TEXAS INSTRUMENTS

Improving Man's Effectiveness Through Electronics

Model 990 Computer

Prototyping System
Operation Guide

MANUAL NO. 945255-9701
ORIGINAL ISSUE 1 MAY 1976

Digital Systems Division

LIST OF EFFECTIVE PAGES

INSERT LATEST CHANGED PAGES DESTROY SUPERSEDED PAGES

Note: The portion of the text affected by the changes is indicated by a vertical bar in the outer margins of the page.

Model 990 Computer Prototyping System Operation Guide (945255-9701)

Original Issue . . . . 1 May 1976

Total number of pages in this publication is 254 consisting of the following:

PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.
Cover	0	A-1 – A-2	0		
Eff. Pages	0	B-1 – B-8	0		
iii – x	0	C-1 – C-8	0		
1-1 – 1-14	0	D-1 – D-12	0		
2-1 – 2-14	0	E-1 – E-2	0		
3-1 – 3-74	0	F-1 – F-2	0		
4-1 – 4-24	0	G-1 – G-4	0		
5-1 – 5-6	0	Alphabetical Index Div	0		
6-1 – 6-8	0	Index-1 – Index-8	0		
7-1 – 7-36	0	User's Response	0		
8-1 – 8-4	0	Business Reply	0		
9-1 – 9-4	0	Cover Blank	0		
10-1 – 10-24	0	Cover	0		
11-1 – 11-6	0				

PREFACE

This manual describes the 990 Prototyping System and gives instructions for installing and operating it on the Model 990/4 Computer. It provides detailed descriptions of the individual modules that comprise the system software and includes techniques for using them.

This publication is intended for users of the 990 Prototyping System package: users who are developing applications programs for the 990/4 Computer, the 990/10 Computer, and the TMS9900 Microprocessor, and users who generate and test read-only memory (ROM) resident programs for use with the TMS9900 Microprocessor.

The information in this manual is divided into the following sections:

I. General Description – Briefly describes the 990 Prototyping System Software, the modules in the system, and the hardware components that support it.

II. System Installation and Operation – Gives the sources of information on unpacking, installing and operating the supporting hardware, with appropriate references. Step-by-step procedures for installing the software, loading the software modules, and operating the modules are included. Interrupts and single instruction execution are explained.

III. Debug Monitor – Describes the I/O operations, loading methods, debugging modes, and operator commands of the debug monitor. Debugging techniques are discussed at length. The commands are explained and detailed descriptions of each of the commands are included. Detailed descriptions of the supervisor calls follow.

IV. Text Editor – Presents detailed loading, initialization and editing procedures for the text editor. This section includes descriptions of each of the text editor commands. Explanations of printed messages and a source program example are also included.

V. One-Pass Assembler – Gives a general description of the assembler and details the procedures for loading and operating it. The operation discussion covers input/output and printed messages. Directives and pseudo-instructions are briefly discussed, error messages are explained, and an example of printed output is included.

VI. Object Code Formats – Explains the two object code formats: standard 990 object code and compressed absolute format object code.

VII. PROM Programmer – Describes the functions and capabilities of the PROM Programmer software module, the data configurations that it handles, and the procedure for programming PROMs. The section includes detailed descriptions of PROM Programmer commands and gives examples of the use of commands. Example programs are included.

VIII. BNPF Dump Module – Explains how to use the BNPF Dump overlay module. Presents a detailed description of the data format and the commands.

IX. HIGH/LOW Dump Module – Explains how to use the HIGH/LOW Dump overlay. Presents a detailed description of the data format and the commands.

X. System Operation and Debugging Example – Presents a complete example that illustrates assembly, loading, debugging and editing. An explanation of each phase of the example program is included.

XI. PROM Programming Examples – Presents examples of procedures for programming PROMs.

The appendixes cover compatibility of the Prototyping System with the DX10 operating system, an explanation of the stand-alone programming procedure, the character set used in the assembly language and in data terminal input/output, a summary of commands and directives, and a list of error messages. The appendixes also include an explanation of memory and PROM mapping parameters and tables of information related to PROM programming.

The following publications contain additional information needed to use the 990 Prototyping System.

Title	**Manual Number**
Model 990/4 Computer System Hardware Reference Manual	945251-9701
Model 990 Computer TMS9900 Microprocessor Assembly Language Programmer's Guide	943441-9701
Model 990 Computer Model 733 ASR/KSR Data Terminal Installation and Operation	945259-9701
Model 990 Computer PROM Programming Module Installation and Operation	945258-9701
Model 990 Computer Programming Card	943440-9701

TABLE OF CONTENTS

Paragraph Title Page

SECTION I. GENERAL DESCRIPTION

SECTION II. SYSTEM INSTALLATION AND OPERATION

SECTION III. DEBUG MONITOR

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Paragraph Title Page

SECTION IV. TEXT EDITOR

SECTION V. ONE-PASS ASSEMBLER

TABLE OF CONTENTS (Continued)

Paragraph Title Page

SECTION VI. OBJECT CODE FORMATS

SECTION VII. PROM PROGRAMMER

SECTION VIII. BNPF DUMP MODULE

TABLE OF CONTENTS (Continued)

APPENDIXES

LIST OF ILLUSTRATIONS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

SECTION I

GENERAL DESCRIPTION

1.1 INTRODUCTION

This section presents an overview of the 990 Prototyping System hardware and software. The first portion of the section describes the purpose and capabilities of the system software. The equipment in the hardware configuration that supports the system software is identified and briefly discussed. The following paragraphs identify the sources of information required to install and operate the hardware, present the memory requirements and configurations for the 990/4 computer, and list the part numbers for the 990 Prototyping System hardware and software components.

The remainder of the section describes the modules that comprise the system software. These modules include the debug monitor, monitor overlay functions, text editor, one-pass assembler, programmer panel and 733 ASR ROM loader firmware, PROM programmer, BNPF Dump Module, and HIGH/LOW Dump Module. Memory requirements and loaders are also discussed, and the prototyping process is described.

1.2 PURPOSE AND CAPABILITIES OF THE SYSTEM

The 990 Prototyping System Software provides interactive generation and development of applications programs for all members of the 990 Computer Family. It operates on the Model 990/4 Computer. The Prototyping System Software package supports up to 28K words of memory.

With this system, the user can develop capabilities previously reserved for electromechanical devices, discrete logic or conventional integrated circuits.

In addition to applications program development, it is particularly suited to generation and testing of firmware (software resident in read-only memory) programs for use with the TMS9900 microprocessor.

1.2.1 PROTOTYPING SYSTEM SOFTWARE DESCRIPTION. The purpose of the Prototyping System Software is to provide the capability to generate, edit, assemble, load and debug user programs for software applications and firmware generation. In addition to the debug functions, the Debug Monitor provides supervisor calls to perform input/output (I/O) operations on the 733 ASR Data Terminal and utility routines such as decimal ASCII to binary, hexadecimal ASCII to binary, binary to decimal ASCII, and binary to hexadecimal ASCII conversion routines. Overlays to the Debug Monitor provide a program trace package, a linking loader, and the capability to dump a program in memory to tape in a compressed absolute format and load it back into memory. In addition, overlays provide a PROM programmer package and BNPF and HIGH/LOW dump programs. The BNPF and HIGH/LOW dump programs provide the capability to create cassette tapes in BNPF or HIGH/LOW format (formats that encode sequences of bits) for prototyping applications. The BNPF overlay also provides the capability to load BNPF format tapes back into memory.

The system software package is available in object format on a read-only magnetic tape cassette and in source format on punched cards; however, the system source must be assembled and linked using a 990/10 Program Development System. The system software provides source and object compatibility with other 990 systems.

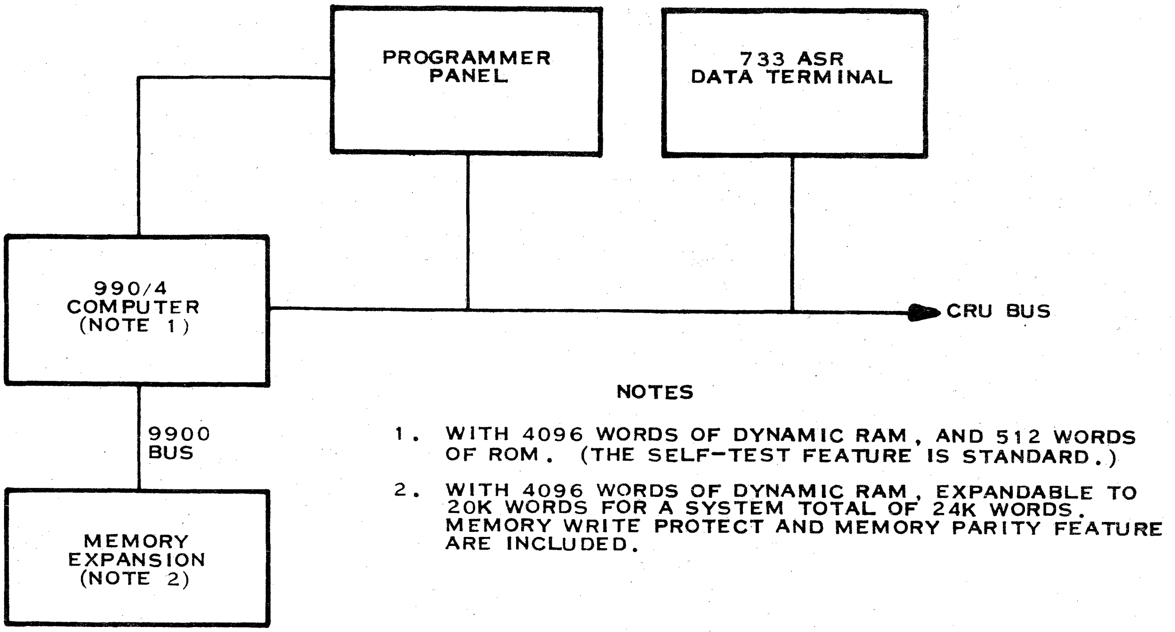
1.2.2 HARDWARE CONFIGURATION REQUIRED FOR SYSTEM SOFTWARE. The Prototyping System Software requires the following hardware configuration:

- Computer – the 990/4 microcomputer. The computer has access to dynamic random access memory (RAM) and on-board read-only memory (ROM) as described in paragraph 1.2.2.1. A chassis, power supply and packaging is available with the computer.
- 733 ASR Data Terminal
- Programmer Panel
- PROM Programming Module (optional)

A simplified block diagram of the hardware configuration is shown in figure 1-1.

1.2.2.1 Model 990/4 Computer. The Model 990/4 Computer consists of the 990/4 microcomputer on a single printed circuit card, one or more memory expansion cards, and a chassis and power supply. The 990/4 microcomputer circuit card contains the CPU, 4K words of on-board dynamic random access memory, and up to 1K of ROM or static RAM.

Detailed information on the Model 990/4 Computer may be found in the *Model 990/4 Computer System Hardware Reference Manual,* Manual No. 945251-9701.



NOTES

1. WITH 4096 WORDS OF DYNAMIC RAM, AND 512 WORDS OF ROM. (THE SELF-TEST FEATURE IS STANDARD.)
2. WITH 4096 WORDS OF DYNAMIC RAM, EXPANDABLE TO 20K WORDS FOR A SYSTEM TOTAL OF 24K WORDS. MEMORY WRITE PROTECT AND MEMORY PARITY FEATURE ARE INCLUDED.

(B)133277

Figure 1-1. 990 Prototyping System Hardware Block Diagram

Central Processing Unit. The 990/4 CPU has the following characteristics:

- Eight interrupts (up to seven external and a power-up trap)
- Real-time clock
- Communications Register Unit (CRU) interface for I/O
- Direct Memory Access (DMA) interface for extended memory, which can be used for processor-independent I/O (when a user-designed controller is implemented)
- Self-test routine
- Fault indicator

Memory. The minimum memory required for the Prototyping System Software is 4096 words of on-board dynamic RAM, 512 words of on-board ROM for the 733 ASR loader and the self-test feature, and 4096 words of dynamic RAM on a memory expansion circuit card. The expansion card may be expanded to 20K words, giving a total of 24K words of dynamic RAM in the Prototyping System configuration. The 512 words of ROM are divided into 256 words of firmware for the 733 ASR ROM loader (both tape cassette and cards) and programmer panel and 256 words for the CPU/memory self-test routines. The card loader is included for compatibility with the 990/10 Computer. ROM or static RAM may be expanded by an additional 512 words.

An additional EPROM memory expansion card is available. This card may contain from 1K to 8K memory words in 1K increments.

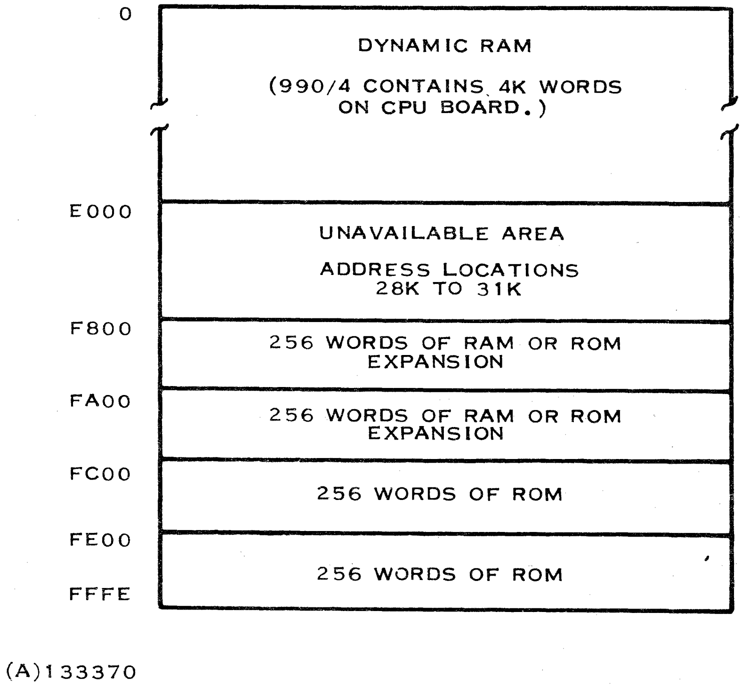

0
DYNAMIC RAM
(990/4 CONTAINS 4K WORDS ON CPU BOARD.)
E000
UNAVAILABLE AREA
ADDRESS LOCATIONS 28K TO 31K
F800
256 WORDS OF RAM OR ROM EXPANSION
FA00
256 WORDS OF RAM OR ROM EXPANSION
FC00
256 WORDS OF ROM
FE00
256 WORDS OF ROM
FFFE
(A)133370

Figure 1-2. Hardware Memory Configuration

Memory write protect is required in the 990 Prototyping System, and is implemented on the 990/4 memory expansion circuit card. A memory parity feature (which provides parity error detection logic and an interrupt signal to the CPU) is standard in the 990 Prototyping System.

The hardware memory configuration is shown in figure 1-2. The numbers at the left are byte addresses.

Chassis. The computer chassis is available in two configurations, one that holds 6 full-size cards and one that holds 13 full-size cards. In addition, a table-top chassis mounting option is available with the 6-slot chassis. The power supply is located in the computer chassis.

1.2.2.2 Interrupt, XOP and Trap Vectors. This discussion covers the different types of vectors and explains the power-up trap.

Vectors. Located in 990/4 memory are dedicated locations reserved for interrupt, XOP and trap vectors. The interrupt and XOP vectors are available for the exclusive use of user programs, except that one XOP may be used for executing supervisor calls. A vector is a two-word pair providing the program counter and workspace for the service routine that handles an interrupt or XOP.

Power-Up Trap. The power-up interrupt traps through a vector at address zero or address $FFFC_{16}$, depending on a jumper wire implemented on the 990/4 CPU board. This allows more flexible memory allocation for dedicated systems that do not have an operator panel. The 990 Prototyping System powers up through trap address $FFFC_{16}$.

Trap addresses are illustrated in figure 1-3.

1.2.2.3 733 ASR Data Terminal. The 733 ASR Data Terminal provides the communication link between the user and the computer system. It is an automatic send-receive terminal, allowing either automatic or manual entry of data and output of data under keyboard or remote control.

The major components of the terminal are the following:

- Keyboard
- Thermal printer
- Two magnetic tape cassette units

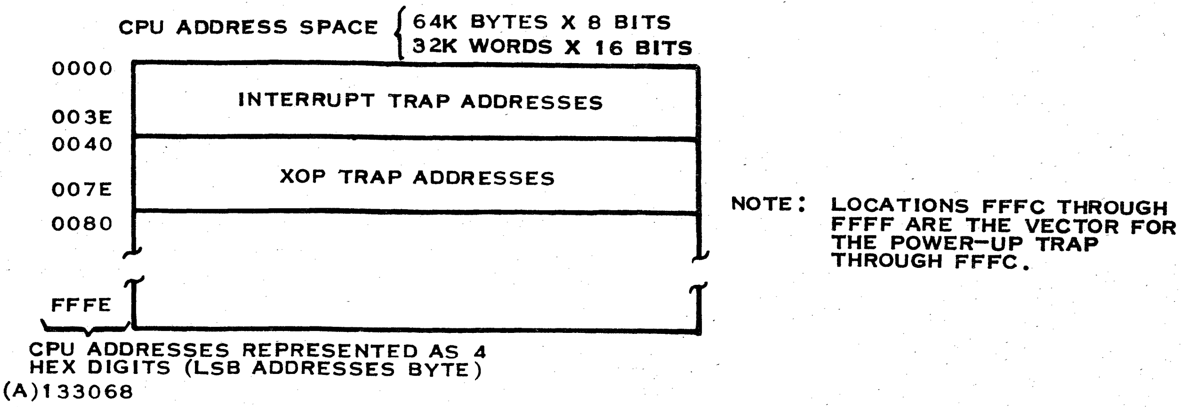

CPU ADDRESS SPACE
64K BYTES X 8 BITS
32K WORDS X 16 BITS
0000
INTERRUPT TRAP ADDRESSES
003E
0040
XOP TRAP ADDRESSES
007E
0080
FFFE
CPU ADDRESSES REPRESENTED AS 4 HEX DIGITS (LSB ADDRESSES BYTE)
(A)133068
NOTE: LOCATIONS FFFC THROUGH FFFF ARE THE VECTOR FOR THE POWER-UP TRAP THROUGH FFFC.

Figure 1-3. Trap Addresses

For more efficient use of the hardware resources, the Prototyping System Software debug monitor recognizes the keyboard and printer as the input and output portions of one I/O device. It also recognizes tape cassette unit 1 and tape cassette unit 2 as distinct I/O devices.

The assembler recognizes the keyboard and printer as separate devices. The keyboard and printer are functionally distinct, and either cassette may function as the record cassette or playback cassette.

The tape cassettes function as a terminal-based data storage system, providing a convenient method for loading software modules, storing data temporarily, and recording data permanently on a compact, easy-to-handle storage medium. File data may be read from cassette to the computer, or computer output may be stored on cassettes. The user may write data to either cassette. When one unit is in the record mode, the other is in playback mode.

1.2.2.4 Programmer Panel. The Programmer Panel gives the user full control of the CPU by entering control information from the panel instead of the data terminal keyboard. Memory may be examined and modified directly from the panel. This capability is useful in applications requiring software troubleshooting.

1.2.2.5 PROM Programming Module. The Programmable Read-Only Memory (PROM) Programming Module, which is optional, enables the user to program his own PROMs. The module chassis includes front panel keylock and indicators, device sockets with a programming adaptor, and a power supply. (A programming adaptor is a plug-in module that provides the control functions for a specific PROM device type.) Plug-in adaptors are available for both PROM and erasable programmable read-only memory (EPROM) devices.

The PROM programming module operates as a CRU device. Software programs have direct control over the PROM address, data, timing and control signals which allow the module to supply the voltages and interconnections needed to program several types of PROMs, both bipolar and MOS.

1.3 SYSTEM PART NUMBERS

The hardware and software system part numbers are listed in table 1-1. The individual hardware components of the 990 Prototyping System and their part numbers are listed in table 1-2. Memory sizes listed are the minimum required; memory components with larger capacities may be substituted for those listed.

1.4 SOFTWARE MODULES

The standard software includes these modules:

- Debug Monitor (PX9MTP) – This monitor supports the tape cassette data medium only. Instruction trace, which allows the user to monitor and analyze an executing program, the linking loader (PX9LAL), Absolute Dump/Absolute Load, BNPF Memory Dump, HIGH/LOW Memory Dump, and the PROM Programmer are routines that are overlays and may be loaded into the monitor transient area when they are to be used.

- One-Pass Assembler (PX9ASM)

- Text Editor (PX9EDT)

- Upfront Loader (PX9UFL) – This loader is placed on the system software cassette tape immediately in front of PX9MTP, PX9EDT and PX9ASM. The upfront loader precedes a file of compressed absolute format code and reduces the loading time.

Table 1-1 Prototyping System Part Numbers

Item	TI Part Number
990 Prototyping System with 733 ASR Data Terminal (packages include both hardware and software)	
8K Memory Words	945202-0001
12K Memory Words	945202-0002
16K Memory Words	945202-0003
20K Memory Words	945202-0004
24K Memory Words	945202-0005
990 Prototyping System without 733 ASR Data Terminal (packages include both hardware and software)	
8K Memory Words	945202-0006
12K Memory Words	945202-0007
16K Memory Words	945202-0008
20K Memory Words	945202-0009
24K Memory Words	945202-0010
Prototyping System Software	
Object on Tape Cassette	943380-0001
Source on Card Deck	943380-0012
Standard Control Information Cassette	943350-0001

In addition, firmware programs are located on the ROM modules for the programmer panel and 733 ASR ROM loader.

1.4.1 GENERAL. The following paragraphs discuss the capabilities and requirements of the Prototyping System Software.

1.4.1.1 Memory Requirements. The Prototyping System expects these minimum amounts of RAM and ROM:

- 4K words of user dynamic RAM
- 4K words of system dynamic RAM
- 256 words of system static RAM
- 256 words of system ROM containing the programmer panel and 733 ASR ROM loader firmware.

Table 1-2. Part Numbers of Hardware Required in 990 Prototyping System

Description	Part Number
Module 990/4 Computer Central Processing Unit	
990/4 CPU with 4K 16-bit words of dynamic RAM	944910-0002
Dynamic RAM Parity Feature	945120-0006
733 ASR ROM Loader (Prototyping) (includes self-test feature)	945121-0005
Static RAM Device Kit	945122-0001
Memory Expansion Module (one must be selected)	
4K words with write protect	944935-0006
8K words with write protect	944935-0007
12K words with write protect	944935-0008
16K words with write protect	944935-0009
20K words with write protect	944935-0010
Memory Parity Feature (must match memory expansion module size)	
4K words	945120-0001
8K words	945120-0002
12K words	945120-0003
16K words	945120-0004
20K words	945120-0005
EPROM Memory	
EPROM Memory Module (optional)	945170-0001
EPROM Device Kit (optional)	945123-0004
6-Slot Chassis with Programmer Panel, 20-ampere power supply	944960-0001
733 ASR Data Terminal Kit	945161-0001
PROM Programming Kit (optional)	
Tabletop	944924-0001
Rack Mount	944924-0002
PROM programming accessory equipment	
PROM Programming Adapter (optional)	945135-0001
EPROM Programming Adapter (optional)	945165-0001
EPROM Erase Kit (optional)	945160-0001

A diagram of the memory configuration is shown in figure 1-4. The numbers at the left are byte addresses and are given for the minimum memory configuration of 4K words of user and 4K words of system memory.

1.4.1.2 System Software Loaders. Loading of programs or program modules is accomplished with one or more of the four available loaders provided in the system software:

- *Standard 990 object loader.* Loads a program in standard 990 object code. The loader resides in a 256-word ROM. One of the standard loader's functions is used to load overlays into the monitor transient area.

- *Compressed absolute format loader.* Loads a program that has been stored in compressed absolute format. The loader is an overlay that must be resident in the monitor transient area.

- *Upfront loader (PX9UFL).* Loads a program in compressed absolute format code. The loader must be located immediately in front of the beginning of the compressed absolute format code. The 733 ASR ROM (standard) loader loads the upfront loader, which in turn loads the compressed absolute format code.

- *Relocating linking loader (PX9LAL).* PX9LAL which must be resident in the monitor transient area, loads program modules in object code, modifies memory addresses in the modules, and links the modules. The program code may specify absolute memory locations or specify relocatable memory locations that allow the entire program module to be placed in any sufficiently large available memory area.

These loaders handle either conventional object code or object code in compressed absolute format. The compressed absolute format code allows faster loading than with standard 990 object code. Object code formats are described in detail in Section VI.

1.4.2 DEBUG MONITOR (PX9MTP). PX9MTP, the control program and system executive for the software system, occupies 4K words of memory.

1.4.2.1 Overview. PX9MTP is a modular program that consists of five major divisions:

- I/O executive
- Keyboard command processor
- Supervisor call interface
- Keyboard commands
    - Debug commands
    - System control commands
- Transient area

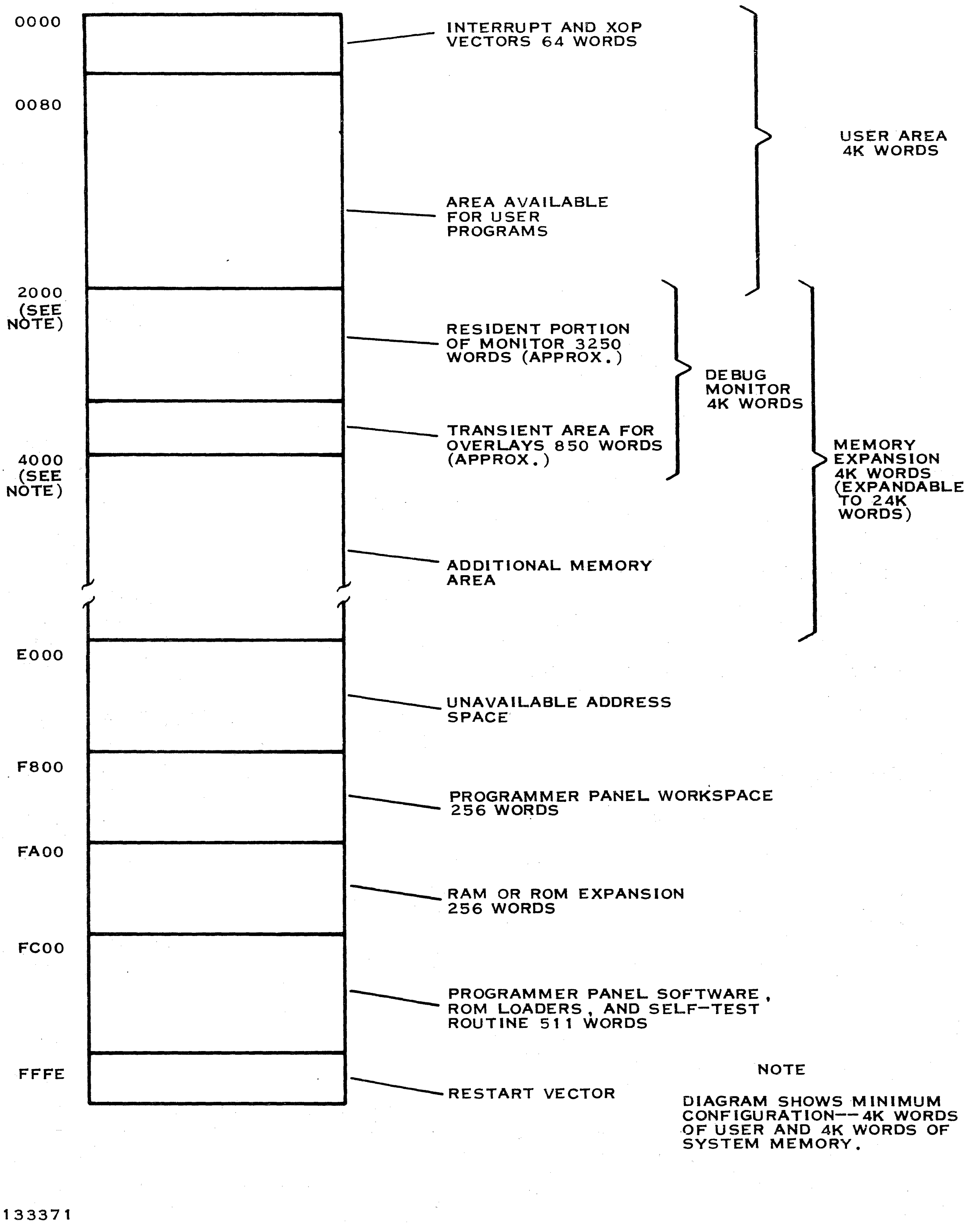

0000
0080
2000 (SEE NOTE)
4000 (SEE NOTE)
E000
F800
FA00
FC00
FFFE
INTERRUPT AND XOP VECTORS 64 WORDS
AREA AVAILABLE FOR USER PROGRAMS
RESIDENT PORTION OF MONITOR 3250 WORDS (APPROX.)
TRANSIENT AREA FOR OVERLAYS 850 WORDS (APPROX.)
ADDITIONAL MEMORY AREA
UNAVAILABLE ADDRESS SPACE
PROGRAMMER PANEL WORKSPACE 256 WORDS
RAM OR ROM EXPANSION 256 WORDS
PROGRAMMER PANEL SOFTWARE, ROM LOADERS, AND SELF-TEST ROUTINE 511 WORDS
RESTART VECTOR
USER AREA 4K WORDS
DEBUG MONITOR 4K WORDS
MEMORY EXPANSION 4K WORDS (EXPANDABLE TO 24K WORDS)
NOTE
DIAGRAM SHOWS MINIMUM CONFIGURATION—4K WORDS OF USER AND 4K WORDS OF SYSTEM MEMORY.
(A)133371

Figure 1-4. 990 Prototyping System Software Memory Configuration

PX9MTP controls user programs and supports the one-pass assembler (PX9ASM) and the text editor (PX9EDT). The debug monitor provides all of the necessary facilities for the following functions:

- Debugging
- Linking
- Loading
- I/O support for user programs.
- Program save and restore

The monitor occupies 4096 words in memory, of which about 3250 words are permanently resident and about 850 words are assigned to an area for overlay modules that may be loaded into memory from cassette when needed. (Refer to Section II for debug monitor loading procedures.)

1.4.2.2 **I/O Supervisor Calls.** The I/O executive decodes and processes 733 ASR I/O supervisor calls from other PX9MTP modules and from user programs. Upward compatibility is maintained because the I/O service request is format compatible with TX990 and DX10 supervisor calls. PX9MTP provides file and record level I/O performed independently of the device type to which the I/O is directed.

1.4.2.3 **Non-I/O Supervisor Calls.** In addition to I/O supervisor calls, the monitor processes such non-I/O calls as user program termination and data format conversion. The formats involved in the conversion routines are binary data and decimal and hexadecimal ASCII character codes. Supervisor calls make use of a block of memory which contains a code for the operation to be performed and parameters associated with the operation.

1.4.2.4 **User Interaction with Monitor.** PX9MTP interacts with the user through the 733 ASR keyboard and printer to receive and decode commands, and to activate the various command processors. Examples of the capabilities offered are:

- LP – Load a program.
- AL – Assign a logical unit number (LUNO) to a specified device.
- EX – Execute a program.
- OV – Overlay the monitor transient area with different cassette-resident commands. (Once they are loaded, transient commands are handled exactly like resident commands).
- PL – Load the PROM Programmer software module.

1.4.3 **MONITOR OVERLAY FUNCTIONS.** In order to limit the memory area occupied by the debug monitor to a size of 4K words, some of the monitor functions are handled as overlays. Overlay-resident commands are extensions of the memory-resident monitor that allow the less frequently used commands to reside on cassette tape. These functions are overlays:

- Link and load
- Absolute dump/absolute load

- Instruction trace
- PROM Programmer
- BNPF format dump
- HIGH/LOW format dump

Overlays are loaded in the transient area of the debug monitor's memory space. Since the PROM programmer is too large to fit in the transient area, part of it is loaded into the highest-numbered address locations of user memory. The overlay-resident commands are used exactly like normal commands once the overlay is loaded into the transient area. Attempts to invoke commands which are not resident will generate error messages.

1.4.3.1 **Linking Loader.** The linking loader, PX9LAL, loads program modules into memory, links the modules as required, and returns control to the monitor after all modules have been loaded. The loaded program modules are object modules produced by one of these assemblers:

- One-Pass Assembler (PX9ASM)
- SDSMAC, the macro-assembler in the 990/10 Program Development System.
- Cross Assembler, which allows the user to assemble code for the 990 Family of computers on an IBM System 360/370 or on certain international timesharing services.

1.4.3.2 **Absolute Dump/Absolute Load.** The Absolute Dump/Absolute Load overlay routine provides two functions: it saves a program or data space in memory by writing that program or data onto 733 ASR cassette tape in compressed absolute data format, and it loads object code that has been stored in compressed absolute data format. The saved memory data sequence is stored in compressed absolute data format, and can be reloaded using either the absolute loader or the upfront loader, both invoked by monitor keyboard commands. The absolute dump can also be used to save an entire memory data sequence complete with the current debug parameters in the data sequence. The memory data sequence can then be reloaded from the start and the debugging continued as if it were never interrupted.

1.4.3.3 **Instruction Trace.** The instruction trace feature allows the user to monitor the contents of internal data sequences and analyze the ongoing progress of an executing program. The user can specify breakpoints and snapshots for interpreting the progress of his program.

1.4.3.4 **PROM Programmer.** The PROM Programmer software package provides flexible control of the PROM programming process through the use of PX9MTP operator commands. The PROM programmer commands inform the control software of memory bounds, PROM characteristics, and mapping parameters. Additional operator commands allow the use of standardized control information for frequently used programming functions. With PROM programmer commands, the user is able to program PROMs and verify the contents of a PROM or ROM circuit.

The PROM programmer requires that the debugged software routine to be transferred to a PROM be resident in memory. The software module selects data from memory and transfers it with other interface data to the PROM Programming Module. The hardware module stores the received data in the PROM as directed by the command.

Two requirements must be met in order to program PROMs:

1. A programming adapter must exist for the chosen device type.

2. The data to be programmed must be loaded into 990/4 memory.

1.4.3.5 BNPF Dump Program. The BNPF Dump software package creates an output file on magnetic tape cassette in a standard BNPF format that can be used to manufacture ROMs. The software package can also read the BNPF format cassette, recreating the memory data sequence used to generate the BNPF cassette tape. The BNPF Dump module requires that the program to be converted into BNPF format be resident in memory.

1.4.3.6 HIGH/LOW Dump Program. The HIGH/LOW Dump software package creates an output file on magnetic tape cassette in TI 256 X 4 HIGH/LOW format that can be used to manufacture ROMs. The HIGH/LOW Dump module requires that the program to be converted into HIGH/LOW format be resident in memory.

1.4.4 USER AREA SYSTEM PROGRAMS. The user area system programs are the packaged system programs that run in the user area of memory. These programs are the Text Editor (PX9EDT) and the One-Pass Assembler (PX9ASM).

1.4.4.1 Text Editor (PX9EDT). PX9EDT is an interactive text editor that runs as a user program invoked by PX9MTP. PX9EDT edits existing source code or creates and saves new source code. It reads an existing program from magnetic tape cassette to a memory buffer for editing and then outputs it to a second cassette.

The text editor processes three different classes of commands:

- Setup commands

- Edit operation commands

- Output commands

Since the object module format for the 990 Family consists of ASCII strings acceptable to the text editor, PX9EDT may also be used to edit object modules.

1.4.4.2 One-Pass Assembler (PX9ASM). PX9ASM is a one-pass assembler that runs as a user program invoked by PX9MTP. PX9ASM accepts source code input from cassette and produces an object program on cassette, a listing and an error summary. The object code produced may contain either relocatable or absolute origin code. It may also contain references to external symbols in other modules and define external symbols. A collection of object modules may then be linked and loaded into memory by the linking loader (PX9LAL).

1.4.5 PROGRAMMER PANEL AND 733 ASR ROM LOADER FIRMWARE. The programmer panel and ROM loader firmware executes from ROM situated in the last 256 words of the memory address space and serves as the system loader. It also gives the operator an elementary level of control over executing programs by means of the programmer panel when the debug monitor (PX9MTP) is not resident.

The ROM firmware handles level 0 interrupts, including power up, HALT and SIE interrupts, and may enter PX9MTP or retain control in the programmer panel firmware depending on the cause of the interrupt.

1.5 PROTOTYPING PROCESS

The Prototyping System provides an efficient mechanism to the TMS9900 microprocessor user for generating and testing stand-alone programs, and for transferring those programs into PROM or ROM devices.

Development of a set of control sequences with the Prototyping System typically evolves through the following steps:

1. *Program development.* Source programs on cassette tapes may be created using the PX9EDT text editor and assembled with the PX9ASM assembler. The assembler generates object code on magnetic tape cassette. Programs may also be developed using the 990/10 Program Development System or the Cross Support System.

2. *Prototype debug.* The program is loaded into the Prototyping System memory and is run and debugged under actual operating conditions. Any problems found are then corrected either (1) in the memory version of the program or (2) by updating the source or object and repeating the development procedure from step 1.

3. *PROM programming.* The tested program is programmed into a PROM using the PROM Programmer software package and the PROM Programming Module. The created PROM is then used in the system in which it will operate for further checking. If other problems occur, a new PROM can be created by repeating the procedure from step 2.

4. *PROM documentation.* The contents of the verified PROM are then dumped to a cassette in BNPF or HIGH/LOW format.

5. *ROM manufacture.* The PROM documentation is used to mass produce copies of the control sequence in ROM circuits.

6. *System production.* The ROM circuits are mated with the microprocessor or microcomputer (as applicable) for the final control system.

SECTION II

SYSTEM INSTALLATION AND OPERATION

2.1 INTRODUCTION

This section presents the hardware and software installation information and operating instructions. The first portion discusses the hardware and includes the following topics:

- Unpacking and installation of hardware
- Hardware operation

This portion includes cabling diagrams of the hardware system and gives references to other manuals in which the unpacking, installation and operating instructions can be found.

The remainder of this section contains detailed instructions for loading and executing the software modules and user programs. The following topics are covered:

- 990 Prototyping System Software cassette generation
- Using the 733 ASR ROM loader
- Operating the monitor
- Entering commands on the terminal keyboard
- Input/output and logical unit assignments
- Loading and executing programs.
- Interrupts and single instruction execution
- Memory write protect

2.2 UNPACKING AND INSTALLATION OF HARDWARE

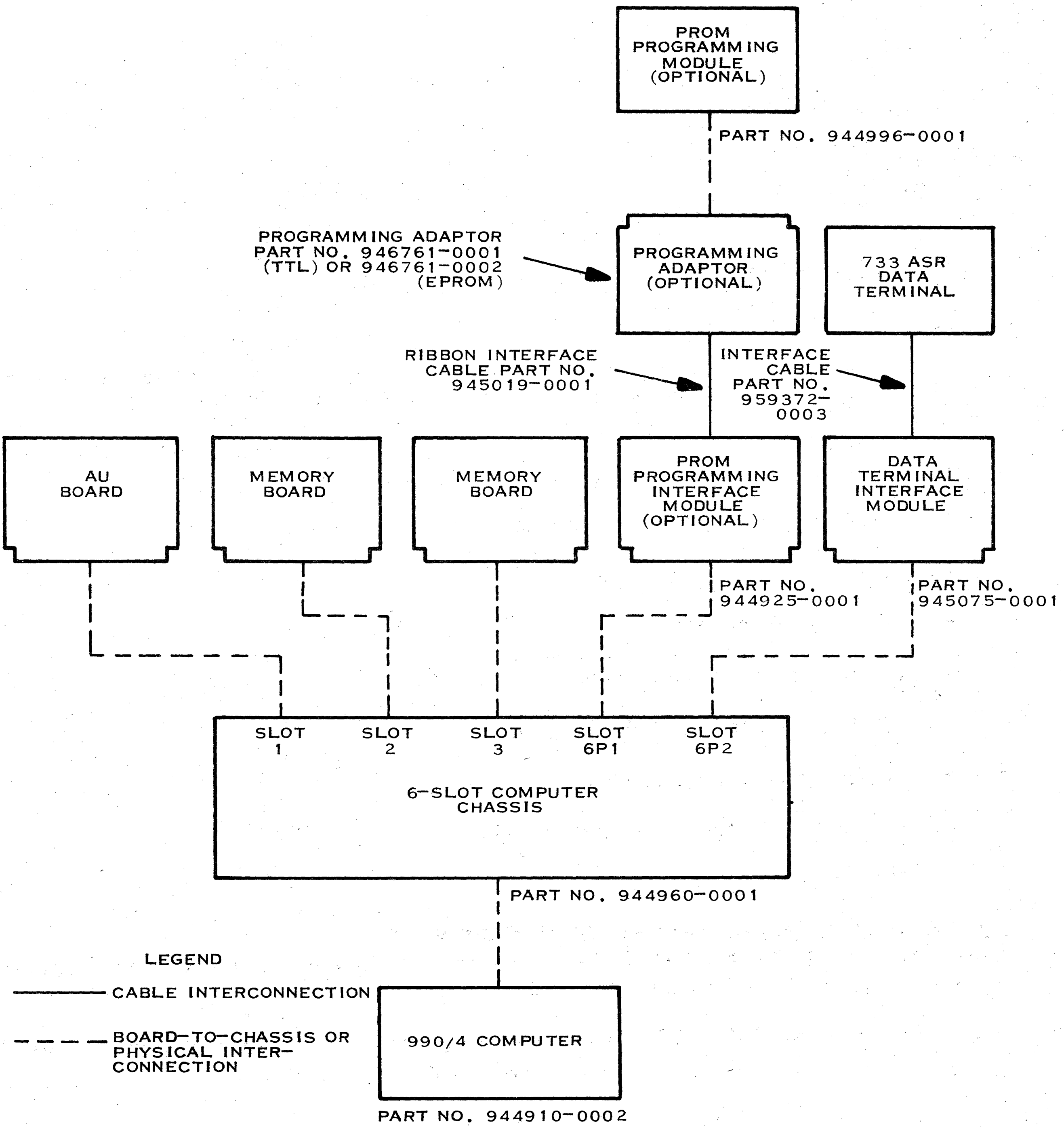
Unpack and install the 990/4 Computer as described in the *Model 990/4 Computer System Hardware Reference Manual,* Manual No. 945251-9701.

Unpack and install the 733 ASR Data Terminal as described in the *Model 990 Computer Model 733 ASR/KSR Data Terminal Installation and Operation,* Manual No. 945259-9701.

Install the interface module for the data terminal and interconnect the computer and data terminal as described in the *Model 990 Computer Model 733 ASR/KSR Data Terminal Installation and Operation,* Manual No. 945259-9701.

A system cabling diagram is shown in figure 2-1.

If the system includes the optional PROM Programming Module, unpack and install it as described in the *Model 990 Computer PROM Programming Module Installation and Operation,* Manual No. 945258-9701. Interconnect the computer and programming module as described in that manual.


PROM PROGRAMMING MODULE (OPTIONAL)
PART NO. 944996-0001
PROGRAMMING ADAPTOR PART NO. 946761-0001 (TTL) OR 946761-0002 (EPROM)
PROGRAMMING ADAPTOR (OPTIONAL)
733 ASR DATA TERMINAL
RIBBON INTERFACE CABLE PART NO. 945019-0001
INTERFACE CABLE PART NO. 959372-0003
AU BOARD
MEMORY BOARD
MEMORY BOARD
PROM PROGRAMMING INTERFACE MODULE (OPTIONAL)
DATA TERMINAL INTERFACE MODULE
PART NO. 944925-0001
PART NO. 945075-0001
SLOT 1
SLOT 2
SLOT 3
SLOT 6P1
SLOT 6P2
6-SLOT COMPUTER CHASSIS
PART NO. 944960-0001
LEGEND
CABLE INTERCONNECTION
BOARD-TO-CHASSIS OR PHYSICAL INTER-CONNECTION
990/4 COMPUTER
PART NO. 944910-0002

(A)133372

Figure 2-1. Cabling Diagram, 990 Prototyping System

2.3 HARDWARE OPERATION

The programmer panel controls and indicators are described in detail in the *Model 990/4 Computer System Hardware Reference Manual,* Manual No. 945251-9701.

The 733 ASR data terminal's controls and indicators and operation of the terminal are described in detail in the *Model 990 Computer Model 733 ASR/KSR Data Terminal Installation and Operation,* Manual No. 945259-9701.

The user should be aware of the following points regarding use of the system software with the data terminal:

- The Prototyping System Software requires that tapes be written in line tape format rather than continuous tape format. In line format, the tape buffer is written to tape when a carriage return is encountered. To place the 733 ASR in line format, set the TAPE FORMAT switch to the LINE position.
- A tape should always be rewound completely:
    - After a cassette tape is installed in a tape transport.
    - After every initialization of power.
- Before removing a cassette tape from a tape transport.
- Before switching off the power to the data terminal.

The PROM Programming Module is a software-controlled module that provides an interface between the 990/4 microcomputer and an external chassis containing power supplies and interchangeable circuitry on an adaptor to program specific types of PROM devices. This module operates as a CRU device.

Detailed information about the PROM Programming module can be found in the *Model 990 Computer PROM Programming Module Installation and Operation,* Manual No 945258-9701.

Software initiates the programming cycle and determines the duty cycle. Software has direct control over the PROM address and the data to be programmed; it has limited control over the width of the pulse used to program the PROMs. The data address and pulse width information are placed into input registers.

Each PROM or family of PROMs has different requirements because of its programming characteristics. The adaptor, a type of interface card, handles these differences. It provides any buffering of the address and data lines and regulates the dc voltages present in the external chassis that are used for control and programming.

2.4 PROTOTYPING SYSTEM SOFTWARE CASSETTE GENERATION

The Prototyping System Software cassette tape consists of 15 files and is a complete object tape for the Prototyping System. The files on this tape are (in order):

1. Text – Description of tape and copying instructions
2. PX9UFL – Upfront loader
3. PX9MTP – Monitor (root segment)

4. PX9MTP – Linking loader overlay
5. PX9MTP – Instruction trace overlay
6. PX9MTP – Absolute dump/absolute load overlay
7. PX9MTP – PROM programmer, part 1
8. PX9MTP – PROM programmer, part 2
9. PX9MTP – BNPF dump overlay
10. PXMTP – HIGH/LOW dump overlay
11. PX9UFL – Upfront loader
12. PX9EDT – Text editor
13. PX9UFL – Upfront loader
14. PX9ASM – One-pass assembler
15. PX9MTP – Monitor (relocatable root segments)

In addition, the system includes the Standard Control Information Cassette.

The user should copy each of the object files to a separate cassette for convenience in using the system. This can be done by copying the master cassette in local mode using the 733 ASR Data Terminal. The upfront loader and the file following it should be copied to the same cassette. PROMPG Part 1 and Part 2 should also be copied on one cassette. The following procedure may be used:

1. Do not rewind this cassette after printing the text of the first file. If the cassette was listed using local mode and continuous start, it will be correctly positioned.
2. Check that the RECORD switch in the bottom row of switches on the upper unit is in the LOCAL position and that the PRINTER switch is in the OFF position. (The PLAYBACK switch should already be set to LOCAL.) The TAPE FORMAT switch should be set to LINE.
3. Insert a cassette in the second drive and ready it.
4. Set the Record Control ON/OFF switch to the ON position.
5. Press the CONT START switch in the Playback Control switch area. The next file should be copied to the record cassette.
6. If the file just copied is the upfront loader or the PROM programmer part 1, repeat step 5 to copy the next file onto the same cassette.
7. Set the Record Control ON/OFF switch to the OFF position. Rewind and remove the record cassette. Set the record enable (write) tab to the record disable position, and label the cassette with the appropriate file name.

8. Repeat steps 3-7 for each of the object files.

9. Rewind and remove the master cassette and store it in a safe place.

Additional information may be found in the *Model 990 Computer Model 733 ASR/KSR Data Terminal Installation and Operation,* Manual No. 945259-9701.

File number 3 on the Prototyping System software master cassette is the PX9MTP monitor in compressed absolute data format. This module must be loaded using the upfront loader. (PX9EDT and PX9ASM must also be loaded with the upfront loader.) The monitor will be loaded in locations 2000_{16} to 4000_{16}. This will place the monitor in the upper 4K words of an 8K word system. If the user has another system configuration, he may wish to load the monitor at a different location. To do this, file 15, the relocatable monitor, must be loaded. By placing a D tag character in the code, followed by a four-digit hexadecimal bias address and end of record, the bias for the relocatable monitor may be specified when the file is being copied from the master cassette.

Example:

```
D8000F

First record of monitor.
```

The monitor will be loaded in locations 8000_{16} to $A000_{16}$. This D tag record may be created in local mode or by using PX9EDT.

Using the monitor residing at location 2000_{16}, an absolute code module of the relocated monitor may be created:

1. Load the absolute code monitor which resides at 2000_{16}.

2. Using this monitor or the programmer panel LOAD switch, load the relocatable monitor at the desired bias.

3. Halt and reset the monitor at 2000_{16}.

4. Load the Absolute Dump/Absolute Load overlay.

5. Copy the upfront loader to the beginning of a tape in LOCAL mode.

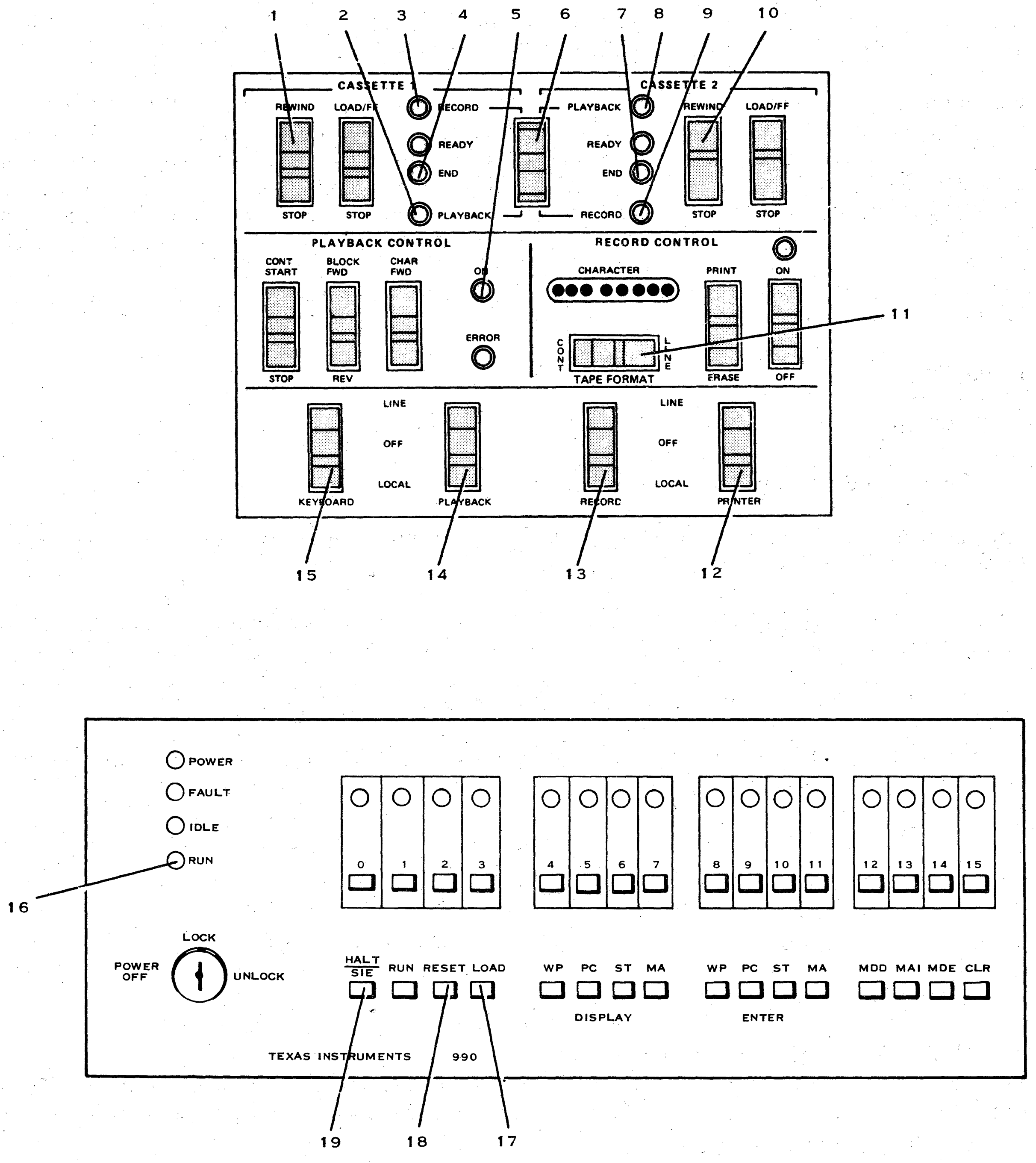
6. Dump the relocated monitor to the tape with an entry point equal to the bias.

2.5 USING THE 733 ASR ROM LOADER

The following paragraphs present a procedure for loading software modules with the 733 ASR ROM loader, describe loading of PX9MTP with the ROM loader, and give some information on loading under PX9MTP control.

2.5.1 LOADING STANDARD 990 OBJECT MODULES. Programs or modules in standard 990 object format may be loaded with the 733 ASR ROM loader by using the following procedure. Refer to figure 2-2; the numbers in parentheses are keyed to the figure.

