i. e., 16, 32, 48, or 64),

T specifies the tumble table to be used.

For each device specified by nn, a 16-byte device packet is created at assembly time. The generalized coding of the #IMCDP routine is shown in Figure 4-3.

The T argument is used to specialize the first statement in the #IMCDP routine. This first statement is the macro call statement:

$ Δ #IMCT(2)

which causes interrupt tumble tables to be created for use of the CIC. There are two macro routines which define interrupt tumble tables within the System Macro Library file: the prototype CIC interrupt tumble table routine (#IMCT1) and the production CIC interrupt tumble table routine (#IMCT2). The prototype routine will be called and inserted in the IOCS monitor being created if the actual value of the T argument is 1, and the production routine will be called and inserted if the actual value of T is 2. The generalized coding of the #IMCT1 and #IMCT2 routines is shown in Figure 4-4.

1.6 Logical Unit Assignment Macro Routine (#IMLAS)

This macro routine contains only comments, and provides user documentation of the logical unit assignments within his particular version of the IOCS monitor. The coding of the #IMLAS macro routine is shown in Figure 4-5. At SYSGEN time, the tags of the Physical Control Blocks (PCBs) of the assigned logical units are inserted in the place of the associated dummy arguments in the comments of the form

```
*#IT(n)     EQU     operand value        (nN)
```

if a logical unit has been assigned.

The #IMLAS comments are printed by the SYSGEN program. They are not, however, incorporated in the assembly-formatted file.
It is the responsibility of the programer to determine the LUN assignments from the documentation of his particular IOCS monitor.

1.7 Physical Input/Output Table Macro Routine (#IMPIT)

This routine is used to create a Physical Input/Output Table (PIOT) area for each device assigned on the ASGP directive input to SYSGEN. That is, the fourth parameter, DSC, in each set of ASGP parameters is used to specialize the beginning address (i. e., the label) of the PIOT

```
?#IMCDP
$ #IMCT(2)  (1)
**************************************************************
**************************************************************
*
*                 CIC DEVICE PACKETS
*
**************************************************************
**************************************************************
*
*
*
        MOD      256
#ITD00  RESV,0   16          DEVICE PACKET 0
        EXDEF    #ITD00
#ITD01  RESV,0   16          DEVICE PACKET 1
        EXDEF    #ITD01
#ITD02  RESV,0   16          DEVICE PACKET 2
        EXDEF    #ITD02
#ITD03  RESV,0   16          DEVICE PACKET 3
        EXDEF    #ITD03
#ITD04  RESV,0   16          DEVICE PACKET 4
        EXDEF    #ITD04
#ITD05  RESV,0   16          DEVICE PACKET 5
        EXDEF    #ITD05
#ITD06  RESV,0   16          DEVICE PACKET 6
        EXDEF    #ITD06
#ITD07  RESV,0   16          DEVICE PACKET 7
        EXDEF    #ITD07
#ITD08  RESV,0   16          DEVICE PACKET 8
        EXDEF    #ITD08
#ITD09  RESV,0   16          DEVICE PACKET 9
        EXDEF    #ITD09
#ITD0A  RESV,0   16          DEVICE PACKET A
        EXDEF    #ITD0A
#ITD0B  RESV,0   16          DEVICE PACKET B
        EXDEF    #ITD0B
#ITD0C  RESV,0   16          DEVICE PACKET C
        EXDEF    #ITD0C
#ITD0D  RESV,0   16          DEVICE PACKET D
        EXDEF    #ITD0D
#ITD0E  RESV,0   16          DEVICE PACKET E
        EXDEF    #ITD0E
#ITD0F  RESV,0   16          DEVICE PACKET F
        EXDEF    #ITD0F
*
#ITD10  RESV,0   16          DEV PACKET 10          (1,E,16)
        EXDEF    #ITD10                              (1,E,16)
#ITD11  RESV,0   16          DEV PACKET 11          (1,E,16)
        EXDEF    #ITD11                             (1,E,16)
#ITD12  RESV,0   16          DEV PACKET 12          (1,E,16)
        EXDEF    #ITD12                             (1,E,16)
#ITD13  RESV,0   16          DEV PACKET 13          (1,E,16)
        EXDEF    #ITD13                             (1,E,16)
#ITD14  RESV,0   16          DEV PACKET 14          (1,E,16)
        EXDEF    #ITD14                             (1,E,16)
#ITD15  RESV,0   16          DEV PACKET 15          (1,E,16)
        EXDEF    #ITD15                             (1,E,16)
#ITD16  RESV,0   16          DEV PACKET 16          (1,E,16)
        EXDEF    #ITD16                             (1,E,16)
#ITD17  RESV,0   16          DEV PACKET 17          (1,E,16)
        EXDEF    #ITD17                             (1,E,16)
#ITD18  RESV,0   16          DEV PACKET 18          (1,E,16)
        EXDEF    #ITD18                             (1,E,16)
#ITD19  RESV,0   16          DEV PACKET 19          (1,E,16)
        EXDEF    #ITD19                             (1,E,16)
#ITD1A  RESV,0   16          DEV PACKET 1A          (1,E,16)
        EXDEF    #ITD1A                             (1,E,16)
#ITD1B  RESV,0   16          DEV PACKET 1B          (1,E,16)
        EXDEF    #ITD1B                             (1,E,16)
#ITD1C  RESV,0   16          DEV PACKET 1C          (1,E,16)
        EXDEF    #ITD1C                             (1,E,16)
```

Figure 4-3. Generalized Coding of the #IMCDP Routine (Sheet 1 of 2)

```
#ITD1D  RESV,Ø   16          DEV PACKET 1D      (1,E,16)
        EXDEF    #ITD1D                         (1,E,16)
#ITD1E  RESV,Ø   16          DEV PACKET 1E      (1,E,16)
        EXDEF    #ITD1E                         (1,E,16)
#ITD1F  RESV,Ø   16          DEV PACKET 1F      (1,E,16)
        EXDEF    #ITD1F                         (1,E,16)
#ITD2Ø  RESV,Ø   16          DP 2Ø      (1,E,16)(1,E,32)
        EXDEF    #ITD2Ø                 (1,E,16)(1,E,32)
#ITD21  RESV,Ø   16          DP 21      (1,E,16)(1,E,32)
        EXDEF    #ITD21                 (1,E,16)(1,E,32)
#ITD22  RESV,Ø   16          DP 22      (1,E,16)(1,E,32)
        EXDEF    #ITD22                 (1,E,16)(1,E,32)
#ITD23  RESV,Ø   16          DP 23      (1,E,16)(1,E,32)
        EXDEF    #ITD23                 (1,E,16)(1,E,32)
#ITD24  RESV,Ø   16          DP 24      (1,E,16)(1,E,32)
        EXDEF    #ITD24                 (1,E,16)(1,E,32)
#ITD25  RESV,Ø   16          DP 25      (1,E,16)(1,E,32)
        EXDEF    #ITD25                 (1,E,16)(1,E,32)
#ITD26  RESV,Ø   16          DP 26      (1,E,16)(1,E,32)
        EXDEF    #ITD26                 (1,E,16)(1,E,32)
#ITD27  RESV,Ø   16          DP 27      (1,E,16)(1,E,32)
        EXDEF    #ITD27                 (1,E,16)(1,E,32)
#ITD28  RESV,Ø   16          DP 28      (1,E,16)(1,E,32)
        EXDEF    #ITD28                 (1,E,16)(1,E,32)
#ITD29  RESV,Ø   16          DP 29      (1,E,16)(1,E,32)
        EXDEF    #ITD29                 (1,E,16)(1,E,32)
#ITD2A  RESV,Ø   16          DP 2A      (1,E,16)(1,E,32)
        EXDEF    #ITD2A                 (1,E,16)(1,E,32)
#ITD2B  RESV,Ø   16          DP 2B      (1,E,16)(1,E,32)
        EXDEF    #ITD2B                 (1,E,16)(1,E,32)
#ITD2C  RESV,Ø   16          DP 2C      (1,E,16)(1,E,32)
        EXDEF    #ITD2C                 (1,E,16)(1,E,32)
#ITD2D  RESV,Ø   16          DP 2D      (1,E,16)(1,E,32)
        EXDEF    #ITD2D                 (1,E,16)(1,E,32)
#ITD2E  RESV,Ø   16          DP 2E      (1,E,16)(1,E,32)
        EXDEF    #ITD2E                 (1,E,16)(1,E,32)
#ITD2F  RESV,Ø   16          DP 2F      (1,E,16)(1,E,32)
        EXDEF    #ITD2F                 (1,E,16)(1,E,32)
#ITD3Ø  RESV,Ø   16          DP 3Ø      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD3Ø                 (1,E,16)(1,E,32)(1,E,48)
#ITD31  RESV,Ø   16          DP 31      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD31                 (1,E,16)(1,E,32)(1,E,48)
#ITD32  RESV,Ø   16          DP 32      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD32                 (1,E,16)(1,E,32)(1,E,48)
#ITD33  RESV,Ø   16          DP 33      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD33                 (1,E,16)(1,E,32)(1,E,48)
#ITD34  RESV,Ø   16          DP 34      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD34                 (1,E,16)(1,E,32)(1,E,48)
#ITD35  RESV,Ø   16          DP 35      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD35                 (1,E,16)(1,E,32)(1,E,48)
#ITD36  RESV,Ø   16          DP 36      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD36                 (1,E,16)(1,E,32)(1,E,48)
#ITD37  RESV,Ø   16          DP 37      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD37                 (1,E,16)(1,E,32)(1,E,48)
#ITD38  RESV,Ø   16          DP 38      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD38                 (1,E,16)(1,E,32)(1,E,48)
#ITD39  RESV,Ø   16          DP 39      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD39                 (1,E,16)(1,E,32)(1,E,48)
#ITD3A  RESV,Ø   16          DP 3A      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD3A                 (1,E,16)(1,E,32)(1,E,48)
#ITD3B  RESV,Ø   16          DP 3B      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD3B                 (1,E,16)(1,E,32)(1,E,48)
#ITD3C  RESV,Ø   16          DP 3C      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD3C                 (1,E,16)(1,E,32)(1,E,48)
#ITD3D  RESV,Ø   16          DP 3D      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD3D                 (1,E,16)(1,E,32)(1,E,48)
#ITD3E  RESV,Ø   16          DP 3E      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD3E                 (1,E,16)(1,E,32)(1,E,48)
#ITD3F  RESV,Ø   16          DP 3F      (1,E,16)(1,E,32)(1,E,48)
        EXDEF    #ITD3F                 (1,E,16)(1,E,32)(1,E,48)
*
        SKIP     P
        END
```

Figure 4-3. Generalized Coding of the #IMCDP Routine (Sheet 2 of 2)

```
?#IMCT1
************************************************************
************************************************************
*
*                    INTERRUPT TUMBLE TABLES
*                        PROTOTYPE CIC
************************************************************
************************************************************
*
*
*------TUMBLE TABLE  1
        MOD      256
#ITIT1  RESV,Ø   128          INTERRUPT TUMBLE TABLE 1
        EXDEF    #ITIT1
*
*
*------TUMBLE TABLE  2
        MOD      256
#ITIT2  RESV,Ø   128          INTERRUPT TUMBLE TABLE 2
        EXDEF    #ITIT2
*
        SKIP     P
        END
?#IMCT2
************************************************************
************************************************************
*
*                    INTERRUPT TUMBLE TABLES
*                        PRODUCTION CIC
************************************************************
************************************************************
*
*
*------TUMBLE TABLE 1
        MOD      64
#ITIT1  RESV,Ø   64           INTERRUPT TUMBLE TABLE 1
        EXDEF    #ITIT1
*
*
*------TUMBLE TABLE  2
        MOD      64
#ITIT2  RESV,Ø   64           INTERRUPT TUMBLE TABLE 2
        EXDEF    #ITIT2
        RESV,Ø   512                    (1,E,16)(1,E,32)
        SKIP     P
        END
```

Figure 4-4. Generalized Coding of the #IMCT1 and #IMCT2 Routines

```
?#IMLAS
         SKIP     P
*****************************************************************
*****************************************************************
*
*                   LUN ASSIGNMENTS
*
*****************************************************************
*****************************************************************
*
*#ISYSF EQU      0           SYSTEM FILE
*#ISYSI EQU      1           SYSTEM INPUT
*#ISYSL EQU      2           SYSTEM LOG
*#ISYSD EQU      3           SYSTEM DATA
*#ISYSB EQU      4           BINARY INPUT
*#ISYST EQU      5           LISTING
*#ISYSO EQU      6           SYSTEM OUTPUT
*#ISYSR EQU      7           SCRATCH
*#ILOG8 EQU      8           LOGICAL UNIT 8
*#ILOG9 EQU      9           LOGICAL UNIT 9
*#ILOGA EQU      A           LOGICAL UNIT A
*#ILOGB EQU      B           LOGICAL UNIT B
*#ILOGC EQU      C           LOGICAL UNIT C
*#IT(1) EQU      D                                      (1N)
*#IT(2) EQU      E                                      (2N)
*#IT(3) EQU      F                                      (3N)
*#IT(4) EQU      10                                     (4N)
*#IT(5) EQU      11                                     (5N)
*#IT(6) EQU      12                                     (6N)
*#IT(7) EQU      13                                     (7N)
*#IT(8) EQU      14                                     (8N)
*#IT(9)  EQU     15                                     (9N)
*#IT(10) EQU     16                                     (10N)
*#IT(11) EQU     17                                     (11N)
*#IT(12) EQU     18                                     (12N)
*#IT(13) EQU     19                                     (13N)
*#IT(14) EQU     1A                                     (14N)
*#IT(15) EQU     1B                                     (15N)
*#IT(16) EQU     1C                                     (16N)
*#IT(17) EQU     1D                                     (17N)
*#IT(18) EQU     1E                                     (18N)
*#IT(19) EQU     1F                                     (19N)
*#IT(20) EQU     20                                     (20N)
*#IT(21) EQU     21                                     (21N)
*#IT(22) EQU     22                                     (22N)
*#IT(23) EQU     23                                     (23N)
*#IT(24) EQU     24                                     (24N)
*#IT(25) EQU     25                                     (25N)
*#IT(26) EQU     26                                     (26N)
*#IT(27) EQU     27                                     (27N)
*#IT(28) EQU     28                                     (28N)
*#IT(29) EQU     29                                     (29N)
*#IT(30) EQU     2A                                     (30N)
*#IT(31) EQU     2B                                     (31N)
*#IT(32) EQU     2C                                     (32N)
*
*
         SKIP     P
         END
```