1. Place the computer in halt mode by pressing the HALT/SIE switch (19) on the programmer panel.



(A)133076

Figure 2-2. Controls and Indicators Used in Loading Procedures

2. Place the 733 ASR data terminal on line. (The device function switches (12 through 15) must be set to the LINE position.)

3. Load and ready the cassette containing the program to be run in either transport drive.

4. Set the TAPE FORMAT switch (11) to LINE.

5. Place the selected cassette in playback mode by setting the PLAYBACK/RECORD switch (6) in the middle of the top row of the data terminal's upper switch panel to the PLAYBACK position for that cassette. The PLAYBACK and RECORD indicator lamps (2, 3, 8 and 9) indicate the mode of each cassette drive.

6. Press the RESET and LOAD switches (18 and 17) on the programmer panel to initiate the load. The Playback Control ON indicator lamp (5) on the data terminal's upper switch panel lights to indicate that data is being transferred.

7. When the load is completed, the loader will transfer control directly to the program if the program contained an entry vector. (An entry vector is a special tag generated by the assembler indicating the starting location for the program. The tag is generated if the user includes the starting address for his program in the END statement.) If there is no entry vector or if an error occurs during loading, the loader returns control to the programmer panel.

8. The cassette should be rewound by pressing the REWIND side of the REWIND/STOP switch (1 or 10) on the upper switch panel of the data terminal and removed to prevent accidental reuse. The tape is finished rewinding when the END indicator lamp (4 or 7) lights.

2.5.2 LOADING COMPRESSED ABSOLUTE FORMAT OBJECT MODULES. Compressed absolute format code may be loaded using the 733 ASR ROM loader by including the upfront loader in front of the compressed absolute format code module. Refer to the Load with Upfront Loader (LU) command in Section III for a further description of the upfront loader.

To load with the upfront loader, follow the procedures described in paragraph 2.5.1. The upfront loader is loaded at the ROM loader default bias address, $A0_{16}$.

After the upfront loader is memory resident, control is passed to it and the compressed absolute load initiated. Once the absolute format module is loaded, control is passed to it if an entry point has been found. If there is no entry point or there is a load error, control is returned to the programmer panel.

If the user wishes to perform a bootstrap load with the upfront loader, but would like the upfront loader at a different point, he may add to the upfront loader object module a D tag (load bias) character as the first record.

2.5.3 LOADING THE MONITOR. To load the monitor, mount the cassette containing the upfront loader and PX9MTP and load it in the manner described in paragraph 2.5.1. A period (.) is printed to indicate that the monitor is loaded and ready to accept commands.

2.6 OPERATING THE MONITOR

When the monitor is loaded by the 733 ASR ROM loader, it will be located at locations 2000_{16}, to 4000_{16} with the entry point at 2000_{16} (assuming an 8K configuration). If a program remains in the execution mode because of an error or is aborted by a programmer panel halt, the monitor will need to be restarted to reenter the command processor mode. To restart the monitor, proceed as follows:

1. Halt the system by pushing the programmer panel HALT/SIE switch. The RUN indicator lamp is extinguished when in halt mode.
2. Clear the data indicator lamps by pressing the CLR switch.
3. Enter 2000_{16} on the data indicator lamps.
4. Press the ENTER PC switch.
5. Press the RESET switch.
6. Press the RUN switch.

At this point, the monitor should respond with a period (.). If the monitor does not respond, repeat steps 1 through 6. If further attempts to restart fail, the monitor may have been destroyed and a reload is necessary.

2.7 ENTERING COMMANDS ON THE TERMINAL KEYBOARD

Commands are entered as a two-character command name and a string of parameters. The command name and each parameter are separated by one or more spaces or a comma. A carriage return will end the record and signal the end of input to the monitor. The RUB OUT key on the keyboard may be used to delete all characters from the present character position to the beginning of the current parameter. CRTL H will delete one character (back to the beginning of the current parameter).

Some keys, such as TAB (CTRL I), the space bar, backspace (CTRL H), ESC and RUB OUT, are interpreted differently depending upon which command processing routine is executing. The special interpretations of these and others are explained in the routines, or programs in which they occur.

The monitor recognizes a number of special control characters which conform to the standard 990 file and data format. Appendix C shows the valid control characters and their functions for keyboard, printer, and cassette I/O as defined in the 990 standard file and data formats.

2.8 INPUT/OUTPUT AND LOGICAL UNIT ASSIGNMENTS

When a program is written, the input and output is device independent and is simply input from or output to a logical unit number. At run time, the user must enter the Assign LUNO (AL) command to assign each LUNO to a physical device if the system default logical unit assignments (described in Section III) are not being used. When the program is run, the monitor takes care of all the device-dependent characteristics required.

2.9 LOADING AND EXECUTING PROGRAMS

The following paragraphs present procedures for loading programs and discuss the user's interface with the software.

2.9.1 LOADING. PX9MTP loads programs using two different object code formats, compressed absolute and standard 990 object format. Any of five commands – LP, OV, PL, LU or LA, all described in Section III – may be used to load programs. The operator interface with PX9MTP is similar for all five types of loads:

1. Place the cassette containing the program or overlay to be loaded on an available cassette transport drive.

2. Set the four switches in the bottom row of the data terminal's upper switch panel (12, 13, 14 and 15, figure 2-2) to the LINE position.

3. Enter the appropriate monitor keyboard command for the type of load being performed followed by the LUNO which has been assigned to the cassette containing the program to be loaded. If no LUNO is entered with the command, the system assumes a default of LUNO 7. If the AL command has not been used to redefine the two cassettes, the system defaults LUNO 7 to CS1 (the left cassette drive) and LUNO 8 to CS2 (the right drive).

4. When the load completes, the system will accept further commands. If an entry vector was specified within the load module, the PC for the user's program is recorded within PX9MTP and may be displayed with the Inspect Registers (IR) command and observed on the programmer panel data indicator lamps. For monitor-controlled I/O, the playback and record modes need not be set since the monitor handles this function.

The standard 990 object code format and the compressed absolute format are described in Section VI.

2.9.2 USER PROGRAM INTERFACE WITH SYSTEM SOFTWARE. Monitor commands are used to load or execute programs in the user area of memory. User programs, the text editor (PX9EDT) and assembler (PX9ASM) are loaded into user memory and executed in free running mode with monitor control or in free running mode. Before executing in either mode, the entry point for programs in the user area must be set in the user's PC. The Inspect Registers (IR) command may be used to determine the starting PC value, and the Modify Registers (MR) command may be used to change it.

User programs may communicate with the resident software system by means of the Extended Operation (XOP) instruction. This is also true of two user area system programs: PX9ASM and PX9EDT.

XOP 15 is used to call PX9MTP to perform I/O and data conversion services as defined in Section III. This XOP vector is initialized by the monitor whenever a Load Program (LP), Load Program in Compressed Absolute Format (LA), Load Overlay (OV), or Load Program in Compressed Absolute Format with Upfront Loader (LU) command is issued. The user program may overlay this vector and supply its own service routine.

2.9.3 EXECUTING A USER PROGRAM. A program may be executed by issuing either an RU or EX command. If the RU command is used, the monitor will control the execution and various run-time debug aids may be utilized. The monitor executes programs using either the SIE feature (see paragraph 2.10) or an interpretive trace (see Section III). A program may be halted and control returned to the command processor at any time by pressing the ESC key on the data terminal keyboard. If an EX command is issued, the program will be executed without monitor control. The program may be halted only by pressing the programmer panel HALT switch and restarting the monitor.

After a program has been executed in either mode, control may be returned to the monitor command processor by an End of Programmer supervisor call (Section III) or by branching to the beginning of the monitor.

2.10 INTERRUPTS AND SINGLE INSTRUCTION EXECUTION

The following paragraphs discuss the interrupt scheme and the role of interrupts in single instruction execution, which is a debugging aid. Single instruction execution is briefly explained.

2.10.1 INTERRUPTS. The 990/4 Computer supports eight levels of interrupts. Any device which is capable of interrupting the 990 is assigned (in the hardware) to an interrupt level. The 990 compares the level of any interrupt with a program-determined value called a mask. If the interrupt is at a higher level (lower numeric value) than the mask, the interrupt is allowed; otherwise, the interrupt is not permitted. For more detailed information about interrupts, refer to the *Model 990/4 Computer System Hardware Reference Manual,* Manual No. 945251-9701.

The highest level (level 0) is used to indicate that power has just been applied to the 990, either initially or following a power failure, and/or that a special interrupt for the programmer panel is active. The level 0 interrupt differs from the other interrupts because it cannot be masked by the program.

The level 0 interrupt is generated whenever one or more of these conditions occurs:

- Monitor-initiated single instruction execution (SIE).
- The operator presses the HALT/SIE pushbutton on the programmer panel.
- A program executes an LREX (Load ROM and Execute) assembly language machine instruction.
- A power-up condition occurs.

The level 0 interrupt trap vector must be connected by jumper cable to location $FFFC_{16}$.

2.10.2 SINGLE INSTRUCTION EXECUTION. It is often convenient for debugging purposes to execute a program one instruction at a time. This feature is provided on the programmer panel and also by PX9MTP. The hardware supports this feature in the following manner:

1. The programmer panel or PX9MTP initiates execution of a single user program instruction.
2. In the process of executing the user program instruction, three distinct actions occur. First, the programmer panel or PX9MTP causes an RTWP (Return with Workspace Pointer) assembly language machine instruction to be executed. This returns control to the process or to the user.
3. Second, the user program instruction is executed.
4. In the third action, the 990 Computer generates a level 0 interrupt which transfers control back to the programmer panel. If the SIE was initiated by PX9MTP, the programmer panel will transfer control back to PX9MTP.

This sequence of actions is repeated for each user program instruction, except under certain conditions. The user must be aware of these exceptions:

- If the instruction was a BLWP (Branch and Load Workspace Pointer) or XOP (Extended Operation), the processor executes an additional instruction before any interrupts occur. (This feature is necessary to support reentrant subroutines using BLWP or XOP instructions for linkage.)

- If there is a lower level interrupt pending, that interrupt is honored instead of the next "user instruction". Therefore, when the programmer panel regains control, the return PC points into the interrupt subroutine rather than the original user program.

2.11 WRITE PROTECT

The 990 Prototyping System is equipped with a write protect feature which permits or prohibits writing to a selected area of memory. The write protect logic basically consists of a seven-bit Upper and Lower Bound register and a Protect/Permit control bit (figure 2-3). The loading of this register and control bit may be accomplished with the Set Write Protect Region (SP) and Clear Write Protect Region (CP) commands (Section III), or by normal CRU communications.

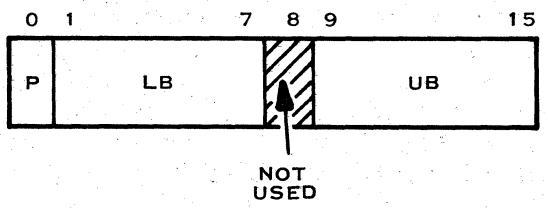
2.11.1 SETTING A WRITE PROTECT REGION. To set a write protect region, the lower and upper bounds must be output to CRU base address $1FA0_{16}$. The most significant bit (bit 0) is the Protect/Permit bit. Bit 0, when set to 1, indicates write permit, and, when set to 0, indicates write protect. To specify the protect region, memory is divided into 256-word blocks. The lower and upper bounds are each seven bits long and serve as an index into the memory addresses to specify which contiguous 256-word block of memory is to be protected. For example, the lower bound of the protect region equal to 2000_{16} would be represented in the Protect register as 10_{16}. The memory block beginning at location 2000_{16} is the sixteenth 256-word (512-byte) memory block. A bound is calculated by dividing the starting address of the memory block by 200_{16} (512_{10}). In this example, 2000_{16} divided by 200_{16} is equal to 10_{16}. The upper bound is not included in the protect region. When outputting to the CRU Protect register to specify the protect bounds, a Load CRU (LDCR) instruction with a count of 16 must be used to set all 16 bits because the Protect register works like a shift register. To protect the memory range 2000_{16} to 4000_{16}, the lower bound is set equal to 10_{16}, the upper bound is set equal to 20_{16}, and the Protect bit is set to 0. Therefore, the Protect register is set to 1020_{16} by outputting these fields to the CRU in the format specified in figure 2-3.

2.11.2 PROTECT VIOLATION FLAG. When an attempt is made to write into a memory location within the protected region, the Protect Violation flag is set to $FFFF_{16}$. This flag, which is 0 normally, can be sensed by reading any of the 16 CRU bits at base $1FA0_{16}$· If this protected region is within the TMS9900 on-board RAM, the write will not be inhibited. If this protect region is on the expansion memory card, the write will be inhibited. Attempts to write are flagged with an error message.

The Protect Violation flag may be cleared in two different ways:

1. I/O RESET (RSET) – This machine instruction clears the violation flag and sets bit 0 of the Protect register to 1 (not protected).

2. Output a 1 to any or all of the 16 bits of the Protect register.



BIT FIELDS

P	PROTECT/PERMIT BIT 0–PROTECT 1–PERMIT
LB	LOWER BOUND
UB	UPPER BOUND

NOTES

THE CRU OUTPUT DATA FORMAT IS THE SAME AS THE FORMAT OF DATA IN MEMORY BEFORE IN LDCR INSTRUCTION IS EXECUTED.

BITS 1 AND 9 ARE THE MOST SIGNIFICANT BITS, AND BITS 7 AND 15 ARE THE LEAST SIGNIFICANT BITS OF THE LB AND UB FIELDS.

(A)133373

Figure 2-3. CRU Output Data Format

When running under monitor control with an RU command, the Protect Violation flag is checked after each user instruction is executed. The monitor also checks for a write protect error when control is returned to the command string processor. This enables the user to detect violation errors incurred during monitor commands such as Modify Memory (MM) and the program loading commands (LP, OV, PL, LL, LU and LA). The monitor prints the error message

```
MX07
```

if a write protect violation occurs. The violation flag is cleared, the protect register restored and the user program halted.

The Protect Violation flag is not checked when executing a user program with the EX command.

When the Protect Violation flag is set, another signal is generated which may be wired to an interrupt level. If the user chooses to do this, an interrupt routine must be provided by the user.

If the program is being executed with the EX command and the interrupt has not been wired in, there is no automatic checking for a protect violation after each instruction.

When the monitor is restarted, the Protect register and Protect Violation flag are initialized. An I/O Reset is performed which clears the Protect Violation flag and sets the Protect register to $FFFF_{16}$.

2.11.3 PROTECTING THE MONITOR. In debugging a user program, the monitor is often destroyed by an incorrect instruction. This may be avoided in most cases by write-protecting the monitor.

The monitor has been constructed so that all of the data areas occur near the end of the monitor. The first 1400_{16} bytes of the monitor may be included within the protect region.

SECTION III

DEBUG MONITOR

3.1 INTRODUCTION

This section discusses the purpose and capabilities of the debug monitor (PX9MTP), explains how to debug under monitor control, and gives detailed descriptions of the monitor keyboard commands available to the user. The following topics are covered:

- A general description of the monitor, including its functions and capabilities. Communication with the monitor. Debugging features. Input/output operations and logical device assignments. Methods for loading programs using PX9MTP, and the different types of loads.

- Debug functions provided by the monitor. Capabilities provided by the different types of debug commands. User-specified parameters that serve as interpretation aids. The two debugging modes: single instruction execution and instruction trace. A comparison of the merits and limitations of the debugging modes.

- Discussion of the use of monitor keyboard commands. The three types of commands: system control, debug, and PROM/ROM process control commands. Mnemonic codes and command parameters. The conditions under which commands may be entered. Processing of commands by the monitor. Error messages. Notational conventions used in the command syntax definitions.

- Descriptions of the commands, including a brief explanation of their purpose, their syntax and parameters, how they function, error messages, application notes if applicable, and examples of how the commands are used. The monitor keyboard commands are listed in table 3-1.

- Supervisor calls. Their purpose. The differences between I/O and non-I/O supervisor calls. Supervisor call formats and examples.

- Debugging techniques. An explanation of preventive, exposure and remedial techniques. General techniques for any debugging situation. Specific techniques for debugging under PX9MTP, including how to plan a debugging session, use of breakpoints, and time-saving and simulation techniques. Patching assembly language code into an existing program.

3.2 GENERAL DESCRIPTION

PX9MTP is a memory-resident system executive that responds interactively to user input from the 733 ASR data terminal keyboard, provides extensive program debug features, and provides a supervisor call interface to user programs.

The operator communicates with the monitor by entering commands through the keyboard of the 733 ASR data terminal. These commands may assign logical unit numbers (LUNOs) to devices for I/O operations, and instruct the system to load and execute specific programs. These supported programs may interface with the monitor through supervisor calls (paragraph 3.5).

Table 3-1. Monitor Keyboard Commands

Mnemonic	Description	Paragraph
AL	Assign LUNO	3.4.2
LP	Load Program	3.4.3
OV	Load Overlay	3.4.4
PL	Load PROM Programmer	3.4.5
LL	Link and Load Program	3.4.6
DP	Dump in Absolute Format	3.4.7
LU	Load Program in Compressed Absolute Format with Upfront Loader	3.4.8
LA	Load Program in Compressed Absolute Format	3.4.9
EX	Execute User Program Directly	3.4.10
RU	Execute User Program under SIE or Trace	3.4.11
MM	Modify Memory	3.4.12
IM	Inspect Memory	3.4.13
MR	Modify Registers	3.4.14
IR	Inspect Registers	3.4.15
MW	Modify Workspace Registers	3.4.16
IW	Inspect Workspace Registers	3.4.17
MC	Modify CRU Register	3.4.18
IC	Inspect CRU Input Lines	3.4.19
SS	Set Snapshot	3.4.20
IS	Inspect Snapshot	3.4.21
CS	Clear Snapshot	3.4.22
SB	Set Breakpoint	3.4.23
CB	Clear Breakpoint	3.4.24
ST	Set Trace Definition	3.4.25
SR	Set Trace Region	3.4.26
CR	Clear Trace Region	3.4.27
FB	Find Byte	3.4.28
FW	Find Word	3.4.29
HA	Hexadecimal Arithmetic	3.4.30
SP	Set Write Protect Region	3.4.31
CP	Clear Write Protect Region	3.4.32

Supervisor call communication with the monitor is accomplished with extended operation 15 (XOP 15) and a parameter block giving the specific details of the request. A supervisor call can be used to request system functions such as:

- Convert decimal numbers in ASCII format to binary values, and binary values to ASCII format decimal numbers.
- Convert hexadecimal numbers in ASCII format to binary values, and binary values to ASCII format hexadecimal numbers.
- Provide I/O operations that are compatible with those for the DX10 operating system.
- Terminate the current program.

PX9MTP also provides debugging aids for stand-alone programs. The program debug functions of the monitor:

- Give the user interactive control over his programs.
- Are independent of the user program.
- Are compatible (where possible) with the corresponding software features of the 990/10 Program Development System.

3.2.1 INPUT/OUTPUT OPERATIONS. PX9MTP I/O operations provided by the supervisor calls are device independent, as described in Section II. The 733 ASR data terminal appears to the monitor as three separate logical devices – cassette unit 1, cassette unit 2, and the printer-keyboard, as described in Section I. The Assign LUNO (AL) monitor keyboard command is used to assign logical unit numbers. The monitor supports the following I/O operations to the 733 ASR data terminal: open file, read ASCII data, write ASCII data, and write end-of-file.

3.2.2 METHODS FOR LOADING PROGRAMS. PX9MTP supports three distinct methods for loading programs into memory: a relocating loader, a relocating and linking loader, and a compressed absolute format loader. The two relocating load operations called by the Load Program (LP) and Link and Load Program (LL) commands handle programs in object format produced by any of the 990 assemblers. Relocation allows a program to be loaded into any available memory area to make efficient use of memory space.

The linking process in the LL command integrates object modules that have been assembled separately into a single, contiguous program. This type of load operation accepts one or more object modules of a program and loads them into memory at addresses specified in the program modules.

The third type of loading operation uses a condensed data format, generated by the Dump in Absolute Format (DP) command, that can be loaded much faster than the equivalent object module format of the program. When a program has been completely debugged and is to be stored for future use, it can be copied to cassette using the DP command to create the condensed data format. Then, by calling the Load Program in Compressed Absolute Format (LA) command, the program will be loaded into the same memory area that it was stored in when originally dumped. This condensed data format module may also be loaded with the Load Program in Compressed Absolute Format with Upfront Loader (LU) command. Refer to paragraphs 3.4.3 through 3.4.9 for detailed command descriptions.

3.3 DEBUG FUNCTIONS

The Program Debug function of the monitor allows the user to test, validate, and remove errors from a program under development. Debugging is accomplished by entering commands for various debug functions from the terminal keyboard. The commands are decoded and processed by the monitor. The debug facilities operate entirely from programs and data stored within the 4096-word memory area reserved for PX9MTP.

The available debug commands may be classified into the following groups.

- *Set commands.* These commands allow the user to define up to four of each of the following aids: program counter breakpoints, formatted snapshots, trace regions, and trace formats.

- *Clear commands.* These commands allow the user to negate the effect of a previous set command.

- *Inspect command.* These commands allow the user to display the contents of AU registers, workspace registers, memory regions, and CRU lines. These commands are also used to force snapshots.

- *Modify commands.* These commands allow the user to examine and optionally modify memory, workspace registers, AU registers, and CRU lines (by inspecting the input and modifying the output).

- *Miscellaneous commands.* These commands include functions such as word and byte memory searches, and hexadecimal arithmetic with automatic decimal conversion.

In debugging a program, the user may print out data on the terminal for examination, modify data, specify program elements (parameters whose values are determined by the user) for interpreting the progress of his program, set and clear these elements, search for specific bit patterns in bytes and words, and perform arithmetic calculations with hexadecimal numbers. These actions may be performed on memory, registers, and CRU input and output lines. They may also be performed on the specifiable program elements: breakpoints, snapshots, and trace regions. They are defined as follows:

- *Breakpoint* – A point during the execution of a program at which control is returned to the debug monitor to allow the user to examine the progress of his program or enter any of the debug commands..

- *Snapshot* – A printed display of the contents of contiguous workspace registers plus the contents of an area in memory as defined by the operator. A snapshot may be printed automatically at a breakpoint.

- *Trace region* – An area of the program about which information concerning the execution of an instruction is output on the printer. This information may be printed following the execution of each instruction, following each branch, or following each change in the contents of a data word.

3.3.1 DEBUGGING MODES. When debugging with the monitor, the user may use either the single instruction execution (SIE) mode or the instruction trace mode by issuing the Execute User Program under SIE or Trace (RU) command. In these modes, after each instruction is executed, the monitor checks whether a breakpoint has been reached. If a breakpoint has not been reached or the run count is not depleted, the monitor continues executing instructions in the same manner.

When running in SIE mode, the monitor uses the hardware-controlled SIE feature described in Section II to execute each user instruction.

When in the instruction trace mode, the user's program is executed by a software interpreter which decodes the user's instructions and then executes the instructions. This allows the system to check and display detailed information on the execution of an instruction. The software interpreter is contained in the instruction trace overlay which must be loaded before the instruction trace feature can be used.

The instruction trace feature allows the user to monitor the contents of internal data sequences, alter these data sequences, and analyze the ongoing progress of an executing program. The user can also specify breakpoints and snapshots for interpreting the progress of his program.

Under instruction trace, all extended operations (XOPs) and interrupts are executed directly by the hardware, not under control of the software.

3.3.2 COMPARISON OF DEBUGGING MODES. SIE is considerably faster than instruction trace. In SIE mode, interrupts and XOPs are executed one instruction at a time under control of the software. XOPs and interrupts under instruction trace, on the other hand, are executed directly by the hardware. SIE is always memory resident, while instruction trace is contained in a separate overlay.

If speed, XOPs, interrupts, or loading the overlay is not a primary consideration, it is suggested that instruction trace be used as the normal mode of execution. If no trace printout is desired, a null trace may be set. (Refer to the description of the Set Trace Definition (ST) command, paragraph 3.4.25.)

3.3.3 SUMMARY. The program debug facilities of PX9MTP are easily used by novice programmers, yet have the power needed by the sophisticated programmer to fully test his programs. The novice needs to learn only four types of operations (Set, Clear, Inspect and Modify) to be applied to any of several debug or machine resources (memory, breakpoints, etc.). An experienced user will learn to associate snapshots and trace formats (by number) with specific breakpoints and trace regions, respectively. Trace formats and snapshots are predefined for the novice but may be modified if desired.

3.4 KEYBOARD COMMANDS

The following paragraphs present background information on the keyboard commands. They describe the components of a command, their significance, and their general characteristics. The individual commands are then described in detail.

3.4.1 GENERAL. As an aid to the use and understanding of the commands, the different types of commands are discussed, and command codes and parameters are explained. These paragraphs also describe entry of commands on the terminal keyboard, explain how commands are processed, and list error messages that may be returned by the system.

3.4.1.1 Types of Commands. The keyboard commands may be classified into three types: system control commands, debug commands, and PROM/ROM process control commands. System control commands include those needed to get the program loaded and running or to initiate a program dump. The debug commands are those entered by the user during execution of a program under development. PROM/ROM process control commands are those needed to program PROMs and produce cassettes for manufacturing ROMs. The commands may also be classified according to the way they are handled by the monitor. Some of the commands are memory-resident; the others reside on tape cassette and are loaded into the transient area of memory from cassette when they are needed. These cassette-resident commands are the overlay commands, and include those less frequently used.

3.4.1.2 System Control Command Codes. System Control Commands are identified by two-letter mnemonic codes, and may be followed by one or more parameters. This group of commands includes the following:

- Assign LUNO (AL)
- Load Program (LP)

- Load Overlay (OV)
- Link and Load Program (LL)
- Dump in Absolute Format (DP)
- Load Program in Compressed Absolute Format with Upfront Loader (LU)
- Load Program in Compressed Absolute Format (LA)
- Load PROM Programmer (PL)
- Execute User Program Directly (EX)
- Execute User Program under SIE or Trace (RU)

The individual system control commands are described in paragraphs 3.4.2 through 3.4.11.

3.4.1.3 Debug Command Codes. Debug commands are identified by a two-letter mnemonic code. The first letter represents the operation performed, and the second letter represents the program debug or machine element on which the command operates. The operation performed may be one of the following:

First Letter of Command	Operation
I	The Inspect operation displays on the printer whatever data or debug element is requested, in the specified size or amount.
M	The Modify operation displays a requested quantity, such as the contents and the register number of a workspace register, and accepts an input which may change the value. This operation automatically increments and displays the next item of the element being modified. The Modify commands operate on memory, workspace registers, machine registers, and the CRU.
S	The Set operation is used in commands to define program debug elements such as breakpoints, snapshots and traces.
C	The Clear operation is used to clear or reinitialize breakpoints, snapshots, and trace regions.
F	The Find operation searches for bit patterns in bytes or words. The patterns are characterized by mask and value. If the specified bits in the mask are the same as the corresponding bits in the value, a pattern match exists.
H	The Hexadecimal operation calculates the sum and difference of two hexadecimal numbers, and prints the results in both hexadecimal and decimal format.

The second letter of a debug command represents the element on which an operation is performed, and may be one of the following:

Second Letter of Command	Element
M	The Memory element represents any RAM or ROM in the hardware system configuration. If nonexistent memory is specified, an undefined bit pattern is returned.
W	The Workspace element represents the user program's current workspace registers, registers 0 through 15. In a Find Word command, W represents *word*.
R	The Registers element represents the user's program counter register, workspace pointer register and status register. In a Set Trace Region (SR) or Clear Trace Region (CR) command, R represents region. A region is a memory area where there will occur a trace of statements executed when running under Instruction Trace. Associated with each region is its index (a number from 0 to 3), trace type, mode of execution (single step or continuous run), and, optionally, variables to be traced.
C	The CRU element represents the Communications Register Unit of the 990 Computer. The data on the CRU input lines and the input line numbers may be displayed, and the data on the CRU output lines may be modified.
B	The Breakpoint element represents the four program counter (PC) breakpoints. Associated with each breakpoint is its index (a number from 0 to 3), a program counter value, a reference counter (optional) and a snapshot index (optional). The user's program counter, workspace pointer and status register are automatically printed along with the breakpoint index number. If a snapshot was associated with the breakpoint at definition time, then the snapshot indicated by the snapshot index is also printed. Breakpoints are detected before instruction execution. In a Find Byte (FB) command, B represents *byte*.
S	The Snapshot element represents a four-element vector of program displays. Each display is characterized by a range of workspace registers and a range of memory. Whenever a snapshot is invoked, the register and memory ranges are dumped to the printer.
T	The Trace element is a four-element vector of trace types. Associated with each element is an index and a string of characters indicating the type of trace. The Set Trace Definition (ST) command modifies an existing trace type of the same index. When the monitor is loaded, each element is assigned a default trace type. (For example, type 1 is PIWSEADEA: program counter, instruction and format, workspace pointer changes, source and destination, effective addresses and their contents after execution.) The ST command is available only if the Instruction Trace overlay is in memory.
P	The Protect element represents a write-protected region of memory.

The individual debug commands are described in paragraph 3.4.12 through 3.4.32.

The individual debug commands are described in paragraph 3.4.12 through 3.4.32.

3.4.1.4 PROM/ROM Process Control Command Codes. PROM/ROM Process Control commands are identified by two-letter mnemonic codes and may be followed by one or more parameters. This group of commands include:

- PROM Programmer Standard (PS)
- PROM Programmer (PP)
- Perform BNPF Operation (DB)
- Perform HIGH/LOW Operation (HL)

The individual process control commands are described in Section VII, VIII and IX.

3.4.1.5 Command Parameters. A keyboard command mnemonic code may be followed by one or more parameters. The list of parameters is separated from the command code by one or more blanks or a comma.

The parameters in the list are delimited by commas or by strings of one or more blanks. Each parameter may be a hexadecimal number of one to four hexadecimal digits or a string of alphanumeric characters.

3.4.1.6 Entry of Commands. Keyboard commands may be entered whenever the software system is in command mode. In this mode, a period (.) is printed as the first character on a new line. Depending on the command executed, the software system may or may not return to command mode. The monitor requests another command by printing another period (.) at the beginning of a line. No program executing under the monitor uses a period in this manner to request user input.

3.4.1.7 Command String Processor. The command string processor parses command input strings, given command definition tables. It validates parameter values and converts those representing hexadecimal numbers to binary. It also indicates the existence and position of all null parameters.

The command string processor can parse up to eight parameters. It passes control to the appropriate command processor after it has recognized a syntactically correct command.

3.4.1.8 Processing of Commands. It is helpful to the user to understand how the commands are executed. The command processor handles commands in the following ways.

- A command string is aborted and data is cleared from the buffers on any error.
- The command is checked against a predefined list of acceptable commands.
- The command and any parameters are validated immediately after the appropriate terminator is processed for that parameter (or command). If an error is detected, the entire input is discarded.

- Use of the backspace (CTRL H) on the terminal keyboard deletes single characters from the current parameter or command mnemonic only. Once a parameter has been validated, it may not be changed.
- Use of the delete (RUB OUT) key on the terminal keyboard removes only the current parameter, not the entire command.

The following application notes apply to the use of commands:

- The user must realize that the command input is being checked as it is being entered. This helps prevent entering a long command with an error. Make sure that a parameter is correct before entering the terminator for it.
- The user may abort the current command by pressing the escape (ESC) key on the terminal keyboard.

3.4.1.9 Error Messages. Following is a list of errors the user may encounter when entering keyboard commands and the meanings of the error codes.

Error Code	Meaning
MP00	Invalid parameter entered, invalid hexadecimal number entered, or maximum parameter list length exceeded.
MS01	Invalid command. The first two characters do not match any known command.
MX03	Overlay resident command not in memory. The command must be loaded into the transient area before it can be executed.

A complete list of the error codes appears in Appendix E.

3.4.1.10 Notational Conventions. The notational conventions used in the syntax definitions of the keyboard commands are as follows:

< > Item to be supplied by the user. The term shown within angle brackets is a generic term.

[ ] Optional item – may be included or left out, at the user's discretion. Items not enclosed in brackets are required.

{ } Choice to be made from two or more items, one of which must be included.

Items in capital letters in the syntax definition are entered into the command statement exactly as shown.

The fields in the command (the command mnemonic and the parameters) are separated by either commas or strings of one or more blanks. This choice is shown symbolically as:

$$\left\{\begin{matrix} , \\ \not{b}\ldots \end{matrix}\right\}$$

When one or more parameters are omitted, two or more field separators may occur in sequence. The user must be sure that he includes the correct number of separators in a sequence; he should be aware of how they are interpreted by the computer. Two strings of blanks run together will be read as a single long string of blanks. A comma preceded or followed by a blank will be read as two separators in sequence. It is suggested, therefore, that commas (without preceding or following blanks) be used to set off omitted parameters.

In the examples of command statements, user-supplied data is underlined to distinguish it from data printed by the monitor. The carriage returns that terminate command statements are not shown in the examples.

3.4.2 **ASSIGN LUNO (AL).** The Assign LUNO command is used to establish the I/O devices that will perform I/O under PX9MTP.

Syntax definition:

AL $\left\{\begin{array}{l} , \\ \text{ƀ}\ldots \end{array}\right\}$ <luno> $\left\{\begin{array}{l} , \\ \text{ƀ}\ldots \end{array}\right\}$ <device>

The command is terminated by a carriage return.

Parameters:

luno Logical unit number – number associated with the I/O device.

device Character string which is the name of a device.

The acceptable device names are as follows:

LOG 733 data terminal keyboard and printer.

DUM Dummy device. Input from DUM returns an end-of-file; output to DUM is discarded.

CS1 Left cassette drive on the 733 data terminal.

CS2 Right cassette drive on the 733 data terminal.

Default LUNO assignments: If the AL command is not entered, a set of default values is used for the PX9MTP LUNO assignments. These default LUNO assignments are:

LUNO (Hexadecimal)	**Device**
0	LOG (cannot be changed)
1-5	DUM
6	LOG
7	CS1
8	CS2
9-F	DUM

Although the AL command may be omitted and default assignments used for PX9MTP LUNOs, neither parameter may be omitted if the AL command is used.

Error message:

MX02 Missing required parameter, invalid device name, or invalid LUNO. Re-enter the command.

Application note: The AL command may be needed when using the assembler, text editor, standard loader, linking relocating loader, upfront loader, absolute loader, and absolute dump facilities. The user should refer to the documentation for the appropriate software component to determine the LUNOs used by that component.

Examples:

```
.AL 2 CS1
.AL,1,DUM
```

The first example assigns LUNO 2 to cassette CS1. The second example assigns LUNO 1 to the dummy device. Both statements are terminated by a carriage return.

3.4.3 **LOAD PROGRAM (LP).** The Load Program command initiates a program load with the standard loader.

Syntax definition:

$$\mathrm{LP}\ \left[\begin{Bmatrix} , \\ \text{ƀ}\ldots \end{Bmatrix}\ \left[\langle\text{luno}\rangle\right]\left[\ \begin{Bmatrix} , \\ \text{ƀ}\ldots \end{Bmatrix}\ \langle\text{bias}\rangle\right]\right]$$

The command is terminated by a carriage return.

Parameters:

luno Logical unit number of the input device.

bias Base address of the relocatable object code.

Parameter default values:

If the logical unit number is not specified, a value of 7 is used. Unless reassigned by an Assign LUNO (AL) command, LUNO 7 is assigned to cassette CS1.

If the bias address is not specified, a value of $A0_{16}$ is used. The bias may be overridden by the appearance of a D tag character in the object code.

Description: The 990 standard object code loader resides as a firmware program in a 256-word ROM. When the software system is under control of PX9MTP, a program is loaded by a call to the ROM loader. After the program is loaded, the program entry point, if it exists, is placed in the user PC; then control is returned to the monitor. The user may execute or debug his program by issuing an EX or RU command to the monitor.

Error messages:

LD00 Invalid tag or I/O error. Re-enter the command. If the error persists, the tape may be bad or not readied.

LD01 Invalid LUNO. Re-enter the command.

Examples:

```
.LP 7,1000
.LP
.LP,,1000
```

The first example has the load LUNO and bias address supplied. The second example loads from a default LUNO of 7 at a bias address of $A0_{16}$. The third example loads from a default LUNO of 7 at the bias address supplied (1000_{16}).

3.4.4 LOAD OVERLAY (OV). The Load Overlay command is used to load an overlay into the monitor transient area.

Syntax definition:

$$\text{OV}\ \left[\begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix} \langle\text{luno}\rangle\right]$$

The command is terminated by a carriage return.

Parameter:

luno Logical unit number of the input device.

Parameter default value: If the logical unit number is not specified, a value of 7, the logical unit number normally assigned to tape cassette CS1, is used.

Description. The monitor has an 850 word transient area reserved for overlays. Overlays consist of one or more command service routines. The following commands are overlays:

Overlay Module	Function	Command Mnemonic
1	Dump in Absolute Format	DP
1	Load Program in Compressed Absolute Format	LA
2	Link and Load Program	LL
3	Set Trace Definition	ST
3	Set Trace Region	SR
4	PROM Programmer Standard	PS
4	PROM Programmer	PP
5	Perform BNPF Operation	DB
6	Perform HIGH/LOW Operation	HL

Overlay modules 1, 3, and 4 contain two commands each. The commands resident in the overlay are printed when the overlay is loaded. Overlay module 4, which is a special overlay (see the description of the PL command in paragraph 3.4.5), must be loaded with the PL command.

The standard loader is called to load the overlay. If a checksum, tag or I/O error occurs during loading, control is returned to the monitor and the error message LD00 printed. After the service routine is loaded, the command is activated. All commands residing in the transient area prior to the overlay are disabled. If an attempt is made to execute a command that normally resides in the overlay but is not presently there, an error message of MX03 is printed.

Error messages:

LD00 Invalid checksum, tag or I/O error occurred during loading.

LD01 Invalid load LUNO.

Example:

```
.OV 8
DP
LA
.
```

The overlay containing the DP and LA commands is loaded from the device assigned to LUNO 8.

3.4.5 LOAD PROM PROGRAMMER (PL). The Load PROM Programmer command is used to load the PROM Programmer software module into memory.

Syntax definition:

PL {ƀ'...} [<luno>[{ƀ'...} <bias>]]

Parameters:

luno Logical unit number of the cassette drive on which the PROM Programmer (PROMPG) is mounted.

bias Load addresses for the extended PROMPG program.

Parameter default values:

If luno is not specified, a value of 7 (cassette CS1) is used.

If bias is not specified, the supplied value is $1C80_{16}$ (the address immediately following the end of user memory minus 380_{16} to cover the length of the extended PROMPG program).

Description: PROMPG consists of an overlay module and a memory extension module. The overlay module is loaded into the monitor transient area, and the memory extension module is loaded into the highest numbered address locations of user memory.

When the PL command is issued, the overlay will be loaded and the following printed:

```
PP
PS
```

The memory extension will then be loaded into user memory at the specified bias address.

If the user attempts to enter a command before the memory extension module has been loaded, error message LD00 will be printed. This may occur even though the overlay module load has been indicated by a printout.

Error messages:

LD00	Invalid tag or I/O error. Reenter the command. If the error persists, the tape may be bad or not readied.
LD01	Invalid LUNO. Reenter the command.

Examples:

```
.PL 8
PP
PS
.
```

The PROM programmer module is loaded from the device assigned to LUNO 8. After the overlay resident module is loaded, PP and PS are printed. The memory extension module is then loaded into the top of user memory.

3.4.6 LINK AND LOAD PROGRAM (LL). The Link and Load Program (LL) command starts up the relocating linking loader, PX9LAL. PX9LAL must be loaded into the transient area with the OV command before executing the LL command.

Syntax definition:

```
LL
```

The command is terminated by a carriage return.

Description: PX9LAL loads program modules into the memory of the Model 990 Computer and performs address modification for relocatable code. PX9LAL also performs the linking defined in the program modules and prints a load map. After all modules have been loaded, control returns to the monitor. Details of PX9LAL operation are contained in the next paragraph.

Example:

```
.LL
```

After the user has entered the command, a series of questions is printed. The answers provide the information needed to complete the linking and loading.

3.4.6.1 Description of PX9LAL Operation. PX9LAL loads the first object module supplied. Subsequent object modules will be loaded only if the first six characters of the Program Identifier (IDT) assembler directive character string of the module match an unsatisfied external reference included in a previously loaded module. This makes it necessary for the user to identify the modules desired since modules not referenced are not loaded. This also allows the user to maintain a library of program modules on cassette.

A program identifier (IDT) assembler directive allows a program name of up to eight characters, but PX9LAL recognizes and prints only the first six. Therefore, the first six characters of all IDT program names must be unique.

End-of-Module and End-of-File Records. A module is terminated by an end-of-module separator record, which is denoted by a colon as the first character of the record. The end-of-module record is generated by the assembler when an "END" statement is encountered. The end-of-file record is generated by the assembler when an end-of-file record is encountered by the assembler on the source input. This enables the user to batch-assemble source and batch-load object modules. The loader will continue loading modules from one cassette until an end-of-file record is encountered.

PX9LAL Symbol Table. The symbol table is built in the user area. The length of the symbol table is determined by the number of symbols externally defined or referenced in the program loaded by PX9LAL. Ten bytes of memory are required for a symbol table entry for each symbol. PX9LAL builds the symbol table toward the low-order addresses in memory, beginning at the top of user memory.

User Program Load Addresses. PX9LAL loads user programs into the user memory address space. The user program may not be loaded in a memory area with a higher address than the current lowest address of the symbol table. If the user attempts this, an error message will be printed.

As described in Section VI, the object code may contain a load point preceded by a tag character of D. The tag character and the associated load point must precede the other tag characters and fields of the program module. PX9LAL loads the relocatable code of the module beginning at the specified load point unless the load point is an odd address (not on a word boundary). In that case, PX9LAL loads the relocatable code beginning at the word boundary preceding the address. When no load point is specified for the first module loaded, PX9LAL loads the relocatable code beginning at a default address ($A0_{16}$). When no load point is specified for a subsequent module, PX9LAL loads the relocatable code beginning at the first word boundary following the last byte of the preceding module.

Loading of Program Modules. PX9LAL loads a program module by placing the data of the module in the proper addresses. The object program may contain both absolute and relocatable code. PX9LAL places data at absolute addresses supplied in the object program, and modifies relocatable addresses to obtain actual memory addresses into which it places the associated data. Absolute data is placed in memory directly, but relocatable data is modified and placed in memory. Relocation and the modifications required for relocation are described in a subsequent paragraph.

As PX9LAL loads a module, it processes the data in the module that specifies the linking to be performed. This data consists of symbols from the operand fields of DEF statements, and symbols from the operand fields of REF statements. PX9LAL maintains a list of symbols and the corresponding memory addresses to perform the required linking, and to define required modules. Linking is described in a subsequent paragraph.

After the user enters "E" in response to the "LOAD/END?" message, PX9LAL prints a list of symbols that represent any unresolved references and the entry point for the program. The user may either prepare a module for loading to resolve the references and request PX9LAL to load it, or return to the monitor.

Relocation. The relocation provided by PX9LAL allows the relocatable segments of program modules to be loaded into available memory sequentially.

Within relocatable segments of a program module, all addresses are relative to the start of the first relocatable segment of the program module. PX9LAL computes the corresponding memory address by adding the memory address of the load point of the program module to the relocatable addresses.

Data within the program module that represents a relocatable address or is derived from a relocatable address (by evaluating an expression, for example) is relocatable even though it may appear at an absolute address. PX9LAL modifies this data by adding the memory address of the load point of the program module to the data.

By modifying addresses and data as previously described, PX9LAL makes the necessary adjustments for executing the program properly from any area of memory. This modification precedes, and is independent of, any required linking.

Linking. The purpose of linking is to integrate two or more program modules as they are loaded, resulting in a program in memory which the computer can execute properly, i.e., any address required by more than one module must be placed in all locations that reference the address.

The object code of each module contains the symbols defined in the module for use in other modules. A value is associated with each symbol.

The object code of each module also contains any symbols required in the module but defined in another module. Associated with each symbol is an address of a location into which the value associated with the symbol must be placed. When the value is required in more than one location in the module, these locations are chained together with each location containing the address of the next location, and the last location in the chain containing zero. As supplied by the assembler, the addresses in the chain may be either absolute or relocatable.

To link the modules, PX9LAL processes the chain associated with each external symbol by placing the corresponding address in each location in the chain until it has placed the address in the location that contains zero, the end of the chain.

The object code of a module may also contain a symbol similar to an external reference, but different in two respects. The symbol is the first six characters of the IDT character string of one of the program modules to be linked, and a zero value is associated with the symbol. The symbol is used by PX9LAL to identify a required module. The zero inhibits any attempt to perform linking. When more than one program module is to be loaded, the first module must contain at least one reference of this type and may contain one for each module of the load.

Printed Output. The printed output of PX9LAL is a full or partial load map. This map shows the name and load point of each module that is loaded. The full map also includes the symbols and the corresponding memory addresses of any external definitions in the module. Both types of maps contain only names of modules that have been loaded because they were referenced.

3.4.6.2 Operational Messages. When PX9LAL is started by the LL command, a series of messages requesting user responses are given. The messages are the following.

LD PT?

Load Point. The user should input the hexadecimal memory location of the load point for the object module. If a carriage return is entered, the default value of 0 will be assumed.

LD BI?

Load Bias. The user should input the hexadecimal value of the load bias for the object module or modules to be loaded. If a carriage return is entered, the default value $00A0_{16}$ will be assumed.

The load point and load bias specified above are used in determining how the code is relocated and the memory address where the code will actually be loaded. Code assembled with an absolute origin (AORG) directive will be loaded at the absolute address determined by the directive plus the load point.

MEMLOC = ABS ADDR + LD PT

Code assembled with a relocatable origin (RORG) directive will be loaded at the relocatable address determined by the directive plus the load bias plus the load point.

MEMLOC = REL ADDR + LD BI + LD PT

Note that the relocation is performed on the code using the load bias only. Specifying a load bias is equivalent to placing a D tag with that bias before the module being loaded. The load point is only used to determine the actual memory location where the code will be loaded.

Object code loaded with a load point not equal to the default 0 is not executable. This feature has been included in the linking loader for special Prototyping System applications.

F/P LIST?

Full or Partial List. The user enters a character to specify the type of memory map desired. When the user enters an "F", PX9LAL prints a full memory map. When the user enters a "P", or any other character, PX9LAL prints a partial memory map. A partial memory map lists the IDT name and load point of each module that is loaded. Multiply-defined references are shown by one or more "Ms" following the load point of the module in which the multiple definition occurs. The partial map does not identify the multiple definitions, but does indicate the number of multiple definitions in each module.

A full memory map lists IDT name and load point of each module that is loaded like the partial memory map. The full memory map also includes the symbols and the corresponding memory addresses of any external definitions in the module. Names of modules that are not referenced do not appear in either type of map. Multiply-defined references are each identified by an "M" at the end of the external definition line.

LOAD/END?

Load or End. To load a program module or modules, the user should position the cassette to the desired object module and enter an "L" which may be followed by the hexadecimal logical unit number of the input device. (See Section II and paragraph 3.4.2.) If no number is input, a default LUNO of 7 will be assumed. If a number is input it must be between hexadecimal 0 and F inclusive and of the form "L<n>" with no embedded blanks.

Example: LOAD/END? L8

When the load option is selected, PX9LAL loads all the object modules on the positioned cassette until an end-of-file is encountered. Unless a fatal error is encountered while loading, PX9LAL will repeat the previous message after the modules have been loaded. At that point, the user may position to another module on cassette to be loaded.

When all modules required for the program have been loaded, the user should enter an "E" to end the load process.

When the user enters an "E", PX9LAL prints any undefined symbols and the following message to identify the entry point of the loaded program.

ENTRY = XXXX

If no entry point was specified, the program assumes a default of $00A0_{16}$.

The following question is then asked.

TERM/CONT?

Terminate or Continue. The user should enter "T" to terminate the load process or "C" to continue. If terminate is selected, control returns to the monitor. At that point the program counter register has been set to the entry printed previously and the user may enter the "EX" or "RU" command to execute or debug his program.

The user may select the continue option in order to load more program modules, possibly to satisfy undefined references. If continue is selected the "LOAD/END" question will be asked again.

3.4.6.3 **Error Messages.** PX9LAL prints an error message when it detects an error. One message is for a command processor error. There are four fatal errors that terminate execution of PX9LAL following the printing of the error message. There are four other error messages that serve as warnings. PX9LAL continues the load operation following the printing of these messages.

Command Processor Error. This message is:

MX03 PX9LAL Not Loaded in Transient Area

PX9MTR prints this message when it determines that PX9LAL is not resident in the monitor transient area. Load the overlay and reenter the command.

Fatal Errors. The first message is:

LL01 Illegal Load Sequence

In a module, PX9LAL will accept no other field except a field having a tag character of D ahead of a field that is preceded by a tag character of 0. PX9LAL will not accept a field preceded by tag character D after it has read a field preceded by the tag character 0. When PX9LAL reads a field of the object code out of sequence, it prints this message. PX9LAL then terminates and restarts. The user may recover from the error by correcting the sequence of the object code and reloading the program.

The second message is:

LL02 Invalid Load Code

PX9LAL prints this message when it reads an invalid character as the tag character. Valid tag characters processed by PX9LAL are the hexadecimal digits 0 through D and F, G, and H. Tag characters G and H are symbol table tags in the 990/10 Disc System Software, but PX9LAL ignores them.

When this error occurs, the PX9LAL module (but not the entire link sequence) terminates and restarts. The user may recover from the error by correcting the object code and reloading the program. The error in the object code may be an error in entering the tag character. It may also be a legitimate tag character used incorrectly, causing PX9LAL to consider a character in a label or a character string as a tag character.

The third message is:

LL03 Missing End Statement

PX9LAL prints this message when a second tag character 0 is followed by a nonblank IDT character string in the same object module (no end-of-module record between the two). An object module may contain more than one field with a tag character of 0, but the IDT character string associated with a subsequent 0 tag must be blank.

When this error occurs, PX9LAL terminates and restarts. The user may recover from the error by correcting the object code and reloading the program. The obvious correction is the insertion of an end-of-module record preceding the second field that has a tag character of 0. However, when the error results from improper concatenation of object code files or omission of one or more object records, additional correction may be required.

The fourth message is:

LL04 Load Address Error

PX9LAL prints this message when a load address is out of the user area or would cause PX9LAL to load data over the symbol table.

If this error occurs, PX9LAL terminates and restarts. The user may recover from this error by changing the load bias if the bias is greater than the default. PX9LAL loads programs in the user area of memory and builds a symbol table at the top of user memory directly below PX9MTP. The symbol table contains the IDT character string of the first module loaded, and each symbol is used in an external reference or definition. Ten bytes of memory are required for each entry in the table.

Nonfatal Errors. The first message is:

LL05 Previous Load Module Error

PX9LAL prints this warning message when the first six characters of the IDT name of the current module match the first six characters of the IDT name of a previously loaded module or match a previously loaded externally defined symbol. PX9LAL then skips over records to the end of the module and positions the tape to the beginning of the next module. The message LOAD/END is then printed. The user should identify the module to which the message applies. When the module is required instead of the previously loaded module, either remove the first module, or place the required module ahead of the other module in the load sequence, and reload. When both modules are required, change the IDT character string of either module, and reload. When the first six characters of the IDT character string are identical to a symbol externally defined in a module of the program, change either the symbol or the IDT character string and reload. When the module is not required, the mesage may be ignored.

The second message is:

LL06 Checksum Error–Retry

Each record of an object module contains a checksum. The checksum is the 2's complement of the sum of the binary values corresponding to the ASCII representations of the characters in the record, including the checksum tag character, and is expressed as four hexadecimal digits. PX9LAL computes a checksum of the record it has read and compares the result with the checksum from the record. When the checksums are not equal, PX9LAL prints this message.

The user may position the object module tape for reading the record again by taking the playback cassette off-line and backspacing the tape one record. (To backspace the tape one record, set the PLAYBACK switch to LOCAL, and press the REV side of the BLOCK FWD/REV switch in the PLAYBACK CONTROL area of the upper switch panel.) If the user does not change the position of the tape, the checksum error will be ignored. To continue the loading process, the user must enter a carriage return on the keyboard.

A checksum error which represents an inaccurate reading of the object code should not be ignored. The record should be reread at least once in the attempt to read it correctly. However, a checksum error may be the result of altering the contents of an object record without removing the checksum field. This type of checksum error may be ignored.

The third message is:

M	Multiply-Defined Symbol

PX9LAL prints an "M" for each multiply-defined external definition encountered when processing a module. When a full memory map is being printed, the "M" is printed on the external definition line. When a partial memory map is being printed, an "M" for each multiply-defined symbol in the module follows the module name and load point.

When the second definition is required in the program instead of the first, change the load sequence to load the program module that contains the desired definition first and reload. When both definitions are required in the program, change one of the symbols and reload. This may also require changing corresponding references to avoid other errors. When the symbol of the definition contains the first six characters of the IDT character string of a previously loaded program module, change the symbol and corresponding references or the IDT character string and its reference, and reload. When the definition is not required in the program, ignore the message.

The fourth message is:

UNDEFINED	Undefined Symbols

When all modules have been processed, and the user enters "E" to the "LOAD/END" option, PX9LAL scans the symbol table to find any symbols that are not defined. If any undefined symbols are found, this message followed by a list of undefined symbols is printed.

An intentionally undefined external reference (dummy reference) that is included in one of the modules of a program permits a type of load-time patching. If external references have been inadvertently omitted, a program module may be generated that uses the deliberately undefined reference as the IDT character string, and consists of absolute external definitions to satisfy program requirements. PX9LAL loads this module, and the program is ready for execution. If the deliberately undefined reference is the only undefined reference, the program may be executed properly with the reference remaining undefined. This technique is intended for use during program development.