Figure 4-5. Generalized Coding of the #IMLAS Macro Routine

associated with each device assigned at system generation time. When the SYSGEN program processes the ASGP directive in its input deck, it generates the macro call

$\$\Delta$ #IMPIT Δ $DSC_1, DSC_2, \ldots\ldots, DSC_n$

When the Assembler processes the macro call, it arranges the DSCs in an argument table, locates the #IMPIT routine on the System Macro Library file, and replaces the dummy arguments with the actual values in the argument table. The generalized coding of the #IMPIT routine is shown in Figure 4-6. Notice the dummy arguments of the form nN at the right side of each source statement. If the corresponding actual argument is not specified, the nN dummy argument causes the source statement to be omitted from the specialized routine. That is, no PIOT area is reserved and the symbolic tag of its beginning location is not completed.

At program execution time, execution of an IOACT service request to the IOCS monitor causes the appropriate programer-defined FIOB information to be moved to the associated IOCQ table entry, from which the monitor subsequently moves it into the PIOT associated with the device on which the I/O operation is to be performed. The PIOT is then used by the device driver and service routines to perform the required action.

```
?#IMPIT
*******************************************************************
*******************************************************************
*
*                 PIOT TABLES
*
*******************************************************************
*******************************************************************
*
*
        MOD       16              START AT 8 WORD BOUNDARY
*
#ITP(1) RESV,0    16              PIOT TABLE 1                (1N)
#ITP(2) RESV,0    16              PIOT TABLE 2                (2N)
#ITP(3) RESV,0    16              PIOT TABLE 3                (3N)
#ITP(4) RESV,0    16              PIOT TABLE 4                (4N)
#ITP(5) RESV,0    16              PIOT TABLE 5                (5N)
#ITP(6) RESV,0    16              PIOT TABLE 6                (6N)
#ITP(7) RESV,0    16              PIOT TABLE 7                (7N)
#ITP(8) RESV,0    16              PIOT TABLE 8                (8N)
#ITP(9) RESV,0    16              PIOT TABLE 9                (9N)
#ITP(10) RESV,0   16              PIOT TABLE 10               (10N)
#ITP(11) RESV,0   16              PIOT TABLE 11               (11N)
#ITP(12) RESV,0   16              PIOT TABLE 12               (12N)
#ITP(13) RESV,0   16              PIOT TABLE 13               (13N)
#ITP(14) RESV,0   16              PIOT TABLE 14               (14N)
#ITP(15) RESV,0   16              PIOT TABLE 15               (15N)
#ITP(16) RESV,0   16              PIOT TABLE 16               (16N)
#ITP(17) RESV,0   16              PIOT TABLE 17               (17N)
#ITP(18) RESV,0   16              PIOT TABLE 18               (18N)
#ITP(19) RESV,0   16              PIOT TABLE 19               (19N)
#ITP(20) RESV,0   16              PIOT TABLE 20               (20N)
#ITP(21) RESV,0   16              PIOT TABLE 21               (21N)
#ITP(22) RESV,0   16              PIOT TABLE 22               (22N)
#ITP(23) RESV,0   16              PIOT TABLE 23               (23N)
#ITP(24) RESV,0   16              PIOT TABLE 24               (24N)
        SKIP      P
         END
```

Figure 4-6. Generalized Coding of the #IMPIT Macro Routine

1.8 Driver Common Macro Routine (#IMCOM)

This routine is always incorporated in any IOCS monitor that is being created. That is, it is always called by SYSGEN via the macro call

$\Delta #IMCOM

The routine contains no dummy arguments, hence no actual argument list appears in the macro call.

The #IMCOM routine performs all device independent processing required to set up for any I/O operation and record the status of the completed operation.

The main functions of the Device Common Routine are:

Setting up for the return of control to the calling routine.

Setting up the PIOT with the information in words 2 through 9 of the IOCQ entry to be processed.

Testing for and recording I/O status in the logical status field of the IOCQ.

Disabling interrupts at the start of the IOCQ entry processing, and enabling interrupts when processing is complete.

Starting the required I/O operation.

1.9 Level Service Macro Routine (#IMLSR)

A specialized version of this routine is prepared for each external interrupt level 1-8. The routine reserves a storage area for saving registers of the interrupted program, establishes linkages to all driver and service routines for the devices assigned to each given interrupt level, enables invalid interrupt message logging, and restores the interrupted processing level.

For each interrupt level assigned on the ASGP directive, the SYSGEN program generates a macro call in the format shown below:

$$\$\Delta\text{#IMLSR}\Delta\,\mathrm{Arg}_1, \mathrm{Arg}_2, \overbrace{\mathrm{Arg}_3, \mathrm{Arg}_4, \mathrm{Arg}_5}^{\text{argument set 1}}, \overbrace{\mathrm{Arg}_6, \mathrm{Arg}_7, \mathrm{Arg}_8}^{\text{argument set 2}}, \ldots$$

$$/\ldots, \overbrace{\mathrm{Arg}_{24}, \mathrm{Arg}_{25}, \mathrm{Arg}_{26}}^{\text{argument set 8}}$$

where:

Arg_1 is the interrupt level to be serviced by a given level service routine

Arg_2 is a logging flag if the System Log device was assigned a LUN ID via the ASGL directive, or 00 if no device was assigned.

The remainder of the arguments in the $Δ#IMLSR statement call list form three argument sets, where the left argument in each set is the first two characters of the device identifier, the second argument is the third character of the device identifier (i.e., the decimal integer portion of the identifier, denoting the specific device of a given type), and the third and last argument in the set is the DSC parameter specified for each device on the ASGP directive. In this usage the DSC parameter effects specialization of the Device Driver and Device Service Routines for the assigned device. Eight argument sets may appear in a given $Δ#IMLSR macro call.

The general coding of the #IMLSR routine is shown in Figure 4-7. Notice that the interrupt level (argument 1) and the logging flag (argument 2) are inserted in numerous places within the routine. As many as eight Device Driver Routine macro calls may be specialized within the #IMLSR coding set. The generalized macro call coding from the #IMLSR routine is as follows:

```
$         #IMD(3)    (3),(4),(2),(1),(5)                        (3N)
$         #IMD(6)    (6),(7),(2),(1),(8)                        (6N)
$         #IMD(9)    (9),(10),(2),(1),(11)                      (9N)
$         #IMD(12)   (12),(13),(2),(1),(14)                     (12N)
$         #IMD(15)   (15),(16),(2),(1),(17)                     (15N)
$         #IMD(18)   (18),(19),(2),(1),(20)                     (18N)
$         #IMD(21)   (21),(22),(2),(1),(23)                     (21N)
$         #IMD(24)   (24),(25),(2),(1),(26)                     (24N)
```

Each Device Driver Routine name (#IMD(n)) is specialized by inserting the two character alphabetic portion of the device identifier in the place of the dummy argument that appears at the right end of the name. The two characters of the identifier also becomes the first actual argument in the call. The second actual argument contains the decimal integer portion of the device identifier. The third actual argument contains the logging flag, and the fourth actual argument contains the interrupt level. The fifth actual argument contains the DSC parameter from the ASGP directive.

Notice also that a given Driver Routine macro call is not generated if an argument set in the $Δ #IMLSR macro call is not specified, as controlled by the dummy arguments of the form (nN) at the right end of the generalized statement coding.

Note also that if no logging device has been assigned (i.e., if the second argument in the $Δ#IMLSR call is 00), the coding of the Invalid Interrupt Logging Routine is to be omitted from the specialized #IMLSR coding, as specified by the dummy argument of the form (n, E, 00) at the end of each statement of the generalized routine coding.

1.10 Device Driver and Service Macro Routines

For each device attached to a given PTS-100, a Device Driver Routine and a Device Service Routine must appear inthe IOCS monitor to be used on the system. For all standard devices supplied with the PTS-100, generalized driver and service macro routines appear on the System Macro Library file. For nonstandard devices, the user must assume the responsibility of designing, implementing, and incorporating the required driver and service routines in the IOCS monitor.

Generalized device driver and device service routines are called via macro calls within the level Service Routine (#IMLSR) for the interrupt level to which the associated device is assigned, as described in subsection 1.9. The generalized device driver and service macro routines are specialized with the three-character device identifier (i.e., arguments 1 and 2 in the macro call specialized within the #IMLSR macro routine). That is, the device identifier replaces dummy arguments 1 and 2 throughout the driver and service macro routines for the assigned device. Generalized macro routine coding for the Modem Send Driver and Modem Send Service routines is illustrated in Figure 4-8. Other driver and service routines for standard PTS-100 devices are coded in the same manner. That is, the device identifier is inserted throughout the generalized routines to specialize addresses, tags, etc.

1.11 Monitor Service Call (MSC) Macro Routine (#IMMSC)

This macro routine is incorporated in each IOCS monitor that is being created. It performs the necessary processing for entering interrupt level 9 and for calling the appropriate MSC routine (e.g., OPEN, CLOSE, INITialization, IOACT, etc.). The SYSGEN program generates the macro call statement:

$Δ#IMMSC

to cause the MSC routine to be incorporated in the IOCS monitor. Since there are no arguments in the call statement, the entire #IMMSC routine is written into the IOCS monitor file by the macro processor of the Assembler.

```
?#IMLSR
************************************************************************
************************************************************************
*
*                  LEVEL (1) SERVICE ROUTINE
*
*          ENTERED FROM NEW PC OF INTERRUPT PACKET
************************************************************************
************************************************************************
*
#IXT1(1) RESV,Ø     6
*------SAVE REGISTERS
#IXSR(1) SX1         #IXT1(1)        SAVE X1
          SX2        #IXT1(1)+2      SAVE X2
         STW         #IXT1(1)+4      SAVE AC
         SKIP        P
$         #IMD(3)    (3),(4),(2),(1),(5)                       (3N)
$         #IMD(6)    (6),(7),(2),(1),(8)                       (6N)
$         #IMD(9)    (9),(1Ø),(2),(1),(11)                     (9N)
$         #IMD(12)   (12),(13),(2),(1),(14)                    (12N)
$         #IMD(15)   (15),(16),(2),(1),(17)                    (15N)
$         #IMD(18)   (18),(19),(2),(1),(2Ø)                    (18N)
$         #IMD(21)   (21),(22),(2),(1),(23)                    (21N)
$         #IMD(24)   (24),(25),(2),(1),(26)                    (24N)
************************************************************** (2,E,ØØ)
************************************************************** (2,E,ØØ)
*                                                              (2,E,ØØ)
*-------INVALID INTERRUPT LOGGING                              (2,E,ØØ)
*                                                              (2,E,ØØ)
************************************************************** (2,E,ØØ)
************************************************************** (2,E,ØØ)
         LX2         #IX(1)PØ                                  (2,E,ØØ)
         JMP         #IZLOG                                    (2,E,ØØ)
         HEX         Ø(1)Ø6                                    (2,E,ØØ)
#IX(1)PØ ADC         *+2                                       (2,E,ØØ)
         SKIP    P                                             (2,E,ØØ)
************************************************************************
************************************************************************
*
*                  LEVEL (1) RESTORE ROUTINE
*
************************************************************************
************************************************************************
*
*------RESTORE REGISTERS
#IX(1)99 LX1         #IXT1(1)        RESTORE X1
          LX2        #IXT1(1)+2      RESTORE X2
          LDW        #IXT1(1)+4      RESTORE AC
         INR
         SKIP    P
          END
```