3.4.6.4 Examples of Load Map Printouts. The following five examples show load map printouts of load operations with and without linking.

Example 1:

```
.OV 8
LL
.LL

LD PT?
LD BI? 100
F/P LIST? F

LOAD/END? L

        XREF     0100
               ◆ PRINTC   0196
               ◆ GETCHR   01A4
               ◆ TERM     01C8
        PARSEM   02B4
               ◆ PARSE    0316
               ◆ DEFPR    0470
               ◆ OPNDPR   0474
               ◆ OPERPR   0472
               ◆ STMT     0466
        CTYPM    04D6
               ◆ CTYP     051E
        PRTBM    0568
               ◆ PRTB     0568
        CSYMM    06B6
               ◆ CSYM     06B6
               ◆ ISYM     0718
               ◆ NXTLOC   072A
               ◆ ENDSYM   1E3A
               ◆ FSTSYM   072C
        SYMRFM   1E46
               ◆ SYMREF   1E46
               ◆ OVFL     1E7C
        SYMDFM   1E9C
               ◆ SYMDEF   1E9C
LOAD/END? E
 ENTRY = 0100

TERM/CONT? T

.
```

Example 1 shows a full load map printout of a link and load of seven modules. The default load point of 0 is taken and a load bias of 0100_{16} is entered. The module names are XREF, PARSEM, CTYPM, PRTBM, CSYMM, SYMRFM, and SYMDFM. The address following XREF shows that the module XREF is loaded at address 100_{16}. The addresses following each of the other module names specify the hexadecimal addresses where each module is loaded. The symbol names preceded by asterisks are the external definitions supplied by the module, and the absolute addresses corresponding to the defined labels are also printed. The external definitions PRINTC, GETCHR, and TERM are defined in the module XREF. PRINTC is at address 0196_{16}, GETCHR is at address $01A4_{16}$, and TERM is at address $01C8_{16}$.

All of the files are on one cassette with an end-of-file after the last module SYMDFM. Therefore, all the modules were loaded, and the LOAD/END question was then printed. Since all modules had been loaded, an "E" was entered to end the load process. The entry point of 0100_{16} was then printed.

Example 2:

```
.LL

LD PT? 0
LD BI? 100
F/P LIST? P

LOAD/END? L

      XREF      0100
      PARSEM    02B4
      CTYPM     04D6
      PRTBM     0568
      CSYMM     06B6
      SYMRFM    1E46
      SYMDFM    1E9C
LOAD/END? E
 ENTRY = 0100

TERM/CONT? T

.
```

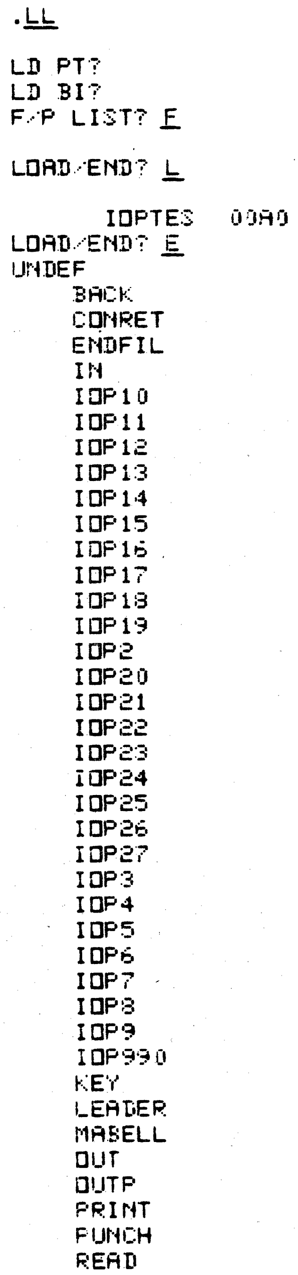
Example 2 shows a partial load map printout of a link and load of the same seven modules loaded in example 1.

Example 3:

```
.LL

LD PT?
LD BI?
F/P LIST? F

LOAD/END? L

      IOPTES    00A0
LOAD/END? LB

      IOP990    10DC
            ◆ MABELL    10DC
      IOP2      10F6
            ◆ ENDFIL    10F6
      IOP3      1106
            ◆ LEADER    1106
      IOP4      1126
            ◆ ENDF      1164
            ◆ ENDL      1168
            ◆ PCHCR1    1154
            ◆ PUNCH     1126
      IOP5      1178
            ◆ CNTCHK    1182
            ◆ CNTRTN    118A
            ◆ CONRET    1178
      IOP6      119E
            ◆ REDPNT    119E
      IOP7      11AC
            ◆ PRINT     11AC
            ◆ PRTIT     11B4
      IOP8      11D0
            ◆ KEY       11D0
      IOP9      1202
            ◆ READ      1202
            ◆ REDIN     122A
      IOP10 1248
            ◆ PUTIN   1248
      IOP11 1276
            ◆ SCREEN  1276
      IOP12 1292
            ◆ TABCHK  1294
      IOP13 12C6
            ◆ CONTRL  12C6
      IOP14 12EC
            ◆ FORM    12EC
      IOP15 1322
            ◆ FORM    1322 M
            ◆ LFCR    1322
            ◆ LFCR2   1326
      IOP16 1358
            ◆ CNTCHK  1366 M
            ◆ CNTRTN  1366 M
            ◆ CONTRL  1366 M
            ◆ FORM    1366 M
            ◆ IN      1358
            ◆ LFCR    1366 M
            ◆ PRTIT   1366 M
            ◆ PUTIN   1366 M
            ◆ SCREEN  1366 M
      IOP17 1368
            ◆ OUTP    1368
            ◆ TABCHK  1368 M
      IOP18 138C
            ◆ FLAG    138C
            ◆ OUT     138E
            ◆ OUTP    138E M
            ◆ REDIN   13B8 M
            ◆ TABCHK  138E M
      IOP19 13C8
            ◆ STATUS  13C8
      IOP20 13CE
            ◆ LOAD    13D4
            ◆ REWND1  13D0
            ◆ REWND2  13CE
      IOP21 13F4
            ◆ BACK    13F4
      IOP22 140C
            ◆ ULOAD1  140E
            ◆ ULOAD2  140C
      IOP23 1414
            ◆ RECRD1  1416
            ◆ RECRD2  1414
      IOP24 1426
            ◆ REWIND  1426
      IOP25 1450
            ◆ RDC     1450
      IOP26 1466
            ◆ STATAS  1466
      IOP27 147A
            ◆ DELAY   147A
            ◆ DELAY1  147E
LOAD/END? E
 ENTRY = 10D8

TERM/CONT? T

.
```

Example 3 shows a full load map printout of a link and load of 28 modules. The default load point and load bias values of 0 and $A0_{16}$ respectively are taken. The first module IOPTES is on one cassette and the other 27 modules are on a second cassette. The "L" response to the first LOAD/END question specifies a load from the tape mounted in the drive assigned to LUNO 7. After the first module is loaded and the end of file encountered, the LOAD/END question is asked again. The response "L8" specifies a load from the tape mounted in the drive assigned to LUNO 8. The Ms printed after the absolute addresses of the external definitions indicate that these references are multiply defined. When a reference is multiply defined, the first encountered definition is used. The response "E" to the final LOAD/END question ends the load process. The entry point of $10D8_{16}$ is then printed.

Example 4:



```
.LL

LD PT?
LD BI?
F/P LIST? F

LOAD/END? L

        IOPTES   00A0
LOAD/END? E
UNDEF
     BACK
     CONRET
     ENDFIL
     IN
     IOP10
     IOP11
     IOP12
     IOP13
     IOP14
     IOP15
     IOP16
     IOP17
     IOP18
     IOP19
     IOP2
     IOP20
     IOP21
     IOP22
     IOP23
     IOP24
     IOP25
     IOP26
     IOP27
     IOP3
     IOP4
     IOP5
     IOP6
     IOP7
     IOP8
     IOP9
     IOP990
     KEY
     LEADER
     MABELL
     OUT
     OUTP
     PRINT
     PUNCH
     READ
```

```
        RECRD1
        RECRD2
        REDPNT
        REWND1
        REWND2
        STATUS
        ULOAD1
        ULOAD2
      ENTRY = 10D8

     TERM/CONT? T

     .
```

Example 4 shows a full load map printout of a load of the first module used in example 3. The "E" response to the second LOAD/END question terminates the load process. Any undefined references are then printed. These undefined references are external references specified in the module IOPTES. To continue loading to satisfy the undefined references, a "C" could be entered when the "TERM/CONT" question is asked. The user could then continue the load process.

Example 5:

```
     .LL

     LD PT? 1000
     LD BI?
     F/P LIST? F

     LOAD/END? L

            IOP990   00A0
                     ◆ MABELL    00A0
     LOAD/END? E
      ENTRY = 00A0

     TERM/CONT? T

     .
```

Example 5 shows a full load map printout of a load of one module. The load point specified is 1000_{16} and the default load bias of $A0_{16}$ is selected. The printout specifies that the load point of IOP990 is $A0_{16}$ and the symbol MABELL is at location $A0_{16}$. These are the addresses at which this program will execute and all relocation is done with this bias. However, the relocatable code is actually loaded starting at $10A0_{16}$, the sum of the load point and the load bias.

3.4.7 DUMP IN ABSOLUTE FORMAT (DP). The Dump in Absolute Format (DP) command is used to dump an area of memory to cassette tape in compressed absolute data format. This command must be loaded into the transient area with the OV command before it can be executed.

Syntax definition:

$$\text{DP} \begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix} \text{<start addr>} \begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix} \text{<end addr>} \left[\begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix} \left[\text{<entry point>}\right] \right.$$

$$\left. \left[\begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix} \text{<program name>} \right] \left[\begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix} \text{P} \right] \right]$$

The command is terminated by a carriage return.

Parameters:

start addr	Memory address of the first byte to be dumped; a hexadecimal number in the range 0 through FFFF.
end addr	Memory address of the last byte to be dumped; a hexadecimal number in the range 0 through FFFF.
entry point	Entry point of the program when it is reloaded; a hexadecimal number in the range 0 through FFFF.
program name	The name of the program; 1 to 15 alphanumeric characters.
P	If P is entered, the end-of-module tag and the end-of-file marker will not be written on tape.

Parameter default values:

The first two parameters, the starting memory address and the ending memory address, are required.

If entry point is not specified, no entry point value is supplied.

If program name is not specified, no program name is supplied.

If P is not specified, the end-of-module tag and the end-of-file marker will be written.

Description: The Dump in Absolute Format command allows the user to store on tape any sections of memory he wishes to save. This is very useful when patches have been made to the object code during a debug session. By dumping the code to tape, these patches need not be recreated when the debugging is resumed.

Storing a program in absolute format is also useful for loading purposes. Whether loaded by the LA command or with the upfront loader and the LU command, the load is considerably faster than with the standard object code loader.

The output is directed to the device assigned to LUNO 7, normally tape cassette CS1. The LUNO assignment may be changed with the Assign LUNO (AL) command.

The partial dump option is useful for dumping noncontiguous portions of memory. The last module dumped must not be a partial dump or an error will occur when loading.

Error messages:

DP13	Low memory address greater than high memory address.
MP00	Parameter specification error. Reenter the command with the correct parameter.

MS05 Required parameter not entered. Reenter the command with the parameter.

MX01 Unrecoverable I/O error.

Examples:

```
.DP 1000,1030,1004,DUMPIT
.

.DP 1000,1040
.
```

In the first example, the bytes from location 1000_{16} up to and including 1030_{16} are dumped. The entry point is 1004_{16}, and the name of the program is DUMPIT. In the second example, no name or entry point was specified when the memory area was dumped. In both examples, because P is not specified, the end-of-module tag and the end-of-file marker will be written.

3.4.8 LOAD PROGRAM IN COMPRESSED ABSOLUTE FORMAT WITH UPFRONT LOADER (LU). The Load Program in Compressed Absolute Format with Upfront Loader (LU) command initiates a load of absolute code.

Syntax definition:

$$\text{LU}\left[\begin{Bmatrix} , \\ \text{ƀ}\ldots \end{Bmatrix} \left[<\text{luno}>\right] \left[\begin{Bmatrix} , \\ \text{ƀ}\ldots \end{Bmatrix} <\text{bias}>\right]\right]$$

The command is terminated by a carriage return.

Parameters:

luno Logical unit number of the input device.

bias Base address of the relocatable upfront loader.

Parameter default values:

If the logical unit number is not specified, a value of 7, normally assigned to tape cassette CS1, is used.

If the bias address is not specified, the upfront loader is loaded at a location $1B0_{16}$ bytes below the beginning of the monitor.

Description: Compressed absolute format code may be loaded by including a short loader (called an upfront loader) at the beginning of the code.

The upfront loader is $1B0_{16}$ bytes of relocatable code, in standard 990 object code module format, placed in front of a load module of compressed absolute format code in order to reduce the loading time. Executing the LU command causes the upfront loader to be loaded by the 733 ASR ROM loader. After the upfront loader is memory resident, control is passed to it and the compressed absolute load initiated. When the user program is loaded, the program entry point is placed in the user's PC register and control is returned to the command string processor. The user must be careful to put the upfront loader at a position in the user memory where it will not be overlayed by the program being loaded.

Error message:

LD00 Load error.

LD01 Invalid LUNO.

Examples:

```
.LU
.LU 7,1BA0
.LU,,1BA0
.
```

The first and third examples load from a default LUNO of 7. The first example loads the upfront loader at a default bias address $1B0_{16}$ bytes below the beginning of the monitor. The second example has the load LUNO and bias address supplied. The third example has the load bias supplied.

3.4.9 LOAD PROGRAM IN COMPRESSED ABSOLUTE FORMAT (LA). The Load Program in Compressed Absolute Format command loads object code that has been stored in a compressed absolute format by the Dump in Absolute Format (DP) command. The LA command must be loaded into the transient area with the OV command before it can be executed.

Syntax definition:

$$\text{LA}\left[\begin{Bmatrix} , \\ \text{b̸...} \end{Bmatrix} \langle\text{luno}\rangle\right]$$

The command is terminated by a carriage return.

Parameter:

luno Logical unit number of the input device.

Parameter default value: If the logical unit number is not specified, a value of 7, normally assigned to tape cassette CS1, is used.

Description: To execute the LA command, the absolute loader must be resident in the transient area. If it is not there, it must be loaded as an overlay by using the Load Overlay (OV) command.

If the load is successful, the module name is printed if it was defined and the entry point address is placed into the user's PC register. Control is returned to the monitor after a successful load or if an error occurs. Refer to Section VI for a description of compressed absolute object code format.

Error messages:

LD00 Load error.

LD01 Invalid LUNO.

MX03 Command not resident in transient area. Load the overlay and reenter the command.

Examples:

```
.LA 8
DUMPIT
.

.LA
.
```

In the first example, the module on LUNO 8, which was created using a Dump in Absolute Format (DP) command, is loaded. (The program name DUMPIT was assigned to the module.) The module is loaded and the program name printed. The entry point is put in the user's program counter register; this address will be displayed in the programmer panel data indicator lamps.

The module loaded in the second example is input from default LUNO 7 and did not have a name associated with it when it was created with the DP command.

3.4.10 EXECUTE USER PROGRAM DIRECTLY (EX). The Execute User Program Directly command is used to start a user program. (The one-pass assembler and the text editor are loaded as user programs.)

Syntax definition:

```
EX
```

The command is terminated by a carriage return.

Description: The program is executed directly by the 990 computer without using the SIE or trace features. Execution is started with the PC, WP and ST that would be displayed if an Inspect Registers (IR) command were executed.

Application notes: In order to regain control from an executing user program, the user must intervene at the programmer panel. The monitor may be restarted by transferring control to its starting memory location (the first word of the monitor memory area).

The processor registers (the WP, PC and ST registers), the contents of which may be displayed by entering the Inspect Registers (IR) command, are not updated when a program is executed with the EX command.

A user program may return control to the monitor by using the end-of-program supervisor call.

If the user runs a stand-alone program, for example using the CRU to perform I/O, he must inspect the processor registers from the programmer panel.

Example:

```
.IR
PC=046C WP=0000 ST=0000
.EX
ASM/TERM? A

ASM/TERM? T

.IR
PC=046C WP=0000 ST=0000
.
```

The EX command begins execution with the PC, WP and ST registers equal to the values obtained when the Inspect Registers (IR) command is invoked. A program run under EX does not change the contents of these registers. The second IR command shows that the contents remain the same.

3.4.11 EXECUTE USER PROGRAM UNDER SIE OR TRACE (RU). The Execute User Program under SIE or Trace command provides controlled execution of the user's program.

Syntax definition:

```
RU [{ ,   } <instruction count>]
    {b̸...}
```

The command is terminated by a carriage return.

Parameter:

instruction count	Maximum number of instructions to be executed before returning to command mode. A value of 0 indicates that no instruction limit applies.

Parameter default value: The value of the instruction count at the last entry into command mode is used as the default value. If the previous RU command has exhausted the instruction count, the default is 0, implying no instruction limit. The system is initially loaded with a default value of 0.

Description: Instructions in the user's program are executed one at a time using either the hardware SIE feature or the software trace interpreter. The user may specify one of these two modes of operation with the Set Trace Region (SR) command (paragraph 3.4.26).

Before the monitor executes a user instruction, it checks whether the instruction is within a defined trace region. If the instruction is within a trace region, the trace interpreter is called and the instruction traced. If the instruction is not within a trace region, the instruction is executed using Single Instruction Execution (SIE, described in paragraph 3.3.1). In both cases, the user's WP, PC, and ST registers are updated after each instruction executed. The monitor checks whether a breakpoint has been reached and if so, prints out the user's registers and snapshot, if defined. If a snapshot is assigned to a breakpoint, the monitor continues execution after the breakpoint has been reached, without operator intervention. If no snapshot was specified, the monitor returns control to the command processor. (Refer to the descriptions of the SB and SS commands in paragraph 3.4.20 and 3.4.23.) If the run count, number of instructions to be executed, is depleted, the monitor returns control to the command processor. Otherwise the monitor continues execution of the user program.

Error message:

MX04 Attempt to execute in trace mode when the instruction trace
overlay is not loaded.

Application notes: Be sure that the processor registers are properly set before beginning execution. The contents of the registers may be inspected with the Inspect Registers (IR) command and modified as needed with the Modify Registers (MR) command. The initial PC is set by the loader when a user program that specifies an entry point is loaded. The starting memory location of a program is specified in the END statement of the program; a label that appears within the program for this purpose is referenced in the END statement. If two or more user programs are competing for specification of the starting location, the last one loaded takes precedence.

The user may regain control of the program which is executing under SIE or instruction trace by pressing the escape (ESC) key on the terminal keyboard. If the user's program is using monitor I/O support, pressing the ESC key may cause an escape character to be returned to the program rather than to the monitor; the user should be aware that the escape character may be handled in these two different ways since the results of program operation may be affected.

Interrupts are processed as they occur by the user program using the SIE mode of execution. When running under the trace mode, interrupts and extended operations (XOPs) are executed directly.

When running under SIE, an IDLE assembly language machine instruction is handled like an NOP instruction. The SIE level 0 interrupt causes the computer to continue execution.

The user must be aware of how the 733 ASR operates when he decides to enable interrupts since interrupts can occur when character keys are pressed. PX9MTP is not interrupt driven; therefore, significant problems may result. It is recommended that the interrupt mask be set if possible so that the 733 ASR cannot interrupt.

The overhead when executing under SIE is approximately 100 instructions for each user instruction. Using trace, the overhead is approximately 170 instructions for each user instruction.

It is often convenient to use the trace mode of execution when no information is being printed (by setting a null trace type). This is similar to executing using the SIE processor except that interrupts run at full processing speed. A variable trace can also be used to detect modification of particular memory locations. (Variable trace is explained in paragraph 3.4.26.)

Examples:

```
.RU
.RU 5
```

In the first example, the maximum number of instructions to be executed before returning to command mode is the value used at the last entry into command mode, or is 0 initially or if the previous RU command has exhausted the instruction count. The second example specifies an instruction count of 5.

3.4.12 MODIFY MEMORY (MM). The Modify Memory command displays the address and contents of a memory word and accepts a new hexadecimal data value from the user.

Syntax definition:

$$\text{MM}\left[\begin{Bmatrix} , \\ \text{b̸}\ldots \end{Bmatrix} \text{<memory address>}\right]$$

The command is terminated by a carriage return.

Parameter:

memory address   Address of memory to be modified.

Parameter default value: If the memory address is not specified, a value of 0 is used.

Description: If the user inputs a new value, the memory location is modified to match the input value. If the user terminates his input with a blank (space), the next location value is printed and the process repeated. If the user terminates his input with a carriage return or comma, the command processing terminates.

Error message:

DP00   An invalid hexadecimal value was input.

Application note: The MM command is useful for setting up desired conditions in order to check out a routine. It is also convenient for creating patches and for examining memory one word at a time.

Example:

```
.MM 1000
1000=FFFF 1
1002=FFFF 3
1004=FFFF
1006=FFFF 8
.
```

These command statements place the value 1 in location 1000, 3 in location 1002, and 8 in location 1006. The user may enter a space (blank) if he does not want to modify a location but wants to go on to the next location. A carriage return terminates the command at any time.

3.4.13 INSPECT MEMORY (IM). The Inspect Memory command is used to display in hexadecimal format the contents of one or more consecutive memory locations.

Syntax definition:

$$\text{IM}\left[\begin{Bmatrix} , \\ \text{b̸}\ldots \end{Bmatrix} \text{<starting mem addr>} \left[\begin{Bmatrix} , \\ \text{b̸}\ldots \end{Bmatrix} \text{<ending mem addr>}\right]\right]$$

The command is terminated by a carriage return.

Parameters:

starting mem addr — Hexadecimal value representing the memory address of the first memory word displayed.

ending mem addr — Hexadecimal value representing the memory address of the last memory word displayed.

Parameter default values:

If neither parameter is specified, all memory is dumped.

If the ending address is not specified, only one word is displayed.

An odd address is changed to the preceding word address before the addressed byte is displayed.

Description: Memory is displayed in groups of four words, two groups per line. The address of the first word on the line is printed at the left. The display may be terminated at any time by pressing the ESC key on the terminal keyboard.

Error message;

DP13 — The ending address specified is less than the starting address specified.

Examples:

```
.IM 1000,1004
1000=1002 C0E0 023E
.

.IM 1006
1006=1004
.
```

3.4.14 **MODIFY REGISTERS (MR).** The Modify Registers command displays the contents of the user's internal registers – workspace pointer (WP), program counter (PC), and status (ST) registers – and allows the user to modify them.

Syntax definitions:

MR

The command is terminated by a carriage return.

Description: The register name and current contents are printed and an input is accepted from the user. If the user inputs a valid hexadecimal number, the contents of the registers are changed. If the user enters a space, the processor prints the name and contents of the next register. If the user enters a carriage return, the command terminates.

Error message:

DP00 An invalid hexadecimal number was input, or the number input was greater than FFFF$_{16}$.

Application notes: Modification of the Workspace Pointer (WP) register causes the registers that would be displayed by the Inspect Workspace Registers (IW) command to change. The Modify Registers command is used to establish the initial environment for a program executed with the Execute User Program Directly (EX) or the Execute User Program under SIE or Trace (RU) command.

Examples:

```
.MR

PC=2000 244
WP=0000 A6
ST=0000
.

.MR

PC=0244
WP=00A6 A2
ST=0000 2
.

.MR

PC=0244 246
.
```

The first example changes the value in the PC register to 244$_{16}$ and the value in the WP register to A6$_{16}$. The second example changes the WP register value to A2$_{16}$ and the ST register value to 2$_{16}$. The third example changes the PC register value to 246$_{16}$.

As in the second example, the user may press the space bar on the terminal keyboard if he does not wish to modify a particular register. As in the third example, he may press the RETURN key on the terminal keyboard after entering a new PC register value to terminate the command.

3.4.15 INSPECT REGISTERS (IR). The Inspect Registers command displays the contents of the user's registers – the program counter (PC), workspace pointer (WP), and status (ST) registers – for the current user program.

Syntax definition:

IR

The command is terminated by a carriage return.

Application note: The displayed register values are those values which are loaded into the processor in response to an EX or RU command.

Example:

```
.IR
PC=0246 WP=0000 ST=0000
.
```

3.4.16 MODIFY WORKSPACE REGISTERS (MW).

The Modify Workspace Registers command is used to display and change the contents of one or more of the user's workspace registers.

Syntax definition:

$$\text{MW}\left[\begin{Bmatrix} , \\ \text{b̸...} \end{Bmatrix} \text{<starting workspace reg>}\right]$$

The command is terminated by a carriage return.

Parameter:

starting workspace reg — The first workspace register to be displayed. (Hexadecimal value.)

Parameter default value:

If the starting workspace register is not specified, a value of 0 is used.

Description: The names and current contents of the workspace registers are displayed. The command processor accepts the user's input, which may be a new value for the register contents and a terminator. If a new value is input, the current contents of the specified register is changed. If the terminator is a blank, the next register is printed for modification. If the terminator is a carriage return or comma, the command processing terminates. The command processing terminates automatically after processing workspace register 15 (F_{16}).

Application note: The user is cautioned to be sure that the workspace pointer actually points to the intended workspace. The Modify Workspace Registers command displays the registers within the current workspace (the workspace defined by displaying the WP in an IR command).

Example:

```
.MW 4
R4=0000 7
R5=0000 89
R6=0000
R7=0000 1000
.
```

This example changes the contents of workspace registers R4, R5 and R7 to 7_{16}, 89_{16} and 1000_{16}, respectively. A carriage return was entered after changing the contents of R7.

3.4.17 INSPECT WORKSPACE REGISTERS (IW).

The Inspect Workspace Registers command is used to display the contents of a sequence of the user's workspace registers.

Syntax definition:

$$\text{IW}\left[\begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix}\left[\text{<starting workspace reg>}\right]\left[\begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix}\text{<ending workspace reg>}\right]\right]$$

The command is terminated by a carriage return.

Parameters:

starting workspace reg	First workspace register to be displayed. Hexadecimal number.
ending workspace reg	Last workspace register to be displayed. Hexadecimal number.

Parameter default values:

If the starting workspace register is not specified, a value of 0 is used.

If the ending workspace register is not specified, the value used is the starting workspace register.

If neither parameter is specified, all 16 registers are displayed.

Description: The set of workspace registers displayed are those pointed to by the WP that would be displayed if an IR command were executed. Workspace registers are displayed with the register number preceding the register contents.

Error message:

DP13	Either the starting workspace register number is greater than the ending workspace register number, or a workspace register number greater than F_{16} was requested.

Examples:

```
.IW
R0=0000 R1=0000 R2=0026 R3=0000 R4=0000 R5=2032 R6=0000 R7=0000
R8=0000 R9=0000 RA=0000 RB=0000 RC=0000 RD=3798 RE=2008 RF=0002
```

If no workspace register or range is specified, all 16 registers are printed.

```
.IW 2,8
R2=0000 R3=0000 R4=0000 R5=0000 R6=0000 R7=0000 R8=0000

.IW 2
R2=0000
.
```

3.4.18 MODIFY CRU REGISTER (MC). The Modify CRU Register command reads and displays the data on CRU input lines, and sets data on CRU output lines.

Syntax definition:

$$\text{MC}\left[\begin{Bmatrix} , \\ \text{ƀ}\ldots \end{Bmatrix}\right]\left[\text{<CRU address>}\right]\left[\left[\begin{Bmatrix} , \\ \text{ƀ}\ldots \end{Bmatrix}\right]\text{<CRU width>}\right]$$

The command is terminated by a carriage return.

Parameters:

CRU address	The CRU word address. A value from 0 to $1FFF_{16}$.
CRU width	The number of bits to be changed in each CRU word (hexadecimal). A value from 1 to 10_{16}. A value of 0 is interpreted as 10_{16}.

Parameter default values:

If the CRU word address is not specified, a value of 0 is used.

If the CRU width is not specified, a value of 10_{16} is used.

Description: When the CRU bit width is less than 16 bits, the data value is displayed right justified in a four-digit hexadecimal value. The user's data may be input as a four-digit value; the rightmost bits, where the bit width is given by the CRU width parameter, are used to modify the CRU value. Enter a new value to change the value, a space to continue on to the next value, and a carriage return to terminate data modification.

The addresses are displayed as they would be used in workspace register 12 (the CRU base address), which is the actual CRU bit address times 2. Also, data is displayed and entered directly as the STCR/LDCR instruction receives/sends it.

If the CRU word address is greater than $1FFF_{16}$, the command is ignored.

Error message:

DP12	CRU bit width parameter too small (negative) or too large (greater than F_{16}). Invalid bit string width.

Application note: The Modify CRU Register command may be used to change the data being sent to an external device during the debugging of a new interface.

Examples:

```
.MC 1000 8
1000=00FF 0080
1010=00FF 0040
.

.MC 1000
1000=FFFF 1000
```

In the first example, only the eight bits to be modified are displayed. After the data is entered, a space causes the next eight CRU bits to be displayed. The address of the next eight bits is equal to the previous address plus 10_{16} (two times eight bits). In the second example, since the CRU bit width is not specified, a value of 10_{16} is used.

3.4.19 INSPECT CRU INPUT LINES (IC). The Inspect CRU Input Lines command is used to display in hexadecimal format the contents of one or more consecutive CRU locations.

Syntax definition:

$$\text{IC}\left[\begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix}\left[\langle\text{CRU lower limit}\rangle\right]\left[\begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix}\langle\text{CRU upper limit}\rangle\right]\right]$$

The command is terminated by a carriage return.

Parameters:

CRU lower limit   CRU address that begins the display. The address must be in the range of 0 to $1FFF_{16}$.

CRU upper limit   CRU address that ends the display. The address must be in the range 0 to $1FFF_{16}$.

Parameter default values:

If the CRU lower limit is not specified, a value of 0 is used.

If the CRU upper limit is not specified and the CRU lower limit is specified, the default value is the CRU lower limit. Sixteen bits are displayed.

If neither parameter is specified, the entire CRU is displayed.

Description: Data is displayed in groups of four words, two groups per line. The address of the first word on the line is printed on the left. The display may be terminated at any time by pressing the ESC key on the terminal keyboard.

The address displayed is the actual CRU bit address times two.

Error message:

DP13   The highest CRU address specified in less than the lowest CRU address specified, or the highest CRU address specified is greater than the highest CRU address permitted ($1FFF_{16}$).

Examples:

```
.IC 1000 1060
1000=FFFF FFFF FFFF FFFF
.

.IC 100
0100=608D
.
```

In the first example, the CRU bits at addresses 1000_{16} through 1060_{16}, in 20_{16} increments, are displayed. Since the CRU addresses are twice the actual bit addresses, the address of the next 10_{16} CRU bits would be a 20_{16} address increment. In the second example, the 16 CRU bits at location 100_{16} are displayed.

3.4.20 **SET SNAPSHOT (SS).** The Set Snapshot command is used to define a set of registers and memory locations to be displayed as a single unit.

Syntax definition:

$$\text{SS}\ \begin{Bmatrix},\\ \text{ƀ}\ldots\end{Bmatrix}\Big[\text{<snapshot no.>}\Big]\Big[\Big[\begin{Bmatrix},\\ \text{ƀ}\ldots\end{Bmatrix}\Big[\text{<starting reg no.>}\Big]\Big[\Big[\begin{Bmatrix},\\ \text{ƀ}\ldots\end{Bmatrix}\Big[\text{<ending reg no.>}\Big]$$

$$\Big[\Big[\begin{Bmatrix},\\ \text{ƀ}\ldots\end{Bmatrix}\Big[\text{<starting memory addr>}\Big]\Big[\Big[\begin{Bmatrix},\\ \text{ƀ}\ldots\end{Bmatrix}\ \text{<ending memory addr>}\Big]\Big]\Big]\Big]$$

The command is terminated by a carriage return.

Parameters:

snapshot no.	Index number of snapshot to be defined. The index is a number in the range 0-3.
starting reg no.	First workspace register to be displayed.
ending reg no.	Last workspace register to be displayed.
starting memory addr	First memory word address to be displayed.
ending memory addr	Last memory word address to be displayed.

Parameter default values:

If the snapshot number is not specified, a value of 0 is used.

If the starting workspace register number is not specified, a value of 0 is used.

If the ending workspace register number is not specified, the value used is the starting register number if the starting register number is specified. Otherwise, the value is 0_{16}.

If the starting memory address is not specified, a value of 0 is used.

If the ending memory address is not specified, the value used is the starting memory address if the starting memory address is specified. Otherwise, it is 0_{16}.

Description: Snapshots may be invoked with the Inspect Snapshot (IS) command or when a breakpoint which references a snapshot is encountered.

Error messages:

DP03	A parameter is greater than the required maximum value. Reenter the command.
DP04	Snapshot is already defined. Reenter the command.
DP13	The ending parameter (register or memory address) is less than the beginning parameter.

Application notes: Snapshots are convenient for defining a frequently used display during a debug session. If certain registers or memory data areas are frequently modified, they are likely choices for snapshots.

Since a snapshot may be attached to a PC breakpoint to dump some data and continue execution, a trace can be constructed which will be activated only when some specified event occurs. A dump may be produced and execution will continue without operator intervention.

Snapshots are useful for extended traces when the user wants to leave the computer running with breakpoints established. This would allow the computer to take an automatic dump when an exceptional condition is encountered and then continue execution.

Examples:

```
.SS 1,2,5,1000,1002

.SS 0,0,F
```

In the first example, the snapshot associated with index 1 displays workspace registers 2 through 5 and memory locations 1000_{16} through 1002_{16}. In the second example, the snapshot associated with index 0 displays workspace registers 0 through F_{16} and memory address 0 (the default). Refer to the IS command examples in paragraph 3.4.21 for the corresponding commands.

3.4.21 INSPECT SNAPSHOT (IS). The Inspect Snapshot command is used to display sequences of workspace registers and memory addresses.

Syntax definition:

$$\text{IS}\ \Bigg[\begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix} \Bigg[\text{<starting snapshot no.>}\Bigg]\Bigg] \Bigg[\begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix} \text{<ending snapshot no.>}\Bigg]\Bigg]$$

The command is terminated by a carriage return.

Parameters:

starting snapshot no. Index number (number of the snapshot in sequence) of the first snapshot to be displayed. A number from 0 to 3.

ending snapshot no. Index number of the last snapshot to be displayed. A number from 0 to 3.

Parameter default values:

If neither the starting snapshot number nor the ending snapshot number is specified, all snapshots are displayed.

If the starting snapshot number but not the ending snapshot number is specified, the named snapshot is displayed.

If the ending snapshot number but not the starting snapshot number is specified, the snapshots from 0 through the specified snapshot are displayed.

Description: Snapshots are defined with the Set Snapshot command. Attempts to display undefined snapshots are ignored.

Error message:

DP13 Either the ending snapshot number is greater than the starting snapshot number, or a snapshot number greater than the permitted maximum was input. Reenter the command with the correct snapshot numbers.

Examples:

```
.IS
SNAP0
R0=0000 R1=0000 R2=0000 R3=0000 R4=0007 R5=0089 R6=0000 R7=0000
R8=0000 R9=0000 RA=0000 RB=0000 RC=0000 RD=0000 RE=0000 RF=0000
0000=0000
SNAP1
R2=0000 R3=0000 R4=0007 R5=0089
1000=0001 0003
.

.IS 1,3
SNAP1
R2=0000 R3=0000 R4=0007 R5=0089
1000=0001 0003
.

.IS 3
.
```

The snapshots in these examples were set in the examples of the Set Snapshot command (paragraph 3.4.20). In the last example, if a snapshot is not set, the monitor will return control without printing anything.

3.4.22 CLEAR SNAPSHOT (CS).

The Clear Snapshot command is used to disable previously specified snapshots.

Syntax definition:

$$\text{CS}\left[\begin{Bmatrix}, \\ \text{ƀ}\ldots\end{Bmatrix}\left[\langle\text{starting snapshot}\rangle\right]\left[\begin{Bmatrix}, \\ \text{ƀ}\ldots\end{Bmatrix}\langle\text{ending snapshot}\rangle\right]\right]$$

The command is terminated by a carriage return.

Parameters:

starting snapshot	The first snapshot to be cleared. A number from 0 to 3.
ending snapshot	The last snapshot to be cleared. A number from 0 to 3.

Parameter default values:

If no parameters are specified, all snapshots are cleared.

If only the first parameter is given, only the specified snapshot will be cleared.

If only the second parameter is given, snapshot 0 through the specified ending snapshot will be cleared.

Description: If an attempt is made to clear a snapshot that has not been set, the command is ignored.

Error message:

DP13	A snapshot index greater than the maximum possible index number (3) was specified, or the ending snapshot index was less than the starting snapshot index number.

Examples:

```
.CS 0,2

.CS 2
```

In the first example, all snapshots except index number 3 are cleared. In the second example, only snapshot 2 is cleared.

3.4.23 SET BREAKPOINT (SB). The Set Breakpoint command is used to define a breakpoint which causes the processor to stop or interrupt execution of a user program at a specified instruction.

Syntax definition:

$$\text{SB} \left\{\begin{matrix} , \\ \text{ƀ...} \end{matrix}\right\} \text{<bkpt no.>} \left\{\begin{matrix} , \\ \text{ƀ...} \end{matrix}\right\} \text{<memory addr>} \left[\left\{\begin{matrix} , \\ \text{ƀ...} \end{matrix}\right\} \left[\text{<ref cnt>}\right] \left[\left\{\begin{matrix} , \\ \text{ƀ...} \end{matrix}\right\} \text{<snapshot no.>}\right]\right]$$

The command is terminated by a carriage return.

Parameters:

bkpt no. — Breakpoint index number. The number may be 0, 1, 2 or 3. Required parameter.

memory addr — Address of an instruction on which the breakpoint is to be set. Required parameter.

ref cnt — The pass number (hexadecimal) on which a breakpoint is to be taken. For example, a reference count of 3 means to break on the third reference to the memory address for an instruction fetch.

snapshot no. — Index number of a previously defined snapshot which is to be displayed when the breakpoint is taken.

Parameter default values:

If the reference count (pass number) is not specified, a value of 1 is used. If the user enters a value of 0, it is equivalent to a reference count of $FFFF_{16}$.

If the snapshot number is not specified, a snapshot is not printed.

Use of breakpoints: The breakpoint is one of the key elements in program debugging because it enables the user to specify conditions under which he wants to receive control. Breakpoints are particularly useful when the user wants to intercept control after an unexpected control transfer occurs from a conditional branch. By setting a breakpoint on the unexpected or error path out of a conditional branch, the program may be allowed to execute without interruption unless some error condition occurs.

When a breakpoint is encountered, the contents of the processor registers are displayed. (The contents are the values that would be displayed if an IR command were to be invoked.) The breakpoint index number is also displayed to aid in determining which breakpoint was encountered.

If an attempt is made to set a breakpoint on an address outside the allowed range, the command is ignored.

Error message:

DP20 Breakpoint specification error. Required index number may be invalid or missing, or the PC value (memory address) may have been omitted.

Application notes: The PC value for a breakpoint must point to the first word of a multiword instruction.

A breakpoint occurs *before* the execution of the instruction to which it points.

If a snapshot is associated with a breakpoint, execution of the user program resumes after the snapshot is printed. If no snapshot is associated with the breakpoint, execution terminates and PX9MTP accepts another command.

If more than one breakpoint is associated with a specific location, only the first (lowest numbered) will be found.

If (1) the execution is under the control of the Execute User Program under SIE or Trace (RU) command with an instruction count, (2) a breakpoint occurs, and (3) a new count is not specified on the next RU command, then, when execution is resumed, counting is continued as if no breakpoint was encountered.

Breakpoints are not active when the user code is executed with the EX command.

An error is not reported when a Set Breakpoint (SB) command redefines an already defined breakpoint. The specified breakpoint is modified to take on the new definition.

When an instruction has been fetched from a breakpoint location a number of times equal to the contents of the reference counter, the breakpoint is activated.

Examples:

```
.SB 0,1000,1,2

.SB 1,1000,1,0

.SB 2,1004
```

The first two examples set a breakpoint at address 1000 on the first reference to that address for an instruction fetch. The first example sets breakpoint index number 0 with snapshot index number 2 to be displayed, and the second example sets breakpoint index number 1 with snapshot index number 0 to be displayed. The third example specifies breakpoint index number 2 to be taken at memory location 1004_{16}. No snapshot is printed, and execution of the user program terminates after the breakpoint is encountered.

3.4.24 CLEAR BREAKPOINT (CB). The Clear Breakpoint command is used to disable previously specified breakpoints.

Syntax definition:

$$\mathrm{CB}\left[\begin{Bmatrix} , \\ \text{ƀ}\ldots \end{Bmatrix} \left[\langle\text{starting breakpoint}\rangle\right] \left[\begin{Bmatrix} , \\ \text{ƀ}\ldots \end{Bmatrix} \langle\text{ending breakpoint}\rangle\right]\right]$$

The command is terminated by a carriage return.

Parameters:

starting breakpoint — The first breakpoint to be cleared. A number from 0 to 3.

ending breakpoint — The last breakpoint to be cleared. A number from 0 to 3.

Parameter default values:

If no parameters are specified, all breakpoints are cleared.

If only the first parameter is given, only the specified breakpoint will be cleared.

If only the second parameter is given, breakpoints 0 through the specified ending breakpoint will be cleared.

Description: If an attempt is made to clear a breakpoint that has not been set, the command is ignored.

Error message:

DP13 — A breakpoint index greater than the maximum possible index number (3) was specified, or the ending breakpoint index was less than the starting breakpoint index number.

Examples:

```
.CB 1,3

.CB
```

The first example clears all breakpoints except number 0. The second example clears all breakpoints.

3.4.25 SET TRACE DEFINITION (ST). The Set Trace Definition command defines parameters that determine what information about instruction trace regions will be printed. This command is implemented as a service routine on the instruction trace overlay module.

Syntax definition:

ST $\begin{Bmatrix} , \\ \text{ƀ}\ldots \end{Bmatrix}$ <format index> $\begin{Bmatrix} , \\ \text{ƀ}\ldots \end{Bmatrix}$ <char string>

The command is terminated by a carriage return.

Parameters:

format index — Trace format index number; a number from 0 to 3.

char string — Character string describing the options to be printed. The string contains from 1 to 27 characters.

Parameter default values: There are no default values. Both parameters are required.

Character string symbols: The character string symbol definitions and the associated trace printouts are as follows:

Character	Trace Output	Description
P	XXXX	Program counter. The program counter is printed for every instruction executed. The program counter value is printed if anything else is printed even if "P" was not specified (example 1).
I	F-IIII	Instruction and format. (Instruction formats are described in the *Model 990 Computer TMS9900 Microprocessor Assembly Language Programmer's Guide,* Manual No. 943441-9701.) The instruction and its format are printed for each instruction executed (example 2).
M	ST=XXXX	Status mask. The contents of the status mask which is placed in the user status register is printed after each instruction executed (example 2).
W	WP=XXXX	Workspace pointer changes. When the user's workspace changes, the new workspace is printed.
T	BT=XXXX	Targets for branch or jump instruction. Whenever a branch or jump occurs, the target address of the branch/jump is printed.
C	C=XXXX	CRU address. When one of the instructions that references the CRU (LDCR, STCR, TB, SBO, SBZ) is executed, the address of the first bit referenced is printed. For example, for TB 2, the address is base (=R12) + 2.
N	(null)	Null trace. No printout occurs. If any other characters occur in the string, the null trace is overridden.
X	X-XXXX	XOP level. When an XOP instruction is executed, the XOP level is printed.
S		Source. Refers to the source register. It is followed by an E, B, A or R.
E	SE=XXXX	Source effective address. This address is the memory location that the source field addresses. It is printed for every instruction (example 2) that has a source operand.
B	SB=XXXX	Contents of source effective address before execution. The contents of the source effective address before execution are printed for every instruction (example 2) with a source operand.
A	SA=XXXX	Contents of source effective address after execution. The contents of the source effective address are printed after each instruction with a source operand is executed (example 2).
R	SR=XXXX	Contents of source workspace register after execution for T_s = 3 (indirect addressing with autoincrement). (T_s is the source addressing mode field in an assembly language machine instruction.) The contents of the source register is printed if an autoincrement is specified.

Character	Trace Output	Description
D		Destination. Refers to the destination. It is followed by an E, B, A or R.
E	DE=XXXX	Destination effective address. This address is the memory address that the destination field addresses. The destination effective address is only printed for Format 1, 3, and 9 assembly language machine instructions. All other instruction format types do not have a destination field (example 2).
B	DB=XXXX	Contents of destination effective address before statement executed. This is printed whenever a destination field exists (example 2).
A	DA=XXXX	Contents of destination effective address after execution. This is printed whenever a destination field exists (example 2).
R	DR=XXXX	Contents of destination workspace register after execution for T_d = 3 (indirect addressing with autoincrement). (T_d is the destination addressing mode field in an assembly language machine instruction.) The contents of the destination register is printed if an autoincrement is specified.

Description: The character string is scanned for proper syntax. If the string conforms to the syntax, a trace print control template is built and placed in the trace format table.

The character string in the ST command allows the user to select only those portions of the trace output that he needs. For tutorial purposes, an extensive trace output could be requested, while minimal traces such as a PC or variable trace are also easily selected. Each character in the character string represents a desired portion of the trace.

If any trace option other than PC is printed, PC is also printed.

A variable trace (paragraph 3.4.26) is implemented by specifying the desired variable.

The character string is scanned from left to right. The characters E, B, A and R are modified by the most recent occurrence of S or D. If E, B, A or R is encountered before an occurrence of S or D, or if an invalid character is encountered, the scan is aborted and an invalid syntax message is issued. A character string consisting entirely of S or D is also an invalid syntax.

All four trace format table elements have initial values as follows when the debug monitor overlay containing the ST command is loaded:

Index Number	Equivalent Character String
0	P
1	PIWSEADEA
2	T
3	PIMWTCXSEBARDEBAR (all trace output options)

Error messages:

DP23 Syntax error in trace format character string. Reenter the command.

DP26 Invalid trace format index number. Reenter the command.

Examples of typical character strings: Some examples of typical character strings are presented here. To invoke a PC trace, the character string is

```
P
```

If a branch trace is desired, the character string is

```
T
```

The character string for a trace that includes PC, instruction and format, workspace pointer changes, and source and destination effective addresses is

```
PIWSEDE
```

To specify all options, the character string is the same as the string equivalent to default trace format index number 3 (above).

Example 1: Trace format 1 in the following example is defined as a program counter trace. The program counter is the only option printed.

```
.ST 1,P
.SR 1,0,2000,1,N
.MR

PC=198C  46C
.RU
046C
0470
0474
1A92
1A96
198C
198E
1992
1994
1996
.
```

Example 2: This example shows the trace format index number 1 set to a full trace.

```
.ST 1,PIMWTCXSEBARDEBAR
.SR 1,24C,260,1,S
.MR
```

```
PC=0250  24C
.RU
024C  8-02E0   ST=0000   SE=00A6   SB=024C   SA=024C
0250  6-04E0   ST=0000   SE=01FC   SB=0054   SA=0000
0254  6-04E0   ST=0000   SE=01B4   SB=C259   SA=0000
0258  6-04E0   ST=0000   SE=01B8   SB=C060   SA=0000
025C  6-0720   ST=0000   SE=01BA   SB=01E6   SA=FFFF
0260  1-C820   ST=C000   SE=021E   SB=109A   SA=109A   DE=00D2
      DB=1850  DA=109A
```

3.4.26 SET TRACE REGION (SR). The Set Trace Region command defines a trace region. This command must be loaded into the transient area with the OV command before it can be executed.

Syntax definition:

$$\text{SR} \begin{Bmatrix} , \\ \text{ƀ...} \end{Bmatrix} \text{<region index>} \begin{Bmatrix} , \\ \text{ƀ...} \end{Bmatrix} \text{<lower mem addr>} \begin{Bmatrix} , \\ \text{ƀ...} \end{Bmatrix} \text{<upper mem addr>}$$

$$\begin{Bmatrix} , \\ \text{ƀ...} \end{Bmatrix} \text{<format index>} \left[\begin{Bmatrix} , \\ \text{ƀ...} \end{Bmatrix} \left[\text{<step region>}\right] \left[\begin{Bmatrix} , \\ \text{ƀ...} \end{Bmatrix} \text{<v1>} \left[\begin{Bmatrix} , \\ \text{ƀ...} \end{Bmatrix} \text{<v2>}\right.\right.\right.$$

$$\left.\left.\left.\left[\begin{Bmatrix} , \\ \text{ƀ...} \end{Bmatrix} \text{<v3>}\right]\right]\right]\right]$$

The command is terminated by a carriage return.

Parameters:

region index — Trace region index number; a number from 0 to 3.

lower mem addr — First memory address in the trace region; a hexadecimal number in the range 0 to FFFE.

upper mem addr — Last memory address in the trace region; a hexadecimal number in the range 0 to FFFE.

format index — Trace format index number; a number from 0 to 3.

step region — If this field contains S, an instruction step region is specified. If it contains N, the field specifies no instruction step. Any other character specifies no instruction step.

v1, v2, v3 — Addresses of variables to be traced while in the designated region. Up to three variables may be specified. The range of values for each variable is 0 to $FFFE_{16}$. In the printed trace data, only changes are shown.

Parameter default values:

The first four parameters in the syntax definitions are required.

If the step region parameter is not specified, a value of N is used.

If none of the parameters v1, v2, and v3 are specified, no variables will be traced in the designated region.

Description: The specified regions of memory are designated as the program area to be executed under control of the interpretive trace.

The trace region index number determines which trace type will be executed as defined by the Set Trace Definition (ST) command. If two overlapping regions have been defined, the region with the lowest index has precedence and the trace type defined in that region is executed. (See example 1.)

The trace format index number indicates the trace type vector assigned to the trace region. When the trace overlay is loaded, each of the four trace type vectors, indices 0 through 3, is assigned an initial value. These vectors may be modified by the Set Trace Definition (ST) command. Trace types may vary from a null trace to a full trace.

The function of the instruction step region is to control the execution of the user program. If the instruction step region is set by entering an S parameter on the terminal keyboard, only one instruction at a time will be executed and traced. To execute another instruction, the user must press the space bar.

If variables have been specified to be traced, only changes will be printed. The format of the output is:

```
AAAA = DDDD
```

Where AAAA is the address of the variable and DDDD is the new value of the variable. These are hexadecimal values.

Error messages:

DP13	The specified last memory address was less than the first memory address. Reenter the command.
DP10	Invalid trace region index number. Reenter the command.
DP26	Invalid trace format index number. Reenter the command.

Example 1: This example shows the setting of two different trace regions, one a PC trace and the other a full trace. The region with the lower index is executed when the two regions overlap. In this manner, the user can get a general trace until he reaches a critical section of the program where he wants everything traced.

```
.ST 1,PIMWTCXSEBARDEBAR
.ST 2,P
.SR 2,0,2000,2,N
.SR 1,24C,260,1,S
.MR

PC=0250  246
.RU
0246
024A
024C  8-02E0   ST=0000    SE=00A6    SB=024C    SA=024C
0250  6-04E0   ST=0000    SE=01FC    SB=0054    SA=0000
0254  6-04E0   ST=0000    SE=01B4    SB=C259    SA=0000
0258  6-04E0   ST=0000    SE=01B8    SB=C060    SA=0000
025C  6-0720   ST=0000    SE=01BA    SB=01E6    SA=FFFF
0260  1-C820   ST=C000    SE=021E    SB=109A    SA=109A    DE=00D2
      DB=1850   DA=109A
0266
026A
0270
0274
0278
027A
027E
.
```

Outside the critical region, a continuous run is desired. Inside the critical region, there is a single instruction step. The operator must press the carriage return or space bar on the terminal keyboard after each statement executed.

Example 2: The trace region is set from 0 to 2000_{16}, with the trace format index number equal to 3. (Trace type 3 defaults to a full trace.) The snapshot prints workspace registers 1 through 4 and memory locations 1000_{16} to 1004_{16}. A breakpoint is set at 0474_{16} with snapshot 1 associated. A Modify Registers (MR) command sets the program counter to $046C_{16}$, and execution is begun by issuing an Execute User Program under SIE or Trace (RU) command.

```
.SR 1,0,2000,3,N

.SS 1,1,4,1000,1004
.SB 1,474,,1
.MR

PC=198C  46C
.RU
046C  8-02E0   ST=2000    WP=044C    SE=1968    SB=0900    SA=0900
0470  1-C2A0   ST=C000    SE=00A6    SB=1A92    SA=1A92    DE=0460
      DB=0000  DA=1A92
BKPT#1
PC=0474    WP=044C    ST=C000
SNAP1
R1=11C0    R2=0000    R3=0000    R4=0000
1000=10D8    C145     1305
0474  6-045A   ST=C000    BT=1A92    SE=1A92    SB=C2A0    SA=C2A0
1A92  1-C2A0   ST=2000    SE=00A8    SB=0000    SA=0000    DE=0460
      DB=1A92  DA=0000
```

```
1A96  6-0420   ST=2000   WP=1968   BT=198C   SE=1988   SB=1968
      SA=1968
198C  6-04C3   ST=2000   SE=196E   SB=FFFF   SA=0000
198E  1
.
```

Following is a listing of the portion of the program executed in this example with all references resolved:

```
 Memory      Object
Location      Code                   Source

  046C        02E0               LWPI    MAINW
  046E        044C
  0470        C240               MOV     @ENTRY,R10
  0472        00A6
  0474        045A               B       *R10
                .                  .
                .                  .
                .                  .
  1A92        C2A0      INIT     MOV     @KBLUNO,R10
  1A94        00A8
  1A96        0420               BLWP    @OPEN
  1A98        1988
                .                  .
                .                  .
                .                  .
  1988        1968      OPEN     DATA    IOWKS
  198A        198C               DATA    OPEN1
  198C        04C3      OPEN1    CLR     R3
```

This is a typical example using snapshots, breakpoints and an instruction trace. Since a snapshot is associated with the breakpoint, the snapshot is printed and execution continued. An exit from the RU command is made by pressing the ESC key on the terminal keyboard.

3.4.27 **CLEAR TRACE REGION (CR).** The Clear Trace Region instruction is used to disable previously specified trace regions.

Syntax definition:

CR $\left[\begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix}\right] \left[\text{<starting trace region>}\right] \left[\left[\begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix}\right] \text{<ending trace region>}\right]$

The command is terminated by a carriage return.

Parameters:

starting trace region   The first trace region to be cleared.
                         A number from 0 to 3.

ending trace region   The last trace region to be cleared.
                       A number from 0 to 3.

Parameter default values:

If no parameters are specified, all trace regions are cleared.

If only the first parameter is given, only the specified trace region will be cleared.

If only the second parameter is given, trace regions 0 through the specified ending trace region will be cleared.

Error message:

DP13 A trace region index greater than the maximum possible index number (3) was specified, or the ending region index was less than the starting region index number.

Examples:

```
.CR 1,3

.CR
```

In the first example, all but region 0 are cleared. In the second example, all regions are cleared.

3.4.28 **FIND BYTE (FB)**. The Find Byte command is used to scan an area of memory for a particular byte value.

Syntax definition:

$$\text{FB}\left\{\begin{matrix}, \\ \text{ƀ...}\end{matrix}\right\}\left[\text{<start mem addr>}\right]\left\{\begin{matrix}, \\ \text{ƀ...}\end{matrix}\right\}\left[\text{<ending mem addr>}\right]\left\{\begin{matrix}, \\ \text{ƀ...}\end{matrix}\right\}$$

$$\text{<desired value>}\left[\left\{\begin{matrix}, \\ \text{ƀ...}\end{matrix}\right\}\text{<mask>}\right]$$

The command is terminated by a carriage return.

Parameters:

start mem addr Memory address at which search begins.

ending mem addr Memory address at which search is terminated.

desired value Hexadecimal value for which the search is made. This value is required.

mask Hexadecimal value to be ANDed with each byte before comparing it with the desired value.

Parameter default values:

If the starting memory address is not specified, a value of 0 is used.

If the ending memory address is not specified, a value of $FFFF_{16}$ is used.

If the mask parameter is not specified, a value of FF_{16} is used.

Description: Each byte in the memory search range is ANDed with the mask and compared to the desired value. The memory location and contents are printed out whenever a match is found. After each match, the user must enter a space on the terminal keyboard to continue the search. If he enters a carriage return, the command terminates.

Error messages:

DP13 The ending address is less than the starting address. Reenter the command.

MS05 A required parameter, the desired value, is missing. Reenter the command.

MX06 The beginning address is an invalid memory address. Reenter the command.

Application notes: No check is made to ensure that the mask does not exclude a bit required by the desired value, thereby making a match impossible. If the monitor is being searched, results may not appear to be correct since the monitor is changing during the search process.

Examples:

```
.FB 0,2000,0,0F
0000=0000
0000=0000
0002=0000
0002=0000
0004=0000
0004=0000
0006=0000
0006=0000
0008=0000
.

.FB 0,2000,06,0F
0300=0456
0644=0556
.
```

In the first example, the high order four bits of each byte are masked so that any byte with a 0 in the low order four bits will be located. The address of the leftmost byte of each word is printed so that if both bytes of a word are printed, an address location will be printed twice. For example, if bytes 0004 and 0005 are printed, the address 0004 will appear twice in the listing.

In the second example, the high order four bits of each byte are masked so that any byte with a 6 in the low order four bits will be located.

3.4.29 FIND WORD (FW). The Find Word command is used to scan an area of memory for a particular word value.

Syntax definition:

$$\mathrm{FW}\ \begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix} \left[\langle\text{start mem addr}\rangle\right] \begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix} \left[\langle\text{ending mem addr}\rangle\right] \begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix}$$

$$\langle\text{desired value}\rangle \left[\begin{Bmatrix} , \\ \not{b}\ldots \end{Bmatrix} \langle\text{mask}\rangle\right]$$

The command is terminated by a carriage return.

Parameters:

start mem addr — Memory address at which search begins.

ending memory addr — Memory address at which search is terminated.

desired value — Hexadecimal value for which the search is made. This value is required.

mask — Hexadecimal value to be ANDed with each word before comparing it with desired value.

Parameter default values:

If the starting memory address is not specified, a value of 0 is used.

If the ending memory address is not specified, a value of FFFF_{16} is used.

If the mask parameter is not specified, a value of FFFF_{16} is used.

Description: Each word in the memory search range is ANDed with the mask and compared to the desired value. The memory location and contents are printed out whenever a match is found. After each match, the user must enter a space on the terminal keyboard to continue the search. If he enters a carriage return, the command terminates.

Error messages:

DP13 — The ending address is less than the starting address. Reenter the command.

MP00 — The beginning address is an invalid memory address. Reenter the command.

MS05 — A required parameter, the desired value, is missing. Reenter the command.

Application notes: No check is made to ensure that the mask does not exclude a bit required by the desired value, thereby making a match impossible. If the monitor is being searched, results may not appear to be correct since the monitor is changing during the search process.

Examples:

```
.FW 0,2999,456,
0300=0456
.FW 0,2000,56,00FF
0300=0456
0644=0556
.
```

In the second example, the monitor searches for words with a 56 in the low order byte. By pressing the space bar on the terminal keyboard, the user can cause the monitor to continue searching for another occurrence of the data word.

3.4.30 **HEXADECIMAL ARITHMETIC (HA).** The Hexadecimal Arithmetic command calculates the sum and difference of two hexadecimal numbers. The 2's complement hexadecimal value and the signed decimal value are printed.

Syntax definition:

```
HA [{,|b̸...} [<value>] [{,|b̸...} <value>]]
```

The command is terminated by a carriage return.

Parameters:

value Hexadecimal number value.

Parameter default values:

If the value parameter is not specified, a default value of 0 is used.

Application note: No overflow checks are made; therefore, two positive numbers may have a negative sum. All results are represented in 16 bits.

Examples:

```
.HA 103A BA2
SUM=1BDC +07132 DIFF=0498 +01176
.

.HA 89 89
SUM=0112 +00274 DIFF=0000 +00000
.
```

```
.HA 8030 EF00
SUM=6F30  +28464  DIFF=9130  -28368
.

.HA EF00 8030
SUM=6F30  +28464  DIFF=6ED0  +28368
.
```

The calculated difference between the specified number values is the first value minus the second value.

3.4.31 **SET WRITE PROTECT REGION (SP).** The Set Write Protect Region command sets the write protect region to the address specified in the command.

Syntax definition:

SP {ƀ’. . .} <lower mem addr> {ƀ’. . .} <upper mem addr>

The command is terminated by a carriage return.

Parameters:

lower mem addr   Lower boundary memory address of the protected region. Required parameter. Hexadecimal number.

upper mem addr   Upper boundary memory address of the protected region. Required parameter. Hexadecimal number.

Description: This command sets the write protect region from the lower to the upper memory bound addresses. If the memory addresses entered are not on 256-word boundaries, the bounds will be set at the next lower 256-word boundary. The lower bound is included within the protect region but the upper bound is not.

The SP command overrides any previously defined protect region.

When the upper and lower bounds are sent to the CRU, the Protect Violation flag is cleared if it has been set.

Error message:

MS05   Parameter specification error. Either a required parameter is missing, or the lower bound is greater than or equal to the upper bound.

Examples:

```
.SP 1000,2000
```

This command protects a region in memory from 1000_{16} to $1FFF_{16}$.

```
.SP 1000,1F00
```

This command protects a region from 1000_{16} to $1DFF_{16}$. The address $1F00_{16}$ is not a 256-word boundary; therefore, the upper bound is set at the next lower 256-word boundary, 1E00.

3.4.32 CLEAR WRITE PROTECT REGION (CP). The Clear Write Protect Region command clears the protect register and removes protection from the write-protected region.

Syntax definition:

```
CP
```

The command is terminated by a carriage return.

Description: The CP command clears the Protect register and sets the Protect/Permit bit to Permit. The Protect Violation flag is cleared if it has been set.

Example:

```
.CP
```

This command clears a write-protected region set previously with an SP command.

3.5 SUPERVISOR CALLS

Supervisor calls are used to:

- Request all monitor I/O operations.
- Perform frequently used services in the form of monitor routines.

3.5.1 INTRODUCTION. The following paragraphs explain invocation of a supervisor call, coding of supervisor calls, types of supervisor calls, and data block formats.

A supervisor call is made with an XOP assembly language machine instruction, using an extended operation code of 15. The XOP instruction specifies an address pointing to a multiple byte block containing the supervisor call and any necessary arguments.

The individual supervisor calls and their operation codes are listed in table 3-2.

Table 3-2. List of Supervisor Calls

Supervisor Call	Supervisor Call Code (Hexadecimal)	I/O Physical Record Block Operation Code (Hexadecimal)
I/O – Open	0	0
I/O – Read ASCII	0	9
I/O – Write ASCII	0	B
I/O – Write End of File	0	D
End of Program	4	–
Binary to Decimal ASCII	A	–
Decimal ASCII to Binary	B	–
Binary to Hexadecimal ASCII	C	–
Hexadecimal ASCII to Binary	D	–

3.5.2 **I/O SUPERVISOR CALLS.** The data block for an I/O supervisor call consists of the following two blocks (contiguous on a full-word boundary):

- A two-byte block that specifies a zero for an I/O call in the first byte and has the second byte set to 0.
- A seven-word control block, called a Physical Record Block (PRB). This control block specifies the type of I/O operation to be performed and the input and output parameters.

The format of the physical record block is as follows:

0 1ST BYTE 7	8 2ND BYTE 15
<i/o op>	<luno>
<sys flags>	<user flags>
<buffer addr>	
<buffer length>	
<char count>	

The parameters are:

i/o op — The I/O operation requested

luno — The logical unit to which I/O is to be performed.

sys flags — Flags indicating the status of a completed I/O operation:

Bit 0 – reserved.

Bit 1 – unrecoverable I/O error.

Bit 2 – end of file was encountered. (The character count indicates whether any data was transferred.)

user flags — Flags indicating additional processing requirements. Bit 3 is the character I/O flag. Character I/O applies only to the logging device (data terminal) and is ignored in cassette I/O. One character at a time will be read to or printed on the logging device. If the character I/O bit is set, any RUB OUT or backspace character encountered in a read from the logging device will be placed in the user's buffer.

buffer addr — The absolute memory address of the start of an I/O buffer.

buffer length — The maximum number of characters which may be input.

char count — The number of characters actually input or output.

An I/O supervisor call may be coded in assembly language as follows:

```
        XOP     @IOC, 15
         .
         .
         .
IOC     BYTE    0,0
PRB     BYTE    9,7       Read from LUNO 7
        DATA    0         System flags/user flags
        DATA    BUFADR    Buffer address
        DATA    80        Buffer length
        DATA    0         Character count
```

3.5.2.1 Open. The open supervisor call forces a playback/record initialization (to allow a change of mode if the mode is incorrect) of a tape cassette.

Supervisor call code: 0

I/O operation code: 0

Calling parameters:

luno — Logical unit number of the drive on which the tape cassette is mounted.

Result: The Open supervisor call is ignored except by the tape cassette, for which it forces a playback/record initialization. When a LUNO outside the range 0 to F_{16} is specified, the command is ignored.

3.5.2.2 **Read ASCII.** The Read ASCII supervisor call reads ASCII data from an input device.

Supervisor call code: 0

I/O operation code: 9

Calling parameters:

luno	Logical unit number of a device from which data is to be read.
buffer addr	Absolute memory address of the first byte of an input buffer.
buffer length	Maximum length of the input buffer.
char count	The number of characters actually transferred. This value is returned by the supervisor.

Result: Data is read from the specified device until either the buffer length is satisfied or a terminating event such as a carriage return occurs. A read from a dummy device will cause the end-of-file flag to be set.

Errors: An unrecoverable I/O error is returned if:

- An I/O error occurs.
- The output cassette is not ready.

3.5.2.3 **Write ASCII.** The Write ASCII supervisor call writes ASCII data to an output device.

Supervisor call code: 0

I/O operation code: B

Calling parameters:

luno	Logical unit number of a device to which data can be written.
buffer addr	Absolute memory address of the first byte of an output buffer.
buffer length	Unused.
char count	The number of characters to be transferred.

Result: Data is written to the specified device until the character count is satisfied. If the character count is greater than 80 when writing to a cassette, only the first 80 characters are written to cassette.

Errors: An unrecoverable I/O error is returned if:

- An I/O error occurs.
- The output cassette is not ready.

3.5.2.4 **Write End of File.** The Write End of File supervisor call writes an end-of-file record to a tape cassette.

Supervisor call code: 0

I/O operation code: D

Calling parameter:

luno Logical unit number of the drive on which the tape cassette is mounted.

Result: This supervisor call causes an end-of-file record to be written to the specified cassette. This call is ignored by other devices.

Error: An unrecoverable I/O error is returned if an I/O error occurs or if the output cassette is not ready.

3.5.3 **NON-I/O SUPERVISOR CALLS.** The data block for a non-I/O supervisor call is a parameter block containing two to eight bytes. It has the following format:

	1ST BYTE	2ND BYTE
0	<op> 03	<error code> 00
2	<sign> OR PART OF <value> 20	20
4	20 <value>	31
6	32	33

USED FOR DATA CONVERSION ONLY

The parameters are:

op Operation code

error code Code which is set to one if an error is encountered

sign Algebraic sign associated with a parameter value – plus (+), minus (-), ASCII zero or blank

value Parameter value

An example of assembly language coding for a non-I/O supervisor call block (decimal ASCII to binary) follows:

```
DAB  BYTE   >B,0    Op code.
     DATA   >2020   Value parameter is right justified
     DATA   >2031   with leading ASCII blanks.
     DATA   >3233
```

3.5.3.1 **End of Program.** The End of Program supervisor call terminates the calling program. The parameter block contains two bytes.

Supervisor call code: 4

Calling argument: 4 (in byte 0)

Result: The calling program terminates. Control returns to PX9MTP.

3.5.3.2 **Binary to Decimal ASCII.** The Binary to Decimal ASCII supervisor call converts binary data to decimal ASCII character code. The parameter block contains eight bytes.

Supervisor call code: A

Calling arguments:

	1ST BYTE	2ND BYTE
0	A	
2	<sign>	<converted values>
4		
6		

Workspace register 0 contains the value to be converted. The sign parameter is set to minus if the value is less than 0 and to a blank if the value is greater than 0. The converted values parameter is the decimal ASCII equivalent of the binary value.

Result: The binary value in workspace register 0 is converted to a signed decimal number (right justified with leading zeros) in the supervisor call parameter block.

3.5.3.3 **Decimal ASCII to Binary.** The Decimal ASCII to Binary supervisor call converts decimal ASCII character code to binary data. The parameter block contains eight bytes.

Supervisor call code: B

Calling arguments:

	1ST BYTE	2ND BYTE
0	B	<error code>
2	<sign>	<value>
4		
6		

The sign parameter may be plus, minus, ASCII zero or a blank. The value parameter is a decimal ASCII value between -32,768 and +32,767, inclusive. The value parameter must be right justified with leading ASCII zeros or blanks. The result is returned in the caller's workspace register 0.

Result: The decimal ASCII value in the supervisor call block is converted to a 2's complement value in the caller's workspace register 0.

Error code: The error code is set to one if there is an invalid character or if the resultant value is outside the range -32,768 to +32,767.

3.5.3.4 Binary to Hexadecimal ASCII. The Binary to Hexadecimal ASCII supervisor call converts binary data to hexadecimal ASCII character code. The parameter block contains six bytes.

Supervisor call code: C

Calling arguments:

	1ST BYTE	2ND BYTE
0	C	
2	<value>	
4		

Workspace register 0 contains the value to be converted. The value parameter is the converted hexadecimal ASCII value.

Result: The value in workspace register 0 is converted to the corresponding ASCII representation in the supervisor call block.

3.5.3.5 Hexadecimal ASCII to Binary. The Hexadecimal ASCII to Binary supervisor call converts hexadecimal ASCII character code to binary data. The parameter block contains six bytes.

Supervisor call code: D

Calling arguments:

	1ST BYTE	2ND BYTE
0	D	<error code>
2	<value>	
4		

The value parameter is four hexadecimal ASCII characters. The result is returned in the caller's workspace register 0.

Result: The four-character hexadecimal value in the supervisor call block is converted to binary in the caller's workspace register 0.

Error code: The error code is set to one if any character is an invalid hexadecimal digit.

3.6 DEBUGGING TECHNIQUES

Debugging techniques may be divided into three basic categories:

1. *Preventive techniques* – those which may be used to decrease the number of errors. Most of these techniques emphasize simplicity. Code should be simple and straightforward enough to make it obvious that the program works.

2. *Exposure techniques* – those which may be used to make the operation of a program easier to follow during the debugging process.

3. *Remedial techniques* – those used when a bug occurs in the user's program. Typically, most programmers' efforts are expended on these techniques.

Programming effort devoted to avoiding errors or making them apparent is important. Debugging and maintenance represent the majority of the cost in software development and support. The following paragraphs briefly discuss debugging in general and the specifics of debugging under PX9MTP.

3.6.1 GENERAL DEBUGGING TECHNIQUES. Several debug techniques will be helpful to the programmer in any debugging situation. These paragraphs offer some suggestions about debugging a program under development.

3.6.1.1 Debug Code in the Source Program. Include debug code in the source program. The user should keep the testing process in mind from the moment he starts to create a program. When referencing or changing data, the programmer should consider how to tell if the change is correct when reconstructing the results of a run. This often involves being aware of what intermediate results of a computation are lost.

For example, if the value of a variable D is calculated by the statement

```
D = A + B
```

and the program later encounters the statement

```
D = C + D
```

the second statement will cause a new value D to replace the previously calculated value. The calculated sum A + B will therefore be lost. If, on the other hand, the program contains the statement

E = A + B

and, later in the program, the statement

D = C + E

the value of E will be preserved when D is calculated by the second statement. The programmer can examine the memory location containing the value of E to determine the calculated sum A + B.

After a computation is completed, reconstruction of the results of a program run involves distinguishing which decision paths have been taken through the program's code and determining what variables are relevant in calculating the results of a computation.

When the source code is written, it is often simple to store intermediate results in extra memory to record those results, branch paths, or the number of passes through loops. Such statements can be flagged with a character string (e.g., **DEBUG**) in the comment field. When the source code is ready for production, PX9EDT can be used to locate and remove the code that stores intermediate results.

3.6.1.2 **Checking the Program.** Once a program has been successfully assembled, a thorough check of the program can often turn up errors which are hard to detect when the program is executing. In addition to making sure that the program is a correct implementation of the algorithm, it is often worthwhile to read through the program looking for specific errors:

- *Register errors.* Using the wrong register; referencing a register not in the current workspace; using a register as an immediate value (e.g., AI R1,R2 instead of A R1,R2 or AI R1,2); using byte-level operations or data where the data is in the wrong half of the register; or using byte-level data with the other half of the register containing incorrect data which affects the computation.

- *Variable names.* Misspelling of variable names such as T0 and TO; or using a single variable to contain different quantities.

- *Initialization errors.* Referencing values which may not have been properly initialized. This often occurs when a program is re-executed.

- *Buffer initialization.* Omitting an instruction to clear an input buffer between input operations when variable length records are read into a common fixed-length buffer.

- *Branch conditions and loop terminations.* Using the wrong branch instruction (especially JH, JL, JGT, JLE, JLT, JHE, or JOC with subtracts); or executing a loop one time too many or one time too few.

- *Inconsistent techniques.* Using conventions or debug elements which are inconsistent with the coding practice for the module.

- *Module interfaces.* Using variables or parameters which were not correctly set up for an interface; using registers or variables within a subroutine which have values that are not to be changed within the calling routine.

- *Boundary conditions.* Checking that the full range of the possible input data to a computation is correctly processed by the algorithm.

3.6.1.3 Execution Tree. In debugging or testing a program, it is often convenient to visualize the possible paths through the program as a tree with each node of the tree representing a conditional branch. Exhaustive testing of a program would then require testing each possible path through the program under all inputs which follow that path. While it is impossible to test all paths of a typical program, examination of the various paths (or small sets of paths) may reveal errors in the original logic.

3.6.2 SPECIFIC DEBUGGING TECHNIQUES. The following paragraphs describe techniques directed specifically to debugging under the PX9MTP monitor.

3.6.2.1 Planning the Debugging Session. Know the status of the debugging effort at all times. As the user interacts with the program through the console, he should be careful to record any changes made to the program and to be aware of the state of the program when examining it. In a debugging session, the user should have a clear idea of what he wants to accomplish and how he intends to accomplish it. Decisions made in the process of debugging should be carefully thought out.

3.6.2.2 Use of Breakpoints. There are three ways of stopping or interrupting the execution of a user's program which is being debugged at a specific location in the program:

1. Set an instruction count on the RUN command.

2. Execute with the single step option under instruction trace.

3. Set appropriate breakpoints.

Breakpoints stop execution at specific points in the user program rather than at arbitrary points controlled by the instruction count. The user may easily determine in advance and check the results of a computation without concerning himself about the state of the program.

When using breakpoints, be sure that the program will actually reach the desired breakpoint. This may involve putting additional breakpoints on the other paths from conditional branches.

Breakpoints are particularly useful when forcing some condition within a program which is not easily created from its parameters, for example, a CRU input. As an illustration of such a condition, an input value is to be read from a pressure transducer in an on-line process control environment. However, if the program is being debugged, a physically connected transducer is usually impractical and the values must be entered by the programmer. Breakpoints may be set prior to the start of a code sequence. When the breakpoint is taken, the user may set or modify the existing conditions in order to cause specific paths to be taken (as if a specific input had been received from the transducer).

The breakpoint reference count can be used to see that a loop is repeated the correct number of times. By setting the reference count equal to the number of iterations through the loop and setting another breakpoint outside the loop, the user may check that the loop is exhausted on the correct iteration. Breakpoints with attached snapshots with dump debug data or key variables yield a good trace aimed at checking the specific progress of a computation.

3.6.2.3 **Excluding Loops from Instruction Traces.** When tracing a program with printout, it is sometimes desirable to exclude printing of small loops which are very frequently executed or which run for many iterations. (See figure 3-1.) These may be excluded by carefully choosing trace regions, which are areas where an instruction trace is to be run within a program. In determining which trace region is applicable (and thus what trace type to use), the system will find the first (lowest numbered) region containing the user's PC. By selecting a high numbered trace (3) for the main trace control and then setting regions within that large region with lower numbered traces which do not print, the user may prevent a large quantity of output where it is not wanted.

An alternate mechanism is to allow the small loops to be executed by SIE and the remaining program traced. (See figure 3-2.) This can be done by setting trace regions to cover all of the program except the small loops or frequently executed parts. Such a mechanism works well unless the user is using XOPs (other than XOP 15 for PX9MTP I/O) or interrupts which are processed differently by SIE and instruction trace.

If the user is performing I/O by means of supervisor calls (XOP 15) to PX9MTP, this XOP is executed directly (without SIE or instruction trace). If XOP 15 is not used for program I/O, it is not executed directly under SIE.

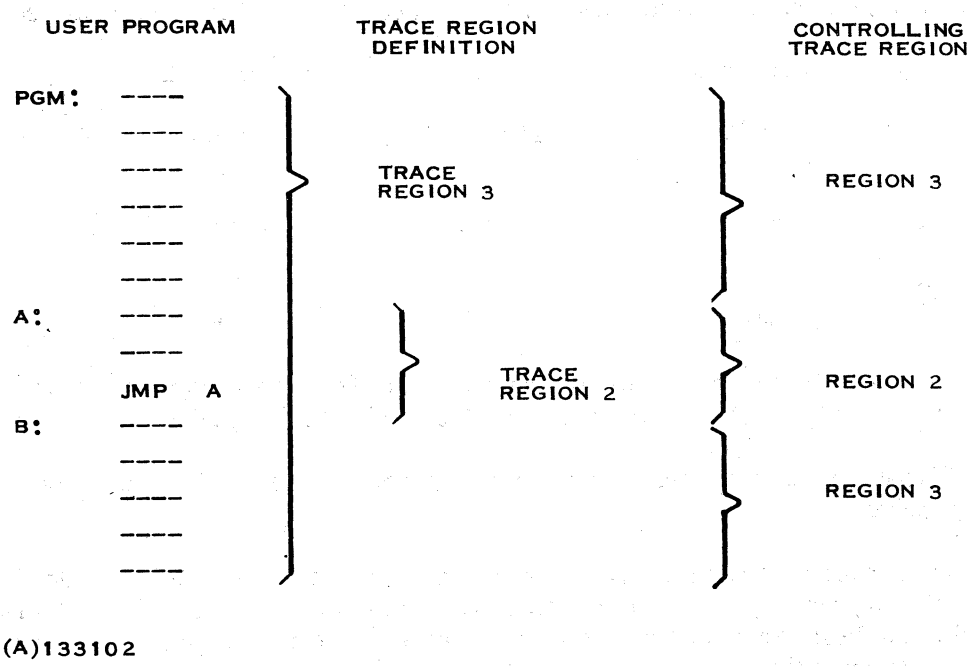


Figure 3-1. Trace Region Precedence of Lower Region Number

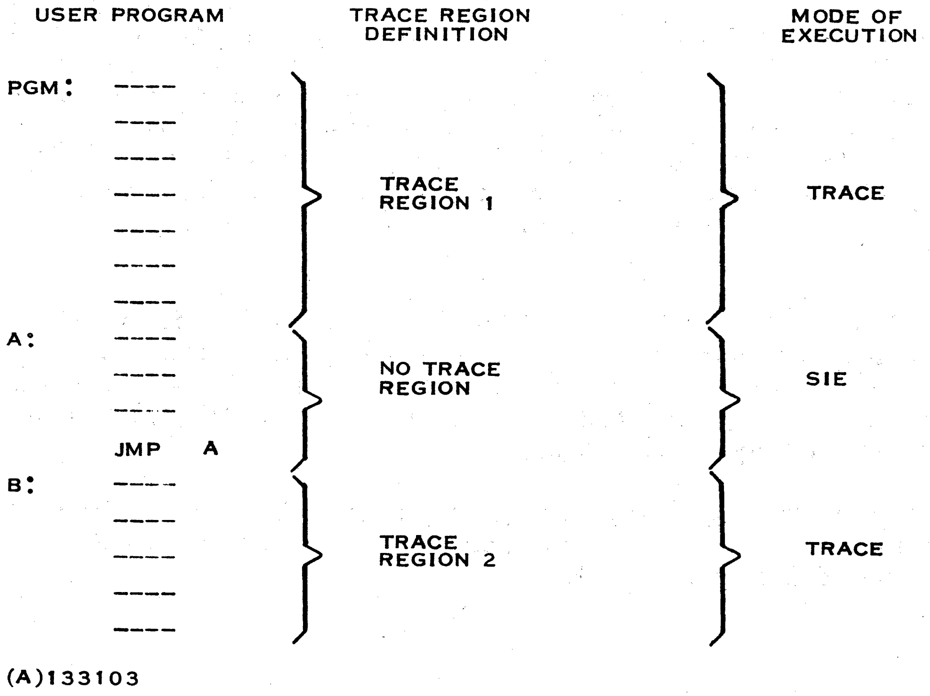


Figure 3-2. Using Both Trace and SIE

3.6.2.4 **Simulating an Interrupt.** A BLWP instruction may be used to control an interrupt routine which is being checked out. This can be handled with the following code sequence:

Instruction	Operand	Generated Code
LIMI	INTLVL	0300 i
BLWP	@INTLVL*4	0420 4*i
JMP	$	10FF

The LIMI sets the interrupt status to the correct level. The BLWP transfers control through the interrupt vector. The quantity i is the value to which INTLVL is equated.

3.6.3 **PATCHING.** Patching (attaching portions of code to existing program code) should be avoided if possible.

During a debug session, it is generally necessary to make patches to object code; however, it is advisable never to leave patches in a completed program (or create ROM firmware from a program with patches). An object program for which there is no corresponding source program is inconvenient and troublesome.

The following paragraphs cover patching techniques. The examples show how to patch a two-address instruction; this instruction is used:

```
MOV *R1,*R2+
```

Because of the number of items to be considered, patching a two-address instruction is one of the more difficult operations. There are two ways to approach it: building a bit image and the additive method.

3.6.3.1 **Patching by Building a Bit Image.** In building a bit image, the user merely fills in each field in the 16-bit word on a bit-by-bit basis. When all fields are complete, the value is converted to hexadecimal for the patch contents.

Example:

Patch the following assembly language instruction:

```
MOV *R1,*R2+
```

by building a bit image.

The MOV instruction has this format:

OP CODE	B	T_d	D	T_s	S

Determine the bits that occupy each field. Starting with the op code field, the hexadecimal op code for a MOV instruction is C000. The first three bits of this op code are 110_2; transfer these bits into the op code field.

The Byte Indicator (B) field specifies whether or not the instruction is a byte instruction. The MOV instruction is a word instruction; therefore, this field is set to 0. (The B field is always 0 for a MOV instruction.) Another way of specifying the same information would be to use the MOV or MOVB instruction (as appropriate) and a four-bit op code.

The D field specifies the destination workspace register. The destination address is *R2+, which indicates workspace register 2 and the workspace register indirect autoincrement addressing mode. The addressing mode for the destination, 11_2, is placed in the T_d field. Transfer the binary value of the register number, 0010_2, into the D field.

Use a similar procedure for the source address, which is *R1. In this case, workspace register 1 is specified and the addressing mode is workspace register indirect. Therefore, transfer 01_2 into the T_s field and 0001_2 into the S field.

The instruction field contents will now be:

110	0	11	0010	01	0001

Now read these 16 bits as a four-digit hexadecimal number.

1100	1100	1001	0001
C	C	9	1

The resulting hexadecimal number is the desired value. The patch value is CC91.

3.6.3.2 **Patching by the Additive Method.** The second approach to the patching problem is the additive method. With a little practice, the patch described in the first approach can be created a little faster by treating each of the fields as a hexadecimal number and adding the results to produce the patch.

Example:

Patch the same assembly language instruction as in the bit image example:

```
MOV *R1,*R2+
```

by using the additive method. This method involves adding hexadecimal values corresponding to each field to the instruction's op code to get the patch value.

The programmer can think of a bit field value as being placed into the instruction word, right justified, and shifted left the number of bits necessary to move it to the appropriate field. This shift is equivalent to binary multiplication, so the bit field value times an appropriate multiplier will give a value to be added to similarly obtained values for other bit fields to yield a sum representing the contents of the instruction word.

Recall that the values for the addressing modes and workspace registers in the previous examples were:

Destination mode (T_d)	3
Destination register (D)	2
Source mode (T_s)	1
Source register (S)	1

In calculating the patch value by the additive method, these values are used.

The first number in the calculation is the hexadecimal op code for the MOV instruction, C000. The B field is always 0 in the MOV instruction; it can be considered part of the instruction op code and ignored in the calculation.

The second number to be added is the value of the destination mode. The code for the address mode is shifted left ten bits, equivalent to multiplication by 400_{16}. The code is 3_{16}; therefore, the value to be added is

$$3_{16} * 400_{16} = 0C00_{16}$$

The third number is the destination register value. To create the value to be added, the register number, 2_{16}, is shifted left six bits, equivalent to multiplication by 40_{16}. The value is

$$2_{16} * 40_{16} = 0080_{16}$$

Calculation of the fourth value involves a code of 1_{16} for the source mode and a four-bit shift (multiplication by 10_{16}). The value is

$$1_{16} * 10_{16} = 0010_{16}$$

Finally, the source register number, 1_{16}, is unshifted. The value to be added is 0001_{16}.

To calculate the required sum, the values are added:

Op code of MOV instruction	C000
Destination mode	0C00
Destination register	0080
Source mode	0010
Source register	0001
Patch value	CC91

The sum, $CC91_{16}$, is the object code to be patched. The patch value is the same as the value obtained in the previous example.

When the same instruction format is used repeatedly, the multiplication constants – 400_{16}, 40_{16} and 10_{16} – do not change and become simple to handle with practice.

3.6.3.3 **Symbolic Versus Indexed Addressing.** The address mode for both symbolic (actual memory address) and register indexed addressing is the same (mode 10_2). The type of addressing is determined by the register field. A register field of zero is symbolic; therefore, no R0 indexing exists. In constructing a patch with a specific address, process it exactly as if it were a register indexed with a register of zero. Refer to the *Model 990 Computer TMS9900 Microprocessor Assembly Language Programmer's Guide*, Manual No. 943441-9701, for further information about symbolic and indexed memory addressing.

3.6.3.4 **Branch Distance Calculations for Jump Instructions.** The signed displacement in an Unconditional Jump (JMP) instruction is a two's complement eight-bit number which represents the number of words to skip forward or backward from the current PC (the PC points to the instruction *following* the jump instruction).

To calculate the displacement for a jump instruction, evaluate

1/2 (target location-(instruction address+2)).

If the target address is less than the instruction address, add 10000_{16} to the target address and perform the subtraction. Note that a forward branch must generate a positive displacement and a backward branch must generate a negative displacement to be in range.

Example 1:

Patch location $17A_{16}$ with a jump to location $1FE_{16}$.

The source address is equal to the instruction address +2, which is 17A+2 = 17C.

The target location minus the source address is 1FE - 17C = 82. Continuing,

1/2 (target location - source address) = 41

The displacement, 41, is positive. The patch value is therefore 1041_{16}, where 10 is the hexadecimal op code for the JMP instruction and 41 is the displacement value.

Example 2:

Patch Location $1FE_{16}$ with a jump to location $17A_{16}$.

The source address is equal to the instruction address+2, which is $1FE_{16}+2_{16} = 200_{16}$. The sum of the target location plus 10000_{16}, minus the source address, is $1017A_{16}-200_{16} = FF7A_{16}$. Continuing

1/2 (target location - source address) = 7FBD = BD (dropping the first two digits)

The displacement, BD, is negative. The patch value is therefore $10BD_{16}$, where 10 is the hexadecimal op code for the JMP instruction and BD_{16} is the displacement value, negative in this case.

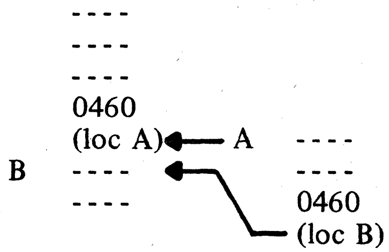
Note that the 7F is generated from the addition of 2_{16} (10000_{16}) and may be discarded. If the high order eight bits of the destination are not equal to 7F, the branch distance is too great to reach with a JMP instruction.

3.6.3.5 **Spin and NOP.** It is sometimes convenient to patch a spin (branch to itself) into a location to intercept control in unexpected situations (the alternate path of a conditional jump, for example). That instruction is a JMP to itself and is a value of $10FF_{16}$. (The corresponding assembly language code is JMP $.)

Unwanted instructions can be replaced with a no-operation (NOP) which is a JMP to the next instruction. The value for an NOP is 1000_{16}.

3.6.3.6 **Out-of-Line Patches.** It is often necessary to patch more instructions into a program than there is room, requiring an out-of-line patch. The simplest mechanism is to use a symbolic address branch instruction to a specific location where the patch is placed. After the patch, use a branch instruction back to the original code.

Example:



Be careful to see that code which is overlayed is moved to the patch area, that it is not a PC relative jump, and that the return pointer comes to the beginning of an instruction.

SECTION IV

TEXT EDITOR

4.1 INTRODUCTION

The first part of this section is a general description of PX9EDT, giving its purpose, its functions, and a brief explanation of how data is edited. The loading procedure for the module is presented, followed by a discussion of how to start execution and assign LUNOs. The initialization messages, user's responses to those messages, and the final message are listed and described.

Specific editing procedures are presented, including procedures for copying from one cassette to another, using editor commands to move the text editor's line pointer, moving text to or from the buffer, and handling file data formats with special terminal keyboard characters. Procedures for coding source or object files and writing a new source program on cassette tape are also described.

The paragraphs that describe text editor commands explain the classes of commands, command operands, and notational conventions used in the command syntax. Detailed descriptions and examples of each of the text editor commands are included.

The error and warning messages, with discussions of user's responses to the messages, are presented. An example of source program editing and a discussion of object code editing conclude this section.

4.2 GENERAL DESCRIPTION

The Text Editor (PX9EDT) is an interactive program for editing source and object programs. The following paragraphs provide a brief general description of the program, list and describe the Text Editor commands, and explain the messages printed by the program.

PX9EDT executes in a Model 990/4 microcomputer or 990/10 minicomputer configured for the 990-733 ASR System Software or 990 Prototyping System. This configuration includes the Model 990/4 microcomputer with 8K or more of memory and a Model 733 ASR Electronic Data Terminal. The Debug Monitor, PX9MTP, must be resident since PX9EDT is loaded by PX9MTP and calls PX9MTP routines for input/output and conversion routines.

PX9EDT may be used to generate source or object data or to edit existing source or object data. PX9EDT provides 18 commands with which the user specifies the desired edit functions. These commands provide the user with the capability to change, add, move, and remove source or object records, and to locate and modify a character string in a group of records.

Data to be edited is read from a cassette tape into an area of memory called the buffer. The data is edited while it is in the memory buffer by using PX9EDT commands. Individual lines of data are identified by a pointer, which may be positioned by PX9EDT commands. The pointer refers to a buffer line number. The edited data is written from the memory buffer to a cassette tape.

The user specifies the buffer size by responding to an initialization message that asks for the number of 4K-word blocks to be assigned to the buffer.

In an 8K system with 4K user memory, PX9EDT contains 3870 bytes of edit buffer space. Lines are placed in the buffer with one character per byte followed by a carriage return, and preceded by a six-byte header. Therefore, a buffer of lines with an average length of 40 bytes, including header and carriage return, would contain approximately 100 lines. Using tabs when inputting source lines causes a tab to be placed in the buffer instead of multiple spaces.

4.3 LOADING AND INITIALIZATION PROCEDURES FOR THE TEXT EDITOR

The following paragraphs describe the loading procedure and the messages output during initialization and termination.

4.3.1 LOADING. PX9EDT is loaded by means of the PX9MTP Load Program in Compressed Absolute Format with Upfront Loader (LU) command. Mount and position the cassette containing the PX9EDT object code and enter this command on the terminal keyboard:

```
LU
```

The LU command assumes that the cassette is mounted in the cassette drive assigned to logical unit number 7. Be sure that logical unit number 7 has not been reassigned to another device. Other acceptable load program commands are the following:

LU,7	Load program with upfront loader from LUNO 7
LU,8	Load program with upfront loader from LUNO 8

4.3.2 STARTING EXECUTION. The user then uses the PX9MTP Execute User Program Directly (EX) command to begin execution of PX9EDT. PX9EDT accepts input from logical unit number 7 and writes its output to logical unit number 8. To use other than system defaults, the PX9MTP Assign LUNO (AL) command should be used before the EX command is entered. (Refer to the discussion of logical unit numbers in Section II and the discussion of the AL command in Section III.)

4.3.3 INITIALIZATION MESSAGES. When PX9EDT is started, it prints a series of messages requesting user responses. The first two messages are printed the first time PX9EDT is executed after being loaded. In subsequent executions and restarting PX9EDT, the first two messages will be deleted.

This message identifies the program name and release information:

```
PX9EDT PART # REV DATE
```

The following message asks for a count of memory blocks:

```
ADD 4K MEM BLOCKS CONFIGURED?
```

The user should input the number of 4K user memory blocks that are configured in his system in addition to the 4K required by PX9EDT. This additional memory will expand PX9EDT's edit buffer space. For example, if the user's system contains 8K of user memory in addition to the 4K required by the monitor, the response should be "1". If the user's system contains 4K of user memory, the response should be "0" or a carriage return.

If the configuration includes more than 4K words of user memory area, PX9MTP will be relocated as described in the system software cassette generation procedure in Section II.

The following message notifies the user that the system is ready for mounting of tape cassettes:

```
POSITION TAPES, ENTER CR
```

The user should mount his cassettes and position them to the correct files, and then enter a carriage return. PX9EDT accepts input from logical unit number 7 and writes its output to logical unit number 8.

A question mark (?) will be printed and the user can enter text editor commands from the keyboard.

4.3.4 FINAL MESSAGE. When the output file has been written in response to a Q or E command, PX9EDT prints the following message:

```
END EDIT
```

PX9EDT then restarts and prints the initial position tapes message.

4.4 TEXT EDITING PROCEDURES

The following paragraphs describe some specific procedures for common text editing tasks. These tasks include copying a tape cassette, moving the PX9EDT buffer pointer, moving lines of text into and out of the buffer, manipulating file data formats, and creating new programs.

4.4.1 COPYING FROM ONE TAPE CASSETTE TO ANOTHER. To copy a cassette with the text editor, the user loads and starts up PX9EDT. When the message

```
POSITION TAPES, ENTER CR
```

is printed, mount and ready the cassette tape to be copied (input) in the cassette drive assigned to LUNO 7 (usually CS1) and mount and ready a scratch cassette (output) in the cassette drive assigned to LUNO 8 (usually CS2).

When a "?" is printed, the user enters the quit command.

```
?Q
```

The input tape is copied to the output tape until an end of file is encountered. The following messages are printed.

```
END EDIT

TERM/CONT?
```

If the input cassette tape contains more than one file to be copied, the user should enter a "C" to restart PX9EDT and restart the procedure. The position tapes message is printed. The tapes are positioned to begin copying the next file. Continue until all files are copied. Enter a "T" to terminate the procedure.

4.4.2 MOVEMENT OF POINTER. The pointer commands are used to move the pointer to any line in the buffer of PX9EDT. Initially, the pointer is at line 1. The Down (D) command may be used to move the pointer down a specified number of lines. Moving the pointer with the Down command to an empty line causes PX9EDT to read source lines or object records from the input file to fill the empty lines, including the line specified in the Down command. When the Down command causes more data to be read from the input file, the pointer is left at the bottom of the buffer after execution of the Down command.

The Up (U) command is used to move the pointer up a specified number of lines. The Top (T) command moves the pointer to the top (first line) of the buffer. The Bottom (B) command moves the pointer to the bottom (last line) of the buffer.

The pointer commands permit the user to move the pointer as desired for effective use of commands that identify lines by specifying the displacement from the pointer.

4.4.3 MOVING LINES TO OR FROM BUFFER. PX9EDT allows the user to edit data in a buffer and identifies lines of data in the buffer with line numbers or with a pointer.

Source lines or object records are placed in the buffer for editing by PX9EDT by reading lines from the input file or by entering lines at the keyboard. The Down (D) command is used to read lines from the input file and move them to the buffer. The Insert (I) and Change (C) commands are used to enter lines from the keyboard. When data is read from the input file, PX9EDT assigns a line number to each line or record. The line number may or may not be printed, but it is not written on the output file. When lines of data are moved in the buffer, each line retains its line number. When lines are removed from the buffer, the line numbers are not reassigned. No line numbers are assigned to lines of data that are entered at the keyboard.

After data has been edited, the buffer may contain lines without line numbers, and the lines in the buffer are not necessarily in line number sequence. When a line has a line number, the line may be specified by line number. Any line may be specified by a displacement from the pointer.

Only data in the buffer may be edited; therefore, it may be necessary to move data from the buffer to the output file to leave more space in the buffer so that additional lines may be read from the input file or entered from the keyboard.

The Keep (K) command writes a specified number of lines from the buffer to the output file. The number of lines specified by the Keep command is written from the top of the buffer regardless of the location of the pointer line. If a number is not specified, all lines in the buffer are written to the output file. This makes these lines unavailable for further editing in this session.

The Quit (Q) command writes lines from the buffer and input file followed by an end-of-file record, and terminates PX9EDT. The Quit command may write lines from the buffer and input file, or from the buffer only. Quit should be used as the final command in an edit session so that an end of file will be written to the output file.

4.4.4 HANDLING OF FILE DATA FORMATS. The following special characters are recognized by the text editor I/O routines. A backspace character (CTRL H) backspaces one character position. A RUB OUT character deletes the current input line. A tab (CTRL I) echoes as one space upon character input, but moves to the nearest tab stop when the line is printed. Tab stops are defined at character positions 8, 13, 31, and 33. An escape (ESC) entered from the keyboard during cassette or print output causes the current I/O operation and the command to be aborted. Control returns to the command handler, and another command may be entered when a "?" is printed.

All other characters from keyboard input, printer output, and cassette read and write are handled as specified in Appendix C.

4.4.5 COMBINING SOURCE OR OBJECT FILES. PX9EDT may be used to combine source files or object files. These files may be on one or more cassette tapes. When the editor is started, the initial messages are printed:

```
PX9EDT PART # REV DATE
ADD MEM BLOCKS CONFIGURED?

POSITION TAPES, ENTER CR
```

The user mounts the tapes and enters a carriage return on the terminal keyboard. The prompt character (?) is displayed. The user enters the D command to read records from the input tape. The K command is then used to write the records in the buffer to the output tape.

```
 ?D150
 ?K150
```

Subsequent D and K commands are entered until the end of file is reached and the entire file has been transferred to the output tape.

```
 ?D150
 ?K150
 ?D150
END OF FILE
 ?K150
```

The user then enters the E command to end the edit process without writing an end of file to the output file.

```
 ?E
```

The terminate or continue question is then asked. The user responds with a C for continue.

```
TERMINATE/CONTINUE ?C
```

The initial tape positioning message is printed. The user repositions the file or replaces the original input file with the next one to be copied and repeats the above procedure until all files have been transferred. When all the records of the last file have been transferred, the user enters the Q command instead of the E command.

```
 ?Q
END EDIT
```

The Q command writes an end of file to the output file and terminates the edit process. The terminate or continue question is printed. Respond with a "T" to terminate.

4.4.6 CREATING NEW PROGRAMS. The following paragraphs describe the use of PX9EDT to enter a new source program on tape. The Insert (I) command is used to input new source statements. Any of the commands may be used to correct any errors made in entering the statements. Because statements entered with the Insert command have no line numbers, the pointer-relative specification is the only available means of specifying a line in a command. The following text describes an example of writing a source program using PX9EDT.

The initial message and the first command, with associated entries, are as follows:

```
POSITION TAPES, ENTER CR
 ?I0
W1      BSS   32
START   RSET
        LWPI  W1
        CLE   R0
```

The I command with an operand of zero causes PX9EDT to place the lines that follow at the top of the buffer. The buffer pointer is not moved as lines are entered and remains ahead of the first line entered. In the above example, an error was made in the operation field of the fourth line, so the user entered an additional carriage return to terminate the command, permitting entry of another command to correct the error.

The next part of the example program is:

```
 ?K3
 ?P1
     CLE   R0
```

The K command cuases PX9EDT to write the first three lines on the output medium. The P1 command causes PX9EDT to print the pointer line to verify that the pointer is at the line that contains the error. An alternative to using the Keep command to write the correct portion of the program is to use a Down command to position the pointer for correction of the error, leaving the first three lines in the buffer.

The next command and the associated entries are as follows:

```
 ?C
     CLR   R0
I1   INC   R0
     JNO   J1
D1   DEC   R0
     JNE   D1
     JMP   I1
     END   START
```

The C command deletes the error line and accepts seven lines of source code. The example source program is now complete, with three lines written on the output medium, and seven lines in the buffer.

The next command and the resulting printing are as follows:

```
 ?F10F'J1"I1'
LAST LINE
0001 FOUND
```

The F command scans the contents of the buffer, replacing the first appearance in each line of string J1 with string I1. The command attempts to scan 10 lines, and prints the message LAST LINE because there are only seven lines in the buffer. The V and P options (paragraph 4.5.4.5) could have been used. This is an alternate method of correcting an error in a source program entered from the keyboard using PX9EDT.

The next command and the resulting printing are as follows:

```
?P10
      CLR  R0
I1    INC  R0
      JNO  I1
D1    DEC  R0
      JNE  D1
      JMP  I1
      END START
LAST LINE
```

The P command causes PX9EDT to print the contents of the buffer and the last line message. This command allows the user to check the program carefully before writing the output file.

The last command and the final message are as follows:

```
?Q0
END EDIT
```

The Q command causes PX9EDT to write the buffer contents on the output medium following the records previously written by the Keep command. An end-of-file record is written following the last record. The 0 specifies that no input records are to be read.

When it is desired to put more than one source module in a file, each module should be terminated with an END statement. The Quit command should be entered in order to output the buffer and write an end of file after all source files have been entered. When the assembler reads the END statement of a module, an end-of-module record is written to the object file. The assembler continues assembling the source modules on the input cassette until an end of file is encountered. The assembler then writes an end of file on the output object tape.

4.5 COMMANDS

The 18 commands of PX9EDT include setup commands, pointer commands, edit commands, print commands, and output commands.

4.5.1 GENERAL. The four setup commands initialize the edit operation. The group includes commands to enable or inhibit printing of line numbers, to set the right margin for printing, and to set left and right limits for the Find command.

PX9EDT edits data in a buffer and identifies lines of data in the buffer with a pointer. Four pointer commands permit the user to position the pointer by moving the pointer down, up, to the top of the buffer, or to the bottom of the buffer. The command that moves the pointer down may also read data from the input file.

The five edit commands of PX9EDT allow the user to change, insert, move, or remove lines of code, and to search for a character string in a set of lines of code. PX9EDT counts the lines in which the string is found and optionally substitutes another character string, verifies substitution of the character string, or prints the line in which the string is found.

Two print commands print lines of code on the printer. One command prints the first and last lines of code in the buffer. The other command prints one or more lines as specified in the command.

PX9EDT provides two output commands to write data on the output file. One command outputs a specified number of lines, or all lines of data from the buffer. The other outputs a specified number of lines, or all lines of data from the buffer, or from the buffer and the input file, and writes an end-of-file record following the last line.

One other command allows the user to terminate the edit process without outputting any lines or writing an end of file.

Commands are entered at the keyboard in response to the printing of a question mark (?).

The command language is free-form, in that one or more spaces may be inserted between characters and operands of the commands. Each command is terminated by entering a carriage return.

4.5.1.1 **Operands.** The operands of the PX9EDT commands specify numbers of lines, line numbers, or displacements from the pointer. The edit commands and one of the print commands may specify a group of lines by first and last line number, or by a number of lines relative to the pointer.

The procedure for moving lines to or from the buffer is described in paragraph 4.4.3.

4.5.1.2 **Conventions.** The following symbols and conventions are used in defining the syntax of PX9EDT commands:

- Angle brackets (< >) enclose items supplied by the user.
- Brackets ([ ]) enclose optional items.
- Braces ({ }) enclose items between which a choice must be made; one, but only one, of the items must be included.
- Items in capital letters and punctuation marks must be entered as shown.

The syntax definitions and examples shown in this manual do not show spaces between the characters of the two-character commands, between the command and operands, or between operands. Spaces may be entered at these points if desired. All operands are decimal numbers.

4.5.2 **SETUP COMMANDS.** The setup commands may be entered immediately following the initial message to initialize limits for the Find command and the right margin for printing, and to enable or inhibit printing of line numbers. These commands may also be entered at any time during the edit to change any of these parameters. When neither of these commands is entered, line numbers are printed, the right margin for lines of print is column 72, and columns 1 through 72 are scanned by the Find command. The setup commands are described in the following paragraphs.

4.5.2.1 **Line Numbers (SL).** The Line Numbers command causes PX9EDT to resume printing line numbers to the left of each statement or record. The syntax for the SL command is as follows:

```
SL
```

The SL command is used to restore printing of line numbers after line number printing has been inhibited by execution of an SN command.

4.5.2.2 **No Line Numbers (SN).** The No Line Numbers command causes PX9EDT to omit printing of line numbers except in the message resulting from the L command. The syntax for the SN command is as follows:

```
SN
```

The SN command may be entered initially or at any time during the edit operation. Omitting the line numbers when editing object code may be desirable to permit printing the entire record.

4.5.2.3 **Print Margin (SP).** The Print Margin command specifies the right margin for printing, except for the message resulting from the L command. The syntax for the SP command is as follows:

```
SP<s>
```

The right margin for printing is column s. The default value for the right margin is column 72. The margin input must be a value between 10 and 80, inclusive. If line numbers are being printed, the line numbers are included in the margin column. The line numbers use six columns, so that if the right margin is comumn 72, only 66 characters plus 6 line numbers and blanks for spacing are printed. The following example shows an SP command that specifies column 60 as the right margin for printing:

```
?SP60
```

4.5.2.4 **Find Margin (SM).** The Find Margin (SM) command specifies left and right limits for the Find command. The syntax for the SM command is as follows:

```
SM<s>,<t>
```

The Find command scans from column s to column t. The SM command may be entered to limit the Find command to a desired field. The default value for the scan limits is from column 1 to column 72. The following example shows an SM command that limits the scan of subsequent Find commands to columns 8 through 25:

```
?SM8,25
```

4.5.3 **POINTER COMMANDS.** The pointer commands may be used to move the pointer to any line in the buffer of PX9EDT. Initially, the pointer is at line 1. Moving the pointer with the Down command to any empty line causes PX9EDT to read source lines or object records from the input file to fill the empty lines, including the line specified in the Down command. Other commands move the pointer up a specified number of lines, or to the top of the buffer, or down to the bottom of the buffer. The pointer commands permit the user to move the pointer as desired for effective use of commands that identify lines by specifying a displacement from the pointer. The pointer commands are described in the following paragraphs.