Figure 4-7. Generalized Coding of the #IMLSR Macro Routine

```
?#IMDMS
**********************************************************************
**********************************************************************
*
*                 MODEM SEND DRIVER
*
*
*------CALLING SEQUENCE
*       LAX2     RETURN
*       JMP,N,X1 4            PCB ADR IN X1
*RETURN NSI
**********************************************************************
**********************************************************************
*
*------DRIVER CONTROLS
         JMP      #IW(1)(2)
        HEX      8000         CHAINING FLAG
#IW(2)T0 ADC      #IT(1)(2)   PCB ADDRESS
*
*------SET UP RETURN
#ID(1)(2) SX2     #IW(2)T1    SET UP RETURN JUMP
*
*------TEST FOR LEGAL OPERATION
        LX2,X1   8            IOCQ-OUT ADR TO X2
        LDW,X2   4            FETCH COMMAND WORD
         AND      #IW(2)C1    =X'0100'
         CNE      #IW(2)C1
         BCB      #IW(2)01    ILLEGAL
        LDW,X2   4            FETCH COMMAND WORD
         AND      #IW(2)C2    =X'0E00'
         CAL      #IW(2)C3    =X'0900'
         BCB      #IW(2)20    LEGAL
         JMP      #IW(2)01    ILLEGAL
*
*------SET UP INTERRUPT MASK
#IW(2)20 LDI      AC,X'60'    FETCH INT MASK
        STB,X2   5            STORE IN IOCQ
*
*------CALL DRIVER COMMON
         SX1      #IW(2)P1    STORE PCB
         LAX2     #IW(2)40
        JMP      #IDCOM       GO TO DRIVER COMMON
#IW(2)P1 HEX      0           PCB ADDRESS
*
*------IS THE CHAIN FLAG SET  ?
#IW(2)40 LDW,X1   0           FETCH COMMAND WORD FROM PIOT
         AND      #IW(2)C4    =X'1000'
         BCB      #IW(2)30    YES
*
*------STOP CHAINING
         LDW,N    #IW(2)T0    FETCH PCB STATUS
         AND      #IW(2)C5    =X'FBFF'
         STW,N    #IW(2)T0    REPLACE PCB STATUS WORD
*
*------EXIT
#IW(2)49 JMP,N    #IW(2)T1    RETURN TO CALLER
*
*------ILLEGAL OPERATION
#IW(2)01 LDI      AC,X'0310',L SET UP STATUS CODE
        STW,X2   2            STORE IN IOCQ
         JMP      #IW(2)49    GO TO EXIT
*
*------CONTINUE CHAINING
#IW(2)30 LDW,N    #IW(2)T0    FETCH PCB STATUS WORD
         XOR      #IW(2)C4    =X'1000' INVERT A/B FLAG
         AND      #IW(2)C6    =X'FBFF'
         ADD      #IW(2)C7    =X'0400' SET SKIP FLAG
         STW,N    #IW(2)T0    REPLACE PCB STATUS WORD
         JMP      #IW(2)49    GO TO EXIT
```

Figure 4-8. Generalized Coding of the Modem Send Device Driver and Device Service Routine (#IMDMS) (Sheet 1 of 3)

```
*
*
*-----CONSTANTS
#IW(2)C1 HEX      0100
#IW(2)C2 HEX      0E00
#IW(2)C3 HEX      0900
#IW(2)C4 HEX      1000
#IW(2)C5 HEX      EBFF
#IW(2)C6 HEX      FBFF
#IW(2)C7 HEX      0400
*
*------VARIABLES
#IW(2)T1 HEX      0            RETURN ADDRESS
*
        SKIP      2
*********************************************************************
*********************************************************************
*
*                 MODEM SEND SERVICE ROUTINE
*
*--------ENTERED  FROM LEVEL  SERVICE ROUTINE
*********************************************************************
*********************************************************************
*
*------READ AND RESET STATUS
#IW(1)(2) LX1     #IW(2)T0     PUT PCB ADR IN X1
        LDW,X1    2            FETCH DEV ADR
          AND     #IW(2)CB     =X'0FFF'
          RTO     #IW(2)T3     READ + RESET ICB
        LX2,X1    8            IOCQ ADR TO X2
*
*------TEST FOR BYTE COUNT = 0
        LDI       AC,X'40'
          AND     #IW(2)T3     BYTE COUNT = 0?
          BCB     #IW(2)50     YES
*
*------TEST FOR DEVICE NOT READY
        LDI       AC,X'20'
          AND     #IW(2)T3     DEVICE NOT READY?
          BCB     #IW(2)70     YES
*
*------GO TO NEXT DSR
          JMP     #IW(2)99     GO TO NEXT DSR
*
*------BYTE COUNT = 0
#IW(2)50 LDW      #IW(2)CC     =X'0302'
        STW,X2    2            UPDATE IOCQ STATUS
*
*------IS CURRENT ORDER CHAINED ?
        LDW,X2    4            FETCH CURRENT ORDER
          AND     #IW(2)C4     =X'1000'
          BCB     #IW(2)52     CURRENT ORDER IS CHAINED
*
*
*
*------UPDATE IOCQ-OUT
        LDW,X2    0            NEW IOCQ ADR
          JMP     #IW(2)57
*
*------UPDATE STATUS OF NEW ENTRY
#IW(2)52  LX2,X2 0             NEW IOCQ TO X2
          SX2     #IW(2)T4     SAVE NEW IOCQ
          LDW     #IW(2)CD     =X'0200'
        STW,X2    2            UPDATE NEW IOCQ STATUS
```

Figure 4-8. Generalized Coding of the Modem Send Device Driver and Device Service Routine (#IMDMS) (Sheet 2 of 3)

```
*
*------IS NEW ORDER CHAINED
#IW(2)54  LDW,X2 4              FETCH NEW ORDER
          AND    #IW(2)C4       =X'1000'
          BCB    #IW(2)53       YES
          JMP    #IW(2)56       NO
*
*------LOOK-AHEAD TO NEXT IOCQ ENTRY
#IW(2)53 LX2,X2  0
        LDB,X2   2              FETCH LOGICAL STATUS
          CNE    #IW(2)C8       =X'0001'
          BCB    #IW(2)56       NOT I/O PENDING
*
*------GO TO MODEM SEND DRIVER
        SX2,X1   8              LOOK-AHEAD ADR TO PCB
          LAX2   #IW(2)56
        JMP,N,X1 4              GO TO DRIVER
*
*------RESTORE NEW IOCQ-OUT
#IW(2)56 LDW     #IW(2)T4
          LX1    #IW(2)T0       PUT PCB ADR IN X1
#IW(2)57 STW,X1  8              RESTORE IOCQ-OUT IN PCB
*
*------EXIT TO LEVEL RESTORE ROUTINE
#IW(2)59 JMP     #IX499         EXIT TO LSRR
*
*--------DEVICE   NOT           READY
#IW(2)70 LDW     #IW(2)C9       =X'0341'
#IW(2)72 STW,X2  2              UPDATE IOCQ STATUS WORD
          JMP    #IW(2)59       GO TO EXIT
*
*
*------CONSTANTS
#IW(2)C8 HEX     0001
#IW(2)C9 HEX     0341           DEVICE NOT READY
#IW(2)CB HEX     0FFF
#IW(2)CC HEX     0302           BYTE COUNT = 0
#IW(2)CD HEX     0200
*
*
*------VARIABLES
#IW(2)T3 HEX     0              ICB STATUS
#IW(2)T4 HEX     0              NEW IOCQ-OUT
*
*------GO TO NEXT DSR
#IW(2)99 EQU     *
*
        SKIP     P
         END
```

Figure 4-8. Generalized Coding of the Modem Send Device Driver and Device Service Routine (#IMDMS) (Sheet 3 of 3)

1.12 EXIT Macro Routine (#IMEXT)

The EXIT macro routine is incorporated in each IOCS monitor that is created. It effects a system exit. The EXIT macro routine is specialized according to the form of macro call that SYSGEN generates:

$Δ #IMEXT Δ Arg
or
$Δ #IMEXT Δ , ,

The first form of the call statement is used if a System Logging Device was assigned via the ASGL directive input to SYSGEN. That is, if the third entry of the ASGL directive was a valid device identifier, the identifier becomes the argument in the $Δ#IMEXT call. The second form of the call statement is generated if the third entry in the ASGL directive was 000 (i.e., if no logging device was assigned).

The generalized coding of the #IMEXT macro routine contains statements to effect message logging if a System Logging Device was assigned. These statements contain dummy arguments of the form (1N). Thus, if the logging device identifier is transmitted inthe $Δ#IMEXT macro call, the logging statements are inserted in the specialized routine; otherwise, they are omitted. The generalized #IMEXT macro routine coding is shown in Figure 4-9.

```
?#IMEXT
***********************************************************************
***********************************************************************
*
*                 .EXIT ROUTINE
*
*--------CALLING SEQUENCE
*        MSC
*        DFC       Ø
***********************************************************************
***********************************************************************
*
*
*
*--------CLEAR LOC Ø AND OLD PC
#IMEXT  LDI       AC,Ø
          STW       #IØØT1
        STW       #ITIP9       CLEAR OLD-PC OF INT LEVEL 9
*                                                              (1N)
*--------LOG END OF JOB                                        (1N)
         LAX2     #IØØP1                                       (1N)
          JMP       #IZLOG          LOG END OF JOB             (1N)
          DFC       ØØØØ                                       (1N)
#IØØP1   FQU        *
*                                                              (1N)
*--------GO TO Ø
          INR                       STALL AT ZERO
*
        SKIP   P
          END
```

Figure 4-9. Generalized Coding of the EXIT Macro Routine

1.13 CLOSE Macro Routine (#IMCLL)

The CLOSE macro routine is specialized for each IOCS monitor that is created. This routine performs the processing required to close a specified device. It is specialized according to the form of macro call that SYSGEN generates:

$Δ #IMCLLΔ Arg

or

$Δ #IMCLLΔ ,

The first form of the call statement is used if a System Logging Device was assigned via the ASGL directive input to SYSGEN. That is, if the third entry of the ASGL directive was a valid device identifier, the identifier becomes the argument in the $Δ #IMCLL call. The second form of the call statement is generated if the third entry in the ASGL directive was 000 (i. e., if no logging device was assigned).

As in the #IMEXT macro routine, statements appear in the generalized #IMCLL macro routine to effect message logging when a System Logging Device has been assigned. When the $Δ #IMCLL call transmits a logging device identifier as its argument, the logging statements (containing dummy arguments of the type (1N)), are inserted in the specialized routine. If a comma is transmitted in the $Δ #IMCLL call statement, the logging statements are omitted by the Assembler's macro processor phase.

1.14 INITialization Macro Routine (#IMINT)

The INITialization routine performs the processing required to initialize all devices in the equipment configuration. This routine is always incorporated in the IOCS monitor being created. SYSGEN generates the statement:

$Δ #IMINT

to cause the routine to be incorporated in the monitor. No arguments are supplied in the call, and no dummy arguments appear in the generalized coding. That is, the entire routine is incorporated in the IOCS monitor.

1.15 OPEN Macro Routine (#IMOPL)

The OPEN routine performs the processing required to OPEN a specified device. It is specialized for each IOCS monitor that is created according to the form of macro call that SYSGEN generates:

$Δ #IMOPLΔ Arg

or $Δ #IMOPLΔ ,

The first form of the call statement is used if a System Logging Device was assigned via the ASGL directive input to SYSGEN. That is, if the third entry of the ASGL directive was a valid device identifier, the identifier becomes the argument in the $Δ #IMOPL call. The second form of the call statement is generated if the third entry in the ASGL directive was 000 (i. e., if no logging device was assigned).

The generalized coding of the #IMOPL macro routine contains statements to effect message logging if a System Logging Device has been assigned. If the logging device identifier is transmitted in the $Δ #IMOPL call, the logging statements (containing the dummy arguments (1N)), are inserted in the specialized routine; otherwise, they are omitted.

1.16 IOACTion Macro Routine (#IMACT)

The IOACTion macro routine is specialized for each IOCS monitor that is created. This routine performs the processing required to set up for and start input/output operations. The SYSGEN program generates the macro call:

$Δ #IMACTΔ Arg

or $Δ #IMACTΔ ,

to effect specialization of the #IMACT routine. The first form of the macro call is used if a System Logging Device was assigned via the ASGL directive input to SYSGEN. That is, if the third entry of the ASGL directive was a valid device identifier, the identifier becomes the argument in the $Δ #IMACT macro call. The second form of the $Δ #IMACT call is used if the third entry in the ASGL directive was 000 (i.e., if no logging device was assigned).

The generalized coding of the #IMACT macro routine contains statements to effect message logging if a System Logging Device was assigned. If the logging device identifier is transmitted in the $Δ #IMACT call, the logging statements (containing the dummy arguments (1N)) are inserted in the specialized routine; otherwise, they are omitted.

1.17 Compute PCB Address Macro Routine (#IMPCB)

This macro routine is always incorporated in the IOCS monitor being created. As its name implies, its function is to compute the addresses of the Physcial Control Blocks (PCBs) for assigned devices. The SYSGEN program generates the macro call statement:

$Δ #IMPCB

to cause the entire routine to be inserted in the IOCS monitor being assembled.

1.18 Error Logging Macro Routine (#IMLOG)

This routine produces the logging messages on the System Logging Device when it has been assigned. The routine contains the LUN Conversion Table, the Message Locate Table, and the Canned Message Table. When a System Logging Device is assigned on the ASGL directive input to SYSGEN, the macro call:

$Δ #IMLOG

is generated. At assembly time, the entire Error Logging Routine is inserted in the IOCS monitor being created.

The Error Logging Routine is called by the MSC service routines when the error logging statements have been incorporated in them. That is, when a logging device identifier is transmitted as the argument in the MSC service routine macro calls generated by SYSGEN, the logging statements are included in the specialized versions of the EXIT, OPEN, CLOSE, etc., service routines.