4.5.3.1 **Down (D).** The Down command causes PX9EDT to move the pointer down a specified number of lines. When the specified move is to a line number greater than the contents of the buffer, PX9EDT adds lines to the buffer and reads records from the input file to fill these lines. The syntax for the D command is as follows:

```
D[<n>]
```

The pointer is moved down n lines. The range of n is 1 to 9999, and the default value when n is omitted is 1. The D command may be entered to read in lines from the input file or to move the pointer to a line farther down in the buffer. Initially, or when the pointer is at the bottom of the buffer, PX9EDT reads n lines from the input file. When the pointer is m lines above the bottom of the buffer and n is greater than m, PX9EDT reads n – m lines from the input file. In each of these cases, the pointer is at the bottom of the buffer after execution of the D command. However, when the pointer is m lines above the bottom of the buffer and m is greater than or equal to n, no lines are read, and the pointer is m – n lines above the bottom of the buffer after execution of the command. The following example shows a D command to move the pointer down 30 lines:

```
?D30
```

4.5.3.2 Up (U). The Up command causes PX9EDT to move the pointer up a specified number of lines. The syntax for the U command is as follows:

```
u[<n>]
```

The pointer is moved up n lines. The range of n is 1 to 9999, and the default value when n is omitted is 1. The U command may be entered to move the pointer up to a specific line in the buffer. The following example shows a U command to move the pointer up 6 lines:

```
?U6
```

4.5.3.3 Top (T). The Top command causes PX9EDT to move the pointer to the top line in the buffer. The syntax for the T command is as follows:

```
T
```

4.5.3.4 Bottom (B). The Bottom command causes PX9EDT to move the pointer to the bottom line in the buffer. The syntax for the B command is as follows:

```
B
```

4.5.4 EDIT COMMANDS. The edit commands add, remove, rearrange, or scan lines of source or object code. These commands act upon a set of the lines in the buffer, specified by line number or by a displacement from the pointer. The edit commands are described in the following paragraphs.

4.5.4.1 Change (C). The Change command deletes a specified set of lines and permits input of one or more lines to replace the deleted lines. The syntax for the C command is as follows:

$$\mathrm{C}\ \begin{Bmatrix} \langle s\rangle\text{-}\langle t\rangle \\ [+]\,[\langle n\rangle] \\ \text{-}\langle n\rangle \end{Bmatrix}$$

Line s through line t are deleted, or n lines with respect to the pointer are deleted. The values of s and t can be equal. Enter as many replacement lines as required. Follow each line with a carriage return; follow the last line with two carriage returns. When n is preceded by a minus sign, n line preceding the pointer line are deleted, but the pointer line is not deleted. When n is unsigned or is preceded by a plus sign, n lines beginning with the pointer line are deleted. When no operand is entered, the pointer line is deleted. When the pointer line is deleted, the pointer is moved to the top line of the buffer. The following example shows a C command to change lines 5 through 7, replacing them with four lines:

```
?C5-7
LOD     MOV  1,4
        AI   4,1
        CI   4,WA+60
        JLT  SUM
```

The following example shows a C command to change the pointer line and the two lines that follow the pointer, replacing them with two lines:

```
?C3
LOD     MOV  1,4
        CI   4,WA+60
```

4.5.4.2 Insert (I). The Insert command permits input of one or more lines following the pointer or a specified line. The syntax for the I command is as follows:

```
I[<k>]
```

Enter as many lines as required. Follow each line with a carriage return; follow the last line with two carriage returns. When k is in the range of 1 to 9999, insert lines following line k. When k is 0, insert lines ahead of the top line in the buffer. When k is omitted, insert lines following the pointer line. The following example shows the use of the I command to insert two lines following line 10:

```
?I10
        CKON
        DEC     7
```

4.5.4.3 Move (M). The Move command moves a specified block of lines to a specified location and deletes the lines at the previous location. The block is specified by first and last line numbers, or by a number of lines preceding or following the pointer. The location is specified as a line number, or as the pointer. The syntax for the M command is as follows:

```
  ⎧<s>-<t>,[<r>]⎫
M ⎨[+]<n>,<r>   ⎬
  ⎩-<n>,<r>     ⎭
```

Line s through line t are moved, or n lines with respect to the pointer are moved. When n is preceded by a minus sign, n lines preceding the pointer line, but not the pointer line, are moved. When n is unsigned or preceded by a plus sign, n lines beginning with the pointer line are moved. The specified lines are placed following line r when r is greater than zero. When r is zero, the specified lines are placed ahead of the top line in the buffer. When r is omitted, the lines are placed following the pointer line. Numbered lines moved by the Move command retain their original line numbers, if any. When the pointer line is moved, the pointer moves with it. When s and t are specified, r must be less than s or greater than t. When n is specified, r may not be omitted. The following example shows an M command to move lines 6 through 8 to follow line 25:

```
?M6-8,25
```

The command in the following examples moves four lines beginning with the pointer line to follow line 30:

```
?M4,30
```

4.5.4.4 Remove (R). The Remove command removes a block of lines. The block is specified by first and last line numbers, or by a number of lines preceding or following the pointer. The syntax for the R command is as follows:

$$\text{R} \left\{ \begin{array}{l} \text{<s>-<t>} \\ \text{[+] [<n>]} \\ \text{-<n>} \end{array} \right\}$$

Line s through t are removed, or n lines with respect to the pointer are removed. When n is preceded by a minus sign, n lines preceding the pointer line, but not the pointer line, are removed. When n is unsigned or preceded by a plus sign, n lines beginning with the pointer line are removed. When no operand is entered, the pointer line is removed. When the pointer line is removed, the pointer is moved to the top line of the buffer. The following example shows an R command to remove line 12:

```
?R12-12
```

The command in the following example removes the three lines preceding the pointer line:

```
?R-3
```

4.5.4.5 Find (F). The Find command scans a block of lines for a specified character string. Optionally, the command may replace the string with or without printing the resulting line, or may print the line and permit the user to specify whether or not to substitute the string. In all cases, the command prints the count of matching lines found. The block is specified by first and last line numbers, or by a number of lines preceding or following the pointer. The syntax for the F command is as follows:

$$\text{F} \left\{ \begin{array}{l} \text{<s>-<t>} \\ \text{[+]<n>} \\ \text{-<n>} \end{array} \right\} \left\{ \begin{array}{l} \text{L} \\ \text{F} \end{array} \right\} \text{<d1><string1><d1>} \left\{ \begin{array}{c} \text{[P]} \\ \text{<d2>[<string2>]<d2>[V] [P]} \end{array} \right\}$$

Line s through line t are scanned, or n lines are scanned. When n is preceded by a minus sign, n lines preceding the pointer line, but not the pointer line, are scanned. When n is unsigned or preceded by a plus sign, n lines beginning with the pointer line are scanned.

When an F is entered following the lines to be scanned, the columns specified in an SM command are scanned. (Columns 1 through 72 are the default for SM.) When an L is entered, the command performs a label scan, beginning at the left limit and extending to the first space.

The character string used in the scan is designated string1, and is enclosed by identical characters, each represented by d1. The character represented by d1 may be any character that does not appear in string1.

When no other parameter is entered, the command scans the specified lines and prints the number of lines in which a match of string1 was found. When P is entered following d1, the command prints each line in which a match of string1 was found, and also prints the number of lines following the last line found.

Character string, string2, enclosed by identical characters, each represented by d2, is the replacing string. String2 may be omitted, or may be longer or shorter than string1. When the replacement is made, the characters of string2, if any, replace the characters of string1 and the length of the resulting line is adjusted as necessary. Character d2 may be any character that does not appear in string2, V, or P.

When no other parameter is entered following string2, the specified lines are scanned and string2 replaces the first appearance of string1 or label string1 each time a match is found. The command prints the number of lines in which the replacement was made after scanning the last line.

Either V or P, or both may be entered following string2. The verify operation, specified by V, prints the line in which the match is found, and prints the question Y/N? on the next line. The user must enter Y or N followed by a carriage return to continue the operation. When the user enters Y the replacement is made. When the user enters N the replacement is not made. The scan continues in either case.

The print operation is specified by P. After the replacement is made, the resulting statement is printed, and the scan continues.

When the specified lines have been scanned, PX9EDT prints the number of lines in which a match was found.

The general rule of PX9EDT which allows spaces between characters or operands does not apply to string1 and string2. Any spaces between the characters represented by d1 are considered part of string1, and any spaces between the characters represented by d2 are considered part of string2.

The following example shows an F command to replace the first appearance in each line of the string EUEN with the string EVEN in lines 34 through 48 and print the resulting lines:

```
?F34-48F*EUEN*$EVEN$P
```

The command in the following example verifies the replacement of label P1 with string PUN1 in each of nine lines beginning with the pointer line:

```
?F9L'P1"PUN1'V
```

NOTE

If a tab character is included between fields of the data being scanned by the F command, the tab character should be used in the comparison character string instead of blanks.

4.5.5 **PRINT COMMANDS.** The print commands cause PX9EDT to print the first and last lines in the buffer, or to print one or more specified lines. The print commands are described in the following paragraphs.

4.5.5.1 Limits (L). The Limits command causes PX9EDT to print the first and last lines in the buffer, including the line number, if any, with the right margin at column 72. The SN and SP commands do not affect the operation of the L command. The syntax for the L command is as follows:

```
L
```

The L command is used to identify the top and bottom lines of the buffer.

4.5.5.2 **Print (P).** The Print command causes PX9EDT to print a block of lines. The block of lines is specified by first and last line numbers, or by a number of lines preceding or following the pointer. The SL and SN commands, when entered, control printing of line numbers, and the SP command, when entered, sets the right margin of the print lines. When these commands are not entered, line numbers are printed and the right margin is column 72. The syntax of the P command is as follows:

```
  ( <s>-<t>    )
P { [+][<n>]   }
  ( -<n>       )
```

Line s through line t are printed, or n lines are printed. When n is preceded by a minus sign, n lines preceding the pointer line, but not the pointer line, are printed. When n is unsigned or preceded by a plus sign, n lines beginning with the pointer line are printed. When no operand is entered, the pointer line is printed. The following example shows a P command to print lines 8 through 10:

```
?P8-10
```

The command in the following example prints the pointer line and the next three lines:

```
?P4
```

The user may terminate the Print command at any time by entering an ESC character at the keyboard. PX9EDT then prints a question mark and awaits input of another command.

4.5.6 **OUTPUT COMMANDS.** PX9EDT provides two commands to write source or object code and one command to end execution of PX9EDT. The Keep command writes the entire buffer or specified lines from the buffer. The Quit command writes specified lines from the buffer, the entire buffer, or the buffer contents and the remainder of the input file, and writes an end-of-file record on the output file. The output commands are described in the following paragraphs.

4.5.6.1 **Keep (K).** The keep command writes a specified number of lines from the buffer on the output device. The syntax of the K command is as follows:

```
K[<n>]
```

The first n lines of the buffer, or all lines in the buffer when n is omitted, are written on the output device. When the pointer line is written, the pointer is moved to the top line remaining in the buffer. The K command is entered to write lines no longer required in the buffer in order to have space in the buffer for additional lines. The following example shows a K command to write the top 15 lines of the buffer:

```
?K15
```

4.5.6.2 **Quit (Q).** The Quit command writes lines from the buffer and input file followed by an end-of-file record. The syntax of the Q command is as follows:

```
Q[<s>]
```

The lines of the input file up to and including line s are written. When line s is in the buffer, lines are written from the buffer only. When line s is not in the buffer, PX9EDT writes the lines in the buffer, reads the additional lines from the input file, and writes these lines. When s is zero, only the lines in the buffer are written. When s is omitted, the lines in the buffer and the remainder of the input file are written. The Q command is entered to write the output file, or the remainder of the output file, including the end-of-file record.

If the output tape is not mounted, the message

```
RDY TAPE-TYPE CR
```

is printed. The user should ready the output cassette and enter a carriage return. The command then proceeds.

4.5.6.3 End (E). The End command stops execution of PX9EDT without writing any more lines to the output file and asks the user whether he wants to continue or terminate execution. An end-of-file is not written. The syntax of the E command is as follows:

```
E
```

The End command is often used to generate stacked modules without ends-of-file between them. In this case, the End command can be used following appropriate Keep commands to write the output file.

4.6 MESSAGES

PX9EDT prints error messages and warning messages. The messages are described in the following paragraphs.

4.6.1 ERROR MESSAGES. The two error messages printed by PX9EDT indicate errors in the entry of commands. When the operator portion of a command is incorrect, PX9EDT prints the following message:

```
INVALID OPERATOR
```

When an operand is not entered correctly or is beyond the range of values for that operand, PX9EDT prints the following message:

```
INVALID OPERAND
```

To recover from either error, the user enters the command correctly, or enters another command.

When an output command, either K or Q, is entered and no output cassette is mounted, the following message is printed:

```
RDY TAPE-TYPE CR
```

The user should ready the output cassette and type a carriage return on the terminal keyboard. The command entered then proceeds.

4.6.2 WARNING MESSAGES. When any command that operates on data in the buffer is entered before data has been placed in the buffer from the input file or from the keyboard (either initially or after writing the entire buffer contents), PX9EDT prints the following message:

```
BUFFER EMPTY
```

To recover, the user should enter a D command, or an I command and data.

When a D, I, or C command attempts to put more data into the buffer than the buffer can contain, PX9EDT prints the following message:

```
BUFFER FULL!
```

The user must enter a K command to write data from the buffer before entering or reading any more data.

When a D command attempts to read more records from the input file than the file contains, PX9EDT prints the following message:

```
END OF FILE
```

PX9EDT will not make any further attempt to read the input file until the program restarts.

When the negative displacement from the pointer line in a C, M, R, F, or P command is greater than the number of lines in the buffer ahead of the pointer line, PX9EDT prints the following message:

```
OFF THE TOP
```

After printing the message, PX9EDT executes the command beginning with the top line of the buffer.

When the positive displacement from the pointer line in a C, M, R, F, or P command is greater than the number of lines in the buffer following the pointer line plus one, the command executes normally until it has processed the last line in the buffer. PX9EDT then prints the following message:

```
LAST LINE
```

PX9EDT prints a question mark and waits for another command.

When the first line in a C, M, R, F, or P command, or the line number in an I command, or the destination line number in an M command is not in the buffer, PX9EDT prints the following message:

```
LINE NOT FOUND
```

PX9EDT does not execute the command, but prints a question mark and waits for another command.

4.7 SOURCE PROGRAM EDITING EXAMPLE

The capabilities of PX9EDT to edit source programs include adding, moving, and removing statements, and replacing a character string in statements. The edited program may include portions of a number of source programs. The purpose of editing is either to combine portions of source programs or to correct or modify a source program. The following paragraphs describe an example of editing a source program and considerations for editing source programs.

4.7.1 DESCRIPTION OF PROGRAM. The source program used as an example is a set of three program modules to be combined into one module. Some changes not related to combining the modules are also made. The source statements for all three modules have been placed in a single file containing 117 records.

The default values for print margin and F command limits are used, and line numbers are printed. No setup command is required.

4.7.2 EXPLANATION OF EXAMPLE. The initialization messages and the first command are as follows:

```
PX9EDT PART # REV DATE
ADD4K MEM BLOCKS CONFIGURED?

POSITION TAPES, ENTER CR

 ?D117
```

The D command moves the pointer down 117 lines, and PX9EDT reads in the source file to fill the buffer as defined by the D command. A smaller value could have been used to read part of the file, followed by a subsequent D command to read the remainder. Had a larger value been entered, PX9EDT would have read the 117 records of the file and printed the end-of-file message. PX9EDT prints the prompt character (?) and awaits another command.

The next command and the resulting printing are as follows:

```
 ?L
0001    TITL 'EDITING EXAMPLE'
0117    END
```

The L command verifies the buffer contents by printing the first and last lines in the buffer. Had the SN and SP commands been entered, they would not have affected the printing of the limits resulting from the L command.

The next command is as follows:

```
 ?T
```

The T command moves the pointer to the top of the buffer (line 1) from line 117 where the first command had placed the pointer. Moving the pointer to the top of the buffer permits using pointer-relative commands for the area at the top of the buffer.

The following commands move line 46 to a position after line 116 and remove line 117.

```
 ?M46-46,116
 ?R117-117
```

The following command is entered.

```
 ?M81-87,115
```

This M command moves lines 81-87 to a position following line 115. This causes the line numbers in the buffer to be out of sequence.

The following commands prepare for verifying the move operation.

```
 ?B
 ?P1
0046                END   START
```

The B command places the pointer on the last line of the buffer, and the P command prints the pointer line to verify that it is on the proper line.

The next command and the resulting printing is as follows:

```
 ?P-13
0111    UP2         MOV   *R10,*R10
0112                JNE   UP1
0113                BL    @ATTOP
0114                MOV   *DUMNXT,TMLOC
0115                JMP   UP3
0081    * ROUTINE COMMON TO UP AND DOWN
0082    UDCOM1      MOV   RTN,R5
0083                BL    @SCANOP
0084                INC   UDCNT
0085                MOV   UDCNT,UDCNT
0086                JEQ   EXIT
0087                B     *R5
0116    *
```

The P command prints the 13 lines preceding the pointer line, and the result shows that lines 81-87 have been placed after line 115. The result also shows the effect of the previous move and remove commands.

The next command and associated entries are as follows:

```
 ?I77
* TITLE = MSGOUT - MESSAGE OUTPUT
MSGOUT  MOV    *R11+,R10
        MOV    @MCOUNT(R10),R10
        BLWP   @PRINT
        B      *R11
```

The I command inserts five lines following line 77. The number of lines inserted is the number of lines entered with the command, and may be one or any number of lines. After the carriage return that terminates the last line, enter an additional carriage return to terminate the command.

The next command and the resulting printing are as follows:

```
 ?P77-78
0077                JMP     EXIT
           * TITLE = MSGOUT - MESSAGE OUTPUT
          MSGOUT    MOV     *R11+,R10
                    MOV     @MCOUNT(R10),R10
                    BLWP    @PRINT
                    B       *R11
0078      EOFEXT    BL      @MSGOUT
```

The P command prints lines 77 through 78, which includes the five unnumbered lines inserted by the previous command. The result shows that the lines have been inserted correctly.

The next command and the resulting interaction are as follows:

```
 ?F1-46F'EXIT"EXTDWN'VP
0071                JMP     EXIT
Y/N? Y
0071                JMP     EXTDWN
0077                JMP     EXIT
Y/N? Y
0077                JMP     EXTDWN
0080      EXIT      RTWP
Y/N? Y
0080      EXTDWN    RTWP
0086                JEQ     EXIT
Y/N? N
0004      FOUND
```

The F command finds the first appearance in a line of the string EXIT in lines 1 through 46. (Remember that line 46 is now the last line, i.e., after line 116.) The entire buffer is scanned because the top line in the buffer is line 1 and the bottom line is line 46. Line numbers greater than 46 between lines 1 and 46 are also scanned. The replacing string is used only when the user enters a Y following the printing of the line found. In the example shown, the replacement was not made in line 86 because the user entered an N following the printing of this line. Lines 71, 77 and 80 were replaced because the user entered a Y following the printing of these lines. The count of lines found is printed after all lines have been scanned. The F command may be used to scan only a portion of the buffer, from one line up to the entire buffer, and replace from one character to the entire statement.

The next three commands are as follows:

```
?R15-15
?R17-17
?R19-19
```

Each R command removes the specified line from the buffer. Three commands that remove one line each are necessary because the lines to be removed are not consecutive. A single R command may remove one or more consecutive lines.

The next command and the resulting printing are as follows:

```
 ?P14-20
0014      DUMNXT  EQU    0
0016      LINAD   EQU    2
0018      LINPTR  EQU    4
0020      CLLOC   EQU    6
```

The P command prints lines 14 through 20. The result shows that the lines specified in the remove command were removed.

The next command is as follows:

```
 ?U2
```

The U command positions the pointer to the second line preceding the pointer line. The pointer could have been moved any number of lines up to the top of the buffer.

The next two commands, the resulting printing of the first command, and the entry associated with the second are as follows:

```
 ?P68-68

0068                A          @MAXLIN,UDCNT
 ?C68-68
          A          @MINLIN,UDCNT
```

The P command prints line number 68 to verify that line 68 is the desired line. The C command changes line 68 to the line entered with the command. One or more consecutive lines may be deleted by a C command, and any number of lines including zero lines may be added. The number of lines added does not have to be equal to the number of lines deleted. The added lines have no line numbers.

The last command and the final message are as follows:

```
 ?Q0
 END EDIT
```

The Q command writes the contents of the buffer and end-of-file record and terminates the PX9EDT run. Omitting the operand following the Q causes the command to write the buffer contents and copy the remainder of the source file. An operand other than zero causes all lines up to and including the specified line to be written. The line may be in the buffer, or in the portion of the input file remaining to be read.

4.8 EDITING OBJECT CODE

The capabilities of PX9EDT to edit object programs include adding, moving, and removing records, and replacing a character string in records. These capabilities allow the user to combine object code, correct object code, and add object code at a machine instruction level. In editing object code, it is necessary to thoroughly understand the object code format and the significance of tag characters, described in the *Model 990 Computer TMS9900 Microprocessor Assembly Language Programmer's Guide*, and summarized in Section V of this manual. Records may be inserted into an object program at any point except that the records that contain the 3 or 4 tag character, the 5 or 6 tag character, and the 1 or 2 tag character must follow all other records in the object file. Further, the record that contains the D tag character, if any, must precede the record that contains the first 0 tag character. Each record must end with tag character F. When the contents of a record are altered, the 7 tag character and associated field must be removed. When the length of relocatable code is increased, the contents of the hexadecimal field associated with the final 0 tag character must be changed. The following paragraph describes an example of editing an object program.

In the example, the purpose of the edit is to add a record to specify a load point, to change instructions that use workspace register 1 to use workspace register 7 instead, to change an instruction, and to add an instruction.

The initialization message and the first command are as follows:

```
POSITION TAPES, ENTER CR

 ?SN
```

The SN command is a setup command that inhibits printing of line numbers. When line numbers are printed, printing of an object record may be truncated because of the length of the print line.

The next command and the associated entry are as follows:

```
 ?I0
D1000F
```

The I command with an operand of 0 inserts the associated line at the top of the buffer. The line will be the first record in the edited object file, and contains load point of 1000_{16}, specified with a D tag character.

The next command and the resulting printing are as follows:

```
 ?D10

END OF FILE
```

The D command causes PX9EDT to read in the object file to be edited. The file contains six records, so the operand used causes PX9EDT to attempt to read past the end-of-file record. This inhibits further reading of any input file in this run of PX9EDT. If more than one file is to be combined in an edit operation, avoid an operand in a D command that will cause PX9EDT to attempt to read more records than the file contains.

The next command and the resulting printing are as follows:

```
 ?L

      D1000F
0006  200CE0010C     7FCABF
```

The L command causes PX9EDT to print the limits. The top line in the buffer is the line entered with the I command, and has no line number. The bottom line is the last line of the object file, line 6.

The next command and the resulting interaction are as follows:

```
 ?F1-6F'B0002''B000E'VP

00000SAMPROG 90040C0000A0020BC06DB00029 0042C0020A0024BC81BC002A7F219F
 Y/N?Y

00000SAMPROG 90040C0000A0020BC06DB000E90042C0020A0024BC31BC002A7F219F
A0028B0241B0000BCB41B0002B0380A00CAC0052C00A2B02E0C0032B0200B0F0F7F1DEF
 Y/N?Y

A0028B0241B0000BCB41B000EB0380A00CAC0052C00A2B02E0C0032B0200B0F0F7F1DEF
 0002 FOUND
```

The F command scans for the character string B0002 with the verify and print options. The replacement string, B000E, changes the memory address of workspace register 1 to that of workspace register 7 in two instructions. Verification and printing provides control and documentation of the changes.

The next command and the resulting interaction are as follows:

```
 ?F1-6F'7F151'''VP

A00D6BC0A0C00CAB04C3BC160C00CCBC1A0C00D0BC1F2B0287B3A00A00ECB02217F151F
 Y/N?Y

A00D6BC0A0C00CAB04C3BC160C00CCBC1A0C00D0BC1F2B0287B3A00A00ECB0221F
 0001 FOUND
```

The F command scans for the character string 7F151, which is a checksum tag character and associated field. The replacement character string is a null string, and the result is to remove the checksum from a record which was changed by an edit command not shown here.

The next command and the associated entry are as follows:

```
?I
A00ECB0227A00F0B06C7A010AB04CTF
A010CB10FFF
```

The I command inserts the associated two lines following the line on which the pointer had been positioned by an edit command not shown. The first line will cause the loader to overlay three words of the original file, which is another way of changing object code. The second line is an added instruction which will increase the size of the program module.

The next three commands, the resulting printing of the second, and the associated entry of the third are as follows:

```
 ?D3

 ?P1

200CE0010C    7FCABF
 ?C
200CE0010E    F
```

The D command moves the pointer line down three lines, and the P command causes PX9EDT to print the pointer line to verify the pointer position. The C command changes the pointer line to modify the number of words of relocatable code in the program. If this is not done, and another module is loaded following this module without specifying a load address for the subsequent module, the subsequent module will overlay the instruction that was added. The pointer line is also changed to delete the checksum.

The last command and the final messages are as follows:

```
 ?Q0

END EDIT
```

The Q command causes PX9EDT to write the contents of the buffer, followed by an end-of-file record, on the output medium.

SECTION V

ONE-PASS ASSEMBLER

5.1 INTRODUCTION

This section describes the purpose of the one-pass assembler and how the assembler functions. It also presents some recommendations for using the assembler. Paragraphs on the loading procedure and operation of the assembler follow. The operation discussion includes the input/output requirements and the operational messages printed. The next part of this section contains a brief discussion of assembler directives and pseudo-instructions and includes references to other publications. Error messages, descriptions of the errors with remedial action required, an explanation of the printed source listing output, and a brief discussion of the object code comprise the remainder of the section.

5.2 GENERAL DESCRIPTION

The One-Pass Assembler (PX9ASM) executes in a Model 990/4 or 990/10 Computer configured for the 990-733 ASR System Software or the 990 Prototyping System. The Debug Monitor, PX9MTP, must be resident since PX9ASM is loaded by PX9MTP and calls PX9MTP routines for input/output and conversion operations. PX9ASM assembles object code for the TMS9900 microprocessor, the 990/4 microcomputer and the 990/10 minicomputer.

A one-pass assembler reads the source statements of a program once only. The assembler maintains a location counter as it reads the statements, and assigns a location counter value to a label (symbol in the label field). The assembler builds a symbol table using these symbols and the assigned values. The assembler also evaluates the expression in the operand field using the values in the symbol table for any symbols in the expression. Then the assembler assembles the appropriate object code according to the operation codes and the values of the operands.

PX9ASM supports the assembly language as described in the *Model 990 Computer TMS9900 Microprocessor Assembly Language Programmer's Guide*, Manual No. 943441-9701. It is recommended that the user read this manual before trying to write any assembly language programs. Because PX9ASM is a one-pass assembler, there is a restriction which allows only one forward reference in an expression.

PX9ASM provides a listing of the source and object code and generates the machine language object code on cassette tape.

5.3 LOADING PROCEDURE FOR THE ASSEMBLER

PX9ASM should be loaded by means of the PX9MTP Load Program in Compressed Absolute Format with Upfront Loader (LU) command. Mount and position the cassette containing the PX9ASM object code and enter this command:

```
LU
```

The LU command with no parameters assumes that the cassette will be mounted in the cassette drive assigned to logical unit number 7. Be sure that logical unit number 7 has not been reassigned to another device. Examples of other acceptable load program commands are the following:

LU,7 Load program from LUNO 7.

LU 8 Load program from LUNO 8.

The user then enters the Execute User Program Directly (EX) command to begin execution of PX9ASM.

5.4 ASSEMBLER OPERATION

The following paragraphs discuss input/output and use of the assembler.

5.4.1 INPUT AND OUTPUT. PX9ASM accepts tapes containing Model 990 Computer/TMS9900 Microprocessor Assembly Language source statements as input. The source tapes may be generated with the Text Editor (PX9EDT). PX9ASM assembles the source lines generating an output listing of the assembled source and object code and a cassette tape object file which may be loaded by the relocating linking loader (LL command of PX9MTP). If no linking is necessary, the LP command may be used to load the object.

PX9ASM accepts input source from logical unit number 7, outputs the listing to logical unit number 6, and outputs the loadable object to logical unit number 8. Under PX9MTP, the following default logical unit number assignments have been made.

Logical Unit Number	Device
6	LOG
7	CS1
8	CS2

If other assignments are required, the Assign LUNO (AL) command of PX9MTP should be used. For example, to assemble a source tape with no printed listing, the user should assign LUNO 6 to DUM, the dummy device. The error messages will continue to be printed.

5.4.2 PX9ASM OPERATIONAL MESSAGES. When PX9ASM is started, it prints a series of messages requesting user responses. The first two messages are printed the first time PX9ASM is executed after being loaded. In subsequent executions and when restarting PX9ASM, the first two messages will not be printed.

The first of these messages is as follows:

```
PX9ASM PART # REV DATE
```

This message identifies the program name and release information.

The second message is:

```
ADD 4K MEM BLOCKS CONFIGURED?
```

The user should input the number of 4K user memory blocks which are configured in his system in addition to the 4K required by PX9ASM and the 4K required by PX9MTP. The maximum number that can be specified is 5. This additional memory will expand PX9ASM's symbol table size. For example, if the user's system contains 8K of user memory space in addition to the 4K required by PX9MTP, the response should be "1". If the user's system contains 4K of user memory space, the response should be "0" or a carriage return only.

The third message is:

```
PREDEFINED REGISTERS?
```

The user should enter "N" for no predefined registers, or "Y" or a carriage return if registers R0 through R15 are to be predefined in the symbol table.

An additional message is the following:

```
ASM/TERM?
```

The user should type "A" to assemble and "T" to terminate and return control to the monitor. For the assemble option response, the user should mount cassettes and position them to the correct files before responding to the message.

5.5 DIRECTIVES AND PSEUDO-INSTRUCTIONS

The following paragraphs briefly describe the assembler directives and pseudo-instructions, explaining how they are used and identifying the publication in which detailed information about them may be found.

5.5.1 ASSEMBLER DIRECTIVES. Assembler directives are used with machine instructions in source programs to supply data to be included in the program and to control the assembly process. The PX9ASM assembler supports the 19 directives listed in Appendix D. The syntax definitions and detailed descriptions of these directives are in the *Model 990 Computer TMS9900 Microprocessor Assembly Language Programmer's Guide*, Manual No. 943441-9701.

5.5.2 PSEUDO-INSTRUCTIONS. A pseudo-instruction is a convenient way to code an operation that is actually performed by a machine instruction with a specific operand. The Model 990 Computer Assembly Language includes two pseudo-instructions: No Operation and Return. The syntax definitions and detailed descriptions of these pseudo-instructions are in the *Model 990 Computer TMS9900 Microprocessor Assembly Language Programmer's Guide*, Manual No. 943441-9701. The pseudo-instructions are summarized in Appendix D.

5.6 ERROR MESSAGES

PX9ASM prints the following error message when it detects an error:

```
** ERR N - STMT XXXX    LAST ERR - STMT YYYY
```

N is an error code as shown in table 5-1. XXXX is the statement number of the source line in which the error was detected. YYYY is the statement number of the source line in which the preceding error, if any, was detected.

Error messages for undefined symbols are printed at the end of the assembly. When a statement allows a forward reference, the reference is not undefined until PX9ASM recognizes an END statement without having recognized a statement defining the symbol. Error messages may be printed at any point, from the lines immediately following the statement in error to lines following the END statement.

Table 5-1. PX9ASM Error Codes

Code	Description
2	Syntax error. The statement corresponding to the error location contains a syntax error.
3	Illegal external reference. The statement corresponding to the error location contains an external reference (and an arithmetic operator) in an expression or an external reference to be placed in a field smaller than 16 bits.
4	Truncation error. The statement corresponding to the error location contains a number that is too large or a character string that is too long. The number may be the result of evaluating an expression. Relocatability of a term or expression may be in error.
5	Multiply defined symbol. A symbol in the statement corresponding to the error location has been previously referenced or defined.
6	Unrecognizable operator. Contents of the operator field of the statement corresponding to the error location is not a mnemonic operation code, a directive, or a name defined as an extended operation.
7	Illegal forward reference. A symbol in the statement corresponding to the error location that should have been previously defined is not previously defined.
8	Illegal term. A term has an illegal value less than zero or greater than 15.

The assembler can accommodate a minimum of 135 symbols in a 4K memory allocation with no predefined registers and 125 symbols in a 4K memory allocation with predefined registers R0 through R15. When the assembler is unable to continue because the area of memory available for symbols and forward references has been filled, the assembler prints the following message:

```
** ABORT **
```

The user may divide the program into two or more modules and assemble them separately. Alternatively, the user may shorten the symbols in the program and reassemble. Since shorter symbols use less space in the symbol table, a symbol table of a given size may contain more, shorter symbols.

Following the last statement of error message, the assembler prints undefined symbols, if there are any, one symbol per line. The undefined symbol may correspond to one of several error codes, or may be a symbol in a DEF directive that does not appear in the label field of a statement.

At the end of the listing is an error summary, as follows:

```
NNNN ERRORS
LAST ERR - STMT XXXX
```

NNNN is the count of errors in the assembly. The second line identifies the last error detected in the assembly. The second lines of the error messages link the error messages so that the user may begin at the error summary message and readily locate all error messages. In an error-free assembly, the final message is:

```
0000 ERRORS
```

5.7 PRINTED OUTPUT

The following paragraphs discuss the source listing and the object code.

5.7.1 **SOURCE LISTING.** The source listings show the source statements and the resulting object code. A typical listing is shown in the example programs in Section VII.

Each page of the source listing has a title line at the top of the page if a title was supplied by a TITL directive. A page number is printed to the right of the title area. The printer skips a line below the title line, and prints a line for each source statement listed. The line for each source statement contains a source statement number, a location counter value, object code assembled, and the source statement as entered. When a source statement results in more than one word of object code, the assembler prints the location counter value and object code on a separate line following the source statement for each additional word of object code. The source listing lines for a machine instruction source statement are shown in the following example:

```
0018  0156  C820    MOV  @INIT+3,@3
      0158  012B'
      015A  0003
```

The source statement number, 0018 in the example, is a four-digit decimal number. Source records are numbered in the order in which they are entered, whether they are listed or not. The TITL, LIST, UNL, and PAGE directives are not listed, and source records between a UNL directive and a LIST directive are not listed. The difference between source record numbers printed indicates how many source records are not listed.

The next field on a line of the listing contains the location counter value, a hexadecimal value. In the example, 0156 is the location counter value. Not all directives affect the location counter, and those that do not affect the location counter leave this field blank. Specifically, of the directives that the assembler lists, the IDT, REF, DEF, DXOP, EQU, and END directives leave the location counter field blank.

The third field contains the hexadecimal representation of the object code placed in the location by the Assembler, C820 in the example. The apostrophe following the third field of the second line in the example indicates that the contents, 012B, is relocatable. All machine instructions and the BYTE, DATA, and TEXT directives use this field for object code. The EQU directive places the value corresponding to the label in the object code field.

In listings printed by PX9ASM, the third field may contain two or four hyphens (-) instead of hexadecimal digits. This occurs when a forward reference determines the values of these digits.

Later, when the forward reference is defined, the assembler prints an additional line in the listing following the statement that defines the forward reference. This line contains the location being resolved, two asterisks (**), and the contents. An error-free listing will include such a line for each location previously printed with hyphens as the contents. The listings printed by the other assemblers do not contain this type of information because all references are either resolved or identified as undefined before the listings are printed.

The fourth field contains the first 60 characters of source statement as supplied to the assembler. Spacing in this field is determined by the spacing in the source statement. The four fields of source statements will be aligned in the listing only when they are aligned in the same character positions in the source statements or when tab characters are used.

The machine instruction used in the example specifies the symbolic memory addressing mode for both operands. This causes the instruction to occupy three words of memory and three lines of the listing. The object code corresponds to the operands in the order in which they appear in the source statement.

5.7.2 OBJECT CODE. The assembler produces standard 990 object code that may be linked to other object code modules or programs and loaded into the Model 990 computer, or may be loaded into the computer directly. Standard 990 object code consists of records containing up to 71 ASCII characters each. The format, described in the next section, permits correction using a keyboard device. Reassembly to correct errors is not always necessary. The object code format is discussed in Section VI.

SECTION VI

OBJECT CODE FORMATS

6.1 INTRODUCTION

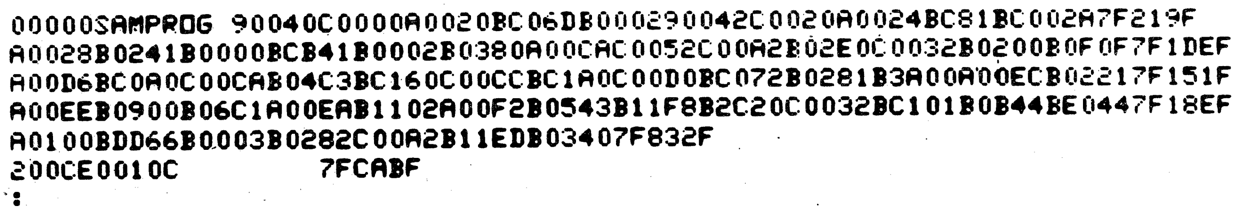
This section describes the two object code formats: standard 990 object code and compressed absolute format object code. The discussion of standard 990 object code covers primarily tag characters. A procedure for changing standard 990 object code is also included. Illustrations of the basic and extended tag formats for compressed absolute format object code are presented.

6.2 STANDARD 990 OBJECT CODE

Standard 990 object code consists of a string of hexadecimal digits, each representing four bits, as shown in figure 6-1.

The object record consists of a number of tag characters, each followed by one or two fields as defined in table 6-1. The first character of a record is the first tag character, which tells the loader which field or pair of fields follows the tag. The next tag character follows the end of the field or pair of fields associated with the preceding tag character. When the assembler has no more data for the record, the assembler writes the tag character 7 followed by the checksum field, and the tag character F, which requires no fields. The assembler then fills the rest of the record with blanks, and begins a new record with the appropriate tag character.

Tag character 0 is followed by two fields. The first field contains the number of bytes of relocatable code, and the second field contains the program identifier assigned to the program by an IDT directive. When no IDT directive is entered, the field contains blanks. The loader uses the program identifier to identify the program, and the number of bytes of relocatable code to determine the load bias for the next module or program. PX9ASM is unable to determine the value for the first field until the entire module has been assembled, so PX9ASM places a tag character 0 followed by a zero field and the program identifier at the beginning of the object code file. At the end of the file, PX9ASM places another tag character zero followed by the number of bytes of relocatable code and eight blanks.



```
00000SAMPROG 90040C0000A0020BC06DB000290042C0020A0024BC81BC002A7F219F
A0028B0241B0000BCB41B0002B0380A00CAC0052C00A2B02E0C0032B0200B0F0F7F1DEF
A00D6BC0A0C00CAB04C3BC160C00CCBC1A0C00D0BC072B0281B3A00A00ECB02217F151F
A00EEB0900B06C1A00EAB1102A00F2B0543B11F8B2C20C0032BC101B0B44BE0447F18EF
A0100BDD66B0003B0282C00A2B11EDB03407F832F
200CE0010C          7FCABF
:
```

(A)132255

Figure 6-1. Object Code Example

Table 6-1. Object Output Tags Supplied by Assemblers

Tag Character	Hexadecimal Field (Four Characters)	Second Field	Meaning
0	Length of all relocatable code	8-character program identifier	Program start
1	Entry address	None	Absolute entry address
2	Entry address	None	Relocatable entry address
3	Location of last appearance of symbol	6-character symbol	External reference last used in relocatable code
4	Location of last appearance of symbol	6-character symbol	External reference last used in absolute code
5	Location	6-character symbol	Relocatable external definition
6	Location	6-character symbol	Absolute external definition
7	Checksum for current record	None	Checksum
9	Load address	None	Absolute load address
A	Load address	None	Relocatable load address
B	Data	None	Absolute data
C	Data	None	Relocatable data
D	Load bias value*	None	Load point specifier
F	None	None	End-of-record
G	Location	6-character symbol	Relocatable symbol definition
H	Location	6-character symbol	Absolute symbol definition

*Not supplied by assembler.

Tag characters 1 and 2 are used with entry addresses. Tag character 1 is used when the entry address is absolute. Tag character 2 is used when the entry address is relocatable. The hexadecimal field contains the entry address. One of these tags may appear at the end of the object code file. The associated field is used by the loader to determine the entry point at which execution starts when the loading is complete.

Tag characters 3 and 4 are used for external references. Tag character 3 is used when the last appearance of the symbol in the second field is in relocatable code. Tag character 4 is used when the last appearance of the symbol is absolute code. The hexadecimal field contains the location of the last appearance. The symbol in the second field is the external reference. Both fields are used by the linking loader to provide the desired linking to the external reference.

For each external reference in a program, there is a tag character in the object code, with a location, or an absolute zero, and the symbol that is referenced. When the object code field contains absolute zero, no location in the program requires the address that corresponds to the reference (an IDT character string, for example). Otherwise, the address corresponding to the reference will be placed in the location specified in the object code by the linking loader. The location specified in the object code similarly contains absolute zero or another location. When it contains absolute zero, no further linking is required. When it contains a location, the address corresponding to the reference will be placed in that address by the linking loader. The location of each appearance of a reference in a program contains either an absolute zero or another location into which the linking loader will place the referenced address.

Figure 6-2 illustrates the chain of the external reference EXTR. The object code contains the following tag and fields:

```
4C00EEXTR
```

At location C00E, the address C00A points to the preceding appearance of the reference. The chain includes both absolute and relocatable addresses and consists of absolute address C00E, C00A, C006, and C002, relocatable addresses 029E, 029A, and 0298, absolute addresses B00E, B00A, B006, and B002, and relocatable addresses 0290 and 028E. Each location points to the preceding appearance, except for location 028E, which contains zero. The zero identifies location 028E as the first appearance of EXTR, the end of the chain.

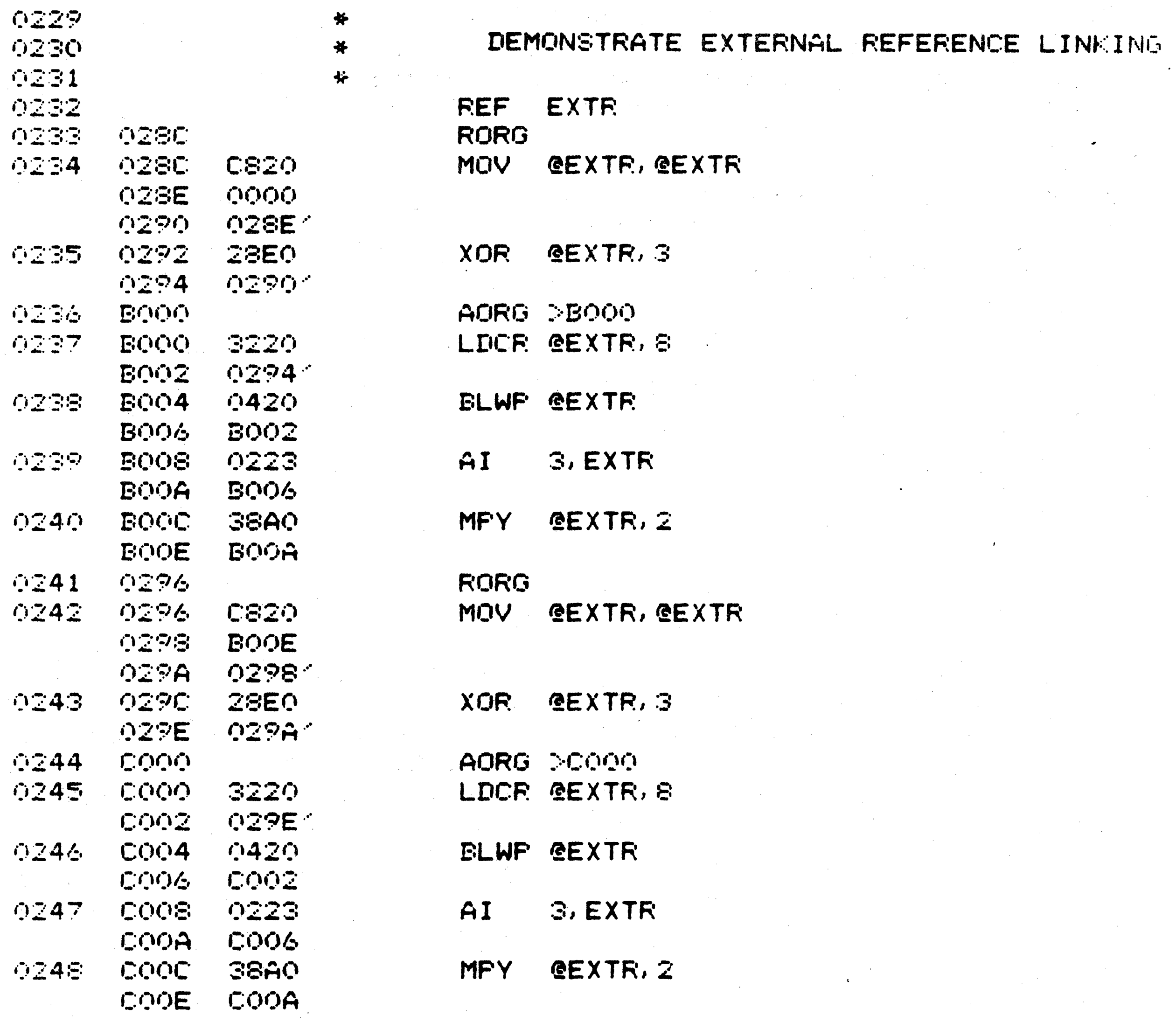
Tag characters 5 and 6 are used for external definitions. Tag character 5 is used when the location is relocatable. Tag character 6 is used when the location is absolute. Both fields are used by the linking loader to provide the desired linking to the external definition. The second field contains the symbol of the external definition.

Tag character 7 precedes the checksum, which is an error detection word. The checksum is formed as the record is being written. It is the 2's complement of the sum of the 8-bit ASCII values of the characters of the record from the first tag of the record through the checksum tag, 7.

Tag characters 9 and A are used with load addresses for data that follows. Tag character 9 is used when the load address is absolute. Tag character A is used when the load address is relocatable. The hexadecimal field contains the address at which the following data word is to be loaded. A load address is required for a data word that is to be placed in memory at some address other than the next address. The load address is used by the loader.

Tag characters B and C are used with data words. Tag character B is used when the data is absolute; an instruction word or a word that contains text characters or absolute constants, for example. Tag character C is used for a word that contains a relocatable address. The hexadecimal field contains the data word. The loader places the data word in the memory location specified in the preceding load address field, or in the memory location that follows the preceding data word.

Tag character F indicates the end of record. It may be followed by blanks.



```
0229                       *
0230                       *         DEMONSTRATE EXTERNAL REFERENCE LINKING
0231                       *
0232                              REF   EXTR
0233   028C                       RORG
0234   028C  C820                 MOV   @EXTR,@EXTR
       028E  0000
       0290  028E'
0235   0292  28E0                 XOR   @EXTR,3
       0294  0290'
0236   B000                       AORG  >B000
0237   B000  3220                 LDCR  @EXTR,8
       B002  0294'
0238   B004  0420                 BLWP  @EXTR
       B006  B002
0239   B008  0223                 AI    3,EXTR
       B00A  B006
0240   B00C  38A0                 MPY   @EXTR,2
       B00E  B00A
0241   0296                       RORG
0242   0296  C820                 MOV   @EXTR,@EXTR
       0298  B00E
       029A  0298'
0243   029C  28E0                 XOR   @EXTR,3
       029E  029A'
0244   C000                       AORG  >C000
0245   C000  3220                 LDCR  @EXTR,8
       C002  029E'
0246   C004  0420                 BLWP  @EXTR
       C006  C002
0247   C008  0223                 AI    3,EXTR
       C00A  C006
0248   C00C  38A0                 MPY   @EXTR,2
       C00E  C00A
```

(A)132256

Figure 6-2. External Reference Example

Tag characters G and H are used when the symbol table option is specified with other 990 assemblers. Tag character G is used when the location or value of the symbol is relocatable, and tag character H is used when the location or value of the symbol is absolute. The first field contains the location or value of the symbol, and the second field contains the symbol to which the location is assigned.

The last record of an object code file has a colon (:) in the first character position of the record, followed by blanks. This record is referred to as an end-of-module separator record.

6.3 PROCEDURES FOR CHANGING STANDARD 990 OBJECT CODE

To correct object code without reassembling a program, change the object code by changing or adding one or more records. One additional tag character is recognized by the loader to permit specifying a load point. The additional tag character, D, may be used in object records changed or added manually.

Tag character D is followed by a load bias (offset) value. The loader uses this value instead of the load bias computed by the loader itself. The loader adds the load bias to all relocatable entry addresses, external references, external definitions, load addresses, and data. The effect of the D tag character is to specify the area of memory into which the loader loads the program. The tag character D and the associated field must be placed ahead of the object code generated by the assembler.

Correction of object code may require only changing a character or a word in an object code record. The user may duplicate the record up to the character or word in error, replace the incorrect data with the correct data, and duplicate the remainder of the record up to the 7 tag character. Because the changes the user has made will cause a checksum error when the checksum is verified as the record is loaded, the user must change the 7 tag character to F.

When more extensive changes are required, the user may write an additional object code record or records. Begin each record with a tag character 9 or A followed by an absolute load address or a relocatable load address, respectively. This may be an address into which an existing object code record places a different value. The new value on the new record will override the other value when the new record follows the other record in the loading sequence. Follow the load address with a tag character B or C and an absolute data word or a relocatable data word, respectively. Additional data words preceded by appropriate tag characters may follow. When additional data is to be placed at a nonsequential address, write another load address tag character followed by the load address and data words preceded by tag characters. When the record is full, or all changes have been written, write tag character F to end the record.

When additional memory locations are loaded as a result of changes, the user must change the hexadecimal field following the tag character 0 that contains the number of bytes of relocatable code. For example, when the object file written by the assembler contained 1000_{16} bytes of relocatable code, and the user has added 8 bytes in a new object record, additional memory locations will be loaded. The user must find the 0 tag character in the object code file and change the value following the tag character from 1000 to 1008; he must also change the 7 tag character to F in that record.

When added records place corrected data in locations previously loaded, the added records must follow the incorrect records. The loader processes the records as they are read from the object medium, and the last record that affects a given memory location determines the contents of that location at execution time.

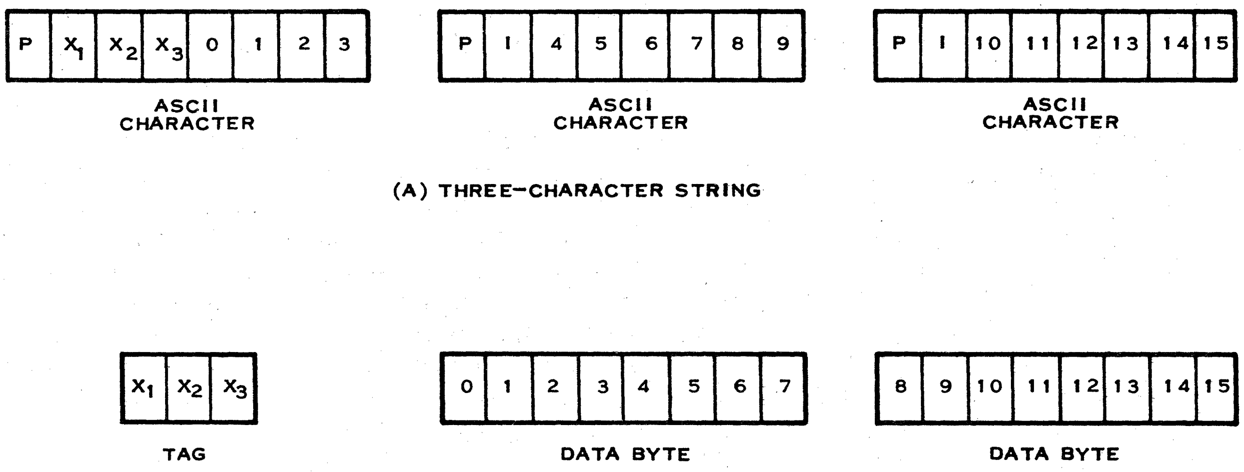
The object code records that contain the external definition fields, the external reference fields, the entry address field, and the final program start field must follow all other object records. An additional field or record may be added to include reference to a program identifier. The tag character is 4, and the hexadecimal field contains zeros. The second field contains the first six characters of the IDT character string. External definitions may be added using tag character 5 or 6 followed by the relocatable or absolute address, respectively. The second field contains the defined symbol, filled to the right with blanks when the symbol contains less than six characters.

6.4 COMPRESSED ABSOLUTE FORMAT OBJECT CODE

Absolute format object code provides the user with a compact object code which can be loaded more rapidly than standard 990 code.

6.4.1 **BASIC TAG FORMAT.** The basic format is a three-character string which maps to a tag and two bytes of data. The formats and tag definitions are shown in figure 6-3.

6.4.2 **EXTENDED TAG FORMATS.** Extended tags (figure 6-4) are used to extend the available data types. An extended tag consists of one or two (the number is tag dependent) bytes which may identify subsequent data. An extended tag with two characters has a six-bit count as the second byte.