1.19 Clock Service Macro Routine (#IMCLK)

The Clock Service Routine performs the processing required when the interval timer interrupt is selected by the user. The interrupt occurs at level 0. It is specialized according to the form of the CALL directive to SYSGEN:

CALL, $Δ #IMCLK, Arg_1, Arg_2

or CALL, $Δ #IMCLK, Arg_1

The first form generates the modified version of the clock. This version of the clock allows linkage to user specified subroutines when a clock expires. Arg_1 must be the letters MT (modified timer). Arg_2 specifies the number of clocks to be generated.

The second form is used to generate the standard version of the clock, which decrements the clock to zero without subroutine linkage. Arg_1 specifies the number of clocks to be generated.

When the CALL directive is processed by SYSGEN, the CALL statement

$Δ #IMCLK

is written onto the Assembler-formatted output file. When the Assembler's macro processor reads the macro call, it writes the Clock Service Routine onto its output work file.

1.20 Parity Error Macro Routine (#IMPAR)

The parity error interrupt level 10 is optional. The user specifies the inclusion or exclusion of the interrupt level via a CALL directive in the format:

CALL, $Δ #IMPAR Δ Arg

or CALL, $ Δ#IMPARΔ ,

as the input to the SYSGEN program.

When the SYSGEN program interprets the CALL directive it generates one of the following macro call statements:

$ Δ#IMPAR ΔArg

or $Δ #IMPARΔ ,

The Parity Error Routine is specialized according to the form of the macro call that is issued. The generalized coding of the Parity Error Routine is shown below:

```
?#IMPAR
***************************************************************************
***************************************************************************
*
*                     PARITY ERROR ROUTINE
*
*         CALLED FROM INTERPUPR LEVEL 10
*
***************************************************************************
***************************************************************************
#IVPAR  LDW     #IVP00
        ADD     #IVC01
        STW     #IVP00
        JMP     #IVP99                                        (1Y)
         LAX2   #IVP99                                        (1N)
        JMP     #I7LOG                                        (1N)
#IVP00  DEC     0013
#IVP99  INR                 EXIT
#IVC01  HEX     0100
        SKIP    P
           END
```

If the first form of the call statement is issued, the first JMP statement is omitted from the specialized #IMPAR routine because of the (1Y) dummy argument, and the LAX2 and second JMP statements are included. If the second form of the call statement is issued by the SYSGEN program, the first JMP statement is included in the specialized Parity Error Routine, and the LAX2 and JMP statements are omitted.

Section 2. USER MACRO LIBRARY FILE

The User Macro Library is a collection of generalized, source-coded routines that may be called within applications source programs to effect the following:

- Creation of FIOBs and IOCQ Table entries
- Monitor service calls to perform the following:

  READ operations on an input device
  WRITE operations on an output device
  REWIND operations on I/O devices
  OPEN devices
  CLOSE devices
  INITialize all devices on the system
  Cause a System EXIT
  Control or interrogate the Watchdog Timer
  Sense device status
  Test and reset lists of devices attached to the Channel Interface Controller

- Inclusion of Translate Tables for use in translating 2260 keyboard code, ASCII keyboard code, and the IPARS keyboard code to the internal ASCII code
- Conversion of Baudot to ASCII code and ASCII to Baudot Code used with the ASR 32 teletype device.

The routines are called via macro call statements within the user's program. The format of the call statement is:

```
$      macroname      argument list
```

where the $ appears in column 1 of the coding form, followed by a blank character. The name of the macro routine then appears, followed by a blank character. The argument list, if required, then appears, where two or more arguments are separated by commas. If a required argument is to be omitted from the list, its omission is indicated by a comma in the argument's position in the list.

When a program contains macro call statements, the User Macro Library file must be input to the Assembler for use by the macro processor in specializing the called macro routines, as described in Part 2 of this Handbook.

The remainder of this section presents detailed descriptions of the User Macro Library routines and the manner in which they are called in source programs.

2.1 IOFIOB Macro Routine

The IOFIOB user macro routine effects specialization of the source statements that construct a File Input/Output Block (FIOB) describing the parameters of each IOACTion service to be performed by IOCS. The generalized coding of the IOFIOB macro routine is shown below:

```
?IOFIOB
(1)      HEX    Ø            SPARE/ERROR CODE
(2)1     HEX    (3)          MODE, FUNCTION, LOG UNIT NO
(2)2     ADC    (4)          BUFFER ADDRESS
(2)3     DEC    (5)          BYTE COUNT
(2)4     ADC    (6)          TRANSLATE TABLE BASE (9Y)
(2)4     HEX    (6)          DISK TRACK-CYLINDER ADDRESS (9N)
(2)5     ADC    (7)          SEARCH TABLE BASE (9Y)
(2)5     HEX    (7)          DISK SECTOR ADDRESS (9N)
         RESV,Ø 4            SPARE
(2)8     HEX    ØØØ(8)       SPARE/LUN EXTENSION NUMBER
 END
```

The macro routine contains nine dummy arguments, and therefore nine arguments must be transmitted in the IOFIOB call statement. These arguments are defined below.

Argument 1:	The starting location (e.g., symbolic tag) of the specific FIOB to be created must appear in the first position of the call statement list. *
Argument 2:	The first portion of a symbolic tag for the constant defining statements in the FIOB must appear as the second argument in the macro call statement. *
Argument 3:	The data transfer mode, device function code, and logical unit number (LUN) of the device on which the IOACTion is to be performed must be specified as argument 3.
Argument 4:	This argument specifies the starting address of the buffer to or from which I/O data is to be transferred.
Argument 5:	This argument specifies the number of bytes of data to be transferred to or from the I/O buffer.
Argument 6:	This argument specifies the starting location of the Translate Table if the I/O action is to be performed on a device other than a disc or the address of the disc track and cylinder if a disc device is to be used in the I/O operation.

*Arguments 1 and 2 supply labels for statements in the FIOB. The first argument supplies a complete label and may be six characters or less; the first character must be alphabetic. Argument 2 specifies five or less characters of a label; the first character must be alphabetic.

Argument 7: This argument specifies the address of the starting location of the Search table if the I/O operation is to be performed on a device other than a disc, or the address of the disc sector if a disc device is to be used in the operation.

Argument 8: The LUN extension number of a specific device on a device controller to which multiple devices may be attached.

Argument 9: This argument must be a 1 if the I/O operation is to be performed on a disc device, or a comma (,) if any other device is to be used for the I/O operation. If a 1 is specified as the ninth actual argument in the $IOFIOB call statement, the Translate and Search Table address constant statements are omitted from the specialized routine and the disc track-cylinder and sector hexadecimal constant statements are included. If a comma is transmitted in position 9 of the call statement argument list, the ADC statements specifying the Translate and Search Table addresses are included in the routine and the disc address statements are omitted from the specialized routine at assembly time.

The call statement:

```
$      IOFIOB      FIOB1,JVC,FFFF,BUFF1,320,0208,0040,0,1
```

causes the following routine to be specialized at assembly time.

```
FIOB1     HEX       0          SPARE/ERROR CODE
JVC1      HEX       FFFF       MODE,FUNCTION, LOG UNIT NO
JVC2      ADC       BUFF1      BUFFER ADDRESS
JVC3      DEC       320        BYTE COUNT
JVC4      HEX       0208       DISK TRACK-CYLINDER ADDRESS
JVC5      HEX       0040       DISK SECTOR ADDRESS
          RESV,0    4          SPARE
JVC8      HEX       0000       SPARE/LUN EXTENSION NUMBER
```

The call statement

```
$      IOFIOB      FIOB2,JVC,2401,BUFF1,80,TRANS,SRCH,4,,
```

causes the following routine to be specialized.

```
FIOB2     HEX       0          SPARE/ERROR CODE
JVC1      HEX       2401       MODE,FUNCTION, LOG UNIT NO
JVC2      ADC       BUFF1      BUFFER ADDRESS
JVC3      DEC       80         BYTE COUNT
JVC4      ADC       TRANS      TRANSLATE TABLE BASE
JVC5      ADC       SRCH       SEARCH TABLE BASE
          RESV,0    4          SPARE
JVC8      HEX       0004       SPARE/LUN EXTENSION NUMBER
```

2.2 IOIOCQ Macro Routine

The IOIOCQ user macro routine effects specialization of source statements required to set up entries in the IOCQ table. The generalized coding of the routine is shown in Figure 4-10.

The IOIOCQ macro may be called to generate a single IOCQ entry or a maximum of eight IOCQ entries. In all cases, the IOCQ name base must appear as the first argument in the list of the $IOIOCQ call statement. Following the name base may be from one to eight modifiers to the name base.* These modifiers uniquely identify individual entries in the IOCQ table being generated. As in all macro call statement lists, the individual arguments in the list are separated by commas, and omitted arguments are indicated by commas.

To effect the creation of a single IOCQ table entry, the user specifies the name base, and a single name base modifier followed by eight commas. For example, the call statement

```
$     IOIOCQ     IOCQ,A,,,,,,,
```

specifies that only one entry is to be created in the IOCQ table. The name base is to be IOCQ, and the modifier A is to be used to specialize the entry. The following coding will be produced as a result of the single entry call.

```
IOCQA     ADC      IOCQA     LINK
          HEX      0         LOGICAL/PHYSICAL STATUS
          HEX      0         MODE,FUNCTION, LOG UNIT NO
          HEX      0         BUFFER ADDRESS
          HEX      0         BYTE COUNT
          HEX      0         TRANSLATE TABLE/DISK ADDRESS
          HEX      0         SEARCH TABLE/DISK ADDRESS
          RESV,0 6           SPARE
```

If several IOCQ entries are to be created, the IOIOCQ macro call statement list contains the name base as the first argument, followed by a comma and the appropriate number of name base modifiers. For example, the call statement

```
$IOIOCQ     IOCQ,A,B,C,D,,,,
```

specifies that four IOCQ entries are to be created. The name base is to be IOCQ, and the four entries are to be specialized with the arguments A, B, C, and D. The following coding will be produced by the call statement:

*Since the name base and the modifiers complete the label fields of the IOCQ table, the combined number of characters must not exceed six characters; the first character must be alphabetic.

```
IOCQA    ADC      IOCQB    LINK
         HEX      0        LOGICAL/PHYSICAL STATUS
         HEX      0        MODE, FUNCTION, LOG UNIT NO
         HEX      0        BUFFER ADDRESS
         HEX      0        BYTE COUNT
         HEX      0        TRANSLATE TABLE/DISK ADDRESS
         HEX      0        SEARCH TABLE/DISK
         RESV,0   6        SPARE
IOCQB    ADC      IOCQC    LINK
          .
          .
          .
IOCQC    ADC      IOCQD    LINK
          .
          .
          .
IOCQD    ADC      IOCQA    LINK
          .
          .
          .
```

2.3 READ Macro Routine

The READ user macro routine effects specialization of the source statements required to perform a read operation on an input device. The generalized coding of the READ macro routine is shown below.

```
?READ
         MSC              MONITOR SERVICE CALL
         HEX      7       I/O ACTION CODE
         ADC      (1)     RETURN ADDRESS
         ADC      (2)     FIOB ADDRESS
END
```

Since there are two dummy arguments in the routine, the call statement to the routine must transmit two actual arguments, as follows:

Argument 1:	This argument specifies the user program address to which IOCS is to return control on completion of the read operation.
Argument 2:	This argument specifies the address of the FIOB* associated with the device on which the read operation is to be performed.