(A) THREE–CHARACTER STRING

(B) CHARACTERS AFTER MAPPING

TAG ($X_1X_2X_3$)	MEANING
100	ABSOLUTE DATA WORD (16 BITS)
101	ABSOLUTE DATA BYTE (8 BITS)
110	ABSOLUTE LOAD ADDRESS
111	EXTENDED TAG
	BIT FIELDS
P	PARITY BIT
I	BIT ALWAYS SET TO ONE

THE NUMBERS 0–15 REPRESENT DATA BIT POSITIONS

(A)133108

Figure 6-3. Basic Tag Format

P	I	I	I	X_1	X_2	X_3	X_4

TAG

P	I	C_1	C_2	C_3	C_4	C_5	C_6

COUNT

TAG $(X_1X_2X_3X_4)$	LENGTH (CHARACTERS)	MEANING
0000	1	END OF MODULE
0001	2	PROGRAM NAME (NOTE 1)
0011	1	ABSOLUTE ENTRY ADDRESS (NOTE 2)
0101	1	CHECKSUM (NOTE 2)
0110	2	ABSOLUTE DATA REPEAT COUNT (NOTES 2,3)

BIT FIELDS

P	PARITY BIT
I	BIT ALWAYS SET TO ONE
$X_1X_2X_3X_4$	TAG
$C_1C_2C_3C_4C_5C_6$	COUNT–NUMBER OF BYTES OF DATA

NOTES

1. FOLLOWED BY CHARACTERS OF NAME. BYTE 2 IS THE NUMBER OF CHARACTERS IN THE NAME, UP TO A MAXIMUM OF 17.
2. FOLLOWED BY ABSOLUTE DATA TAG AND 16 BITS OF DATA IN THE BASIC FORMAT.
3. WHEN THE SEQUENCE OF IDENTICAL WORDS IS ENCOUNTERED DURING THE DUMP, A REPEAT COUNT IS COMPUTED SO THAT THE DATA NEED NOT BE REPEATED. BYTE 2 IS THE NUMBER OF IDENTICAL WORDS.

(A)133109

Figure 6-4. Extended Tag Formats

SECTION VII

PROM PROGRAMMER

7.1 INTRODUCTION

This section describes the PROM Programmer software module and includes the following information:

- General description, including the functions and capabilities of the module, the role of the Standard Control Information Cassette in PROM programming, and an explanation of standard and nonstandard data configurations.
- Loading procedure.
- Detailed discussion of the PROM programming process, covering data formats, PROM and ROM characteristics, mapping parameters, and examples of different levels of looping.
- Detailed descriptions of the PROM Programmer commands.
- Methods for performing some specific programming tasks, such as standardizing nonstandard memory and PROM configurations, and programming EPROMs.
- Programming examples.

7.2 GENERAL DESCRIPTION

The PROM Programmer software module (PROMPG) controls the PROM Programming Module used with the 990 Computer Family. It provides flexible user control of the programming process as well as standardized programming options. PROMPG operates on a prototyping system containing a 990/4 Computer, 733 ASR Data Terminal, and a PROM Programming Module. This software package is an overlay that is loaded into the PX9MTP transient area of memory and extends into the high address locations of user memory.

7.2.1 FUNCTIONS AND CAPABILITIES. PROMPG has a set of commands that perform the following functions:

- Describe standard data configuration in memory and PROM.
- Describe nonstandard data configurations in memory and PROM.
- Provide information for the PROM Programming Module.

With PROMPG, the user can:

- Program data from memory into a PROM.
- Store data from a PROM or ROM into memory.

- Display data from memory.

- Display data from ROM or PROM.

- Compare data in memory and PROM or ROM.

The software package includes a Standard Control Information Cassette that:

- Contains control information for standard data configurations in memory and PROM.

- Supports all PROMs which are supported by hardware programming adaptor cards.

The software package allows the user to replace or add control information to the Standard Control Information Cassette.

7.2.2 STANDARD CONTROL INFORMATION CASSETTE. The Standard Control Information Cassette contains the control information for the most commonly used memory and PROM data configurations. Included in these is information necessary to program all PROMs which are supported by hardware programming adaptor cards.

Each record on the Standard Control Information Cassette contains a memory or PROM designator, a label, the bit string width, and mapping parameters. Records containing PROM control information also contain PROM characteristics. Appendix G contains a table of all the standard configurations on the Standard Control Information Cassette and two other tables which contain additional information about the supported configurations.

7.2.3 PROGRAMMING STANDARD VERSUS NONSTANDARD DATA CONFIGURATIONS. The control information needed to transfer data between memory and PROM may be supplied in one of two ways:

- By reading the information from the Standard Control Information Cassette.

- By specifying the information through the PROM programmer keyboard commands.

Standard data configurations are those configurations which are defined on the Standard Control Information Cassette. Nonstandard data configurations are those which are not defined on the Standard Control Information Cassette.

To program standard data configurations, the necessary control information is read from the Standard Control Information Cassette using the PROM Programmer Standard (PS) command. When programming nonstandard data configurations, the necessary control information may be input using the Define Memory Data Configuration Mapping Parameters (MI), Define ROM/PROM Data Configuration Mapping Parameters (RI), Define String Width (SW), and Define PROM/ROM Characteristics (RC) subcommands.

Once the control information is specified by one of the above methods, the memory and PROM bounds may be set with the Define Memory Bounds (MB) and Define PROM/ROM Bounds (RB) subcommands. The appropriate actions may be specified with the Set Toggles (TS) subcommand and the programming cycle initiated with the Go (GO) subcommand.

7.2.4 PROM PROGRAMMER FUNCTIONS. The PROM Programmer software package allows the user to perform one or more of the following functions simultaneously:

- Perform one of three data transfers:

  (1) Program PROMs.

  (2) Read PROM or ROM data into memory.

  (3) Store nonstandard memory and PROM control information on the Standard Control Information Cassette.

- Display data from memory.

- Display data from PROM or ROM.

- Compare data in memory and PROM or ROM.

The functions to be performed during the programming cycle may be specified with the TS subcommand before the programming cycle is initiated.

7.3 PROM PROGRAMMER LOADING PROCEDURE

The Program, PROMPG, consists of an overlay module and an extension which is loaded into the top of user memory. PROMPG is loaded by means of the Load PROM Programmer (PL) command, which is described in detail in Section III.

Mount and position the cassette containing the PROMPG object code and enter this command on the terminal keyboard:

```
PL {ƀ'...} [<luno> [{ƀ'...} <bias>]]
```

where luno is the logical unit number of the cassette drive on which PROMPG is mounted and bias is the load address for the extension. The default LUNO value is 7. If the bias is not given, the PX9MTP loader loads the extension into the top of user memory at default bias $1C80_{16}$.

When the PL command is issued, the overlay will be loaded and the following printed:

```
PP
PS
```

The memory extension will then be loaded into user memory at the specified bias address.

7.4 PROM PROGRAMMING PROCESS

The PROM programming process allows the user to transfer data from memory to a PROM or vice versa and to display or compare memory and PROM/ROM data. To accomplish these tasks, certain control information must be specified. The information includes memory and PROM/ROM bounds, bit string width, PROM/ROM characteristics, and mapping parameters. The control information may be specified using the PROM programmer keyboard commands and/or by reading in the information from the Standard Control Information Cassette.

7.4.1 BIT STRING WIDTH. Bit strings are the basic unit of data moved between the 990 memory and the PROM. The bit string width specifies the number of bits to be transferred during a programming cycle. The width may be from one to eight bits.

7.4.2 MEMORY AND PROM/ROM BOUNDS. The memory bounds specify the memory locations which contain the data to be transferred to or from PROM. The PROM/ROM bounds define the lower and upper bounds. PROM/ROM addresses are numbered by words; the word size is determined by the PROM/ROM word width.

7.4.3 PROM/ROM CHARACTERISTICS. Each PROM/ROM has a different set of characteristics which must be specified to transfer data to and from the PROM/ROM. The characteristics include word width, output conditions, pulse width, number of retries, duty cycle, and programmable bit width.

- The word width refers to the number of bits per word in the PROM/ROM physical organization. For example, the SN74S287 PROM (256 X 4) has a word width of four bits.

- The output conditions specify whether high or low level logic outputs are to be programmed. The value is 0 if low and 1 if high. Some PROMs are initialized to ones and must be programmed with zeros (low level logic).

- The pulse width is entered as an index value which is used by the hardware to produce the corresponding pulse width in milliseconds to be used in programming PROMs.

- The number of retries refers to the number of times PROM programming is to be retried using the specified pulse width if a programming failure occurs.

- The programming cycle includes the programming time and a delay time. The duty cycle is the percentage of the total time that the programming pulse is on. The typical duty cycle varies between 16 percent and 50 percent.

- The programmable bit width specifies the number of bits that can be physically programmed simultaneously. The programmable bit width cannot be greater than the bit string width.

CAUTION

Errors may be introduced if the programmable bit width is too large for certain PROMs. For example, TTL PROMs require a programmable bit width of one. (Bit widths are listed in Appendix G.)

7.4.4 MAPPING PARAMETERS. The memory and PROM/ROM mapping parameters are used by the software to determine the addresses of the bit strings to be used in the programming cycle. In specifying mapping parameters, the PROM/ROM or memory words within the defined bounds are considered to be a continuous string of bits. Mapping is needed so that portions of 16-bit memory words may be programmed into PROMs with smaller word widths. The mapping parameters include bit increments, number of iterations, and initial bit displacements for each of three loop levels.

- The initial bit displacement is used to determine the starting bit address of the bit string to be transferred between PROM/ROM and memory.

- The bit increments are used to determine the successive bit addresses of the bit strings to be transferred between PROM/ROM and memory.
- The number of iterations is the number of bit strings to be transferred between PROM/ROM and memory.

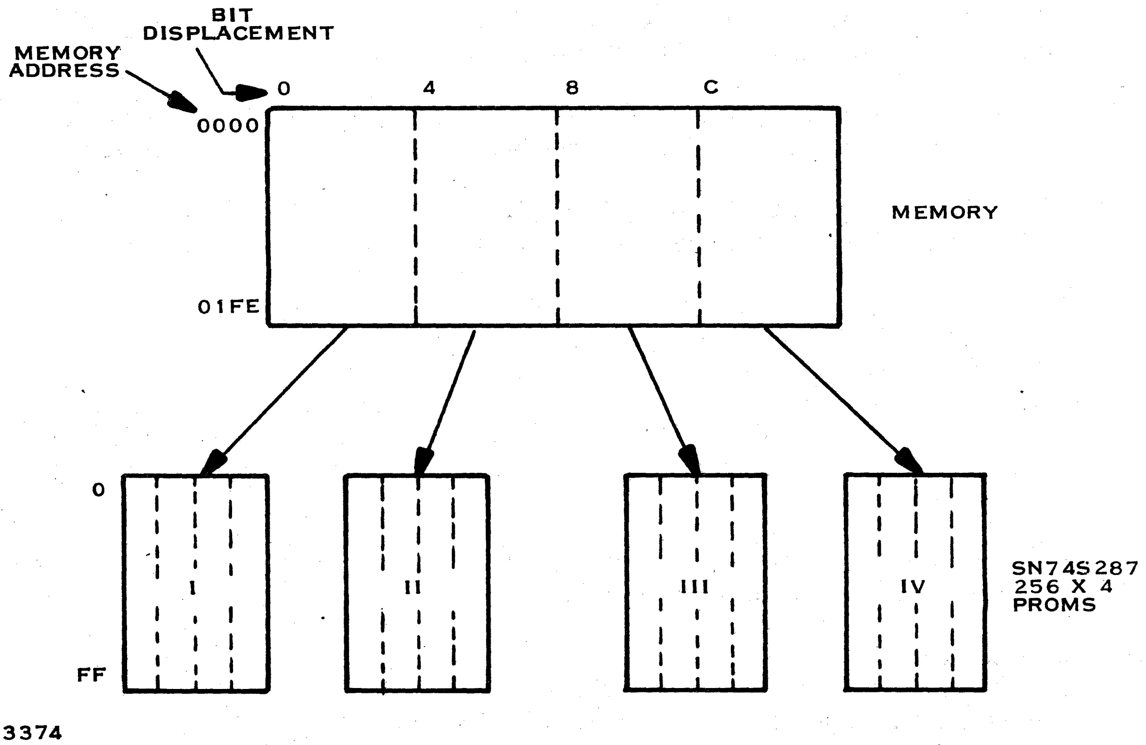
The number of bits in each bit string transferred between PROM/ROM and memory is defined by the bit string width.

Three levels of mapping are allowed where each level contains an initial bit displacement, a bit increment and an iteration count.

Level 1 is used to define the mapping pattern. Level 2 is used to repeat the pattern generated by the level 1 parameters. Level 3 is used to repeat the total pattern generated by levels 1 and 2.

Bit increments and bit displacements on level 2 and 3 are used to determine the initial addresses of the bit strings defined by level one. The number of iterations on levels 2 and 3 are used to determine the number of times to repeat the bit string pattern. This three-level looping scheme is analogous to FORTRAN nested DO loops with level one defining the innermost DO loop and level three the outermost DO loop.

The following is an example of the mapping parameters required to transfer memory into PROM using SN74S287 PROM devices which are 256 X 4 (256 PROM words of 4 bits each). Refer to figure 7-1.



(A)133374

Figure 7-1. Transfer of Data from Memory into PROM

The PROM/ROM bounds are set to 0 and FF_{16}, giving a string of 256 X 4 = 1024 bits. The memory bounds are set to 0 and $1FF_{16}$, giving a memory string of 256 X 16 = 4096 bits. The bit string width is set to 4, which is equal to the PROM word width. The PROM/ROM mapping parameters are the following:

Initial bit displacement	= 0
Bit increment	= 4
Number of iterations	= 100_{16}

The initial bit displacement is 0 so that the string will be transferred to the PROM starting at bit 0 of the PROM word. The bit increment is 4 so that each string of 4 bits will be stored consecutively in the PROM. The number of iterations is 100_{16} (256_{10}) so that 256 bit strings of 4 bits each will be transferred.

The memory mapping parameters for PROM I are the following:

Initial bit displacement	= 0
Bit increment	= 10_{16}
Number of iterations	= 100_{16}

The initial bit displacement selects the bit string starting at bit 0 of the memory word. The bit increment is 10_{16} (16_{10}) so that each successive bit string will begin at bit 0 of each 16-bit memory word. The number of iterations is 100_{16} (256_{10}) so that 256 memory bit strings of 4 bits each will be transferred.

Initial bit displacements of 4_{16}, 8_{16}, and C_{16} are used to transfer the remaining bits of the memory words to PROMs II, III, and IV, respectively. These bit displacements will select the 4 bit strings beginning at bits 4_{16}, 8_{16}, and C_{16} of each 16-bit memory word.

The mapping parameters in this example define the level one looping. Level two and three looping are not used in this example since the pattern defined by the level one looping is not repeated.

7.4.5 EXAMPLES USING ONE, TWO, AND THREE-LEVEL LOOPING. The following examples take the user through the PROM programming process step by step using various levels of looping.

7.4.5.1 One-Level Looping. An address is determined in the following manner. Starting with the beginning memory/PROM/ROM address (indicated by the MB or RB subcommand, paragraph 7.5.3.1 or 7.5.3.2), the initial bit displacement for loop level 1 is added to that address. The resulting bit address is the beginning address of the first bit string. To get each consecutive bit string beginning address, the number of bits indicated by the bit increment value for loop level 1 is added to the previous address. This process continues until the number of addresses determined has reached the maximum value for loop level 1.

As an example, assume a user wishes to select the first four bits of each word of a 256-word block of data in memory. The parameters needed would be:

Loop level	= 1
Bit increment (1)	= 16 (10_{16})
Number of iterations (1)	= 256 (100_{16})
Initial bit displacement (1)	= 0
Bit string width	= 4

Assume the beginning memory address is 0000_{16}. Since the displacement is 0, the first bit string begins at bit 0 and consists of the first four bits at byte 0. The beginning address of the second bit string is determined by adding the bit increment 10_{16} to the previous beginning address of 0. The second four-bit string addressed begins at memory location 0002_{16}. The bit increment of 10_{16} is repeatedly added to the previous address until 100_{16} four-bit strings have been selected from the memory block. The last bit string will be the first four bits at address $01FE_{16}$.

In another example, a user may wish to select four bit strings, each four bits wide, from one word of memory. The parameters needed would be:

Loop level	= 1
Bit increment (1)	= 4
Number of iterations (1)	= 4
Initial bit displacement (1)	= 0
Bit string width	= 4

Assume the beginning memory address is 0100_{16}. Since the displacement is 0, the first bit string will be the first bits of memory address 0100_{16}. By adding the bit increment of four, the second bit string is determined to be the second four bits of memory address 0100_{16}. The third and fourth bit strings are determined similarly.

Consider what would happen if the user decided to select similar strings from 32 words of memory memory beginning with address 0100_{16}. By changing the number of iterations to 128 (4 X 32) and beginning with address 0100_{16}, consecutive four bits of each word could be selected. The parameters needed would be:

Loop level	= 1
Bit increment	= 4
Number of iterations	= 128 (80_{16})
Initial bit displacements	= 0
Bit string width	= 4

By incrementing four bits at a time and selecting four 4-bit strings from each word for 32 words, the 128 four-bit strings can be addressed.

7.4.5.2 Two-Level Looping. The programming sequence in the last example can also be done by two-level looping, a concept which involves using the parameters of loop level 2 to determine a number of beginning addresses. Each address determined by loop level 2 is used as a beginning address for loop level 1.

In the last example the parameters would now be:

Loop level	= 1
Bit increment (1)	= 4
Number of iterations (1)	= 4
Initial bit displacement (1)	= 0
Bit string width	= 4
Loop Level	= 2
Bit increment (2)	= 16 (10_{16})
Number of iterations (2)	= 32 (20_{16})
Initial bit displacement (2)	= 0
Bit string width	= 4

Assume that the beginning memory word is at location 0100_{16}. The displacement in loop level 2 is 0; therefore, the first beginning address to be used by loop level 1 is 0100_{16}. Since loop level 1 has a displacement of 0, the first bit string has a beginning address of bit 0 of address 0100_{16}. Proceeding by adding the bit increment of 4 to each bit address, the next three bit strings can be selected. Loop level 1 has now been completed. Going back to loop level 2 and the previous beginning address in memory (bit 0 of address 0100_{16}), add the bit increment of 10_{16} to that address. The new beginning address in memory is bit 0 of address 0102_{16}, which is now used by loop level 1 to select the next four bits strings. When those strings have been selected, loop level 2 then determines the third beginning address in memory by adding the bit increment 10_{16} to the previous address of bit 0 of address 0102_{16}. Selecting a new beginning address and using that address to increment through loop level 1, loop level 2 continues until 20_{16} beginning addresses have been selected and loop level 1 has been processed 20_{16} times.

7.4.5.3 **Three-Level Looping.** Loop level 3 can be used for reiterative programming. Assume the user wishes to program a 256 by 4 PROM using the same memory data configuration as in the previous example. Since the previous example only selects 128 four-bit strings, the first 128 words and the last 128 words of the PROM could be programmed with the same data from memory.

The PROM data configuration is standard and the control information can be read from the Standard Control Information Cassette. The memory data configuration would have the following parameters:

Loop level	= 1
Bit increments (1)	= 4
Number of iterations (1)	= 4
Initial bit displacement (1)	= 0
Bit string width	= 4

Loop level	= 2
Bit increment (2)	= 10_{16}
Number of iterations (2)	= 20_{16}
Initial bit displacement (2)	= 0
Bit string width	= 4

Loop level	= 3
Bit increment (3)	= 0
Number of iterations (3)	= 2
Initial bit displacement (3)	= 0
Bit string width	= 4

Assuming the beginning memory address is 0100_{16}, the loop level 3 displacement 0 is added to that address to get the beginning address for loop level 2. The increment described for two-level looping now is performed. When the incrementing is complete (128 bit strings of four bits have been selected), loop level 3 then determines the next beginning address for loop level 2. Since the bit increment for loop level 3 is 0, the second beginning address is the same as the first one. Therefore the two-level looping increments through the same memory configuration and selects the same 128 bit strings to program the second 128 four-bit words of the PROM.

The standard PROM control information for a 256 by 4 PROM includes the parameters:

Loop level	= 1
Bit increment (1)	= 4
Number of iterations (1)	= 256 (100_{16})
Initial bit displacement (1)	= 0
Bit string width	= 4

Since the PROM is four bits wide, each increment of four bits gets a new bit string address which also is a new PROM word address. Therefore, as bit strings are selected from memory, they are programmed into consecutive words of PROM for 256 words.

7.5 COMMANDS

The following paragraphs contain detailed descriptions of the PROM Programmer commands. The following symbols and conventions are used in defining the syntax of the commands:

- Angle brackets (<>) enclose items supplied by the user.
- Brackets ([ ]) enclose optional items.
- An ellipsis (. . .) indicates that the preceding item may be repeated.
- Braces ({ }) enclose two or more items of which one must be chosen.

7.5.1 PROM PROGRAMMER STANDARD (PS). The PROM Programmer Standard command searches the Standard Control Information Cassette for the specified records which contain the memory and/or PROM control information.

Syntax definition:

```
PS {ƀ'...}  <char string 1>  [{ƀ'...}  <char string 2>]
```

Parameters:

char string 1 — Name of first record of control information for a standard PROM or memory configuration. Required parameter.

char string 2 — Name of second record of control information for a standard PROM or memory configuration.

Parameter default value:

If char string 2 is not specified, it is omitted.

Description: This command is used to input the control information for the standard memory and PROM data configurations. The user may specify a search of the tape for both memory and PROM control information to be used in the programming process or may specify a search for only memory or PROM control information. If only one data configuration, either memory or PROM, is specified, any control information previously defined for the other type of data configuration remains unchanged.

When all character strings given in the command have been matched to a record on the Standard Control Information Cassette, control is returned to the monitor. The user need not rewind the cassette if the next record of information to be read from the cassette when the user inputs the command again is positioned further along the cassette from the last record which was read.

The Standard Control Information Cassette must be positioned on the cassette drive assigned to logical unit number 7. Records are read from the cassette, and if a record with a name matching either character string is found, the record is stored for use by the program. The search is continued from that point for the other character string if the string is specified in the command. If a record with a matching name is found, the record is stored for use by the program.

The Standard Control Information Cassette and its contents are explained in paragraph 7.2.2 and Appendix G.

Error messages:

PP01 Required parameter missing.

PP03 Bit string widths of memory and PROM configurations do not match.

PP04 Specified record not found on Standard Control Information Cassette.

Application note: If the two character strings specify control information for data configurations both in ROM or both in memory, the control information of the second configuration encountered on the cassette overrides the first.

Example:

```
.PS,MS287-0,S287
```

This command causes the Standard Control Information Cassette to be searched for records containing the control information for memory configuration MS287-0 and PROM configuration S287.

7.5.2 PROM PROGRAMMER (PP). The PROM Programmer command is followed by PROM Programmer subcommands and allows the operator to control the PROM programming process.

Syntax definition:

PP {ƀ'...} <subcommand>

The command is terminated by a carriage return. The command is followed by a subcommand and the appropriate parameters. Refer to the descriptions of the individual subcommands for the syntax definitions.

Parameter:

subcommand Subcommand used with the PP command.

Description: The PROM programming functions are explained in the descriptions of the individual subcommands.

7.5.3 PROM PROGRAMMER SUBCOMMANDS. The following paragraphs contain detailed descriptions of the PROM Programmer subcommands.

7.5.3.1 Define Memory Bounds (MB). The Define Memory Bounds subcommand informs the control software of the lower and upper bounds of the memory data to be used in the programming process.

Syntax definition:

PP {ƀ’. . .} MB {ƀ’. . .} <lower bound> {ƀ’. . .} <upper bound>

Parameters:

lower bound — Byte address of the first byte of the block of memory which contains the memory data configuration. Required parameter. Hexadecimal number.

upper bound — Byte address of the last byte of the block of memory which contains the memory data configuration. Required parameter. Hexadecimal number.

Description: This command defines the lower and upper bounds of the block of memory which contains the memory data configuration. Any bit string to be transferred must be contained entirely in this specified region. An attempt to reference a bit string out of these memory bounds during a programming cycle will cause an error. The lower bound is used as a starting address for the data configuration. When the PROM programmer is loaded, the lower and upper bounds default to 0 and FFF$_{16}$ respectively.

Error messages:

PP01	Required parameter missing.
PP04	Invalid address. The upper bound is less than the lower bound.

Example:

```
.PP,MB,500,5FF
```

This command informs the software that the lower bound of the memory data configuration is 500$_{16}$ and the upper bound of the memory data configuration is 5FF$_{16}$.

7.5.3.2 Define PROM/ROM Bounds (RB). The Define PROM/ROM Bounds command informs the control software of the lower and upper ROM or PROM bounds to be used.

Syntax definition:

PP {ƀ’. . .} RB {ƀ’. . .} <lower bound> {ƀ’. . .} <upper bound>

Parameters:

lower bound Word address of the first physical PROM/ROM word of the block of PROM/ROM words which contains the PROM/ROM data configuration. Required parameter. Hexadecimal number. Initially, the parameter value is 0.

upper bound Word address of the last physical PROM/ROM word of the block of PROM/ROM words which contains the PROM/ROM data configuration. Required parameter. Hexadecimal number. Initially, the parameter value is FFF_{16}.

Description: This command defines the lower and upper bounds of the block of PROM/ROM words which contains the PROM/ROM data configuration. Any bit string referenced must be contained entirely within this specified region. An attempt to reference a bit string out of these bounds during a programming cycle will cause an error.

When the PROM Programmer is loaded, the default values of the lower and upper bounds are 0 and FFF_{16} respectively. If only standard PROM/ROM configurations, which always begin at address 0, are being used, the RB subcommand is not needed. The programming cycle will stop when the region defined by the mapping parameters has been satisfied.

Error Messages:

PP01 Required parameter missing.

PP04 Invalid address. The upper bound is less than the lower bound.

Example:

```
.PP,RB,10,20
```

This command informs the software that the lower bound of the PROM/ROM data configuration is 10_{16} and the upper bound of the PROM/ROM data configuration is 20_{16}.

7.5.3.3 Set CRU Interface Base Address (CS). The Set CRU Interface Base Address command informs the control software of the CRU base address for the PROM Programming Module.

Syntax definition:

PP {b̸'...} CS {b̸'...} <base addr>

Parameter:

base addr The parameter value indicates the CRU base address for the chassis slot in which the PROM programming module interface card is inserted. Required parameter.

Description: When the PROM Programmer is loaded, the base address parameter value is 020_{16}, which is the CRU base address for the chassis slot most frequently used to hold the PROM programming module interface card. After the CS subcommand is used, the software recognizes the given CRU address until a different address is entered with the CS subcommand. There can be no interaction with the PROM programming module unless the control software is informed by default or by the CS subcommand of the correct CRU base address.

Error messages:

PP01 Required parameter missing.

PP02 Base address is greater than $1FFE_{16}$.

Example:

```
.PP,CS,0E0
```

This command informs the control software that the CRU base address of the PROM programming module is $0E0_{16}$.

7.5.3.4 **Set Toggles (TS).** The Set Toggles subcommand sets numeric parameters that inform the control software of the actions to be taken. These numeric parameters are known as toggles. The selected actions are not actually initiated until the PP, GO command is entered.

Syntax definition:

```
PP {ƀ’...} TS [[{ƀ’...} [<mem disp>][[{ƀ’...} [<prom disp>][[{ƀ’...} [<transfer>]
   [[{ƀ’...} <compare>]]]]
```

Parameters:

mem disp Value that specifies whether memory bit strings and addresses are to be displayed. The value is 0 if no memory strings are to be displayed and is 1 if memory bit strings and addresses are to be displayed.

prom disp Value that specifies whether PROM or ROM bit strings and addresses are to be displayed. The value is 0 if no ROM or PROM strings are to be displayed and is 1 if ROM or PROM bit strings and addresses are to be displayed.

transfer Value that specifies the data transfer option:

0 No data transfer between memory and ROM or PROM.

1 PROM is to be programmed from memory data configuration.

2 Memory is to be loaded from ROM or PROM.

3 Nonstandard control information is to be stored on the Standard Control Information Cassette. (Refer to paragraph 7.6.1)

compare Value that specifies whether ROM or PROM bit strings are to be compared to bit strings in the memory data configuration. The value is 0 if no comparison is to be made and 1 if a comparison is to be made. The strings specified by mapping parameters and bit string width are compared. If a comparison fails, the unmatched bit strings and their addresses are to be displayed.

Parameter default values:

If a toggle parameter is not specified, the value specified by a previous TS subcommand or the default value when the PROM Programmer overlay was loaded is used. The default values set up when the PROM Programmer is loaded are the following:

mem disp	= 0	(No display)
prom disp	= 0	(No display)
transfer	= 1	(PROM is to be programmed from memory)
compare	= 1	(Compare bit strings in memory to PROM or ROM)

Description: The toggle parameters specify the action to be taken when the GO subcommand is entered. If the memory display toggle is set, the memory region specified by the memory bounds, bit string width, and the mapping parameters is displayed in the following format.

Mxxxx.yy=zz

where

xxxx = memory byte address

yy = displacement of start of bit string within memory byte ($0 \leqslant yy \leqslant 7$)

zz = right justified bit string (displayed in hexadecimal)

A maximum of four entries may be displayed per line.

If the PROM display toggle is set, the PROM or ROM region specified by the PROM/ROM bounds, bit string width, and the mapping parameters is displayed in the following format.

Raaaa.bb=cc

where

aaaa = PROM/ROM word address

bb = displacement of start of bit string within PROM/ROM word

cc = right justified bit string (displayed as hexadecimal)

A maximum of four entries may be displayed per line.

The transfer toggle specifies the type of data transfer to be performed during the programming cycle. The user may specify programming PROM from memory, loading memory from PROM or ROM, or saving control information for a memory or PROM data configuration on the Standard Control Information Cassette. (Refer to paragraph 7.6.1 for further explanation of this process.) The user may specify no data transfer if only a memory and PROM or ROM comparison or display are desired. If the transfer toggle is set to 1 or 2, data transfer occurs in the memory or PROM/ROM region specified by the memory bounds, bit string width, and mapping parameters.

If the compare toggle is set, the memory and PROM or ROM regions specified by the memory and PROM/ROM bounds, bit string width, and mapping parameters are compared. Any compare errors found are displayed in the following format.

```
>Mxxxx.yy=zz Raaaa.bb=cc
```

The fields for memory and PROM or ROM are the same as defined for the display toggles. One entry of compared data preceded by a *greater than* character is displayed per line. The *greater than* character alerts the user to the compare error. Each entry contains the memory and PROM or ROM contents which failed to compare.

The user may terminate any display by pressing the escape (ESC) key. Control of the program returns to the monitor. Also, if the transfer toggle is set to 3 to save control information on the Standard Control Information Cassette and the user decides not to save the information, the user may reply to the PROM Programmer questions

MEM ID?

or

ROM ID?

with an ESC character. (Refer to paragraph 7.6.1.) The ESC character causes an exist.

Examples:

```
.PP,TS,1,0,0,0
.PP,GO
M0000.00=00  M0002.00=00  M0004.00=00  M0006.00=04
M0008.00=01  M000A.00=00  M000C.00=00  M000E.00=00
.
```

In this example, the memory display toggle is set. When the programming cycle is initiated, the memory region is displayed. The mapping parameters for this region are defined with an initial displacement of 2, bit increment of 10_{16}, and number of iterations set to 8_{16}. The bit string width is set to 4. This example shows that each memory byte address displayed contains the bit string (shown to the right of the equal sign) in bits 2, 3, 4 and 5 of the memory byte. Memory location 6 contains the following bit string: xx0100xx.

```
PP,TS,0,1,0,0
.PP,GO
R0000.00=0F  R0001.00=0F  R0002.00=0F  R0003.00=0F
R0004.00=0F  R0005.00=0F  R0006.00=0F  R0007.00=0F
R0008.00=0F  R0009.00=00  R000A.00=01  R000B.00=01
R000C.00=0C  R000D.00=03  R000E.00=03  R000F.00=00
.
```

In this example, the PROM display toggle is set. When the programming cycle is initiated, the PROM/ROM region defined by the mapping parameters is displayed. The mapping parameters are defined with initial displacement set to 0, bit increment set to 4, and number of iterations set to 10_{16}. The bit string width is set to 4.

```
.PP,TS,0,0,1,1
.PP,GO
>M0000.00=03  R0000.00=01
>M0008.00=04  R0004.00=00
>M000A.00=02  R0005.00=00
>M0022.00=0C  R0011.00=08
>M0042.00=05  R0021.00=04
.
```

In this example, the transfer toggle is set to program PROM from memory and the compare toggle is also set. When the programming cycle is initiated, one bit string at a time will be transferred from memory to PROM until the mapping parameters have been satisfied. After each string is transferred, the value is read back from PROM and compared to the memory bit string. In this example, some compare errors were found during the cycle and the corresponding memory and PROM contents were displayed.

```
PP,TS,0,0,2,1
.PP,GO
.
```

In this example, the transfer toggle is set to load memory from PROM or ROM and to compare memory to the PROM or ROM. No compare errors were found in this example.

7.5.3.5 Go (GO). The Go subcommand initiates the programming cycle specified by the Set Toggles (TS) subcommand.

Syntax definition:

PP $\left\{ b', \ldots \right\}$ GO

Description: When the GO subcommand is entered, the memory and PROM/ROM control information is checked, and the programming cycle defined by the toggles is initiated. The PROM programmer software initiates no transfer of data until this subcommand is entered.

Error messages:

MX01 Tape I/O error, or unrecoverable I/O error.

PP02 Mapping parameters specified a bit string out of the defined memory or PROM/ROM bounds. An example is an attempt to program 512 words of a PROM with the PROM boundaries indicated as 100_{16} through $1FF_{16}$ (256 words).

PP03 The bit string width parameters for memory and PROM or ROM do not match, or the total number of bit strings in the PROM/ROM and memory data configuration defined by mapping parameters do not match. An example is an attempt to map a PROM data configuration containing 256 bit strings from a memory data configuration 512 bit strings.

PP05 Hardware error.

PP06 PROM programming module is not on line.

Example:

```
.PP,GO
```

This command initiates the programming cycle.

7.5.3.6 Define Memory Data Configuration Mapping Parameters (MI). The Define Memory Data Configuration Mapping Parameters subcommand defines the control information needed to determine the addresses of the bit strings in the memory data configuration to be used in the programming cycle.

Syntax definition:

PP {ƀ'...} MI {ƀ'...} <level n> [{ƀ'...}[<imn>][{ƀ'...} [<mmn>][{ƀ'...} <dmn>]]]

Parameters:

level n — Memory mapping level indicator. Its value is 1, 2, or 3. Required parameter.

imn — Bit increment used to determine the successive bit addresses of the bit strings to be used in the programming cycle for the level specified by the level n parameter. Hexadecimal number.

mmn — The number of bit strings to be used in the programming cycle for the level specified by the level n parameter. Hexadecimal number.

dmn — Initial bit displacement used to determine the starting bit address of the first bit string to be used in the programming cycle for the level specified by the level n parameter. Hexadecimal number.

Parameter default values:

If imn is not specified, a value of 0 is used.

If mmn is not specified, a value of 1 is used.

If dmn is not specified, a value of 0 is used.

Description: This subcommand is used to specify the memory mapping parameters for a data configuration not defined on the Standard Control Information Cassette or to modify the mapping parameters of a configuration input from the Standard Cassette. The memory data configuration mapping parameters are explained in detail in paragraph 7.4.4 and Appendix F. The command parameters imn, mmn and dmn correspond to IM_n, MM_n and DM_n in the computations in Appendix F.

If a two- or three-level data configuration mapping has been specified and the user wishes to specify a data configuration using only level one mapping, the looping parameters for levels two and three must be reset to the default values. If three-level mapping has been previously specified and level two mapping will be used, the looping parameters for level three must be reset to the default values. This can be accomplished by typing the MI subcommand and level, leaving off any of the looping parameters. The following commands:

```
.PP,MI,2
.PP,MI,3
```

reset the looping parameters for levels two and three and allow the user to proceed with level one programming.

Error message:

PP02 Parameter value outside the permissible range.

Examples:

```
.PP,MI,1,10,100,4
.PP,MI,2,0,2
```

The first example defines the mapping parameters as follows:

Loop level	= 1
Bit increment	= 10_{16} = 16_{10}
Maximum iteration count	= 100_{16} = 256_{10}
Bit displacement	= 4

The second example defines the mapping parameters as follows:

Loop level	= 2
Bit increment	= 0
Maximum iteration count	= 2
Bit displacement	= 0 (default)

7.5.3.7 Define PROM/ROM Data Configuration Mapping Parameters (RI). The Define PROM/ROM Data Configuration Mapping Parameters subcommand defines the control information needed to determine the addresses of the bit strings in the PROM/ROM data configuration to be used in the programming cycle.

Syntax definition:

PP {ƀ'...} RI {ƀ'...} <level n> [[{ƀ'...} [<irn>][[{ƀ'...} [<mrn>][[{ƀ'...} <drn>]]]

Parameters:

level n PROM/ROM data configuration mapping level indicator. Its value is 1, 2 or 3. Required parameter.

irn	Bit increment used to determine the successive bit addresses of the bit strings to be used in the programming cycle for the level specified by level n. Hexadecimal number.

mrn	Number of bit strings to be used in the programming cycle for the level specified by level n. Hexadecimal number.

drn	Initial bit displacement used to determine the starting bit address of the first bit string used in the programming cycle for the level specified by level n. Hexadecimal number.

Parameter default values:

If irn is not specified, a value of 0 is used.

If mrn is not specified, a value of 1 is used.

If drn is not specified, a value of 0 is used.

Description: This subcommand is used to specify the PROM or ROM mapping parameters for a data configuration not defined on the Standard Control Information Cassette or to modify the mapping parameters of a configuration input from the Standard Cassette. The PROM/ROM data configuration mapping parameters are explained in paragraph 7.4.4 and Appendix F. The command parameters irn, mrn and drn correspond to IR_n, MR_n and DR_n in the computations in Appendix F.

If two- or three-level data configuration mapping has been used and the user wishes to specify another data configuration using only level one mapping, the looping parameters for levels two and three must be reset to the default values. If three-level mapping has been used and the user is going to specify two-level mapping, the looping parameters for level three must be reset to the default values. This can be accomplished by entering the RI subcommand and specifying the level, but omitting the looping parameters. The following command:

```
.PP,RI,3
```

resets the looping parameters for level three to allow the user to proceed with level two and one mapping.

Error message:

PP02	Parameter value is outside the permissible range.

Examples:

```
.PP,RI,1,4,100
.PP,RI,3,1,4,3
```

The first example defines the ROM/PROM characteristics as follows:

Loop level	= 1
Bit increment	= 4
Maximum iteration count	= 100_{16} = 256_{10}
Bit displacement	= 0 (default)

The second example defines the ROM/PROM characteristics as follows:

Loop level	= 3
Bit increment	= 1
Maximum iteration count	= 4
Bit displacement	= 3

7.5.3.8 **Define PROM/ROM Characteristics** (RC). The Define PROM/ROM Characteristics subcommand defines the physical hardware characteristics needed to transfer data to the PROM/ROM.

Syntax definition:

PP {ƀ'...} RC {ƀ'...} <width> {ƀ'...} <high or low> {ƀ'...} <pwl> [{ƀ'...}
[<retries>] [{ƀ'...} [<duty cycle>] [{ƀ'...} <pgmable bits>]]]

Parameters:

width — Number of bits per word in the PROM/ROM physical organization. Required parameter. Hexadecimal number.

high or low — Value that specifies whether high or low logic level output conditions are to be programmed. The value is 0 if low and is 1 if high. Required parameter.

pwl — Normal pulse width to be used for programming. The pulse width is entered as an index value between 1 and 6 obtained from a table in Appendix G. Required parameter.

retries — Number of times programming is to be retried using the normal pulse width if a programming failure occurs. Hexadecimal number.

Duty cycle — Duty cycle to be maintained while programming a PROM. Hexadecimal number. The value is the percentage of the total time (programming time plus delay time) that the programming pulse is on. The normal duty cycle varies between 16% and 50%. For example, a value of 20_{16} is a duty cycle of 32%.

pgmable bits — Number of bits that can be physically programmed simultaneously.

Parameter default values:

If the retries parameter is not specified, a value of 0 is used.

If duty is not specified, a value of 19_{16} (25%) is used.

If the pgmable bits parameter is not specified, a value of 1 is used.

Description: This subcommand is used to define the physical characteristics needed to transfer data to the PROM during the programming cycle. This subcommand may be used when a standard PROM data configuration is not desired or the PROM being used is not supported on the Standard Control Information Cassette. The PROM characteristics are explained in detail in paragraph 7.4.3.

The pulse width is entered as an integer number from 1 to 6. This number is then mapped into a 0.5 millisecond to 16.0 millisecond pulse according to the table of pulse widths in Appendix G. This appendix also contains a table of the range of pulse widths allowed for the supported PROMs.

The HIGH/LOW parameter specifies whether a PROM is to be programmed with 1s or 0s. For example, the S287 PROM is initially all 1s and must be programmed with 0s. The programmable bits parameter specifies the number of bits in the bit string to be physically transferred into the PROM at a time. In most cases, with the exception of the erasable programmable read-only memory (EPROM), only one bit should be programmed at a time. As the number of bits is increased, the reliability of the programming process decreases.

When programming EPROMs, the entire bit string is programmed at once. For a description of the EPROM programming process, see paragraph 7.6.2.

The retries parameter specifies the number of times to repeat the programming process if, after programming the number of bits specified by the programmable bits parameter, a programming failure occurs. The same bits will be reprogrammed until the correct data is transferred or the retry count is depleted. When programming EPROMs, the retry parameter should always be 0 because of the special process involved in EPROM programming (paragraph 7.6.2).

The duty cycle determines the percentage of time that the programmable pulse (which causes the actual transfer of data to the PROM) is on with respect to total cycle time (which includes a delay time). Appendix G contains a table of the range of duty cycles allowed for the supported PROMs.

Error messages:

PP01	Required parameter missing.
PP02	Parameter value outside permissible range.

Examples:

```
.PP,RC,4,1,3,2,10,1
.PP,RC,8,0,4
```

The first example defines the ROM/PROM characteristics as follows:

ROM/PROM word width = 4 bits
High logic level output conditions (program 1s)
Pulse width = 3
Number of retries = 2
Duty cycle = 10_{16} = 16 percent
Program 1 bit at a time

The second example defines the ROM/PROM characteristics as follows:

ROM/PROM word width = 8 bits
Low logic level output conditions (program 0s)
Pulse width = 4
Number of retries = 0 (default)
Duty cycle = 25 percent (default)
Program 1 bit at a time (default)

7.5.3.9 Define String Width (SW). The Define String Width command informs the control software of the width of the bit strings to be transferred between PROM/ROM and memory, displayed, or compared to other bit strings.

Syntax definition:

PP {ƀ’...} SW {ƀ’...} <width>

Parameter:

<width> The number of bits per bit string. A number in the range 1 to 8. Required parameter.

Description: This subcommand sets the memory and PROM/ROM bit string width.

Error messages:

PP01 Required parameter missing.

PP02 String width outside the permissible range.

Example:

```
.PP,SW,1
```

This command defines the width of the string to be 1.

7.6 PROGRAMMING CONSIDERATIONS

The following paragraphs discuss the methods for performing some specific programming tasks of which the user should be aware. The tasks include:

- Standardizing nonstandard memory and PROM configurations.
- Programming erasable programmable read-only memory (EPROM)
- Creating PROMs for memory addresses not in the hardware configuration

7.6.1 STANDARDIZING NONSTANDARD MEMORY AND PROM CONFIGURATIONS. After setting the direction toggle in the TS command to 3 and before typing the GO subcommand, the user should mount the Standard Control Information Cassette on the device assigned to logical unit number 7 and a scratch cassette on the device assigned to logical unit number 8.

This toggle is processed after all other toggles. For example, if the toggle to display memory is also set, the complete memory data configuration is displayed before the PROM programmer begins the standardization process.

When the GO subcommand is typed, PROMPG responds with:

```
MEM ID?
```

and waits for the user's reply. The user enters a name (Character string of 1 to 12 characters) followed by a carriage return to identify the control information for the present memory data configuration. Entering only a carriage return indicates that the user does not wish to retain the present memory configuration's control information on cassette. PROMPG now responds with:

```
ROM ID?
```

and awaits the user's reply. The user's reply is a name identifying the control information for the present PROM/ROM data configuration. Again, the user may indicate with only a carriage return his desire not to retain the present ROM configuration's control information on cassette. PROMGP copies the information from the current Standard Control Information Cassette to the scratch cassette until it encounters control information with an identifying name which matches either the PROM/ROM or memory ID name. If a match is found, the new control information is written on the scratch cassette. If a match of either the memory or PROM/ROM ID name has been found before an end-of-file is encountered on the Standard Control Information Cassette, the new control information is added to the end of the scratch cassette, which now becomes the updated Standard Control Information Cassette.

7.6.2 PROGRAMMING EPROMs. Since EPROMs are metal oxide semiconductor (MOS) devices, they must be programmed in a different manner than TTL PROM devices. EPROMs are charge storage devices which must be programmed by repetitively transferring charge to the EPROM bits. This repetition may be accomplished by looping through the programming process defined by the data configurations. The number of required repetitions to transfer sufficient change to each bit or bit string is defined by the following formula.

100 ms = pulse width X repetitions

Therefore, using a pulse width of 0.5 ms, 200 repetitions must be used to successfully program the EPROM.

There must be a delay after each attempt to program a bit string before trying to program the same bit string again. This delay is necessary to allow the charge to diffuse into the EPROM device without a buildup of excess charge on the surface.

Because of the required delay, each bit string of the EPROM should be attempted once before repeating the programming cycle. To ensure this delay, the number of retries for programming each bit string (defined in the RC subcommand) must be set to zero. Each bit of the EPROM will not appear to have the correct value (0 or 1) until sufficient charge has been transferred to it.

In the early stages of programming, the bits may not have acquired sufficient charge to have the correct value. This will appear as a programming failure if the number of retries is set to a nonzero value, and the bit string will be programmed again without the required delay time. For the same reason, the compare toggle (defined by the TS subcommand) should not be set during the programming cycle, since compare errors will be found in the early stages of programming an EPROM.

Since the programming cycle for an EPROM repeats many times, the display toggles (defined by the TS subcommand) should not be set during the programming cycle since the memory or PROM data will be printed for each repetition.

Therefore to program, compare, and display, the process must be done in two steps. First the toggles must be set to program. After completion of programming the EPROM, the toggles may be set to compare and/or display. The number of repetitions defined must be changed to one before the second step to compare and/or display.

The following example shows how to program a 1024 X 8 EPROM from a 1024 word block of memory. The following commands define the memory and PROM data configurations, bit string width, PROM characteristics, memory bounds, and toggles, and initiate the programming process.

```
PP,MI,1,10,400,0
PP,RI,1,8,400,0
PP,MI,2,0,C8,0
PP,RI,2,0,C8,0
PP,SW,8
PP,RC,8,0,1,0,32,8
PP,MB,0,3FF
PP,TS,0,0,1,0
PP,GO
```

The level 2 mapping defines the repetition count to be $C8_{16}$ = 200. The toggles are set to program memory to PROM.

To perform the compare to check for programming failures, the following commands are needed.

```
PP,MI,2
PP,RI,2
PP,TS,0,0,0,1
PP,GO
```

The RI and MI subcommands define the repetition count to the default value of 1. The toggles are set to compare memory to PROM.

7.6.3 CREATING PROMs FOR MEMORY ADDRESSES NOT IN HARDWARE CONFIGURATION. By specifying a load point for the linking loader different from the default, PROMs may be generated to be used in memory addresses for which memory is not configured in the current system or cannot be loaded with the linking loader.

An example is to generate a PROM to be used at location $FE00_{16}$. Since the ROM for the programmer panel and loader is at location $FE00_{16}$, object code cannot be loaded there. The linking loader provides the capability to load programs with a specified load point and load bias. This allows the user to load programs at a location in memory different from the location at which they will execute, $FE00_{16}$ in this case.

The load point and load bias specified by the user are used in determining how the code is relocated and the memory address where the code will actually be loaded. Code assembled with an absolute origin (AORG) directive is loaded at the absolute address determined by the directive plus the load point.

MEMLOC = ABS ADDR + LDPT

In this example, if the object code to be programmed into PROM is assembled with an absolute origin of $FE00_{16}$ but the user wants to load it at location 200_{16}, he should enter a load point of 400_{16}.

$$200_{16} = FE00_{16} + 400_{16}$$

The load bias entered is not used since the object code is absolute.

Code assembled with a relocatable origin (RORG) directive is loaded at the relocatable address determined by the directive plus the load bias plus the load point.

MEMLOC = REL ADDR + LDBI + LDPT

In this example, if the object code to be programmed into PROM is assembled with a relocatable origin of 0, and the user wants it to be executable at location $FE00_{16}$ but wants to load it at location 200_{16}, he should enter a load point of 400_{16} and a load bias of $FE00_{16}$.

$$200_{16} = 0 + FE00_{16} + 400_{16}$$

Note however, that object code loaded with a load point other than the default 0 is not executable.

7.7 PROGRAMMING EXAMPLES

The following paragraphs present examples of command sequences used to program PROMs with the PROM programmer and examples of command sequences for using the additional PROM programmer capabilities. Additional programming examples are presented in Section XI.

7.7.1 EXAMPLE 1. Generate a 256 X 16 memory with PROMs by programming a 256 word block of memory, located at $7A0_{16}$, into four 256 x 4 PROM devices. Refer to figure 7-2.

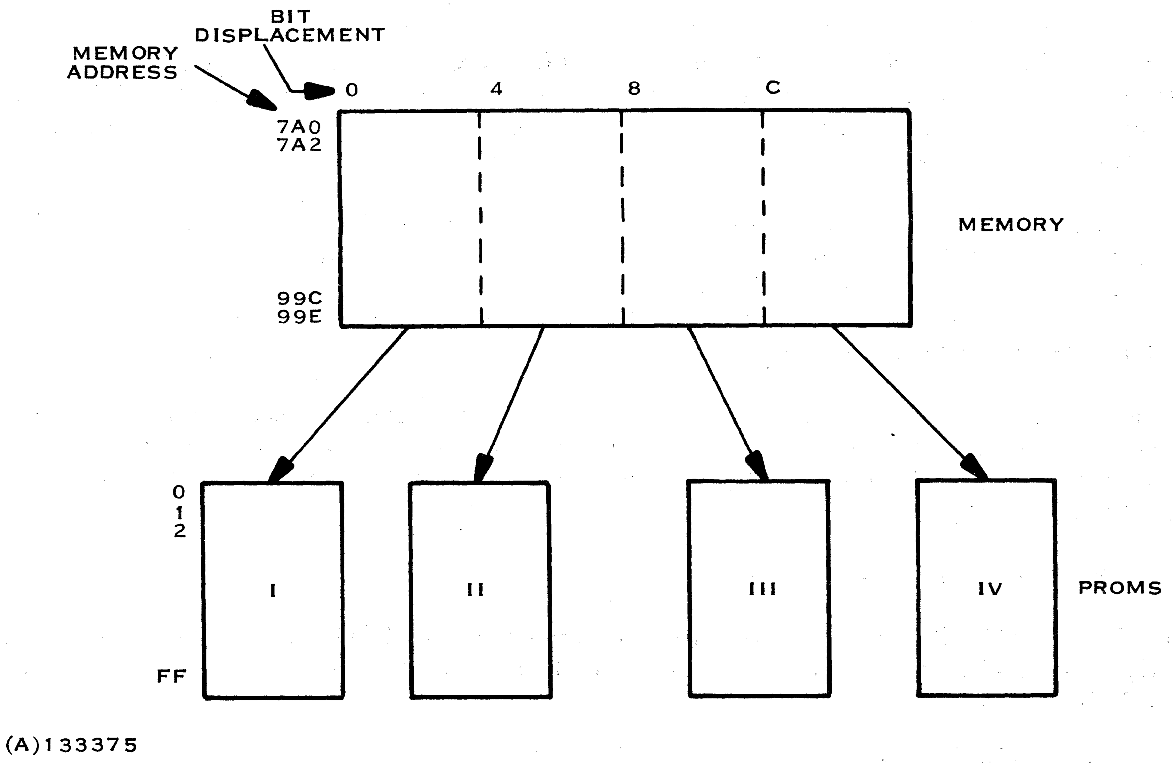


Figure 7-2. Mapping Example 1

Mount the Standard Control Information Cassette on LUNO 7.

Command	*Commentary*
.PL	Load PROM programmer software.
.PS,MS287-0,S287	Standard configuration MS287-0, S287. (Memory configuration initial bit displacement equals 0.)
.PP,MB,7A0,99F	Memory bounds 7A0-99F (PROM bounds default to 0 and FFF_{16}).
.PP,GO	Program PROM I. The toggles were defaulted to program PROM and compare when PROM programmer was loaded by PL.
Change the PROM.	
.PS,MS287-4	Load standard memory configuration MS287-4 with initial bit displacement equal to 4. PROM/ROM configuration does not change.
.PP,GO	Program PROM II.
Change the PROM.	

.PP,MS287-8 Load standard memory configuration MS287-8 with initial bit displacement equal to 8.

.PP,GO Program PROM III.

Change the PROM.

.PS,MS287-C Load standard memory configuration MS287-C with initial bit displacement equal to C_{16}.