*Prior to the appearance of a READ call statement, an FIOB with a READ function code must have been established in the calling program.

```
~~?IOIOCQ~~
(1)(2)    ADC    (1)(2)        LINK (3Y)
(1)(2)    ADC    (1)(3)        LINK (3N)
          HEX    Ø             LOGICAL/PHYSICAL STATUS (2N)
          HEX    Ø             MODE, FUNCTION, LOG UNIT NO (2N)
          HEX    Ø             BUFFER ADDRESS (2N)
~~          HEX    Ø             BYTE COUNT (2N)~~
          HEX    Ø             TRANSLATE TABLE/DISK ADDRESS (2N)
          HEX    Ø             SEARCH TABLE/DISK ADDRESS (2N)
          RESV,Ø  6            SPARE (2N)
(1)(3)    ADC    (1)(2)        LINK (4Y)(3N)
(1)(3)    ADC    (1)(4)        LINK (4N)
~~          HEX    Ø             LOGICAL/PHYSICAL STATUS (3N)~~
          HEX    Ø             MODE, FUNCTION, LOG UNIT NO (3N)
          HEX    Ø             BUFFER ADDRESS (3N)
          HEX    Ø             BYTE COUNT (3N)
          HEX    Ø             TRANSLATE TABLE/DISK ADDRESS (3N)
          HEX    Ø             SEARCH TABLE/DISK ADDRESS (3N)
~~          RESV,Ø  6            SPARE (3N)~~
(1)(4)    ADC    (1)(2)        LINK (5Y)(4N)(3N)
(1)(4)    ADC    (1)(5)        LINK (5N)
          HEX    Ø             LOGICAL/PHYSICAL STATUS (4N)
          HEX    Ø             MODE, FUNCTION, LOG UNIT NO (4N)
          HEX    Ø             BUFFER ADDRESS (4N)
          HEX    Ø             BYTE COUNT (4N)
          HEX    Ø             TRANSLATE TABLE/DISK ADDRESS (4N)
          HEX    Ø             SEARCH TABLE/DISK ADDRESS (4N)
          RESV,Ø  6            SPARE (4N)
(1)(5)    ADC    (1)(2)        LINK (6Y)(5N)(4N)(3N)
(1)(5)    ADC    (1)(6)        LINK (6N)
~~          HEX    Ø             LOGICAL/PHYSICAL STATUS (5N)~~
          HEX    Ø             MODE, FUNCTION, LOG UNIT NO (5N)
          HEX    Ø             BUFFER ADDRESS (5N)
          HEX    Ø             BYTE COUNT (5N)
          HEX    Ø             TRANSLATE TABLE/DISK ADDRESS (5N)
          HEX    Ø             SEARCH TABLE/DISK ADDRESS (5N)
~~          RESV,Ø  6            SPARE (5N)~~
(1)(6)    ADC    (1)(2)        LINK (7Y)(6N)(5N)(4N)(3N)
(1)(6)    ADC    (1)(7)        LINK (7N)
          HEX    Ø             LOGICAL/PHYSICAL STATUS (6N)
          HEX    Ø             MODE, FUNCTION, LOG UNIT NO (6N)
          HEX    Ø             BUFFER ADDRESS (6N)
~~          HEX    Ø             BYTE COUNT (6N)~~
          HEX    Ø             TRANSLATE TABLE/DISK ADDRESS (6N)
          HEX    Ø             SEARCH TABLE/DISK ADDRESS (6N)
          RESV,Ø  6            SPARE (6N)
(1)(7)    ADC    (1)(2)        LINK (8Y)(7N)(6N)(5N)(4N)(3N)
(1)(7)    ADC    (1)(8)        LINK (8N)
~~          HEX    Ø             LOGICAL/PHYSICAL STATUS (7N)~~
          HEX    Ø             MODE, FUNCTION, LOG UNIT NO (7N)
          HEX    Ø             BUFFER ADDRESS (7N)
          HEX    Ø             BYTE COUNT (7N)
~~          HEX    Ø             TRANSLATE TABLE/DISK ADDRESS (7N)~~
          HEX    Ø             SEARCH TABLE/DISK ADDRESS (7N)
~~          RESV,Ø  6            SPARE (7N)~~
 (1)(8)   ADC    (1)(2)        LINK (9Y)(8N)(7N)(6N)(5N)(4N)(3N)
~~(1)(8)    ADC    (1)(9)        LINK (9N)~~
          HEX    Ø             LOGICAL/PHYSICAL STATUS (8N)
~~          HEX    Ø             MODE, FUNCTION, LOG UNIT NO (8N)~~
          HEX    Ø             BUFFER ADDRESS (8N)
~~          HEX    Ø             BYTE COUNT (8N)~~
          HEX    Ø             TRANSLATE TABLE/DISK ADDRESS (8N)
~~          HEX    Ø             SEARCH TABLE/DISK ADDRESS (8N)~~
          RESV,Ø  6            SPARE (8N)
~~(1)(9)    ADC    (1)(2)        LINK (9N)~~
          HEX    Ø             LOGICAL/PHYSICAL STATUS (9N)
~~          HEX    Ø             MODE, FUNCTION, LOG UNIT NO (9N)~~
          HEX    Ø             BUFFER ADDRESS (9N)
~~          HEX    Ø             BYTE COUNT (9N)~~
          HEX    Ø             TRANSLATE TABLE/DISK ADDRESS (9N)
          HEX    Ø             SEARCH TABLE/DISK ADDRESS (9N)
          RESV,Ø  6            SPARE (9N)
~~END~~
```

Figure 4-10. Generalized Coding of the IOIOCQ Macro Routine

To illustrate the manner in which the READ routine is called, assume that the return address in the program is FINISH, and the address of the FIOB to be used is FIOB1. The call statement

```
$     READ     FINISH,FIOB1
```

will cause the READ routine to be specialized as shown below:

```
MSC               MONITOR SERVICE CALL
HEX     7         I/O ACTION CODE
ADC     FINISH    RETURN ADDRESS
ADC     FIOB1     FIOB ADDRESS
```

2.4 WRITE Macro Routine

The WRITE user macro routine effects specialization of the source statements required to perform a write operation on an output device. The generalized coding of the WRITE macro routine is shown below:

```
?WRITE
          MSC          MONITOR SERVICE CALL
          HEX    7     I/O ACTION CODE
          ADC    (1)   RETURN ADDRESS
          ADC    (2)   FIOB ADDRESS
```

Since there are two dummy arguments in the routine, the call statement must transmit two actual arguments as follows:

Argument 1:	This argument specifies the user program address to which the IOCS monitor is to return control on completion of the write operation.
Argument 2:	This argument specifies the address of the FIOB* associated with the device on which the operation is to be performed.

To illustrate the WRITE call statement, assume that the return address in the program is PRINT and the address of the FIOB is PRINTR. The call statement

```
$     WRITE     PRINT,     PRINTR
```

will cause the WRITE routine to be specialized as shown below.

```
MSC               MONITOR SERVICE CALL
HEX     7         I/O ACTION CODE
ADC     PRINT     RETURN ADDRESS
ADC     PRINTR    FIOB ADDRESS
```

*Prior to the appearance of a WRITE call statement, an FIOB with a WRITE function code must have been established in the calling program.

2.5 REWIND Macro Routine

The REWIND user macro routine effects specialization of the source statements required to perform a rewind operation on an I/O device. The generalized coding of the REWIND macro routine is shown below.

```
?REWIND
          MSC             MONITOR SERVICE CALL
          HEX     7       I/O ACTION CODE
          ADC     (1)     RETURN ADDRESS
          ADC     (2)     FIOB ADDRESS
END
```

Since there are two dummy arguments in the routine, the call statement must transmit two actual arguments, as follows:

Argument 1:	This argument specifies the user program address to which IOCS is to return control on completion of the rewind operation.
Argument 2:	This argument supplies the address of the FIOB* associated with the device on which the operation is to be performed.

For example, assume that the program return address is RDREC and the address of the associated FIOB is RWFIOB. The call statement

```
$   REWIND   RDREC,RWFIOB
```

will cause the REWIND routine to be specialized as follows:

```
MSC               MONITOR SERVICE CALL
HEX     7         I/O ACTION CODE
ADC     RDREC     RETURN ADDRESS
ADC     RWFIOB    FIOB ADDRESS
```

2.6 OPEN Macro Routine

The OPEN user macro routine effects specialization of the source statements required to open a specific I/O device. The generalized coding of the macro routine is shown below.

*Prior to the appearance of a REWIND call statement, an FIOB with a REWIND function code must have been established in the calling program.

```
? OPEN
               MSC              MONITOR SERVICE CALL
               HEX      6       OPEN CODE
               ADC      (1)     RETURN ADDRESS
               ADC      *+2     PARAMETER LIST ADDRESS
               HEX      (2)     LUN
               ADC      (3)     IOCQ ADDRESS
  (4)          RESV,0   2       ERROR FIELD
END
```

The OPEN macro routine has four dummy arguments; hence, four actual arguments must be transmitted to the routine via the call statement list, as described below:

Argument 1:	This argument specifies the user program address to which IOCS is to return control when the specified device has been opened.
Argument 2:	This argument supplies the logical unit number (LUN) of the device to be opened.
Argument 3:	The address of the associated entry in the IOCQ table is specified as the third actual argument.
Argument 4:	The label of the error code field is defined in argument 4. This argument must consist of six characters or less, the first of which must be alphabetic.

Shown below is a call statement with the proper arguments for the OPEN routine.

```
$     OPEN    START1,1,IOCQX,ERROR1
```

When the call statement is processed by the Assembler, the following specialized coding is produced for the calling source program:

```
            MSC                  MONITOR SERVICE CALL
            HEX        6         OPEN CODE
            ADC        START1    RETURN ADDRESS
            ADC        *+2       PARAMETER LIST ADDRESS
            HEX        1         LUN
            ADC        IOCQX     IOCQ ADDRESS
ERROR1      RESV,0     2         ERROR FIELD
```

2.7 CLOSE Macro Routine

The CLOSE user macro routine effects specialization of the source statements required to close a specific I/O device. The generalized coding of the macro routine is shown below.

```
? CLOSE
            MSC                  MONITOR SERVICE CALL
            HEX        1         CLOSE CODE
            ADC        (1)       RETURN ADDRESS
            ADC        *+2       PARAMETER LIST ADDRESS
            HEX        (2)       LUN
  (3)       RESV,0     2         ERROR FIELD
END
```

There are three dummy arguments in the CLOSE macro routine. Hence, a user macro call statement must contain three actual arguments to be used to specialize the coding to be incorporated in the calling program. The actual arguments are described below:

Argument 1:	The calling program return address appears as the first argument in the call statement list.
Argument 2:	The second argument supplies the logical unit number (LUN) of the device to be closed.
Argument 3:	The label of the error code field is specified as argument 3. This argument may consist of six characters or less, the first of which must be alphabetic.

The call statement

```
$    CLOSE    FIN1,1,ERROR2
```

will cause the following specialized code to be produced by the Assembler's macro processor:

```
          MSC                    MONITOR SERVICE CALL
          HEX        1           CLOSE CODE
          ADC        FIN1        RETURN ADDRESS
          ADC        *+2         PARAMETER LIST ADDRESS
          HEX        1           LUN
ERROR2    RESV,0     2           ERROR FIELD
```

2.8 INIT Macro Routine

The INIT user macro routine effects specialization of the source statements required to reset all devices on the system. The generalized coding of the routine is shown below.

```
?INIT
          MSC                    MONITOR SERVICE CALL
          HEX        2           INIT CODE
          ADC        (1)         RETURN ADDRESS
END
```

The address constant statement contains a dummy argument instead of the user program address to which IOCS is to return control after the initialization operation is completed. The macro call statement to the INIT routine must therefore specify the return address as its only argument. For example, the call statement

```
$    INIT    START
```

will cause the following specialized routine to be inserted in the calling program at assembly time:

```
MSC                    MONITOR SERVICE CALL
HEX        2           INIT CODE
ADC        START       RETURN ADDRESS
```

2.9 EXIT Macro Routine

The EXIT user macro routine effects the specialization of source statements required to request IOCS to log the end of job and cause a system exit. The generalized coding of the macro routine is shown below.

```
? EXIT
           MSC                MONITOR SERVICE CALL
           HEX        0       ALL DONE CODE
END
```

There are no dummy arguments in the macro routine; hence no actual arguments are transmitted to the routine in the call statement. A sample call statement is illustrated below:

```
$     EXIT
```

The following code will appear in the calling source program.

```
MSC          MONITOR SERVICE CALL
HEX     0    ALL DONE CODE
```

2.10 Watchdog Timer (WDTMSC) Macro Routine

The WDTMSC user macro routine effects specialization of source statements required to control or interrogate the watchdog timer. The generalized coding of the watchdog timer macro routine is shown below.

```
? WDTMSC
             MSC               MONITOR SERVICE CALL
             DEC     3         WATCHDOG TIMER CODE
             ADC     (1)       RETURN ADDRESS
             ADC     (2)       PARAMETER LIST ADDRESS
    (2)      DEC     000(3)    REQUEST CODE
             HEX     0         POWER STATUS FIELD
             HEX     0         ERROR FIELD
```

Since there are three dummy arguments in the macro routine, the watchdog timer call statement must transmit three actual arguments, as follows:

Argument 1:	The first argument specifies the user program address to which IOCS is to return control when the watchdog timer service has been performed.
Argument 2:	The tag assigned to the parameter list. This argument may consist of six characters or less, the first of which must be alphabetic.
Argument 3:	The request code, which may be one of the following:

Code	Requested Service
1	Requests that IOCS reset the watchdog timer. The watchdog timer must be reset once every 34 seconds or automatic program restart will occur.
2	Requests the monitor to start the watchdog timer under program control and automatically initialize the counter to zero.
3	Requests the monitor to turn off the watchdog timer under program control. NOTE Code 3 should be issued only in special cases, since it causes the watchdog timer to be totally disabled.
4	Requests that the monitor read the power status and return the reading to the power status field in the specialized routine.
Other	Any code other than one of the above is illegal and will cause the monitor to return an illegal request code of 1 to the Error Field defined in the last statement of the WDTMSC routine. No watchdog timer action is taken when an illegal code is issued.

When the watchdog timer routine is called, the Assembler substitutes the actual arguments in the call statement for the dummy arguments to produce a specialized routine.

At program execution time, control passes to the IOCS monitor, and the specified timer action is performed.

2.11 Device Sensing (DVSMSC) Macro Routine

The DVSMSC user macro routine effects specialization of the statements required to request the IOCS monitor to sense the status of a specific device. The generalized coding of the device sensing macro routine is shown below.

```
? DVSMSC
             MSC              MONITOR SERVICE CALL
             DEC     5        DEVICE SENSING CODE
             ADC     (1)      RETURN ADDRESS
             ADC     (2)      PARAMETER LIST ADDRESS
   (2)       HEX     00(3)    LUN NUMBER
             HEX     0        STATUS FIELD
             HEX     0        ERROR FIELD
END
```

Since three dummy arguments appear in the macro routine, the user call statement must transmit three actual arguments, as follows:

Argument 1: The calling program address to which IOCS is to return control when the device sensing service has been performed.