.PP,GO Program PROM IV.

7.7.2 **EXAMPLE 2.** Program a 32 by 8 PROM from a 16 word block of memory beginning at memory address 40_{16}.

Assume that the CRU base address is $1A0_{16}$.

Position the Standard Control Information Cassette. Refer to figure 7-3.

.PS,S288,MS288A Standard Control Information for ROM/PROM configuration S288. Standard Control Information for memory configuration MS288A. This configuration has an initial displacement of 0 with a bit increment of 8 bits, and a bit string width of 8.

.PP,MB,40,5F Beginning memory address 40_{16}. Ending memory address $5F_{16}$.

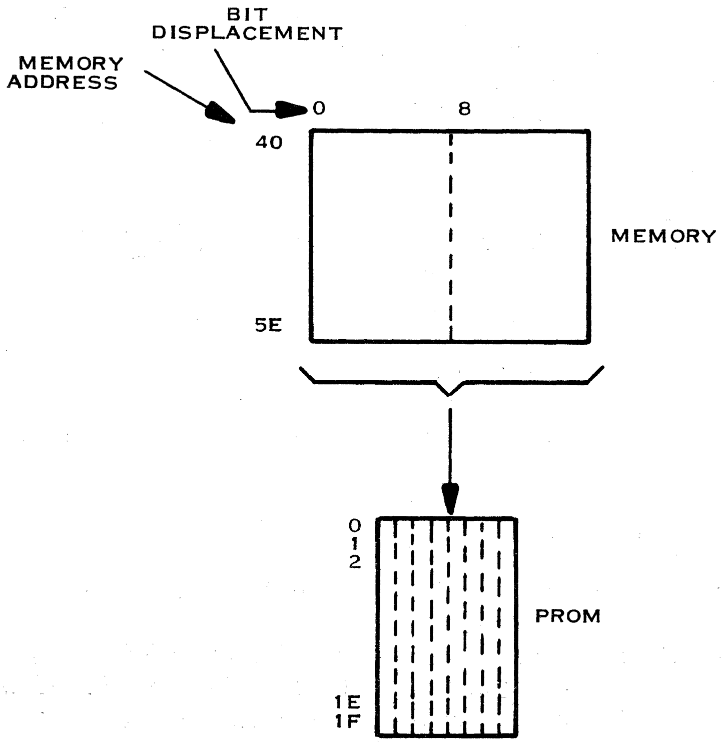
.PP,CS,1A0 CRU ROM interface base address $1A0_{16}$.

.PP,TS,0,0,1,1 Set toggle to program PROM and compare.

.PP,GO

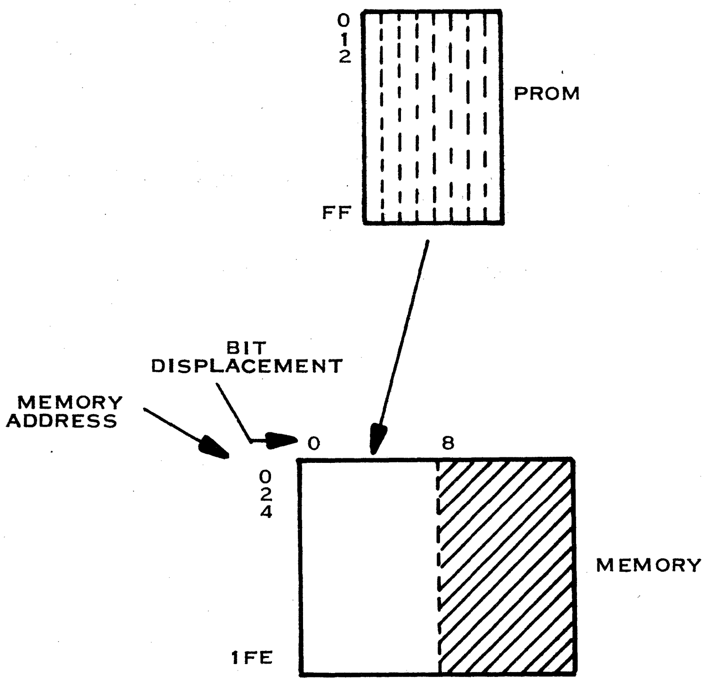
7.7.3 **EXAMPLE 3.** Load the most significant bytes of a 256 word block of memory beginning at memory address 0 from a 256 by 8 PROM.

Assume that the CRU base address is 120_{16}. Refer to figure 7-4.



(A)133376

Figure 7-3. Mapping Example 2



(A)133377

Figure 7-4. Mapping Example 3

Position the Standard Control Information Cassette.

.PS,MS471-0,S471	Standard Control Information for PROM/ROM configuration S471. Standard Control Information for memory configuration MS471. This configuration has initial displacement equal to 0, bit increment equal to 10_{16}, and a bit string width of 8.
.PP,MB,0,1FF	Beginning memory address 0. Ending memory address 1FF.
.PP,CS,120	CRU ROM interface base address 120_{16}.
.PP,TS,0,0,2,0	Load memory from PROM
.PP,GO	

7.7.4 EXAMPLE 4. This example is in several parts.

a. Program a 512 by 8 EPROM from a 256 word block of memory beginning at memory address 80_{16}.

Assume CRU base address 120_{16} (unchanged from previous setting).

Position the Standard Control Information Cassette. Refer to figure 7-5.

.PS,ME2704A,E2704	Standard Control Information for memory configuration ME2704A. Standard Control Information for PROM/ROM configuration E2704.
.PP,MB,80,27F	Beginning memory address 80_{16}. Ending memory address $27F_{16}$.
.PP,TS,0,0,1,0	Program PROM from memory data configuration. (See the note below.)
.PP,GO	

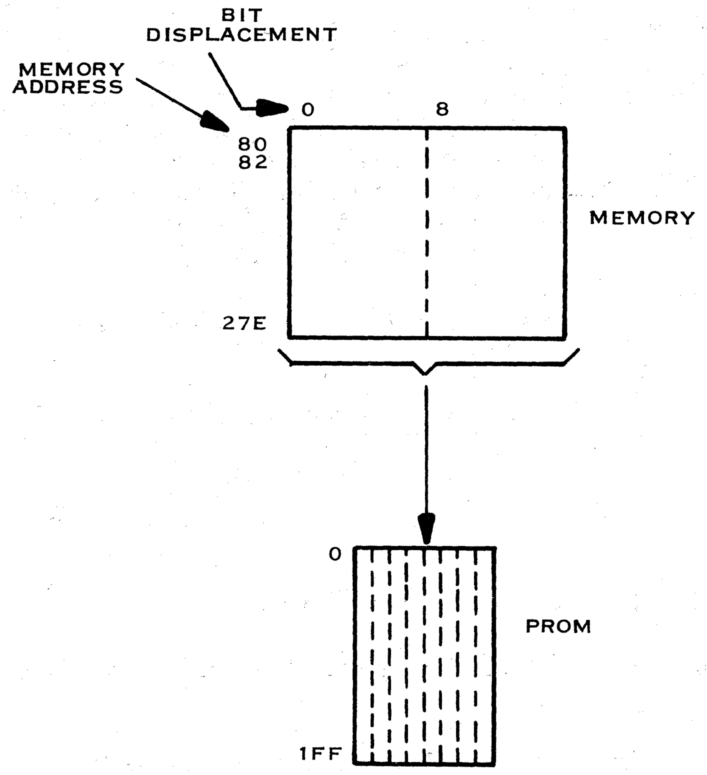


Figure 7-5. Mapping Example 4

b. Compare PROM in a. to memory configuration used to program the PROM.

.PP,MI,2	Clear second level looping.
.PP,RI,2	
.PP,TS,0,0,0,1	Compare PROM and memory.
.PP,GO	

c. Display PROM programmed in a.

.PP,TS,0,1,0,0	Display PROM.
.PP,GO	

NOTE

Because of the nature of programming the EPROM, the EPROM should be compared to memory only after the programming cycle has ended by resetting the toggles and initiating the compare as in b. and the comparison of the PROM to the memory configuration used to program the PROM. (Refer to paragraph 7.6.2.)

7.7.5 **EXAMPLE 5.** Generate a 1024 X 8 memory with PROMs from a 1024 word memory block. Data is loaded in memory from location 200_{16} through location $9FE_{16}$ in even-numbered bytes. Refer to figure 7-6.

Assume that this programming sequence is not standard.

Command	*Commentary*
.PP,MI,1,10,400	Level one memory mapping. ● Increment 4 bits ● 1024 times
.PP,RI,1,4,400	Level one PROM mapping. ● Increment 4 bits ● 1024 times
.PP,SW,4	Program 4 bits at a time.
.PP,MB,200,9FE	Beginning memory address = 200_{16}. Ending memory address = $9FE_{16}$.
.PP,RB,0,3FF	Beginning ROM address = 0. Ending memory address = $3FF_{16}$.
.PP,RC,4,1,1,8,14,1	ROM Characteristics. ● ROM word width of 4 bits ● Program high-logic-level outputs ● Normal pulse width "1", 8 retries ● 20% duty cycle ● Program 1 bit at a time
.PP,GO	Program PROM set I.
Change the PROMs.	
.PP,MI,1,10,400,4	Change initial displacement to 4 bits
.PP,GO	Program PROM set II.

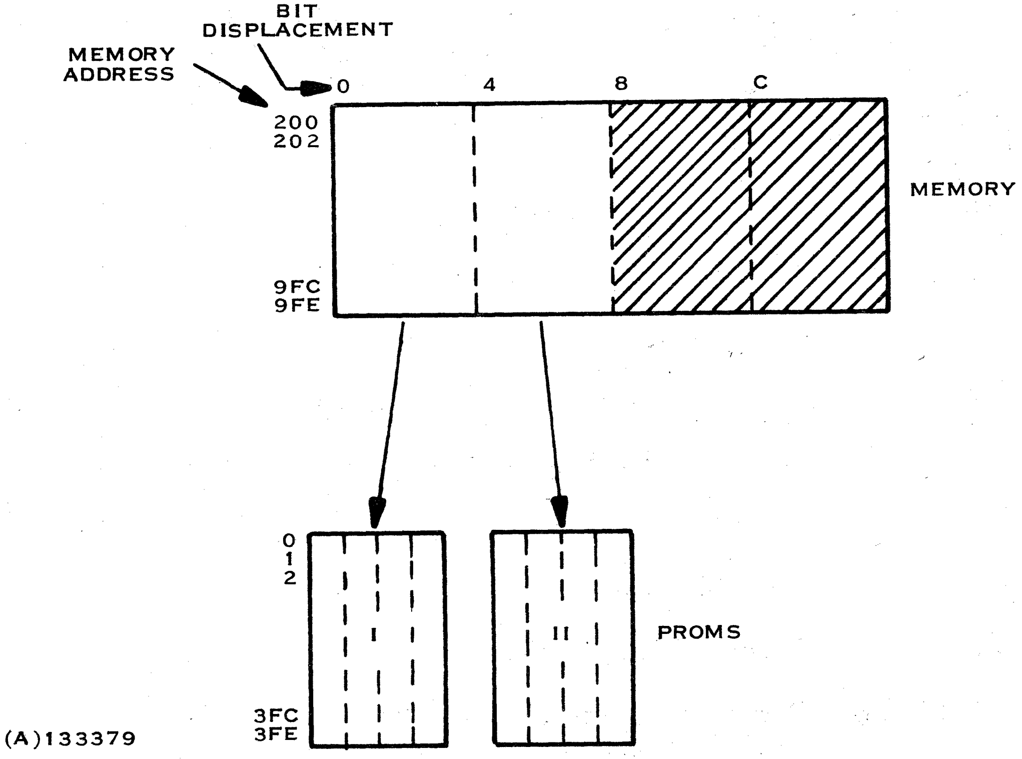


Figure 7-6. Mapping Example 5

7.7.6 **EXAMPLE 6.** Save the control information in example 5. (Refer to paragraph 7.4.5.)

Mount the Standard Control Information Cassette on the device assigned to LUNO 7. Mount the scratch cassette on the device assigned to LUNO 8.

.PP,MI,1,10,400,0	Change displacement back to 0.
PP,TS,0,0,3,0	Set toggle to save information.
.PP,GO	
	Program replies with MEM ID?
MEM ID? MOB4	
	Program replies with ROM ID?
ROM ID? ROB4	
	ROM identifier. Tape I/O occurs.

Example 2 may now be run replacing

.PP,MI,1,10,400

and

```
.PP.RI,1,4,400

      and

.PP,SW,4

      and

.PP,RC,4,1,1,8,14,1

      with

.PS,MOB4,ROB4
```

7.7.7 **EXAMPLE 7.** Twenty-four 4-bit fields are arranged in 16-bit words as shown in the illustration. These 24 fields are to be programmed repetitively in the first 384 four-bit words of a 512 X 4 PROM with characteristics similar to a TI SN74S287 (two 287s with a programming adaptor card to make them appear as a 512 X 4 device). Refer to figure 7-7.

Assume that this programming sequence is not standard. .

Command	*Commentary*
.PP,MI,1,6,3	Level one memory mapping (go across word). ● Increment 6 bits ● 3 times
.PP,MI,2,10,8	Level two memory mapping (step from word to word). ● Increment 16 bits ● 8 times
.PP,MI,3,0,10	Level three memory mapping (provide repetitions of memory data configuration). ● Increment 0 bits ● 16 times
.PP,RI,1,4,180	Level one ROM mapping ● Increment 4 bits ● 384 (= 16 X 8 X 3) times
.PP,SW,4	Program four bits at a time.

.PP,MB,2A0,2AE Beginning memory address.

Ending memory address.

.PP,RB,0,17F Beginning ROM address.

Ending ROM address.

.PP,RC,4,0,1,8,1 ROM characteristics.

- ROM word width 4 bits
- Program low-logic-level outputs
- Normal pulse width "1", 8 retries
- Duty cycle 25% (default)
- Physically program one bit at a time.

.PP,GO

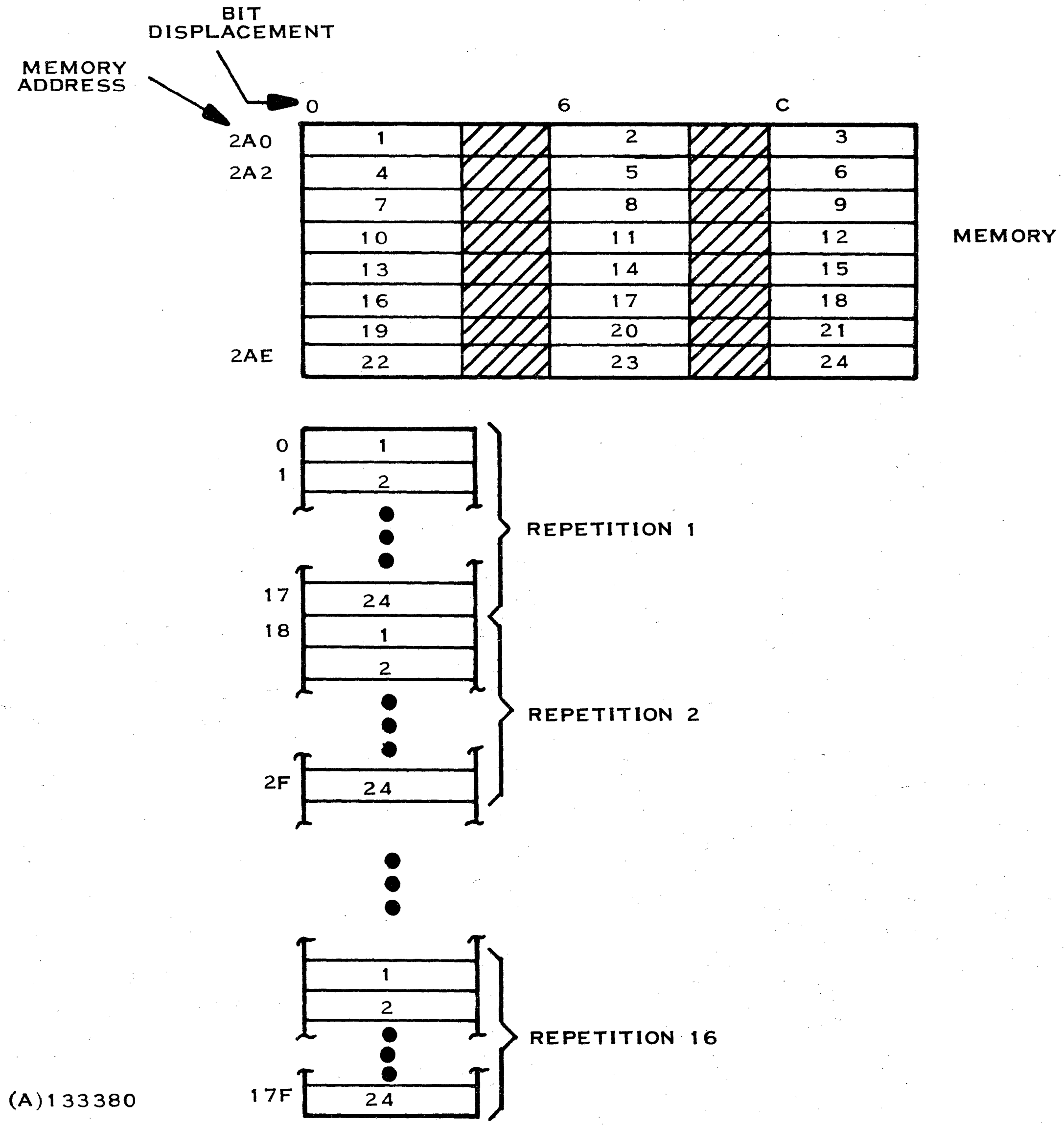


Figure 7-7. Mapping Example 7

SECTION VIII

BNPF DUMP MODULE

8.1 FUNCTIONS AND OPERATION

The BNPF Dump (DMBNPF) overlay, when resident in the monitor transient area, allows the user to produce a BNPF-formatted cassette tape, check that the correct format has been produced, and load the BNPF-formatted load module from cassette into memory. These functions may be initiated by the DB monitor keyboard command.

Instructions for loading the BNPF Dump overlay module into memory are included in the discussion of system software cassette generation in Section II and the OV command in Section III.

8.2 , BNPF FORMAT

The standard format of the DMBNPF output has the following appearance:

decimal byte address ƀ B $\underbrace{\text{xxxxxxxx}}_{\text{first 8-bit byte of P's and N's}}$ F . . . B $\underbrace{\text{xxxxxxxx}}_{\text{sixth 8-bit byte of P's and N's}}$ F

The decimal byte address is the address of the first byte of information contained on the line. It contains no leading zeros and must begin in column 1. Each record contains at most six bytes. The N and P characters represent the bit values 0 and 1 respectively.

8.3 BNPF DUMP COMMANDS

The commands used by the BNPF Dump software module are described in detail in the following paragraphs.

8.3.1 PERFORM BNPF OPERATION (DB). The Perform BNPF Operation command, along with a subcommand, causes a BNPF dump, load or data comparison to occur.

Syntax definition:

DB {ƀ'. . .} <subcommand>

Parameter:

subcommand	Command which specifies a dump, load, or data comparison. If it specifies a dump, additional parameters are required (paragraph 8.3.2.1).

Error message:

MP00	Invalid subcommand

8.3.2 DB SUBCOMMANDS. The DB command is used with a D, C or L subcommand. These subcommands are described in the following paragraphs.

8.3.2.1 Dump Memory to Cassette in BNPF Format (D). The Dump Memory to Cassette in BNPF Format subcommand causes each byte within the specified memory range to be converted to BNPF format and stored on tape.

Syntax definition:

DB {ƀ'...} D {ƀ'...} <start addr> {ƀ'...} <end addr>

Parameters:

start addr — Address of first byte to be dumped. Required parameter. Hexadecimal number.

end addr — Address of last byte to be dumped. Required parameter. Hexadecimal number.

Description: The memory range is specified by the starting and ending addresses. BNPF format, the format in which data is stored on tape, is described in paragraph 8.2. This command dumps to the device assigned to LUNO 7.

Error messages:

DP03 — Dump is larger than 8192 (2000_{16}) bytes Starting address is greater than the ending address.

MS05 — Required parameter missing.

MX01 — Unrecoverable I/O error. (Output cassette may not be ready.)

Example: The following example dumps memory locations 500_{16} to $50F_{16}$ to cassette in BNPF format:

```
DB D,500,50F
.
```

The contents of memory, when printed using the IM command, appear as follows:

```
IM 500 50F
0500=0000  1111  2222  3333  >4444  5555  6666  7777
.
```

After the memory words have been stored on cassette, they appear as follows:

```
1280 BNNNNNNNNF BNNNNNNNNF BNNNPNNNPF BNNNPNNNPF BNNPNNNPNF BNNPNNNPNF
1286 BNNPPNNPPF BNNPPNNPPF BNPNNNPNNF BNPNNNPNNF BNPNPNPNPF BNPNPNPNPF
1292 BNPPNNPPNF BNPPNNPPNF BNPPPNPPPF BNPPPNPPPF
$
```

A dollar sign ($) in the first character of a record denotes the end of the dump. The memory addresses printed are decimal numbers.

8.3.2.2 Compare BNPF Format on Cassette to Memory (C). The Compare BNPF Format on Cassette to Memory subcommand can be used to verify that the correct data was written on cassette tape by the D subcommand (paragraph 8.3.2.1).

Syntax definition:

$$\text{DB} \left\{ \text{b̶'} \ldots \right\} \text{C}$$

Description: After a memory block is dumped to tape, reposition the cassette assigned to LUNO 7 to the first record and enter the DB command and C subcommand. Each BNPF-formatted byte is reconverted to hexadecimal and compared to the byte in memory. If the comparison fails, each byte from the cassette and the corresponding byte from memory are displayed with the hexadecimal address. Control is returned to the command string processor without printing if no comparison errors occur.

Error message:

MX01 Unrecoverable I/O error

Example:

```
DB C
BEG ADDR=0500
END ADDR=050F
.
```

The contents of the tape is compared to memory. The beginning and ending addresses are printed. Because no compare errors have been detected, nothing else is printed.

```
DB C
BEG ADDR=0500
T0502=1100   M0502=0000   T0503=1100   M0503=FF00
T0506=3300   M0506=0000   T0507=3300   M0507=FF00
T050A=5500   M050A=0000   T050B=5500   M050B=FF00
T050E=7700   M050E=0000   T050F=7700   M050F=FF00
END ADDR=050F
.
```

In this example, a number of compare errors have been detected, The memory and tape byte values are displayed, left justified in the field. Pressing the ESC key on the terminal keyboard terminates printing of compare errors.

8.3.2.3 Load BNPF-Formatted Data Module into Memory (L). The Load BNPF-Formatted Data Module into Memory subcommand reads a BNPF-formatted data module from the device assigned to LUNO 7, converts the data to hexadecimal, and stores the data in the memory addresses corresponding to those on the cassette..

Syntax definition:

DB {ƀ'...} L

Error message:

MX01 Unrecoverable I/O error

Example:

```
.DB,L
.
```

SECTION IX

HIGH/LOW DUMP MODULE

9.1 FUNCTIONS AND OPERATION

The HIGH/LOW Dump (DMHL) overlay, when resident in the monitor transient area, allows the user to produce a TI 256 by 4 HIGH/LOW-formatted cassette tape and check that the correct format has been produced. Because DMHL is an overlay, it must be loaded into the transient area before being used.

A program function is initiated by a monitor keyboard command which sets memory bounds and lets the user specify the option desired: either to produce the tape or to perform a data comparison to check the tape.

Instructions for loading the HIGH/LOW Dump overlay module into memory are included in the discussion of system software cassette generation in Section II and the OV command in Section III.

9.2 HIGH/LOW FORMAT

The standard format of the DMHL output has the following appearance:

$$\underbrace{\text{xxx}}_{\substack{\text{beginning}\\ \text{decimal}\\ \text{address}\\ \text{(3 digits)}}} - \underbrace{\text{xxx}}_{\substack{\text{ending}\\ \text{decimal}\\ \text{address}\\ \text{(3 digits)}}}\ \text{ƀƀ}\ \underbrace{\text{yyyy}}_{\substack{\text{first 4-bit}\\ \text{word of H's}\\ \text{and L's}}}\ \text{ƀ}\ \underbrace{\text{yyyy}}_{\substack{\text{second 4-bit}\\ \text{word of H's}\\ \text{and L's}}}\ \text{ƀ}\ \ldots\ \text{ƀ}\ \underbrace{\text{yyyy}}_{\substack{\text{eighth 4-bit}\\ \text{word of H's}\\ \text{and L's}}}$$

The first seven characters of a record must contain the first and last address of the eight data sets described in the remaining columns. As an example, the first record must contain 000 through 007. The addresses must be three-digit right justified zero-filled integers separated by a hyphen (minus sign). The last record must contain 248-255. All 32 records must contain eight consecutive address groups so that the dump starts with 000 and ends with 255.

Each record contains eight 4-bit words of Hs and Ls. The H and L characters represent the bit values 1 and 0 respectively. An example follows:

```
000-007  LLLL LLLH LLHL LLHH LHLL LHLH LHHL LHHH
008-015  HLLL HLLH HLHL HLHH HHLL HHLH HHHL HHHH
016-023  LLLL LLLH LLHL LLLL LLLL LLLL LLLL LLLL
024-031  LLLL LLLL LLLL LLLL LLLL LLLL LLLL LLLL
```

.
.
.

9.3 HIGH/LOW DUMP COMMANDS

The commands used by the HIGH/LOW Dump software module are described in detail in the following paragraphs.

9.3.1 PERFORM HIGH/LOW OPERATION (HL). The Perform HIGH/LOW Operation command, along with a subcommand, causes a HIGH/LOW dump or data comparison to occur.

Syntax definition:

HL {ƀ'...} <subcommand>

Parameter:

subcommand Command which specifies a dump, or data comparison. The subcommands are described in paragraph 9.3.2.

Error message:

MP00 Invalid subcommand

9.3.2 HL SUBCOMMANDS. The HL command is used with a D or C subcommand. These subcommands are described in the following paragraphs.

9.3.2.1 Dump in HIGH/LOW Format (D). The Dump in HIGH/LOW Format subcommand converts four bits of each word of a selected 256-word memory block to HIGH/LOW format and writes the converted format to tape.

Syntax definition:

HL {ƀ'...} D {ƀ'...} <start addr> {ƀ'...} <end addr> [{ƀ'...} <bit>]

Parameters:

start addr Address of the first word in the memory block. Required parameter. Hexadecimal number.

end addr Address of the last word in the memory block. Required parameter. Hexadecimal number.

bit Starting bit of four-bit string in each word. The number of the bit position.

Parameter default values:

If the bit parameter is not specified, it is set to 0.

Description: DMHL writes to the device assigned to LUNO 7. If a block of less than 256 (100_{16}) words is specified, the HL command fills out the 256 words on tape with 4-bit words of Hs. To check whether the correct information was recorded on cassette, reposition the cassette and enter the HL command with the Compare (C) subcommand (paragraph 9.3.2.2).

Error Messages:

DP03	Dump was greater than 256 words. Starting address is greater than the ending address.
MP00	Illegal parameter value. Address was not on word boundary. D parameter missing. Bit parameter value is greater than C_{16} .
MS05	Required parameter (other than subcommand) missing.
MX01	Unrecoverable I/O error or output cassette not ready.

Example:

```
HL D 500 6FE
```

The cassette has the first four bits of each word, with the bit parameter equal to its default value 0, of a 256-word block beginning at 500_{16} converted to HIGH/LOW format.

```
HL D 500 520 8
```

The cassette has four bits beginning at bit 8 of each word of a 16-word block beginning at 500_{16} converted to HIGH/LOW format. The cassette is filled with records of Hs until a 256-word format has been created.

9.3.2.2 Compare HIGH/LOW Format on Cassette to Memory (C). This subcommand is used to verify that the correct data was written on tape by the D subcommand.

Syntax definition:

HL {ƀ'...} C <start addr> {ƀ'...} <end addr> [{ƀ'...} <bit>]

Parameters:

start addr	Address of first word in memory block. Required parameter. Hexadecimal number.
end addr	Address of last word in memory block. Required parameter. Hexadecimal number.
bit	Starting bit of four-bit string in each word. The number of the bit position. Hexadecimal number.

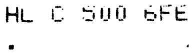
Parameter default value: If the bit parameter is not specified, it is set to 0.

Description: Each four-bit string on cassette is compared to four bits of binary data in each word of the designated memory block. If the comparison fails, the addresses and the cassette and memory data values are printed.

Error messages:

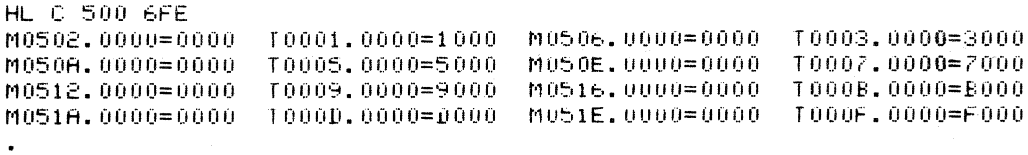
DP03	Block larger than 256 words.
MP00	Illegal parameter value. Address was not on word boundary. Bit parameter value is greater than C_{16} . C parameter missing.
MS05	Required parameter missing.
MX01	Unrecoverable I/O error.

Examples:



```
HL C 500 6FE
.
```

The contents of the cassette are compared to the first four bits of each word from 500_{16} to $6FE_{16}$. No compare errors are detected.



```
HL C 500 6FE
M0502.0000=0000  T0001.0000=1000  M0506.0000=0000  T0003.0000=3000
M050A.0000=0000  T0005.0000=5000  M050E.0000=0000  T0007.0000=7000
M0512.0000=0000  T0009.0000=9000  M0516.0000=0000  T000B.0000=B000
M051A.0000=0000  T000D.0000=D000  M051E.0000=0000  T000F.0000=F000
.
```

In this example, a number of compare errors are found, The relative address of the word on tape, its contents, and the memory address and its contents are displayed. Only the four bits of the memory word that are being compared are displayed. The bits are left justified in the content display.

SECTION X

SYSTEM OPERATION AND DEBUGGING EXAMPLE

10.1 INTRODUCTION

A complete example of system operation and debugging is presented in this section. The example includes assembly of program modules, the loading sequence, debugging, editing, reassembly of the edited module, relinking and loading of all modules, and execution of the final version of the program.

It is assumed that the user has read the manual and has been introduced to these functions. Included with each step in this example is a brief explanation of procedure, a listing of the actual procedure followed, and a reference to the section in the manual where more information can be found.

This program creates a concordance of all the symbols used in a program. The user may specify labels, operators, and/or operands to be included in the concordance printout. To run the program, follow these steps:

1. Mount and ready the source tape in cassette drive 1 (the left-hand drive).
2. Execute the program using the EX command or using the RU command for debugging.

In the concordance program, an error is included in the print routine (PRTBM) so that the user may be exposed to the process involved in creating and debugging a working program. The steps in the process are:

1. Assemble the source programs and create object modules using PX9ASM.
2. Link and load the object modules into user memory using PX9LAL.
3. Using the monitor, debug the program.
4. Using PX9EDT, edit the source module which contains the error.
5. Reassemble the source module.
6. Link and load all the modules into memory again.
7. Execute the final program to see that the error has been corrected and the program executes correctly.

10.2 ASSEMBLING MODULES WITH PX9ASM

The first part of the program is the assembly of modules using PX9ASM. For a description of how to use the assembler, refer to Section V. The assembler must first be loaded using the LU command (Section III). The source modules require the predefined register definitions; therefore, in answer to the question:

```
PREDEFINED REGISTERS?
```

Enter “Y”.

Assembly listings of the routines other than PRTB (IDT 'PRTBM') used in the concordance program are not shown, but are printed when the programs are assembled. The routines not shown are DRIVER, PARSE, CTYP, CSYM, SYMREF, and SYMDEF.

```
.LU 8
.EX

PX9ASM  945393 ** 15MAR76
ADD 4K MEM BLOCKS CONFIGURED? 0
PREDEFINED REGISTERS? Y

ASM/TERM? A
```

```
ASM/TERM? A




                                                                  PAGE 0001

0001
0003                            IDT   'PRTBM'
0004                   *
0005                   *        PRTB WILL READ THE SYMBOL TABLE ONE SYMBOL
0006                   *        AT A TIME AND PRINT THE SYMBOL NAME,
0007                   *        THE STATEMENT NUMBER WHERE THE SYMBOL
0008                   *        WAS DEFINED, AND THE LIST OF STATEMENT
0009                   *        NUMBERS WHERE THE SYMBOL WAS REFERENCED.
0010                   *
0011                   *        CALLING SEQUENCE:
0012                   *          NO INPUT PARMS
0013                   *
0014                   *          REGISTERS DESTROYED - R0,R1,R2,R3,R4,R5,R6,R9,R10
0015                   *
0016                   *
0017                            DEF   PRTB
0018                            REF   FSTSYM
0019                   *
0020                            DXOP SVC,15
0021                   *
0022         0002  SMSYM   EQU   2
0023         0008  SMDEF   EQU   8
0024         000A  SMREF   EQU   >A
0025         0002  REFVAL  EQU   2
0026         0004  SYSFLG  EQU   4                   PRB FLAGS
0027         0006  BFADR   EQU   6
0028         0008  BFLTH   EQU   8
0029         000A  CCOUNT  EQU   10
```

```
PRINT SYMBOL TABLE                                        PAGE 0002

 0031                 *
 0032                 *
 0033                 *
 0034       0000'  PRTB   EQU  $
 0035  0000 C18B          MOV  R11,R6                 SAVE RETURN
 0036  0002 C820          MOV  @FSTSYM,@NXTSYM        POINTER TO FIRST SYMBOL ENTRY
       0004 0000
       0006 ----
 0037                 *
 0038       0008'  PRTB01 EQU  $                      **PRINT A SYMBOL
 0039  0008 13--          JEQ  PRTBXT                 IF DONE
 0040  000A 06A0          BL   @BLNKLN                SET OUTBUF TO BLANKS
       000C ----
 0041  000E C0A0          MOV  @NXTSYM,R2             MOVE SYMBOL TO BUFFER
       0010 ----
 0042  0012 C042          MOV  R2,R1
 0043  0014 0221          AI   R1,SMSYM
       0016 0002
 0044  0018 0200          LI   R0,OUTBUF
       001A ----
 0045  001C CC31          MOV  *R1+,*R0+
 0046  001E CC31          MOV  *R1+,*R0+
 0047  0020 CC31          MOV  *R1+,*R0+
 0048                 *
 0049  0022 C062          MOV  @SMDEF(R2),R1          MOVE SYMBOL DEF TO OUTBUF
       0024 0008
 0050  0026 0281          CI   R1,>FFFF                  (IF IT EXISTS
       0028 FFFF
 0051  002A 13--          JEQ  PRTB02
 0052  002C C281          MOV  R1,R10
 0053  002E 0209          LI   R9,OUTBUF+8            CONVERT BIN TO DECIMAL
       0030 ----
 0054  0032 06A0          BL   @CONV
       0034 ----
 0055                 *
 0056       0036'  PRTB02 EQU  $                      PROCESS REFERENCES
       002A**1305
 0057  0036 C0A0          MOV  @NXTSYM,R2
       0038 ----
 0058  003A C162          MOV  @SMREF(R2),R5
       003C 000A
 0059  003E 0203          LI   R3,OUTBUF+16
       0040 ----
 0060  0042 0204          LI   R4,7                   7 REFERENCES PER LINE
       0044 0007
 0061       0046'  PRTB03 EQU  $
 0062  0046 C145          MOV  R5,R5                  IF END OF REF CHAIN
 0063  0048 13--          JEQ  PRTB05
 0064                 *
 0065  004A C2A5          MOV  @REFVAL(R5),R10        OUTPUT REF TO LINE
       004C 0002
 0066  004E C243          MOV  R3,R9
 0067  0050 06A0          BL   @CONV
       0052 ----
```

```
PRINT SYMBOL TABLE                                              PAGE 0003

 0068  0054  0223          AI    R3,8                 NEXT LINE POSITION
       0056  0008
 0069  0058  0604          DEC   R4                   IF LINE FULL AND MORE REFS REM
 0070  005A  15--          JGT   PRTB04
 0071  005C  C555          MOV   *R5,*R5
 0072  005E  13--          JEQ   PRTB04
 0073                *
 0074  0060  06A0          BL    @PRNTLN              PRINT CURRENT LINE
       0062  ----
 0075  0064  06A0          BL    @BLNKLN              RESET LINE POINTERS
       0066  ----
 0076  0068  0203          LI    R3,OUTBUF+16
       006A  ----
 0077  006C  0204          LI    R4,7
       006E  0007
 0078                *
 0079        0070' PRTB04 EQU   $
       005A**150A
       005E**1308
 0080  0070  C155          MOV   *R5,R5               CHAIN TO NEXT REF
 0081  0072  10E9          JMP   PRTB03
 0082                *
 0083        0074' PRTB05 EQU   $
       0048**1315
 0084  0074  06A0          BL    @PRNTLN              PRINT LAST LINE
       0076  ----
 0085                *
 0086        0078' PRTB06 EQU   $
 0087  0078  C0A0          MOV   @NXTSYM,R2           CHAIN TO NEXT SYMBOL
       007A  ----
 0088  007C  C812          MOV   *R2,@NXTSYM
       007E  ----
 0089  0080  10C3          JMP   PRTB01
 0090                *
 0091  0082  0456    PRTBXT B     *R6                  RETURN
       0008**133C
 0092                *
```

```
PRINT SYMBOL TABLE                                              PAGE 0004

 0094                     *
 0095                     *       BLANK LINE BUFFER
 0096                     *
 0097                     *         REGISTERS USED - R0,R1,R2
 0098                     *
 0099            0084'  BLNKLN EQU   $
       000C**0084'
       0066**0084'
 0100  0084   0202             LI    R2,40
       0086   0028
 0101  0088   0201             LI    R1,'  '
       008A   2020
 0102  008C   0200             LI    R0,OUTBUF
       008E   ----
 0103         0090'  BLNK01 EQU   $
 0104  0090   CC01             MOV   R1,*R0+
 0105  0092   0602             DEC   R2
 0106  0094   15FD             JGT   BLNK01
 0107  0096   045B             RT
 0108                     *
 0109                     *       STRIP TRAILING BLANKS AND PRINT OUTPUT LINE
 0110                     *
 0111         0098'  PRNTLN EQU   $
       0062**0098'
       0076**0098'
 0112  0098   0200             LI    R0,OUTBUF+79       LAST BUFFER POSITION
       009A   ----
 0113         009C'  PR1    EQU   $
 0114  009C   9810             CB    *R0,@BLANK
       009E   ----
 0115  00A0   16--             JNE   PR2
 0116  00A2   0600             DEC   R0                 ASSUME AT LEAST ONE CHAR IN BU
 0117  00A4   10FB             JMP   PR1
 0118         00A6'  PR2    EQU   $
       00A0**1602
 0119  00A6   0201             LI    R1,OUTBUF-1
       00A8   ----
 0120  00AA   6001             S     R1,R0
 0121  00AC   C800             MOV   R0,@WTCC           OUTPUT CHAR COUNT
       00AE   ----
 0122  00B0   2FE0             SVC   @WTPRB             PRINT LINE
       00B2   ----
 0123  00B4   045B             RT
 0124                     *
 0125                     **      CONVERT BINARY TO DECIMAL.
 0126                     *
 0127                     *
 0128                     *         REGISTERS USED - R0,R1,R2
 0129                     *         CALLING SEQUENCE:
 0130                     *           R10 - VALUE TO BE CONVERTED
 0131                     *           R9  - POINTER TO BUFFER FOR RESULT
 0132                     *
 0133         00B6'  CONV   EQU   $
       0034**00B6'
```

```
PRINT SYMBOL TABLE                                          PAGE 0005

        0052**00B6'
  0134  00B6  C08A          MOV  R10,R2             USE REPEATED DIVIDE
  0135  00B8  0200          LI   R0,1000            AND LOOK UP QUOTIENT IN TABLE
        00BA  03E8
  0136  00BC  04C1          CLR  R1
  0137  00BE  3C40          DIV  R0,R1
  0138  00C0  DE61          MOVB @CLIST(R1),*R9+
        00C2  ----
  0139  00C4  0200          LI   R0,100
        00C6  0064
  0140  00C8  04C1          CLR  R1
  0141  00CA  3C40          DIV  R0,R1
  0142  00CC  DE61          MOVB @CLIST(R1),*R9+
        00CE  ----
  0143  00D0  04C1          CLR  R1
  0144  00D2  0200          LI   R0,10
        00D4  000A
  0145  00D6  3C40          DIV  R0,R1
  0146  00D8  DE61          MOVB @CLIST(R1),*R9+
        00DA  ----
  0147  00DC  DE62          MOVB @CLIST(R2),*R9+   REMAINDER IS LAST DIGIT
        00DE  ----
  0148  00E0  045B          RT
  0149                *
  0150                *
  0151  00E2    30  CLIST   TEXT '0123456789'       CONVERT BIN TO DEC
        00C2**00E2'
        00CE**00E2'
        00DA**00E2'
        00DE**00E2'
  0152  00EC  0A0D          DATA >0A0D              CR,LF
  0153  00EE        OUTBUF  BSS  80
        001A**00EE'
        0030**00F6'
        0040**00FE'
        006A**00FE'
        008E**00EE'
        009A**013D'
        00A8**00ED'
  0154  013E  0000  WTPRB   DATA 0,>B00,0,OUTBUF-2,80 OUTPUT PRB
        0140  0B00
        0142  0000
        0144  00EC'
        0146  0050
        00B2**013E'
  0155  0148  0000  WTCC    DATA 0                  OUTPUT CHAR COUNT
        00AE**0148'
  0156                *
  0157  014A    20  BLANK   BYTE ' ',0
        014B    00
        009E**014A'
  0158  014C  0000  NXTSYM  DATA 0
        0006**014C'
        0010**014C'
        0038**014C'
```

```
PRINT SYMBOL TABLE                                    PAGE 0006

       007A◆◆014C'
       007E◆◆014C'
  0159                       END

0000 ERRORS
```

10.3 LOADING MODULES WITH PX9LAL

The second part of the program is the loading of modules. Using PX9LAL, link and load the object modules into memory. (Refer to the software loading procedures in Section II.) Before PX9LAL can be used, it must be loaded into the monitor transient area using the OV command (Section III). When PX9LAL asks for

```
LD PT?
LD BI?
```

enter a carriage return after each to specify the default values of 0 and $A0_{16}$, respectively. In answer to

```
F/P LIST?
```

enter either F (full) or P (partial). The object modules may be loaded from either cassette drive. When PX9LAL prints

```
LOAD/END?
```

enter either L or L7 to load from cassette 1 or L8 to load from cassette 2.

When all of the modules have been loaded, the program entry point is printed. This value is placed in the user's PC register.

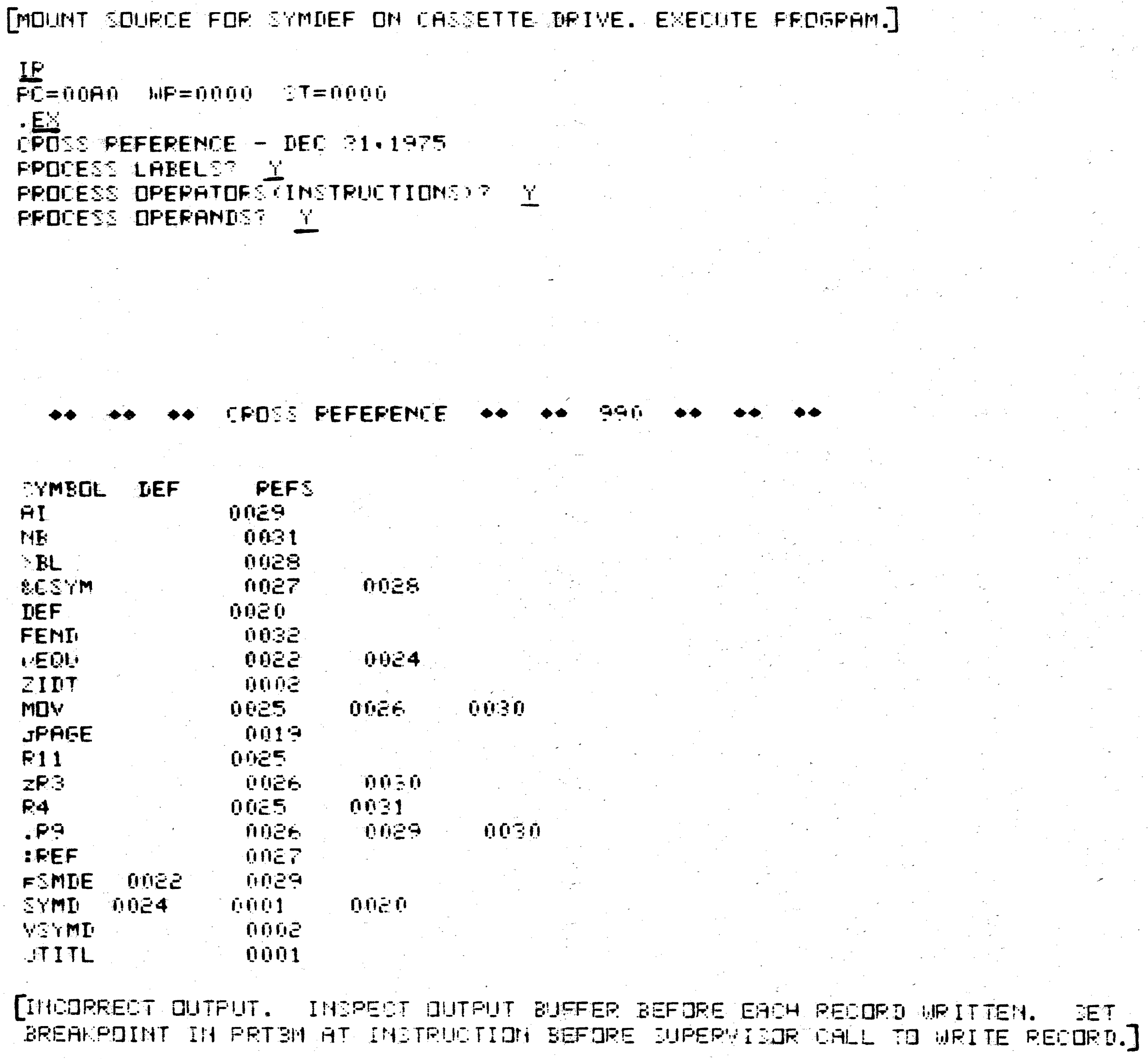
```
.OV ?
LL

.LL

LD PT?
LD BI?
F/P LIST? E

LOAD/END? L

      XREF     00A0
            ◆ PRINTC   0136
            ◆ GETCHR   0144
            ◆ TERM     0168
LOAD/END? L

      PARSEM   0254
            ◆ PARSE    02B6
            ◆ DEFERR   0410
            ◆ OPNDRR   0414
            ◆ OPERRR   0412
            ◆ STMT     0406
LOAD/END? L8

      CTYPM    0476
            ◆ CTYP     04BE
LOAD/END? L

      PRTBM    0508
            ◆ PRTB     0508
LOAD/END? L8

      CSYMM    0652
            ◆ CSYM     0652
            ◆ ISYM     06B4
            ◆ NXTLOC   06C6
            ◆ ENDSYM   1DD6
            ◆ FSTSYM   06C8
LOAD/END? L8

      SYMRFM   1DE2
            ◆ SYMREF   1DE2
            ◆ OVFL     1E18
LOAD/END? L8

      SYMDFM   1E38
            ◆ SYMDEF   1E38
LOAD/END? E
 ENTRY = 00A0

TERM/CONT? T

.
```

10.4 DEBUGGING THE PROGRAM

The third part of the program is the debugging. Execute the program using the EX command (Section III). Use the source module named SYMDEF as the input to the concordance program. (This is the shortest source module.)

The source tape may be positioned to the beginning of SYMDEF by setting the PLAYBACK switch on the data terminal to the LOCAL position and the PRINTER switch to the OFF position. By pressing the CONT START switch in the Playback Control area on the data terminal upper switch panel, the tape will be read to an end-of-file marker and positioned at the beginning of the next file. Repeat this process until the tape is correctly positioned to SYMDEF. Set the PLAYBACK and PRINTER switches to the LINE position.

Follow the debugging procedure outlined in the computer printout in this paragraph. The debugging in this example occurs in PRTBM. For descriptions of the individual monitor keyboard commands, refer to Section III.



```
[MOUNT SOURCE FOR SYMDEF ON CASSETTE DRIVE. EXECUTE PROGRAM.]

 IR
 PC=00A0  WP=0000  ST=0000
 .EX
 CROSS REFERENCE - DEC 31,1975
 PROCESS LABELS?  Y
 PROCESS OPERATORS(INSTRUCTIONS)?  Y
 PROCESS OPERANDS?  Y




   ◆◆  ◆◆  ◆◆  CROSS REFERENCE  ◆◆  ◆◆  990  ◆◆  ◆◆  ◆◆


 SYMBOL  DEF     REFS
 AI             0029
 NB              0031
 >BL             0028
 &CSYM           0027     0028
 DEF            0020
 FEND            0032
 vEQU            0022     0024
 ZIDT            0002
 MOV            0025     0026     0030
 jPAGE           0019
 R11            0025
 zR3             0026     0030
 R4             0025     0031
 .R9             0026     0029     0030
 :REF            0027
 FSMDE  0022     0029
 SYMD  0024     0001     0020
 VSYMD           0002
 JTITL           0001

[INCORRECT OUTPUT.  INSPECT OUTPUT BUFFER BEFORE EACH RECORD WRITTEN.  SET
 BREAKPOINT IN PRTBM AT INSTRUCTION BEFORE SUPERVISOR CALL TO WRITE RECORD.]
```

[TO FIND THE ABSOLUTE ADDRESS OF THE DESIRED INSTRUCTION, ADD THE MODULE LOAD POINT (AS SPECIFIED IN LOAD MAP) TO THE RELATIVE ADDRESS WITHIN THE MODULE.]

```
.HA 508 A8
SUM=05B0   +01456   DIFF=0460   +01120
.SB 1,5B0
.MP

PC=00A0
WP=0000
ST=0000  2
.
```

[REPOSITION TAPE TO SYMDEF.]

```
RU
CROSS REFERENCE - DEC 31,1975
PROCESS LABELS?  Y
PROCESS OPERATORS(INSTRUCTIONS)?   Y
PROCESS OPERANDS?  Y




  **  **  **  CROSS REFERENCE  **  **  990  **  **  **


SYMBOL  DEF      REFS
BKPT#1
PC=05B0  WP=016A  ST=D002
.
```

[LOOK AT OUTPUT BUFFER.]

```
HA 508 EA
SUM=05F2  +01522  DIFF=041E  +01054
.IM 5F2 632
05F2=0806  4149  2020  2020  >2020  2020  2020  2020
0602=3030  3239  2020  2020  >2020  2020  2020  2020
0612=2020  2020  2020  2020  >2020  2020  2020  2020
0622=2020  2020  2020  2020  >2020  2020  2020  2020
0632=2020
.
```

```
[FIRST TWO CHARACTERS OF BUFFER INVALID. POINTER TO SYMBOL TABLE
 NEEDS TO BE INCREMENTED BY TWO. INSERT THE STMT  AI  R1,2   AFTER
 STMT 44. SINCE THERE IS NO ROOM TO INSERT THIS TWO WORD INSTRUCTION,
 WE MUST MAKE A PATCH TO A UNUSED PORTION OF MEMORY, INSERT THE
 INSTRUCTIONS (STMT 45 WHICH WE MUST OVERLAY WITH A BRANCH, AND STMT
 THAT WE ARE INSERTING), AND BRANCH BACK TO THE CODE WE CAME FROM.]

[TO DO THIS PATCH AT STMT 45:

      B    @PATCH              WHERE PATCH = IF00

 AT 1F00 INSERT:

      AI   R1,2
      LI   R0,OUTBUF
      B    @STMT_46            WHERE STMT_46 = 508+18 = 520
 THE CODE FOR THE PATCHES IS :

      0460
      1F00


      0221
      0002
      0200
      05F2
      0460
      0520      ]


 MM 51C
 051C=0200  0460
 051E=05F2  1F00
 .MM 1F00
 1F00=0000  0221
 1F02=0000  2
 1F04=0000  200
 1F06=1E18  5F2
 1F08=4F56  460
 1F0A=464C  520
 .
```

```
[BY ENTERING THE RU COMMAND , THE PROGRAM WILL CONTINUE EXECUTING
 FROM THE BREAKPOINT. THE RECORD WITH THE ERROR IN IT WILL BE PRINTED
 AND THE NEXT RECORD BUILT. THE PROGRAM WILL HALT AT THE BREAKPOINT
 BEFORE PRINTING THE NEXT RECORD. WE CAN INSPECT THE BUFFER TO
 DETERMINE IF OUR PATCH WAS CORRECT.]

 RU
 AI                 0029
 BKPT#1
 PC=05B0  WP=016A  ST=D402
 .IM 5F2 632
 05F2=4220  2020  2020  2020  2020  2020  2020  2020
 0602=3030  3331  2020  2020  2020  2020  2020  2020
 0612=2020  2020  2020  2020  2020  2020  2020  2020
 0622=2020  2020  2020  2020  2020  2020  2020  2020
 0632=2020
 .


[THE BUFFER APPEARS CORRECT. REMOVE THE BREAKPOINT AND PRINT THE
 REST OF THE SYMBOL TABLE.]


 CB,1.
 .RU
 B                  0031
 BL                 0028
 CSYM               0027       0028
 DEF                0020
 END                0032
 EQU                0022       0024
 IDT                0002
 MOV                0025       0026       0030
 PAGE               0019
 R11                0025
 R3                 0026       0030
 R4                 0025       0031
 R9                 0026       0029       0030
 REF                0027
 SMDEF     0022     0029
 SYMDEF    0024     0001       0020
 SYMDFM             0002
 TITL               0001

 .



[NOW THAT WE HAVE DETERMINED THAT THE PATCH IS CORRECT, RELOAD THE
 EDITOR AND INSERT THE MISSING STATEMENT, RE-ASSEMBLE, AND VERIFY
 THAT EVERYTHING IS CORRECT.]
```

10.5 EDITING WITH PX9EDT

The fourth part of the program is editing using PX9EDT. For a description of how to use the text editor, refer to Section IV. The text editor must be loaded using the LU command (Section III).

```
.LU 8
.EX

PX9EDT  945394 ** 15MAR76
ADD 4K MEM BLOCKS CONFIGURED? 0

POSITION TAPES, ENTER CR

 ?D50
 ?P-15
 0035 PRTB    EQU   $
 0036         MOV   R11,R6                 SAVE RETURN
 0037         MOV   @FSTSYM,@NXTSYM        POINTER TO FIRST SYMBOL ENTRY
 0038 *
 0039 PRTB01  EQU   $                      **PRINT A SYMBOL
 0040         JEQ   PRTBXT                 IF DONE
 0041         BL    @BLNKLN                SET OUTBUF TO BLANKS
 0042         MOV   @NXTSYM,R2             MOVE SYMBOL TO BUFFER
 0043         MOV   R2,R1
 0044 *      AI   R1,SMSYM           REMOVED TO CREATE ERROR
 0045         LI    R0,OUTBUF
 0046         MOV   *R1+,*R0+
 0047         MOV   *R1+,*R0+
 0048         MOV   *R1+,*R0+
 0049 *
 ?C44-44
 AI R1,SMSYM                       CORRECTED ERROR

 ?P42-46
 0042         MOV   @NXTSYM,R2             MOVE SYMBOL TO BUFFER
 0043         MOV   R2,R1
              AI    R1,SMSYM                  CORRECTED ERROR
 0045         LI    R0,OUTBUF
 0046         MOV   *R1+,*R0+
 ?Q
END EDIT
TERMINATE/CONTINUE?T

.
```

10.6 REASSEMBLING, RELINKING AND LOADING MODULES AND EXECUTING THE PROGRAM

The fifth part of the program is reassembly of the edited module. The sixth part of the program is the relinking and loading of all modules. The seventh part of the program is execution of the final version of the program.