Argument 2: The address of the parameter list, which becomes the operand of the second ADC statement and the label of the HEX statement that defines the logical unit number (LUN) of the device to be tested. This argument may consist of six characters or less, the first of which must be alphabetic.

Argument 3: The LUN of the device whose status is to be sensed.

When the call statement is processed, the Assembler inserts the actual arguments in the place of the three dummy arguments to produce the specialized routine.

When the routine is executed, the device's hardware status is returned to the status field, unless an erroneous LUN (i.e., an unassigned LUN) was specified, in which case an error code of 2 is returned to the error field. If no error occurred, the error field setting remains all zeros.

2.12 CICMSC Macro Routine

The CICMSC user macro routine effects specialization of source statements required to test or reset busy or off-line bits of devices attached to the Channel Interface Controller (CIC). The generalized coding of the routine is shown below.

```
? CICMSC
               MSC               MONITOR SERVICE CALL
               DEC       4       CIC CODE
               ADC       (1)     RETURN ADDRESS
               ADC       (2)     PARAMETER LIST ADDRESS
  (2)          HEX       00(3)   LUN NUMBER
               DEC       000(4)  REQUEST CODE
               HEX       0       STATUS FIELD
               HEX       0       ERROR FIELD
END
```

Since there are four dummy arguments in the macro routine, the call to the routine must transmit four actual arguments, as follows:

Argument 1: The calling program address to which IOCS is to return control when the CIC service has been performed.

Argument 2: The address of the parameter list, which becomes the operand of the second ADC statement and the label of the HEX statement that defines the logical unit number (LUN). This argument must be six characters or less, the first of which must be alphabetic.

Argument 3: The LUN of the CIC device to be tested.

Argument 4: The request code, which may be one of the following:

Code	Requested Service
1	Requests that the busy bit of the specified device be tested and set. The current status of the busy bit is returned in bit 0 of the status field, where a 1 = busy and a 0 = not busy. The remaining bits of the status field are undefined. The busy bit is then set.
2	Requests a resetting of the busy bit of the specified device.
3	Requests that the off-line bit of the specified device be tested and set. The current status of the off-line bit is returned in bit 0 of the status field, where 1 = off-line and 0 = on-line. The remaining bits of the status field are undefined. The device off-line bit is then set.
4	Requests a resetting of the off-line bit of the specified device.
Other	Any code other than one of the above is illegal and will be reported via a 1 code in the error field defined in the last statement of the CICMSC routine. No other action is taken by IOCS when an illegal request code is specified in position 4 of the call statement.

When the $CICMSC call statement is processed the Assembler inserts the actual arguments in the place of the four dummy arguments to produce the specialized routine.

When the routine is executed the specified action is taken unless an erroneous LUN (i.e., an unassigned LUN) was specified, in which case an error code of 2 is returned in the error field. If no error occurred, the error field setting remains all zeros.

2.13 TT2260 Macro Routine

This routine causes the 2260 Translate Table to be inserted in the calling program. The table contains the conversions codes to be used by the device controller to translate 2260 keyboard code to the ASCII code used internally in the PTS-100. The macro routine coding, shown in Figure 4-11 contains a dummy argument in the label field of the first statement. The call statement to the routine must therefore transmit a label to be identified as the starting location (i.e., the Translate Table base address) in the user's program.

Assume that the symbolic tag TRNS1 is to be used as the base address of this Translate Table in the user's program. The call statement

```
$      TT2260      TRANS1
```

will cause the first statement of the routine to be specialized as follows:

```
TRANS1    HEX    0000
```

The remainder of the Translate Table will be inserted in the calling program.

```
?TT2260
        SKIP      P
*
*********************************************************************
*
*       SOFTWARE TRANSLATE   TABLE 2260
*
*********************************************************************
~~(1)   HEX    0000~~
        HEX    00DC          TAB                 START
        HEX    F05A          RETURN              Z
        HEX    5958          Y                   X
        HEX    5756          W                   V
        HEX    5554          U                   T
        HEX    5352          S                   R
        HEX    5150          Q                   P
        HEX    4F4E          O                   N
        HEX    4D4C          M                   L
        HEX    4B4A          K                   J
        HEX    4948          I                   H
        HEX    4746          G                   F
        HEX    4544          E                   D
        HEX    4342          C                   B
        HEX    4100          A
        HEX    0000
        HEX    00DC          TAB                 START
        HEX    F05A          RETURN              Z
        HEX    3D58          = EQUALS            X
        HEX    5756          W                   V
        HEX    2354          NUMBER SIGN         T
        HEX    5352          S                   R
        HEX    5150          Q                   P
        HEX    4F4E          O                   N
        HEX    4D4C          M                   L
        HEX    4B4A          K                   J
        HEX    4948          I                   H
        HEX    4746          G                   F
        HEX    4544          E                   D
~~      HEX    4342          C                   B~~
        HEX    4100          A
        HEX    2F2E          / SLANT             . PERIOD
        HEX    2D2C          - HYPHEN            , COMMA
        HEX    2600          # AMPERSAND
        HEX    3938          9                   8
~~      HEX    3736          7                   6~~
        HEX    3534          5                   4
        HEX    3332          3                   2
        HEX    3130          1                   0
        HEX    3F3A            QUESTION MARK       COLON
        HEX    2D3B          - HYPHEN              SEMI COLON
~~      HEX    2700          ' APOSTROPHE~~
        HEX    282A           OPEN PAREN         *
        HEX    372B          7                   + PLUS
        HEX    2534            PERCENT           4
        HEX    3340          3                   #
        HEX    2429           DOLLAR SIGN        CLOSE PAREN
~~      HEX    FAEC          STORE-RESTORE       RIGHT ONE CHAR~~
        HEX    F8F6          UP ONE CHAR         DELETE
        HEX    00EE                              HOME
        HEX    EAF2          LEFT ONE CHAR       DOWN ONE CHAR
        HEX    F400          INSERT
        HEX    E2E0          ENTER               PRINT
~~      HEX    DEFC          LOCAL-REMOTE        CLEAR~~
        HEX    FE00          CANCEL              TAB
        HEX    0033                              3
        HEX    3639          6                   9
        HEX    0030                              0
        HEX    3235          2                   5
~~      HEX    3820          8                   SPACE~~
        HEX    0031                              1
        HEX    3437          4                   7
        HEX    2000          SPACE
         END
```

Figure 4-11. Generalized Coding of the TT2260 Macro Routine

2.14 TTASCI Macro Routine

This routine causes the ASCII Keyboard Translate Table to be inserted in the calling program. The table contains the conversion codes to be used by the device controller to translate ASCII keyboard input code to the ASCII code used internally in the PTS-100. The macro routine coding, shown in Figure 4-12, contains a dummy argument in the label field of the first statement. The call statement to the routine must therefore transmit a label to be identified as the starting location (i. e., the Translate Table base address) in the user's program.

The call statement

```
$      TTASCI      ASCTRN
```

will cause the first statement of the routine to be specialized as follows:

```
ASCTRN      HEX      0000
```

The remainder of the table will be inserted in the calling program.

2.15 TTIPAR Macro Routine

This macro routine causes the IPARS Keyboard Translate Table to be inserted in the calling program. The table contains the conversion codes to be used by the device controller to translate IPARS keyboard input code to the ASCII code used internally in the PTS-100. The macro routine coding, shown in Figure 4-13, contains a dummy argument in the label field of the first statement. The call statement to the routine must therefore transmit a label to be identified as the starting location (i. e., the Translate Table base address) in the user's program. The call statement

```
$      TTIPAR      IPRSIN
```

will cause the first statement of the routine to be specialized as follows:

```
IPRSIN      HEX      0000
```

The remainder of the table will be inserted in the calling program.

2.16 Baudot to ASCII Converter Macro Routine

This routine performs the processing required to convert the five-bit Baudot code input via the ASR 32 teletype device to the seven-bit ASCII code used internally in the PTS-100. If this macro routine is to be called, the following calling sequence must be established via source code in the calling program:

```
         LAX2      TAG        LOAD POINTER TO PARAMETER LIST
         JMP,N     TAG3       GO TO CONVERSION ROUTINE
TAG      ADC       INPUT      INPUT BUFFER POINTER - BAUDOT CODE
         ADC       OUTPUT     OUTPUT BUFFER - WHERE TO PUT ASCII
         HEX       XXXX       NUMBER OF BAUDOT CHARACTERS TO BE XLMTED
         RESV,0    2          NUMBER OF ASCII CHARACTERS
         RESV,0    2          ERROR WORD
TAG3     ADC       #JVBAC     POINTER TO CONVERSION ROUTINE
         EXREF     #JVBAC     EXTERNAL REFERENCE STATEMENT FOR #JVBAC
         JMP       TAG2       RETURN AFTER CONVERSION
```

```
?TTASCI
         SKIP      P
*************************************************************************
*
*        SOFTWARE TRANSLATE   TABLE ASCII
*
*************************************************************************
(1)      HEX       0000
         HEX       7E3D            OVERLINE           = EQUALS
         HEX       215A            EXCLAMATION        Z
         HEX       5958        Y                      X
         HEX       5756        W                      V
         HEX       5554        U                      T
         HEX       5352        S                      R
         HEX       5150        Q                      P
         HEX       4F4E        O                      N
         HEX       4D4C        M                      L
         HEX       4B4A        K                      J
         HEX       4948        I                      H
         HEX       4746        G                      F
         HEX       4544        E                      D
         HEX       4342        C                      B
         HEX       415C        A                        REVERSE SLANT
         HEX       0000
         HEX       7D2B            CLOSE BRACE        + PLUS
         HEX       5D00            CLOSE BRACKET
         HEX       0000
         HEX       0000
         HEX       0000
         HEX       0000
         HEX       0000
         HEX       0000
         HEX       0000
         HEX       0000
         HEX       0000
         HEX       0000
         HEX       0000
         ~~HEX       0000~~
         HEX       007B                                 OPEN BRACE
         HEX       2F2E        / SLANT                . PERIOD
         HEX       2D2C        - HYPHEN               , COMMA
         HEX       3B5C          SEMICOLON              REVERSE SLANT
         HEX       3938        9                      8
         HEX       3736        7                      6
         HEX       3534        5                      4
         HEX       3332        3                      2
         HEX       3130        1                      0
         HEX       3F00          QUESTION MARK
         HEX       2200          QUOTATION MARK
         ~~HEX       3A5B          COLON                  OPEN BRACKET~~
         HEX       282A        OPEN PAREN             * ASTERISK
         HEX       2627        # AMPERSAND            ' APOSTROPHE
         HEX       2524          PERCENT                DOLLAR SIGN
         HEX       2322        = NUMBER SIGN          ' COMMERCIAL AT
         HEX       2129          EXCLAMATION          CLOSE PAREN
         ~~HEX       0000~~
         HEX       0000
         HEX       0000
         HEX       0000
         HEX       0000
         HEX       E2E0        ENTER                  PRINT
         ~~HEX       DEFC        LOCAL-REMOTE           CLEAR~~
         HEX       FE00        CANCEL                 ROLL UP
         HEX       0000        ROLL DOWN              ERASE EOS
         HEX       0000        ERASE EOL              TAB
         HEX       FAEC        STORE-RESTORE          RIGHT ONE CHAR
         HEX       F8F6        UP ONE CHAR            DELETE CHAR
         ~~HEX       00EE                               HOME~~
         HEX       EAF2        LEFT ONE CHAR          DOWN ONE CHAR
         HEX       F400        INSERT CHAR
         HEX       2000        SPACE
          END
```

Figure 4-12. Generalized Coding of the TTASCI Macro Routine

```
?TTIPAR
        SKIP        P
****************************************************************************
*
*       SOFTWARE TRANSLATE    TABLE IPAR
*
****************************************************************************
(1)     HEX      0000
        HEX      002A                            ASTERISK
        HEX      F05A        RETURN              Z
        HEX      5958        Y                   X
        HEX      5756        W                   V
        HEX      5554        U                   T
        HEX      5352        S                   R
        HEX      5150        Q                   P
        HEX      4F4E        O                   N
        HEX      4D4C        M                   L
        HEX      4B4A        K                   J
        HEX      4948        I                   H
        HEX      4746        G                   F
        HEX      4544        E                   D
        HEX      4342        C                   B
        HEX      4140        A                   COMMERCIAL AT
         HEX     0000
        HEX      002A                            DSPL
        HEX      F05A        RETURN              Z
        HEX      5958        Y                   X
        HEX      5756        W                   V
         HEX     5554        U                   T
        HEX      5352        S                   R
        HEX      5150        Q                   P
        HEX      4F4E        O                   N
        HEX      4D4C        M                   L
        HEX      4B4A        K                   J
        HEX      4948        I                   H
        HEX      4746        G                   F
         HEX     4544        E                   D
         HEX     4342        C                   B
        HEX      4109        A                   CHNG
        HEX      2F2E        SLANT               PERIOD
         HEX     2D23        HYPHEN              NUMBER SIGN
         HEX     2400        DOLLAR SIGN
        HEX      3938        9                   8
        HEX      3736        7                   6
        HEX        3534      5                   4
        HEX          3332    3     2
        HEX      3130        1                   0
        HEX      2F2E        IAS                 CSS
        HEX      2D23        NAME                ENDI
        ~~HEX      2400        RDUC~~
        HEX      3938        FCNE                TL
        HEX      3736        TKT                 RCVD
        HEX      3534        RMKS                FAX
        HEX      3332        GFAX                FLFU
        HEX      3130        RLOC                SEG
        ~~HEX      E2C8        ENTER~~               UNSMG
        HEX      C2C5        IGN                 RPT
        HEX      0000
        HEX      0000
        HEX      0000
        HEX      00E0                            PRINT
        ~~HEX      DEFC        RESET~~               CLEAR
         HEX     FE00        CANCEL
2        HEX      0000
         HEX     0000
11      HEX      CIFA        EI                  STORE-RESTORE
        HEX      ECF8        FWDSP               UPONE
10      ~~HEX      F6E2        DELETE              ENTER~~
        HEX      EEEA        HOME                BCKSP
9       ~~HEX      F2F4        DWNONE~~              INSERT
        HEX      2000        SPACE
8       ~~HEX      2000        SPACE~~
         END
```

Figure 4-13. Generalized Coding of the TTIPAR Macro Routine

Note that the input buffer contains the Baudot code to be converted to ASCII code and stored in the output buffer. All Baudot data characters are directly convertible to ASCII characters. That is, a one-for-one conversion of data characters is performed. In addition to data characters, the input buffer contains shift characters. The shift characters are used by the Converter routine to perform the conversion, but are not written into the output buffer. Hence, the output buffer need not be any larger than the input buffer, and may in fact be smaller by the number of shift characters transmitted from the ASR 32 teletype to the input buffer. If the programer wishes, he may use the same buffers for input and output, thus overlaying the Baudot input with converted ASCII code.

Normal termination of the Baudot to ASCII Converter Routine may be specified in one of the following ways:

1. Specifying the exact number of input bytes to be processed in the operand field of the HEX statement (i.e., in the third word of the parameter list which starts with TAG in the calling sequence code).

2. Storing a flag of X'FF' in the last byte of the input (Baudot) buffer.

3. If a separate buffer is used for output, storing a flag of X'FF' in the last byte of the ASCII buffer.

If the Baudot byte count is specified in the HEX statement, the routine will return control to the calling program when the specified number of input characters has been translated. If a flag of X'FF' is used in either of the input or output buffers, the routine will discontinue processing when the flag is encountered, unless the byte count is satisfied before the flag is found. That is, if the flag termination is to be used, the HEX statement should specify the hexadecimal value FFFF so that the byte count will never be satisfied before the buffer flag is encountered.

The actual character value of Baudot code appears in the right-most five bit positions of the input byte. The upper, or left-most, three bits indicate whether the code was input from the teletype keyboard (i.e., the first three bits are zeros) or from the high speed paper tape (i.e., the first three bits are ones). The conversion routine zeros these three bits and performs the conversion on the valid five bits of the input Baudot bytes.

The Baudot to ASCII converter routine contains the translate table of ASCII codes to be used in performing the conversion.

To call the converter routine, the programer writes the call statement

```
$       #JMBAC
```

in his source program. Since no dummy arguments are used in the macro routine coding, no actual arguments are required in the call statement. The entire converter routine will be incorporated in the calling program at assembly time.

At execution time, the calling sequence will be performed and control transferred to the assembled converter coding. When the conversion is completed, control will return to the seventh word in the calling program's parameter list.

2.17 ASCII to Baudot Converter Macro Routine

This user macro routine performs the processing required to convert the seven-bit internal ASCII code to the five-bit Baudot code for output to the ASR 32 teletype device. If this macro routine is to be called by a user program, the calling sequence must be established in the calling program as shown below.

```
        LAX2    TAG1            LOAD POINTER TO PARAMETER LIST
        JMP,N   TAG3            GO TO CONVERSION ROUTINE
TAG1    ADC     INPUT           INPUT BUFFER POINTER (ASCII CODE)
        ADC     OUTPUT          OUTPUT BUFFER POINTER (CONVERTED BAUDOT CODE)
        HEX     (BYTE COUNT)    NUMBER OF ASCII BYTES TO TRANSLATE
        RESV,0  2               NUMBER OF BAUDOT BYTES TRANSLATED
        RESV,0  2               ERROR WORD
TAG3    ADC     #JQABC          POINTER TO STARTING LOCATION IN CONVERTER
        EXREF   #JQABC          EXTERNAL REFERENCE STATEMENT FOR #JQABC
        JMP     TAG2            RETURN ADDRESS IN CALLING PROGRAM
         .
         .
         .
TAG2    EQU     *
```

Note that the input buffer contains the ASCII code to be converted to Baudot code and stored in the output buffer.

Normal termination of the ASCII to Baudot converter routine occurs when one of the following conditions is encountered:

1. The specified number of input bytes has been processed.

2. A flag of X'FF' is encountered in the input buffer.

Abnormal termination of the converter routine may occur because of one of the following conditions:

1. The input and output buffers are equal in size.

2. The end of the output buffer was reached before all input code was processed.

These error condition terminations may be prevented by ensuring that the output buffer is large enough to accommodate all of the characters generated by the routine. The total number of characters generated by the routine includes all of the converted Baudot data characters, plus a shift character each time the code conversion switches from alpha data to numeric data, or from numeric data to alpha data. That is, the converter routine outputs a shift character to cause the teletype to switch from upper case to lower case and vice versa when the keyboard position of the current output character differs from the previous output character's position on the keyboard. The total number of output characters will always include at least one shift character, and could possibly include as many shift characters as data characters, thus doubling the output buffer size requirement.

In setting up the output buffer area, the programer should reserve a storage space twice the size of the input buffer, or estimate the number of shift characters that will be generated and set up an output buffer equal to the estimated number of shift characters plus the number of characters reserved for the input buffer. If storage space is critical, the programer may conserve space by overlapping the input and output buffer in one of the following ways:

1. Set up a buffer area equal to the estimated maximum number of shift characters plus the number of input characters, and then perform the following:

   Set the output buffer pointer at the upper location of the buffer area.

   Set the input buffer pointer below the output pointer at the point equal to the estimated number of shift characters.

   Set a termination flag of X'FF' in the last byte of the shared buffer and set the byte count in the HEX statement of the parameter list to X'FFFF'.

2. Set up a buffer area twice the size of the input buffer requirement, then perform the following:

   Set the output buffer pointer at the upper location of the buffer area.

   Set the input buffer pointer at the middle of the buffer area.

   Set a termination flag of X'FF' in the last byte of the shared buffer and set the byte count in the HEX statement of the parameter list to X'FFFF'.

When either of the above actions has been taken in the source program, at execution time the converter routine retrieves the first input code from the starting input pointer location, converts it, and places the output code at the starting output pointer location. As the conversion proceeds, the buffer pointers move down each time code is converted. Eventually, the output data and shift characters write into locations previously occupied by input ASCII code. When the X'FF' flag is encountered, the conversion process terminates and control is transferred to the calling program at the seventh word in the parameter list.

The ASCII to Baudot converter routine contains the translate table of Baudot codes used in performing the conversion.

To call the converter routine, the programer writes the call statement in one of the following formats:

$ #JMABC Arg$_1$, Arg$_2$, Arg$_3$, Arg$_4$

or

$ #JMABC ,,,,

The actual arguments in the first call statement specify that error checking statements in the generalized macro routine are to be eliminated. That is, dummy arguments of the forms (1Y), (2Y), (3Y), and (4Y), appear in the error checking statements. These arguments specify that the Assembler is to omit the statements in which they appear if actual arguments appear in positions 1, 2, 3, and 4 of the call statement list. The actual arguments may be any characters except commas. The actual arguments must be separated by commas.

The second form of the call statement omits the actual argument list via a series of four commas. This call statement causes the error checking statements to be included in the specialized routine that the Assembler inserts in-line in the calling program.

The only advantage in using the first form of the call statement is to eliminate approximately 30 bytes of storage for the error checking statements and to save the time required to process them. Since the storage space and execution time requirements are relatively inconsequential, the programer is advised to use the second form of the call statement to include the error checking unless storage space is critical. The error checking statements test for the following conditions:

1. Input and output buffers are equal in size, in which case an error termination will occur, indicated by an error code of 1 returned in the error word of the calling program's parameter list.

2. There is no equivalent Baudot code for an ASCII code to be converted, indicated by an error code of 2 returned in the error word of the calling program's parameter list.

3. The end of the output buffer was reached before all of the input was converted, in which case an error termination will occur, indicated by an error code of 4 returned in the parameter list error word.

If the error checking statements are excluded, no error indicators will be returned to the calling program. The converter routine will still terminate, however, when the output buffer is too small (i.e., when error conditions 1 and 3 above occur).

In all cases, the ASCII to Baudot converter routine returns control to the calling program at the seventh word of the parameter list.

2.18 ALIOCS Macro Routine

The ALIOCS macro routine is a listing of all the input/output coding used by the PTS-100 cassette native assembler. The lengthy listing is not included in this handbook since it is used only for assembly.

2.19 Disc Logical Input/Output Macro Routine

The Disc Logical Input/Output (LIOCSD) macro routine contains the coding necessary to carry out the actions requested by the disc action and status macros, which are branches to certain routines in the Disc Logical Input/Output macro. This is the main processing macro for the disc.

The Disc Logical Input/Output macro picks up the File Control Block pointer specified by an action or status macro, and uses the fields in that File Control Block both as sources of information about the file and as working storage for the processing of the file.

2.20 Disc File Control Block Description Macro Routine

The Disc File Control Block Description (FCBD) macro establishes a file control block, a work area 100 bytes long, in which the logical input/output maintains all information about the file and the current state of its processing. The generalized coding of the File Control Block Description macro routine is shown in Figure 4-14.

Since there are 10 dummy arguments in the routine, the call statement to the routine, must transmit 10 actual arguments, as follows:

Argument 1:	The label of the File Control Block; this label is referred to in all the action and status macros to identify the specific file to be accessed.
Argument 2:	A number from 0 to 7, identifying the disc drive on which the file resides.
Argument 3:	Type of buffering. S = single buffer. D = double buffer (may be specified only for sequential files).
Argument 4:	First buffer address.
Argument 5:	Second buffer address. (This should be specified as 0 for single buffering.)
Argument 6:	Address of file name, a 10-byte field containing the name that was assigned to the file in the Volume Directory by the Disc Allocator program.
Argument 7:	Address of error word, a one-word field which will be set to an error identifying number if an error occurs in processing this file.
Argument 8:	Relative position of key in record, for direct access. For type K random files this gives the byte location, relative to byte 0 of the record, of the start of the key field.
Argument 9:	The number of bytes in the key field.
Argument 10:	The length of the buffer, in sectors. This is the number of 320-byte sectors that can be read or written at one time, based on the length of the buffer(s) provided.

Each action or status macro contains a pointer to the File Control Block of the file on which t is to operate.

```
?FCBD
*FILE CONTROL BLOCK
(1)      HEX        Ø                      FIOB ERROR WORD
         HEX        Ø                      FUNCTION, LOGICAL UNIT
         ADC        (4)                    FIOB BUFFER ADDRESS
         HEX        Ø                      FIOB BYTE COUNT
         HEX        Ø                      FIOB TRACK AND CYL. ADDRESS
         HEX        Ø                      FIOB SECTOR ADDRESS
         HEX        Ø
         HEX        Ø
         DEC        (2)                    FIOB LUN EXTENSION
         TEXT       ' (3)'                 SINGLE OR DOUBLE BUFFERS
         ADC        (4)                    PROCES BUFFER ADDRESS
         HEX        Ø                      PROCESS BUFFER TOTAL BYTES
         HEX        Ø                      PROCESS BUFFER CURRENT BYTES
         ADC        (4)                    PROCESS BUFFER POINTER
         ADC        (5)                    I/O BUFFER ADDRESS
         HEX        Ø                      I/O BUFFER TOTAL BYTES
         HEX        Ø                      I/O BUFFER STATUS
         ADC        (6)                    FILE NAME ADDRESS
         HEX        Ø                      LEVEL 1 SUBR. EXIT
         HEX        Ø                      LEVEL 2 SUBR. EXIT
         HEX        Ø                      END-FILE INDICATOR
         HEX        Ø                      ERROR INDICATOR
         ADC        (7)                    ERROR WORD ADDRESS
         DEC        (8)                    RELATIVE POSITION OF KEY
         DEC        (9)                    KEY LENGTH
         DEC        (1Ø)                   NUMBER OF SECTORS IN BUFFER
         HEX        Ø                      FILE ORGANTZATION
         HEX        Ø                      FIRST CYLINDER IN FILE
         HEX        Ø                      LAST CYLINDER IN FILE
         HEX        Ø                      NUMBER OF BYTES PER RECORD
         HEX        Ø                      FIXED OR VARIABLE LENGTH
         HEX        Ø                      SECTORS PER BLOCK
         HEX        Ø                      START OF OVERFLOW AREA
         HEX        Ø                      HIGHEST TRACK NUMBER
         HEX        Ø                      OPENED STATUS OF FILE
         HEX        Ø                      ERROR EXIT
         HEX        Ø                      END-FILE EXIT
         HEX        Ø                      WORK AREA ADDRESS
         HEX        Ø                      TYPE OF ACCESS
         HEX        Ø                      ADDRESS OF KEY IN PROGRAM
         HEX        Ø                      RELATIVE ADDRESS
         HEX        Ø                      TEST INDICATOR ADDRESS
         HEX        Ø                      COMPLETION FLAG
         HEX        Ø                      WORK AREA POINTER
         HEX        Ø                      RECORD MOVE COUNT
         HEX        Ø                      END-ACTION ADDRESS
         HEX        Ø                      WORKING STORAGE
         HEX        Ø                      WORKING STORAGE
         HEX        Ø                      DELETION FLAG
         HEX        Ø                      EXC SUBROUTINE EXIT
         END
```

Figure 4-14. Generalized Coding of the FCBD Macro Routine.

2.21 Disc Action and Status Macros

The first parameter of every disc action macro is the label of the File Control Block that identifies the file to be accessed. This must be the same as the first parameter of the file control block description macro for that file.

The action macros are Open, Close, Get, Put, Read, Write, and Delete. Open and Close apply to all file types. Get and Put apply only to sequential files. Read, Write, and Delete apply only to random files.

Test and Wait are status macros and do not result in any file accessing. One of the two must be used after each of the action macros to assure that one action has been completed before the next one is requested. The Test or Wait macro need not follow the action macro immediately but must be issued before another action call.

2.21.1 Open Disc (OPEND)

The Open macro must be issued before any accessing can be done on a file. It searches for the file in the Volume Directory (established by the Disc Allocator utility program), and places in the File Control Block various pieces of information describing the file. It also opens the logical unit through the IOCS monitor. The generalized coding of the routine is shown below.

```
?OPEND
                    LAX2          *+6
                    JMP           #LDO0,L
                    ADC           (1)
                    TEXT          ' (2)'
                    ADC           (3)
                    END
```

Since three dummy arguments appear in the macro routine, the user call statement must transmit three actual arguments, as follows:

Argument 1:	The label of-the File Control Block.
Argument 2:	This argument designates the type of open. I = open for input. O = open for output.
Argument 3:	The label of the error exit is specified as argument 3. This is the location to branch to if the file cannot be opened. The exact reason for the failure is indicated by the error word pointed to in the File Control Block.

2.21.2 Close Disc (CLOSED)

The Close macro is issued when all file accessing has been completed. For output sequential files it causes the last record to be written on the disc, followed by an end of file indicator. If there are no other files open, it also closes the logical unit. The generalized coding of the routine is shown below.

```
?CLOSED
                    LAX2          *+6
                    JMP           #LDC0,L
                    ADC           (1)
                    ADC           (2)
                    END
```

There are two dummy arguments in the Close Disc macro routine:

Argument 1:	The label of the file control block.
Argument 2:	Error exit. This is the location to branch to if an error occurs such that the Close process cannot be completed.

2.21.3 Get (GETD)

The Get disc macro, applicable only to sequential files, causes the next logical record to be moved from the buffer to the specified work area. Buffer switching and disc reads are executed when necessary. The generalized coding of the routine is shown below.

```
?GETD
                    LAX2          *+6
                    JMP           #LDG0,L
                    ADC           (1)
                    ADC           (2)
                    ADC           (3)
                    ADC           (4)
                    END
```

There are four dummy arguments in the Get disc macro routine:

Argument 1:	The label of the File Control Block.
Argument 2:	The address of the work area into which the next record is to be moved.
Argument 3:	End of file exit specifying location to branch to if end of file is found.
Argument 4:	The label of the error exit.

2.21.4 Put (PUTD)

The Put disc macro, applicable only to sequential files, causes the logical record in the designated work area to be moved to an output buffer. Buffer switching and disc writes are executed when necessary. The generalized coding of the routine is shown below:

```
?PUTD
                    LAX2          *+6
                    JMP           #LDP0,L
                    ADC           (1)
                    ADC           (2)
                    ADC           (3)
                    END
```

There are three dummy arguments in the Put disc macro routine:

Argument 1:	The label of the File Control Block.
Argument 2:	The address of the work area from which the record is to be taken.
Argument 3:	The label of the error exit.

2.21.5 Read (READD)

The Read disc macro, applicable only to random files, inputs one record from the disc into a specific work area. The generalized coding of the routine is shown below:

```
?READD
                    LAX2          *+6
                    JMP           #LDR0,L
                    ADC           (1)
                    ADC           (2)
                    ADC           (3)
                    ADC           (4)
                    TEXT          '(5)'
                    ADC           (6)
                    END
```

There are six dummy arguments in the Read disc macro routine:

Argument 1:	The label of the File Control Block.
Argument 2:	The address of the work area into which the record is to be moved.
Argument 3:	End-of-file exit.
Argument 4:	Uncorrectable error exit.
Argument 5:	Type of access. D = direct access. S = sequential access.

Argument 6: Address of direct access parameter list. (This argument is zero if there is no list.) If there is a list, it contains two fields, as follows:

a. Address of key field. This is the address of the left end of the field with which the key in the accessed record is to be compared. The value is zero if there is no key.

b. Relative address. For K type files this is a relative track address; for N type files it is a relative record address.

2.21.6 Write (WRITED)

The Write disc macro, applicable only to random files, outputs one record to the disc from a specific work area. The generalized coding of the routine is shown below:

```
?WRITED
                      LAX2          *+6
                      JMP           #LDW0,L
                      ADC           (1)
                      ADC           (2)
                      ADC           (3)
                      ADC           (4)
                      END
```

There are four dummy arguments in the Write disc macro routine:

Argument 1:	The label of the File Control Block.
Argument 2:	The address of the work area from which the record is to be moved.
Argument 3:	Uncorrectable error exit.
Argument 4:	Address of direct access parameter list, which contains only the relative address. For K type files this is interpreted as a relative track address; for N type files it is a relative record address.

2.21.7 Delete (DELD)

The Delete disc macro, applicable only to random files with keys, locates a specific record in the file and marks it deleted by changing its banner word from hexadecimal 0001 to hexadecimal 0000. The generalized coding of the routine is shown below:

```
?DELD
                 LAX2          *+6
                 JMP           #LDD0,L
                 ADC           (1)
                 ADC           (2)
                 ADC           (3)
                 END
```

There are three dummy arguments in the Delete disc macro routine:

Argument 1:	The label of the File Control Block.
Argument 2:	Uncorrectable error exit.
Argument 3:	Address of direct access parameter list. The list contains two fields, as follows:

a. Address of key field. This is the address of the left end of the field with which the key in the accessed record is to be compared.

b. Relative track address.

2.21.8 Wait (WAITD)

The Wait disc macro checks whether the last requested action on the indicated file has been completed. The Wait macro is applicable to the actions Open, Close, Get, Put, Read, Write, and Delete. The generalized coding of the Wait disc macro routine is shown below.

```
?WAITD
                     LAX2           *+6
                     JMP            #LDWA0,L
                     ADC            (1)
                     END
```

The single dummy argument is the label of the File Control Block.

2.21.9 Test (TESTD)

The Test disc macro provides an alternate means of checking the completion of an action macro, without causing a program stall if the action is found to be incomplete. This status macro is applicable to the actions Open, Close, Get, Put, Read, Write, and Delete. The generalized coding of the Test disc macro routine is shown below:

```
?TESTD
                     LAX2           *+6
                     JMP            #LDT0,L
                     ADC            (1)
                     ADC            (2)
                     END
```

There are two dummy arguments in the Test disc macro routine:

Argument 1:	The label of the File Control Block.
Argument 2:	The address of the indicator word. This word is set to 1 if the last action was completed; it is set to 0 if the action was not completed.

INDEX TO MACRO ROUTINES

APPENDIX A

PTS-100 CHARACTER SET

APPENDIX A

PTS-100 CHARACTER SET

				ASCII HEX	
Symbol	Hollerith	Extended Code$_{10}$	ASCII$_8$	8 Bit	7 Bit
@	0-2-8	50	300	C0	40
A	12-1	132	301	C1	41
B	12-2	130	302	C2	42
C	12-3	134	303	C3	43
D	12-4	129	304	C4	44
E	12-5	133	305	C5	45
F	12-6	131	306	C6	46
G	12-7	135	307	C7	47
H	12-8	144	310	C8	48
I	12-9	136	311	C9	49
J	11-1	68	312	CA	4A
K	11-2	66	313	CB	4B
L	11-3	70	314	CC	4C
M	11-4	65	315	CD	4D
N	11-5	69	316	CE	4E
O	11-6	67	317	CF	4F
P	11-7	71	320	D0	50
Q	11-8	80	321	D1	51
R	11-9	72	323	D2	52
S	0-2	34	323	D3	53
T	0-3	38	324	D4	54
U	0-4	33	325	D5	55
V	0-5	37	326	D6	56
W	0-6	35	327	D7	57
X	0-7	39	330	D8	58
Y	0-8	48	331	D9	59
Z	0-9	40	332	DA	5A
[	12-5-8	149	333	DB	5B
\	0-6-8	51	334	DC	5C
]	11-5-8	85	335	DD	5D
↑	7-8	23	336	DE	5E
←	2-8	18	337	DF	5F
blank	No Punch	00	240	A0	20
!	11-2-8	82	241	A1	21
"	0-5-8	53	242	A2	22
#	0-7-8	55	243	A3	23
$	11-3-8	86	244	A4	24
%	11-7-8	87	245	A5	25
&	12-7-8	151	246	A6	26
'	4-8	17	247	A7	27
(	0-4-8	49	250	A8	28
)	12-4-8	145	251	A9	29
*	11-4-8	81	252	AA	2A
+	12	128	253	AB	2B
,	0-3-8	54	254	AC	2C
-	11	64	255	AD	2D
.	12-3-8	150	256	AE	2E
/	0-1	36	257	AF	2F
0	0	32	260	B0	30
1	1	4	261	B1	31
2	2	2	262	B2	32
3	3	6	263	B3	33
4	4	1	264	B4	34
5	5	5	265	B5	35
6	6	3	266	B6	36
7	7	7	267	B7	37
8	8	16	270	B8	38
9	9	8	271	B9	39

Symbol	Hollerith	Extended Code$_{10}$	ASCII$_8$	ASCII HEX 8 Bit	ASCII HEX 7 Bit
:	5-8	21	272	BA	3A
;	11-6-8	83	273	BB	3B
<	12-6-8	147	274	BC	3C
=	3-8	22	275	BD	3D
>	6-8	19	276	BE	3E
?	12-2-8	146	277	BF	3F
NUL	12-9-1	140	200	80	00
SOH	12-9-2	138	201	81	01
STX	12-9-3	142	202	82	02
ETX	12-9-4	137	203	83	03
EOT	12-9-5	141	204	84	04
ENQ	12-9-6	139	205	85	05
ACK	12-9-7	143	206	86	06
BEL	11-9-1	76	207	87	07
BS	11-9-2	74	210	88	08
HT	11-9-3	78	211	89	09
LF	11-9-4	73	212	8A	0A
VT	11-9-5	77	213	8B	0B
FF	11-9-6	75	214	8C	0C
CR	11-9-7	79	215	8D	0D
SO	0-9-1	44	216	8E	0E
SI	0-9-2	42	217	8F	0F
DLE	0-9-3	46	220	90	10
DC1	0-9-4	41	221	91	11
DC2	0-9-5	45	222	92	12
DC3	0-9-6	43	223	93	13
DC4	0-9-7	47	224	94	14
NAK	12-0-8	176	225	95	15
SYN	12-1-8	148	226	96	16
ETB	11-0-8	112	227	97	17
CAN	11-1-8	84	230	98	18
EM	0-1-8	52	237	99	19
SUB	12-8-9	152	232	9A	1A
ESC	11-8-9	58	233	9B	1B
FS	0-8-9	56	234	9C	1C
GS	8-9	24	235	9D	1D
RS	9-1	12	236	9E	1E
US	9-2	10	237	9F	1F
DEL	1-8	20	377	FF	7F

USER COMMENTS

DOES THIS PUBLICATION SATISFY YOUR NEEDS? Please use this form to advise RDS of any errors or omissions or to recommend additions or deletions. (All replies become the property of Raytheon Data Systems Company.)

Publication Title and No. ____________________

Date of Issue (from title page) ____________________

Please list comments below by page number. (Enclose additional sheets if desired.)

fold

fold

fold

fold

NO POSTAGE REQUIRED IF MAILED IN USA.
Fold in thirds and staple or tape closed.

USER COMMENT FORM

Does this publication satisfy your need for information?

Is it clear?

Can you suggest improvements?

RDS will appreciate your frank comments on this entire publication or any part of it. All replies will be carefully reviewed and all suggestions considered before the next edition is published.

Please write your comments on the reverse side of this form. Thank you.

fold inside

NO POSTAC
NECESSAR
IF MAILEC
IN THE
UNITED STA

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 25529 NORWOOD, MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

RAYTHEON COMPANY
RAYTHEON DATA SYSTEMS
Marketing Department/Technical Writing Services
1415 Boston-Providence Turnpike
Norwood, Massachusetts 02062

fold outside

User's Name (optional) ______________________________

Title ______________________________

Company ______________________________

Address ______________________________

RAYTHEON

FORM NO. 96-0206