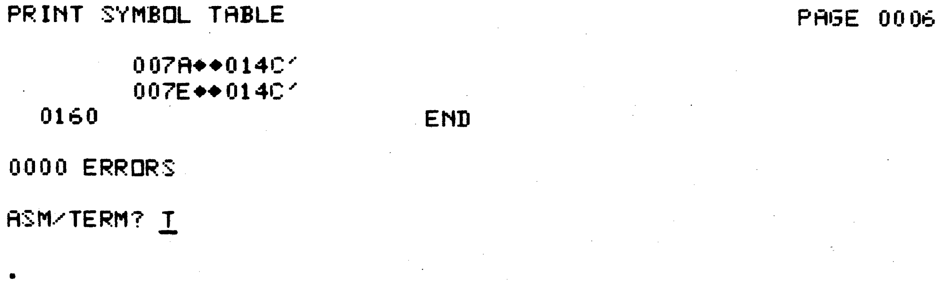
```
.LU 7
.EX

PX9ASM  945393 ◆◆ 15MAR76
ADD 4K MEM BLOCKS CONFIGURED? 0
PREDEFINED REGISTERS? Y

ASM/TERM? A




                                                  PAGE 0001

0001
0002
0004                     IDT  'PRTBM'
0005              ◆
0006              ◆      PRTB WILL READ THE SYMBOL TABLE ONE SYMBOL
0007              ◆      AT A TIME AND PRINT THE SYMBOL NAME,
0008              ◆      THE STATEMENT NUMBER WHERE THE SYMBOL
0009              ◆      WAS DEFINED, AND THE LIST OF STATEMENT
0010              ◆      NUMBERS WHERE THE SYMBOL WAS REFERENCED.
0011              ◆
0012              ◆      CALLING SEQUENCE:
0013              ◆        NO INPUT PARMS
0014              ◆
0015              ◆        REGISTERS DESTROYED - R0,R1,R2,R3,R4,R5,R6,R9,R10
0016              ◆
0017              ◆
0018                     DEF  PRTB
0019                     REF  FSTSYM
0020              ◆
0021                     DXOP SVC,15
0022              ◆
0023        0002  SMSYM  EQU  2
0024        0003  SMDEF  EQU  3
0025        000A  SMREF  EQU  >A
0026        0002  REFVAL EQU  2
0027        0004  SYSFLG EQU  4                   PRB FLAGS
0028        0006  BFADR  EQU  6
0029        0003  BFLTH  EQU  3
0030        000A  CCOUNT EQU  10
```

```
PRINT SYMBOL TABLE                                              PAGE 0002

 0032                *
 0033                *
 0034                *
 0035          0000' PRTB   EQU  $
 0036   0000   C18B         MOV  R11,R6             SAVE RETURN
 0037   0002   C820         MOV  @FSTSYM,@NXTSYM    POINTER TO FIRST SYMBOL ENTRY
        0004   0000
        0006   ----
 0038                *
 0039          0008' PRTB01 EQU  $                  **PRINT A SYMBOL
 0040   0008   13--         JEQ  PRTBXT             IF DONE
 0041   000A   06A0         BL   @BLNKLN            SET OUTBUF TO BLANKS
        000C   ----
 0042   000E   C0A0         MOV  @NXTSYM,R2         MOVE SYMBOL TO BUFFER
        0010   ----
 0043   0012   C042         MOV  R2,R1
 0044   0014   0221         AI   R1,SMSYM
        0016   0002
 0045   0018   0200         LI   R0,OUTBUF
        001A   ----
 0046   001C   CC31         MOV  *R1+,*R0+
 0047   001E   CC31         MOV  *R1+,*R0+
 0048   0020   CC31         MOV  *R1+,*R0+
 0049                *
 0050   0022   C062         MOV  @SMDEF(R2),R1      MOVE SYMBOL DEF TO OUTBUF
        0024   0008
 0051   0026   0281         CI   R1,>FFFF              (IF IT EXISTS
        0028   FFFF
 0052   002A   13--         JEQ  PRTB02
 0053   002C   C281         MOV  R1,R10
 0054   002E   0209         LI   R9,OUTBUF+8        CONVERT BIN TO DECIMAL
        0030   ----
 0055   0032   06A0         BL   @CONV
        0034   ----
 0056                *
 0057          0036' PRTB02 EQU  $                  PROCESS REFERENCES
        002A**1305
 0058   0036   C0A0         MOV  @NXTSYM,R2
        0038   ----
 0059   003A   C162         MOV  @SMREF(R2),R5
        003C   000A
 0060   003E   0203         LI   R3,OUTBUF+16
        0040   ----
 0061   0042   0204         LI   R4,7               7 REFERENCES PER LINE
        0044   0007
 0062          0046' PRTB03 EQU  $
 0063   0046   C145         MOV  R5,R5              IF END OF REF CHAIN
 0064   0048   13--         JEQ  PRTB05
 0065                *
 0066   004A   C2A5         MOV  @REFVAL(R5),R10    OUTPUT REF TO LINE
        004C   0002
 0067   004E   C243         MOV  R3,R9
 0068   0050   06A0         BL   @CONV
        0052   ----
```

```
PRINT SYMBOL TABLE                                                  PAGE 0003

 0069   0054   0223            AI    R3,8               NEXT LINE POSITION
        0056   0008
 0070   0058   0604            DEC   R4                 IF LINE FULL AND MORE REFS REM
 0071   005A   15--            JGT   PRTB04
 0072   005C   C555            MOV   ◆R5,◆R5
 0073   005E   13--            JEQ   PRTB04
 0074                   ◆
 0075   0060   06A0            BL    @PRNTLN            PRINT CURRENT LINE
        0062   ----
 0076   0064   06A0            BL    @BLNKLN            RESET LINE POINTERS
        0066   ----
 0077   0068   0203            LI    R3,OUTBUF+16
        006A   ----
 0078   006C   0204            LI    R4,7
        006E   0007
 0079                   ◆
 0080          0070'    PRTB04 EQU   $
        005A◆◆150A
        005E◆◆1308
 0081   0070   C155            MOV   ◆R5,R5             CHAIN TO NEXT REF
 0082   0072   10E9            JMP   PRTB03
 0083                   ◆
 0084          0074'    PRTB05 EQU   $
        0048◆◆1315
 0085   0074   06A0            BL    @PRNTLN            PRINT LAST LINE
        0076   ----
 0086                   ◆
 0087          0078'    PRTB06 EQU   $
 0088   0078   C0A0            MOV   @NXTSYM,R2         CHAIN TO NEXT SYMBOL
        007A   ----
 0089   007C   C812            MOV   ◆R2,@NXTSYM
        007E   ----
 0090   0080   10C3            JMP   PRTB01
 0091                   ◆
 0092   0082   0456     PRTBXT B     ◆R6                RETURN
        0008◆◆133C
 0093                   ◆
```

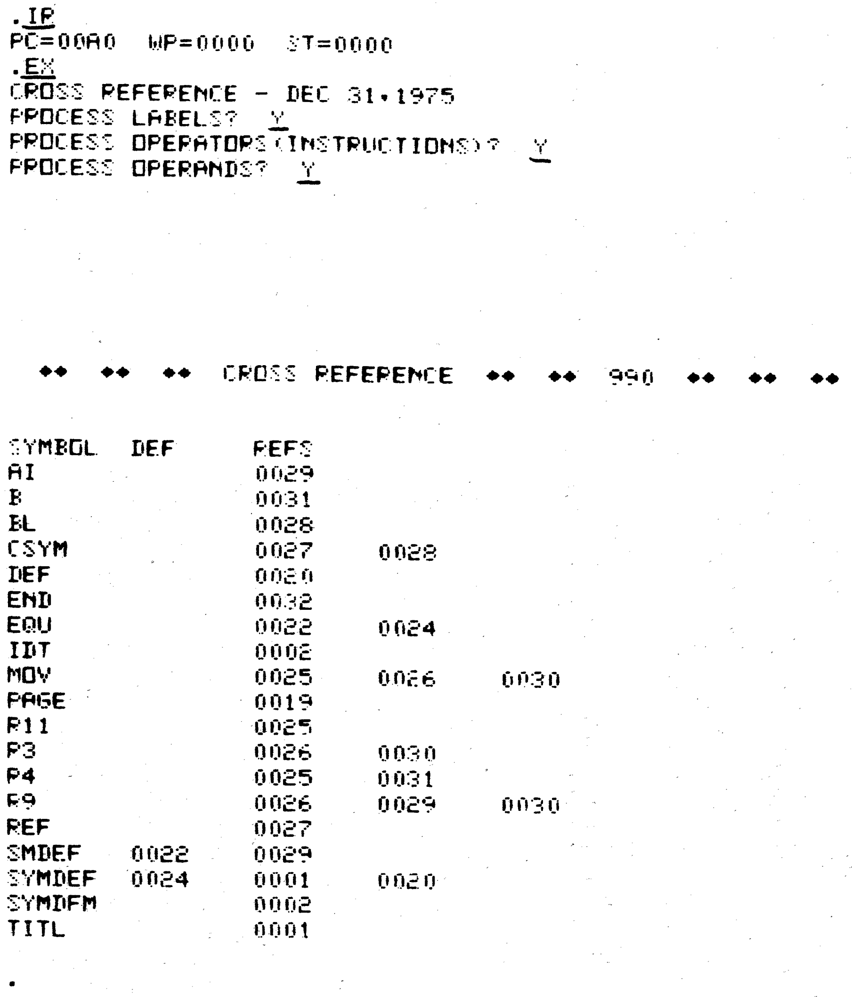
```
PRINT SYMBOL TABLE                                          PAGE 0004

 0095                   *
 0096                   *      BLANK LINE BUFFER
 0097                   *
 0098                   *        REGISTERS USED - R0,R1,R2
 0099                   *
 0100            0084'  BLNKLN EQU   $
       000C**0084'
       0066**0084'
 0101  0084     0202           LI    R2,40
       0086     0028
 0102  0088     0201           LI    R1,'  '
       008A     2020
 0103  008C     0200           LI    R0,OUTBUF
       008E     ----
 0104            0090'  BLNK01 EQU   $
 0105  0090     CC01           MOV   R1,*R0+
 0106  0092     0602           DEC   R2
 0107  0094     15FD           JGT   BLNK01
 0108  0096     045B           RT
 0109                   *
 0110                   *      STRIP TRAILING BLANKS AND PRINT OUTPUT LINE
 0111                   *
 0112            0098'  PRNTLN EQU   $
       0062**0098'
       0076**0098'
 0113  0098     0200           LI    R0,OUTBUF+79       LAST BUFFER POSITION
       009A     ----
 0114            009C'  PR1    EQU   $
 0115  009C     9810           CB    *R0,@BLANK
       009E     ----
 0116  00A0     16--           JNE   PR2
 0117  00A2     0600           DEC   R0                 ASSUME AT LEAST ONE CHAR IN BU
 0118  00A4     10FB           JMP   PR1
 0119            00A6'  PR2    EQU   $
       00A0**1602
 0120  00A6     0201           LI    R1,OUTBUF-3
       00A8     ----
 0121  00AA     6001           S     R1,R0
 0122  00AC     C800           MOV   R0,@WTCC           OUTPUT CHAR COUNT
       00AE     ----
 0123  00B0     2FE0           SVC   @WTPRB             PRINT LINE
       00B2     ----
 0124  00B4     045B           RT
 0125                   *
 0126                   **     CONVERT BINARY TO DECIMAL
 0127                   *
 0128                   *
 0129                   *        REGISTERS USED - R0,R1,R2
 0130                   *        CALLING SEQUENCE:
 0131                   *          R10 - VALUE TO BE CONVERTED
 0132                   *          R9  - POINTER TO BUFFER FOR RESULT
 0133                   *
 0134            00B6'  CONV   EQU   $
       0034**00B6'
```

```
PRINT SYMBOL TABLE                                           PAGE 0005

          0052**00B6'
  0135    00B6  C08A             MOV  R10,R2              USE REPEATED DIVIDE
  0136    00B8  0200             LI   R0,1000             AND LOOK UP QUOTIENT IN TABLE
          00BA  03E8
  0137    00BC  04C1             CLR  R1
  0138    00BE  3C40             DIV  R0,R1
  0139    00C0  DE61             MOVB @CLIST(R1),*R9+
          00C2  ----
  0140    00C4  0200             LI   R0,100
          00C6  0064
  0141    00C8  04C1             CLR  R1
  0142    00CA  3C40             DIV  R0,R1
  0143    00CC  DE61             MOVB @CLIST(R1),*R9+
          00CE  ----
  0144    00D0  04C1             CLR  R1
  0145    00D2  0200             LI   R0,10
          00D4  000A
  0146    00D6  3C40             DIV  R0,R1
  0147    00D8  DE61             MOVB @CLIST(R1),*R9+
          00DA  ----
  0148    00DC  DE62             MOVB @CLIST(R2),*R9+     REMAINDER IS LAST DIGIT
          00DE  ----
  0149    00E0  045B             RT
  0150                     *
  0151                     *
  0152    00E2    30       CLIST  TEXT '0123456789'       CONVERT BIN TO DEC
          00C2**00E2'
          00CE**00E2'
          00DA**00E2'
          00DE**00E2'
  0153    00EC  0A0D             DATA >0A0D               CR,LF
  0154    00EE             OUTBUF BSS  80
          001A**00EE'
          0030**00F6'
          0040**00FE'
          006A**00FE'
          008E**00EE'
          009A**013D'
          00A8**00EB'
  0155    013E  0000       WTPRB  DATA 0,>B00,0,OUTBUF-2,80 OUTPUT PRB
          0140  0B00
          0142  0000
          0144  00EC'
          0146  0050
          00B2**013E'
  0156    0148  0000       WTCC   DATA 0                  OUTPUT CHAR COUNT
          00AE**0148'
  0157                     *
  0158    014A    20       BLANK  BYTE ' ',0
          014B    00
          009E**014A'
  0159    014C  0000       NXTSYM DATA 0
          0006**014C'
          0010**014C'
          0038**014C'
```



```
PRINT SYMBOL TABLE                                          PAGE 0006

         007A◆◆014C´
         007E◆◆014C´
  0160                           END

0000 ERRORS

ASM/TERM? T

.
```

```
.LL

LD PT?
LD BI?
F/P LIST? P

LOAD/END? L8

      XREF     00A0
LOAD/END? L8

      PARSEM   0254
LOAD/END? L7

      CTYPM    0476
LOAD/END? L7

      PRTBM    0508
LOAD/END? L8

      CSYMM    0656
LOAD/END? L

LOAD/END? L8

      SYMRFM   1DE6
LOAD/END? L8

      SYMDFM   1E3C
LOAD/END? E
 ENTRY = 00A0

TERM/CONT? T
```



```
.IR
PC=00A0  WP=0000  ST=0000
.EX
CROSS REFERENCE - DEC 31,1975
PROCESS LABELS?  Y
PROCESS OPERATORS(INSTRUCTIONS)?  Y
PROCESS OPERANDS?  Y




  **  **  **  CROSS REFERENCE  **  **  990  **  **  **

SYMBOL  DEF      REFS
AI               0029
B                0031
BL               0028
CSYM             0027     0028
DEF              0020
END              0032
EQU              0022     0024
IDT              0002
MOV              0025     0026     0030
PAGE             0019
R11              0025
R3               0026     0030
R4               0025     0031
R9               0026     0029     0030
REF              0027
SMDEF   0022     0029
SYMDEF  0024     0001     0020
SYMDFM           0002
TITL             0001

.
```

SECTION XI

PROM PROGRAMMING EXAMPLES

11.1 INTRODUCTION

This section contains two examples of PROM programming processes. The first example shows how to program a user generated program into PROMs. It assumes that the user has already assembled his source program to create an object file on cassette tape. The example shows how the object file is loaded into memory and the steps required to program the PROMs using control information for the memory and PROM data configurations obtained from the Standard Control Information Cassette.

The second example shows how to use a PROM created in the first example to program another PROM with the same data. The data from the first PROM is transferred into memory, the memory and PROM data configurations are set up using the PROM programmer keyboard commands, and the data is then programmed into the PROM.

11.2 EXAMPLE 1

The first step is to load the object file into memory. This may be done with the Load Program (LP) command since the object file is in standard 990 object format and does not need to be linked. The PROM Programmer Standard (PS) command is then used to define the control information for the memory and PROM data configurations. The PROM to be programmed is an SN74S287 PROM which consists of 256 words of 4 bits each. In this example, the memory configuration will be set up to program from the first 4 bits of each memory word in a 256 word block. The memory and PROM bounds are defined with the MB and RB subcommands.

```
.LP,8,0
.PS,MS287-0,S287
.PP,MB,0,1FF
.PP,RB,0,FF
.
```

The LP command loads the object code to be programmed into the PROM beginning at memory location 0. The PS command defines the standard control information for programming a 256 X 4 (SN74S287) PROM with the first four bits of each of 256 words of memory. The MB and RB subcommands specify a transfer of data into PROM word addresses 0-FF_{16} from memory addresses 0 through $1FF_{16}$. The memory data may be displayed to see what will be programmed into the PROM.

```
.PP,TS,1,0,0,0
.PP,GO
M0000.00=00  M0002.00=01  M0004.00=00  M0006.00=06
M0008.00=01  M000A.00=01  M000C.00=00  M000E.00=00
M0010.00=00  M0012.00=00  M0014.00=01  M0016.00=01
M0018.00=0C  M001A.00=03  M001C.00=03  M001E.00=00
M0020.00=00  M0022.00=00  M0024.00=00  M0026.00=01
M0028.00=00  M002A.00=01  M002C.00=00  M002E.00=01
M0030.00=00  M0032.00=01  M0034.00=00  M0036.00=01
M0038.00=01  M003A.00=01  M003C.00=00  M003E.00=01
M0040.00=01  M0042.00=02  M0044.00=01  M0046.00=00
M0048.00=00  M004A.00=00  M004C.00=01  M004E.00=00
M0050.00=00  M0052.00=01  M0054.00=01  M0056.00=01
M0058.00=00  M005A.00=01  M005C.00=0C  M005E.00=0C
M0060.00=0C  M0062.00=0C  M0064.00=0C  M0066.00=0C
```

The toggle is set to display memory with the TS subcommand, and the display is printed when the GO subcommand is entered.

Once the user verifies the data in memory, it is ready to be programmed into PROM. The toggles to program PROM and compare memory and PROM are set with the TS subcommand. The PROM should be inserted in the PROM Programming Module. The programming process is initiated when the GO subcommand is entered.

```
.PP,TS,0,0,1,1
.PP,GO
.
```

The compare is successful, the PROM programming returns to the monitor, and the prompt character (.) is displayed while waiting for the next command. If any compare errors are found, they will be printed before the PROM programmer returns to the monitor.

The following printout shows a programming process in which compare errors were found.

```
.PP,TS,0,0,1,1
.PP,GO
>M0000.00=01   R0000.00=FF
>M0010.00=F4   R0008.00=F9
>M001C.00=00   R000E.00=75
>M0042.00=05   R0021.00=12
>M0046.00=85   R0023.00=FF
.
```

The user may display the PROM after it has been programmed to see what was programmed into PROM and compare it to the memory data display. To display PROM the toggle is set with the TS subcommand.

```
.PP,TS,0,1,0,0
.PP,GO
R0000.00=00  R0001.00=01  R0002.00=00  R0003.00=06
R0004.00=01  R0005.00=01  R0006.00=00  R0007.00=00
R0008.00=00  R0009.00=00  R000A.00=01  R000B.00=01
R000C.00=0C  R000D.00=03  R000E.00=03  R000F.00=00
R0010.00=00  R0011.00=00  R0012.00=00  R0013.00=01
R0014.00=00  R0015.00=01  R0016.00=00  R0017.00=01
R0018.00=00  R0019.00=01  R001A.00=00  R001B.00=01
R001C.00=01  R001D.00=01  R001E.00=00  R001F.00=01
R0020.00=01  R0021.00=02  R0022.00=01  R0023.00=00
R0024.00=00  R0025.00=00  R0026.00=01  R0027.00=00
R0028.00=00  R0029.00=01  R002A.00=01  R002B.00=01
R002C.00=00  R002D.00=01  R002E.00=0C  R002F.00=0C
```

To program the second four bits of each memory word into a PROM, a new PROM is inserted and the following command is entered to get the needed control information for the memory configuration from the Standard Control Information Cassette. The PROM control information does not need to be changed as long as an SN74S287 PROM is being programmed.

```
.PS,MS287-4
.
```

Display the memory data configuration.

```
.PP,TS,1,0,0,0
.PP,GO
M0000.04=02  M0002.04=0F  M0004.04=02  M0006.04=07
M0008.04=06  M000A.04=0D  M000C.04=04  M000E.04=02
M0010.04=00  M0012.04=04  M0014.04=0F  M0016.04=03
M0018.04=00  M001A.04=02  M001C.04=02  M001E.04=00
M0020.04=02  M0022.04=00  M0024.04=06  M0026.04=06
M0028.04=06  M002A.04=06  M002C.04=04  M002E.04=0D
M0030.04=06  M0032.04=03  M0034.04=06  M0036.04=03
M0038.04=0F  M003A.04=06  M003C.04=06  M003E.04=0F
M0040.04=03  M0042.04=08  M0044.04=00  M0046.04=04
M0048.04=05  M004A.04=0A  M004C.04=07  M004E.04=04
M0050.04=00  M0052.04=00  M0054.04=00  M0056.04=00
```

The output displays the bit string beginning at bit four of each word of memory.

```
.PP,TS,0,0,1,1
.PP,GO
.
```

The toggles are set to program the PROM and compare. No compare error display indicates the PROM has been programmed with the data displayed from memory.

The third four bits of each word of memory can be programmed into a PROM using the following commands to get the control information for memory from cassette and set toggles to program PROMs and compare. A new PROM should be inserted before each programming process is initiated.

```
.PS,MS287-8
.PP,TS,0,0,1,1
.PP,GO
.
```

A similar set of commands can be used to program the fourth four bits of each word of memory.

```
.PS,MS287-C
.PP,TS,0,0,1,1
.PP,GO
.
```

11.3 EXAMPLE 2

This example loads the memory data from the first PROM programmed in the previous example, and uses this data to program another PROM. The first step is to define the memory and PROM data configurations to be used in the transfer from PROM to memory and then from memory to PROM. The keyboard commands are used for tutorial purposes in this example to set up the data configurations instead of using the control information on the Standard Control Information Cassette. The keyboard subcommands needed to define the same information found on the Standard Control Information Cassette are MI, RI, SW, and RC. The memory and PROM bounds are defined with the MB and RB subcommands.

```
.PP,MI,1,10,100,0
.PP,RI,1,4,100,0
.PP,SW,4
.PP,RC,4,0,2,0,19,1
.PP,MB,0,1FF
.PP,RB,0,FF
.
```

The MI, SW, and RC subcommands set up the control information for an SN74S287 PROM and for the first 4 bits of each word of a 256 word block of memory. The MB and RB subcommands specify a transfer of data between PROM word addresses 0 through FF_{16} and memory addresses 0 through $1FF_{16}$.

The MI subcommand defines the memory data configuration as follows:

Loop level	= 1
Bit increment	= 10_{16}
Number of iterations	= 100_{16}
Initial bit displacement	= 0

The RI subcommand defines the PROM data configuration as follows:

Loop level	= 1
Bit increment	= 4
Number of iterations	= 100_{16}
Initial bit displacement	= 0

The SW command defines the bit string width for memory and PROM to be 4.

The RC subcommand defines the following PROM characteristics:

High/low level output	= 0
Pulse width	= 2
Number of retries	= 0
Duty cycle	= 19_{16}
Programmable bits	= 1

The user should insert the PROM containing the data configuration to be transferred to memory in the PROM programming module. The PROM data may be displayed by setting the toggles to display PROM with the TS subcommand. The display is printed when the GO subcommand is entered.

```
.PP,TS,0,1,0,0
.PP,GO
R0000.00=00  R0001.00=00  R0002.00=00  R0003.00=0E
R0004.00=03  R0005.00=06  R0006.00=03  R0007.00=07
R0008.00=0F  R0009.00=0F  R000A.00=0F  R000B.00=0F
R000C.00=0F  R000D.00=0F  R000E.00=0F  R000F.00=0F
R0010.00=0F  R0011.00=0F  R0012.00=0F  R0013.00=0F
R0014.00=0F  R0015.00=0F  R0016.00=0F  R0017.00=0F
R0018.00=0F  R0019.00=0F  R001A.00=0F  R001B.00=0F
R001C.00=0F  R001D.00=0F  R001E.00=0F  R001F.00=0F
R0020.00=0F  R0021.00=0F  R0022.00=0F  R0023.00=0F
R0024.00=0F  R0025.00=0F  R0026.00=0F  R0027.00=0F
R0028.00=0F  R0029.00=0F  R002A.00=0F  R002B.00=0F
R002C.00=0F  R002D.00=0F  R002E.00=0F  R002F.00=0F
```

The user may transfer the PROM data into memory and verify the transfer by setting the toggles to transfer PROM to memory and compare with the TS subcommand.

```
.PP,TS,0,0,2,1
.PP,GO
.
```

When the GO subcommand is entered, the PROM data is transferred to memory and each bit string is compared after it is loaded to verify that the correct data is transferred to memory.

When the data is in memory and is correct, it may be programmed into PROM by setting the toggles to program PROM and compare with the TS subcommand. The new PROM to be programmed should be inserted in the PROM programming module and the programming process initiated with the GO subcommand.

```
.PP,TS,0,0,1,1
.PP,GO
.
```

APPENDIX A

COMPATIBILITY WITH DX10

A program developed for the Prototyping System may be run under DX10 if several conventions are followed:

1. The first three words of the program should be:

DATA WP	Workspace
DATA START	Entry point
DATA END-ACTION	Address of point to branch to on an unrecoverable error

2. The program must be terminated with an end-of-program supervisor call.

3. An open supervisor call should be issued before a read or write to a file-oriented device.

4. All interrupts are handled by the DX10 operating system.

5. Absolute code, created by an AORG instruction, is loaded with the same load bias as relocatable code. Code at AORG 0 and RORG 0 are both loaded at the first location of the user's address space.

A more extensive explanation of these points can be found in the *Model 990 Computer DX10 Operating System Programmer's Guide*, Manual No. 945257-9701.

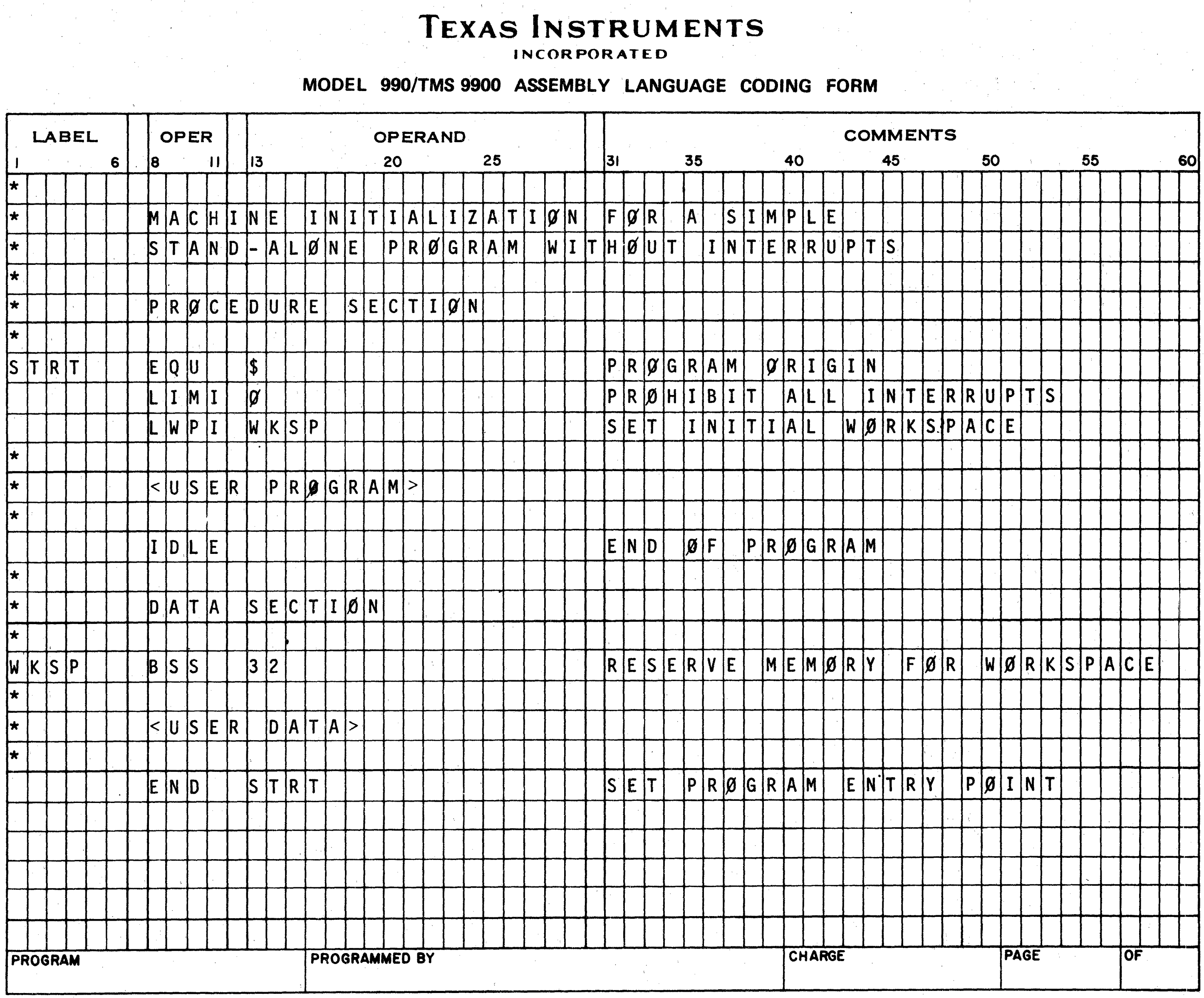
APPENDIX B

STAND-ALONE PROGRAMMING

To run a stand-alone program on the 990, the user must provide initialization procedures for the computer. Generally, these are the initialization of a workspace, the status register, and possibly the interrupt vectors.

The simplest case, shown in figure B-1, can be used for a program that will run without interrupts. Note that a power-up (level 0) interrupt may still occur and will not be handled. The first two instructions set the initial status and workspace pointer. The END statement causes the assembler to pass information to the loader about the starting location (STRT) of the program.

Figure B-2 is an example which initializes some interrupt vectors and supports five levels of interrupts. A routine, provided for the real time clock, counts the number of seconds since power-up. Note that if the routine is reused without reloading the program, the initialization should include resetting the seconds and individual clock interval counters in the workspace for the real time clock interrupt. Also note that the interrupt processor for memory errors resets the interrupt by communicating through the CRU.



TEXAS INSTRUMENTS
INCORPORATED

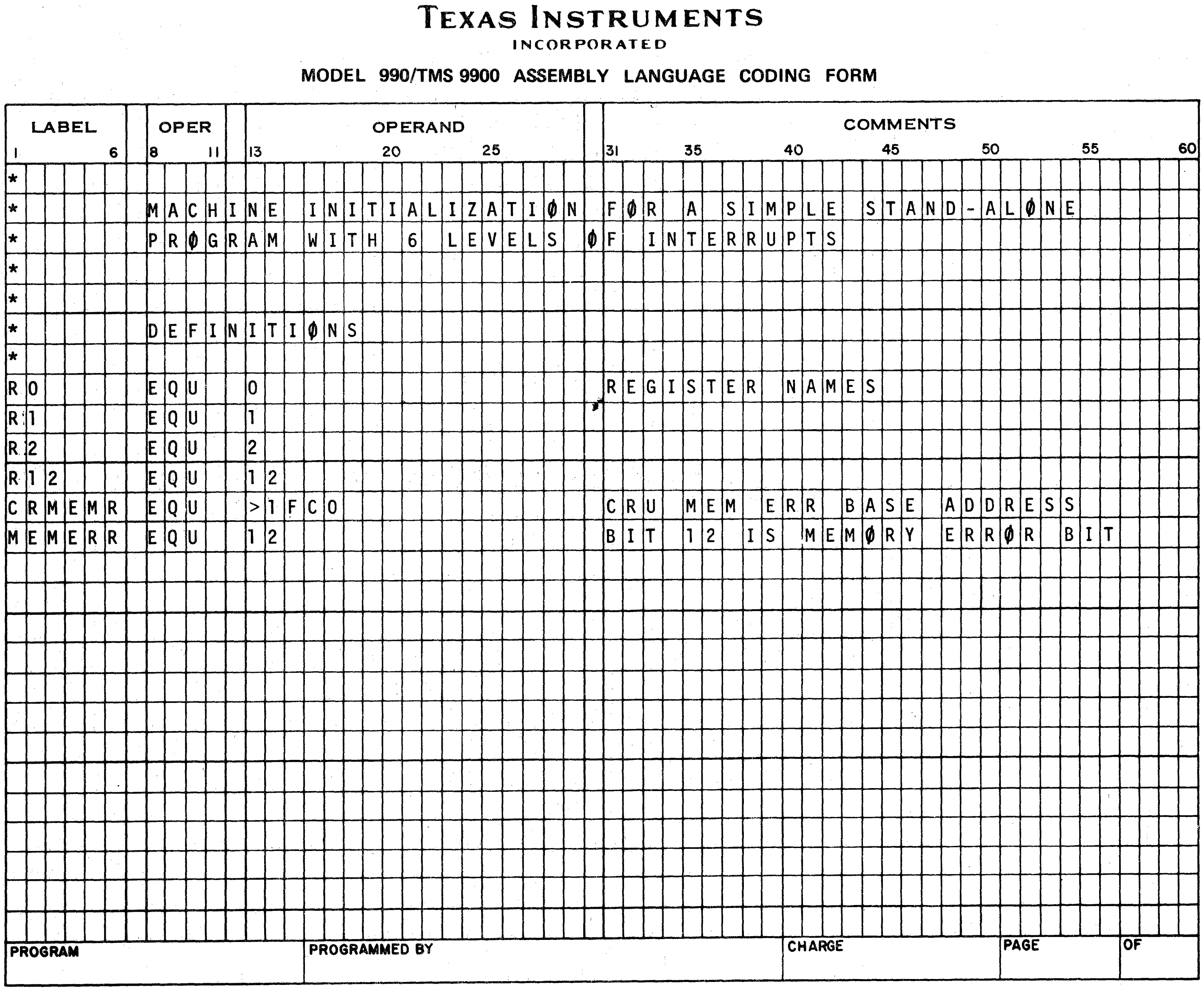
MODEL 990/TMS 9900 ASSEMBLY LANGUAGE CODING FORM

```
LABEL   OPER  OPERAND                 COMMENTS
1    6  8  11 13     20   25          31  35   40   45   50   55   60
*
*      MACHINE INITIALIZATIØN FØR A SIMPLE
*      STAND-ALØNE PRØGRAM WITHØUT INTERRUPTS
*
*      PRØCEDURE SECTIØN
*
STRT   EQU  $                 PRØGRAM ØRIGIN
       LIMI Ø                 PRØHIBIT ALL INTERRUPTS
       LWPI WKSP              SET INITIAL WØRKSPACE
*
*      <USER PRØGRAM>
*
       IDLE                   END ØF PRØGRAM
*
*      DATA SECTIØN
*
WKSP   BSS  32                RESERVE MEMØRY FØR WØRKSPACE
*
*      <USER DATA>
*
       END  STRT              SET PRØGRAM ENTRY PØINT
```

PROGRAM | PROGRAMMED BY | CHARGE | PAGE | OF

(A)133104

Figure B-1. Assembly Language Programming Example No.1



TEXAS INSTRUMENTS
INCORPORATED

MODEL 990/TMS 9900 ASSEMBLY LANGUAGE CODING FORM

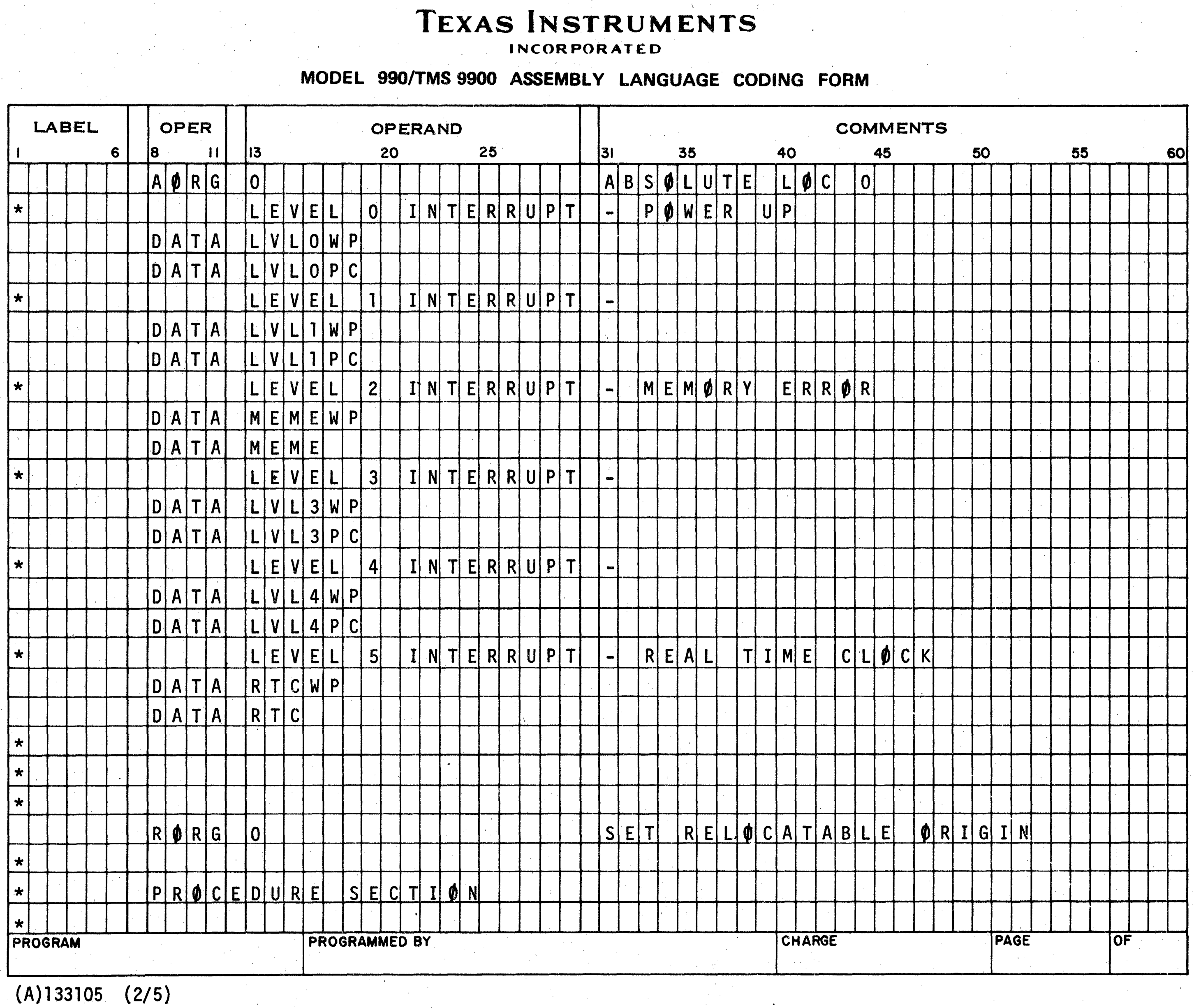
LABEL 1 6	OPER 8 11	OPERAND 13 20 25	COMMENTS 31 35 40 45 50 55 60

```
*
*      MACHINE INITIALIZATIØN FØR A SIMPLE STAND-ALØNE
*      PRØGRAM WITH 6 LEVELS ØF INTERRUPTS
*
*
*      DEFINITIØNS
*
R0     EQU  0                 REGISTER NAMES
R1     EQU  1
R2     EQU  2
R12    EQU  12
CRMEMR EQU  >1FC0             CRU MEM ERR BASE ADDRESS
MEMERR EQU  12                BIT 12 IS MEMØRY ERRØR BIT
```

PROGRAM	PROGRAMMED BY	CHARGE	PAGE	OF

(A)133105 (1/5)

Figure B-2. Assembly Language Programming Example No. 2 (Sheet 1 of 5)



TEXAS INSTRUMENTS
INCORPORATED

MODEL 990/TMS 9900 ASSEMBLY LANGUAGE CODING FORM

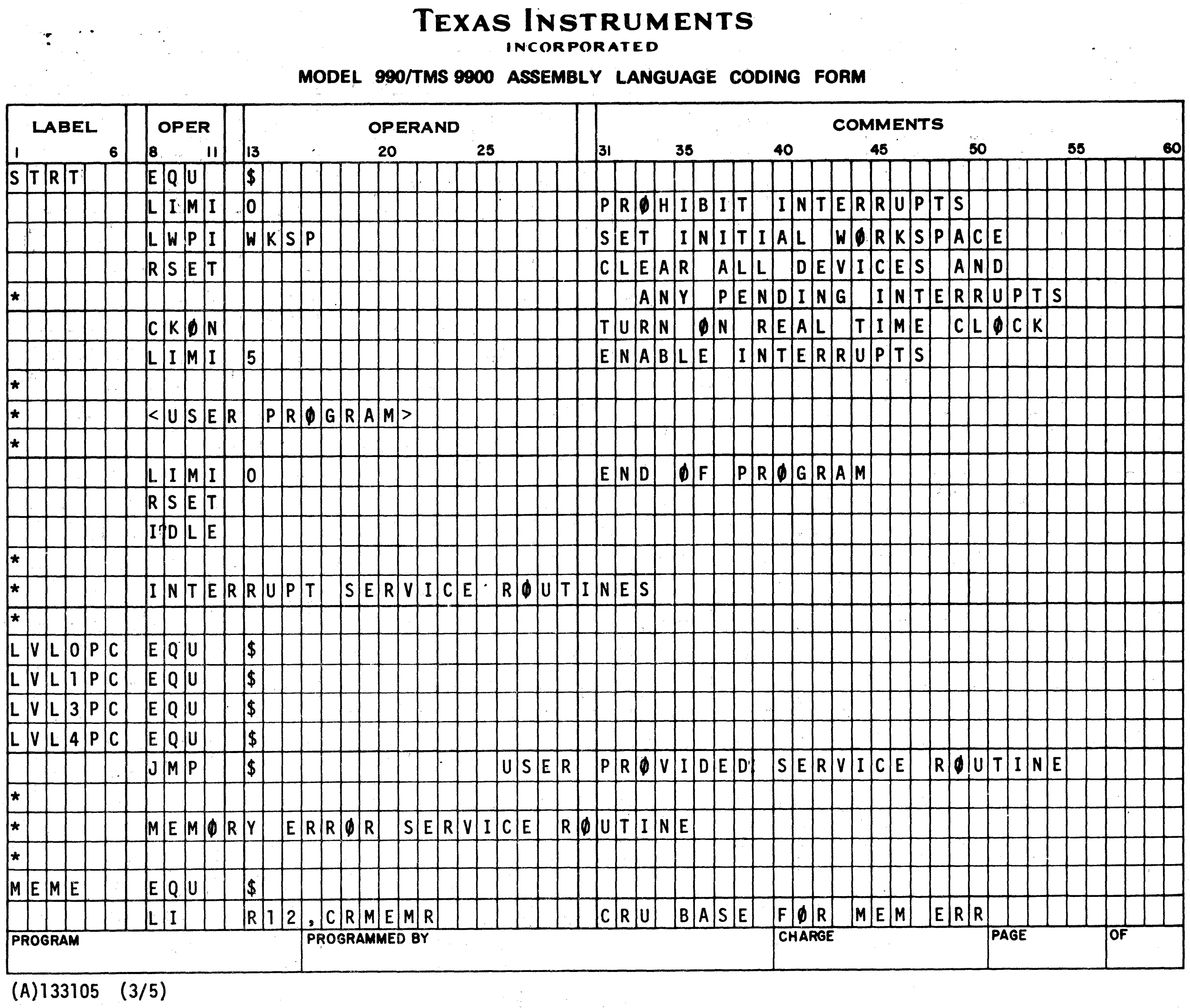
LABEL 1 6 | OPER 8 11 | OPERAND 13 20 25 | COMMENTS 31 35 40 45 50 55 60

```
       AØRG 0                 ABSØLUTE LØC 0
*           LEVEL 0 INTERRUPT - PØWER UP
       DATA LVL0WP
       DATA LVL0PC
*           LEVEL 1 INTERRUPT -
       DATA LVL1WP
       DATA LVL1PC
*           LEVEL 2 INTERRUPT - MEMØRY ERRØR
       DATA MEMEWP
       DATA MEME
*           LEVEL 3 INTERRUPT -
       DATA LVL3WP
       DATA LVL3PC
*           LEVEL 4 INTERRUPT -
       DATA LVL4WP
       DATA LVL4PC
*           LEVEL 5 INTERRUPT - REAL TIME CLØCK
       DATA RTCWP
       DATA RTC
*
*
*
       RØRG 0                 SET RELØCATABLE ØRIGIN
*
*      PRØCEDURE SECTIØN
*
```

PROGRAM	PROGRAMMED BY	CHARGE	PAGE	OF

(A)133105 (2/5)

Figure B-2. Assembly Language Programming Example No. 2 (Sheet 2 of 5)



TEXAS INSTRUMENTS
INCORPORATED

MODEL 990/TMS 9900 ASSEMBLY LANGUAGE CODING FORM

```
LABEL  OPER     OPERAND                     COMMENTS
1    6 8  11 13     20   25           31  35   40   45   50   55   60
STRT   EQU  $
       LIMI 0                 PRØHIBIT INTERRUPTS
       LWPI WKSP              SET INITIAL WØRKSPACE
       RSET                   CLEAR ALL DEVICES AND
*                               ANY PENDING INTERRUPTS
       CKØN                   TURN ØN REAL TIME CLØCK
       LIMI 5                 ENABLE INTERRUPTS
*
*      <USER PRØGRAM>
*
       LIMI 0                 END ØF PRØGRAM
       RSET
       IDLE
*
*      INTERRUPT SERVICE RØUTINES
*
LVL0PC EQU  $
LVL1PC EQU  $
LVL3PC EQU  $
LVL4PC EQU  $
       JMP  $            USER PRØVIDED SERVICE RØUTINE
*
*      MEMØRY ERRØR SERVICE RØUTINE
*
MEME   EQU  $
       LI   R12,CRMEMR        CRU BASE FØR MEM ERR
```

PROGRAM	PROGRAMMED BY	CHARGE	PAGE	OF

(A)133105 (3/5)

Figure B-2. Assembly Language Programming Example No. 2 (Sheet 3 of 5)

TEXAS INSTRUMENTS
INCORPORATED

MODEL 990/TMS 9900 ASSEMBLY LANGUAGE CODING FORM

```
LABEL  OPER  OPERAND                COMMENTS
1    6 8  11 13     20   25         31  35   40   45   50   55   60
       SBZ  MEMERR            CLEAR MEMØRY ERRØR
       JMP  $                 USER'S SERVICE RØUTINE
*
*      REAL-TIME CLØCK SERVICE RØUTINE
*
*           KEEP CØUNT ØF NUMBER ØF SECØNDS SINCE PØWER UP
RTC    EQU  $
       DEC  R1                CØUNT DØWN TICKS TILL SECØND
       JGT  RTCXIT            IF NØT A FULL SECØND
       INC  R0                CØUNT SECØND
       LI   R1,120            RESET TICK'S FØR NEXT SECØND
RTCXIT EQU  $
       CKØF                   CLEAR CLØCK INTERRUPT
       CKØN                   RE-ENABLE CLØCK
       RTWP                   END ØF INTERRUPT
*
*      DATA SECTIØN
*
WKSP   BSS  32                PRØGRAM WØRKSPACE
LVL0WP EQU  $
LVL1WP EQU  $
MEMEWP EQU  $
LVL3WP EQU  $
LVL4WP EQU  $
       BSS  32                WØRKSPACE FØR INTERRUPTS
```

PROGRAM	PROGRAMMED BY	CHARGE	PAGE	OF

(A)133105 (4/5)

Figure B-2. Assembly Language Programming Example No. 2 (Sheet 4 of 5)

TEXAS INSTRUMENTS
INCORPORATED

MODEL 990/TMS 9900 ASSEMBLY LANGUAGE CODING FORM

```
LABEL   OPER  OPERAND           COMMENTS
1    6  8  11 13     20   25    31  35   40   45   50   55   60
*
*      WØRKSPACE FØR REAL-TIME CLØCK INTERRUPT
*
RTCWP  DATA 0                 #  SECØNDS  (WP REG = R0)
       DATA 120               #  TICKS TØ NEXT SECØND
*                                         (WP REG = R1)
       BSS  28
*
*      <USER DATA>
*
       END  STRT
```

PROGRAM | PROGRAMMED BY | CHARGE | PAGE | OF

(A)133105 (5/5)

Figure B-2. Assembly Language Programming Example No. 2 (Sheet 5 of 5)

APPENDIX C

CHARACTER SET

C.1 ASSEMBLY LANGUAGE CHARACTERS

The Model 990 Assembly Language uses the ASCII characters listed in table C-1. The table includes the ASCII code for each character, represented as a hexadecimal value and as a decimal value. The table also shows the corresponding Hollerith code. In addition to the characters listed in table C-1, Model 990 Assembly Language defines six characters that are undefined in ASCII. Table C-2 lists these characters, hexadecimal and decimal representations, corresponding Hollerith codes, and the corresponding character on the Model 29 keypunch.

Table C-1. Character Set

Hexadecimal Value	Decimal Value	Character	Hollerith Code
20	32	Space	Blank
21	33	!	11-8-2
22	34	“	8-7
23	35	#	8-3
24	36	$	11-8-3
25	37	%	0-8-4
26	38	&	12
27	39	‘	8-5
28	40	(	12-8-5
29	41	)	11-8-5
2A	42	*	11-8-4
2B	43	+	12-8-6
2C	44	,	0-8-3
2D	45	-	11
2E	46	.	12-8-3
2F	47	/	0-1
30	48	0	0
31	49	1	1
32	50	2	2
33	51	3	3
34	52	4	4
35	53	5	5
36	54	6	6
37	55	7	7
38	56	8	8
39	57	9	9
3A	58	:	8-2
3B	59	;	11-8-6
3C	60	<	12-8-4
3D	61	=	8-6

Table C-1. Character Set (Continued)

Hexadecimal Value	Decimal Value	Character	Holerith Code
3E	62	>	0-8-6
3F	63	?	0-8-7
40	64	@	8-4
41	65	A	12-1
42	66	B	12-2
43	67	C	12-3
44	68	D	12-4
45	69	E	12-5
46	70	F	12-6
47	71	G	12-7
48	72	H	12-8
49	73	I	12-9
4A	74	J	11-1
4B	75	K	11-2
4C	76	L	11-3
4D	77	M	11-4
4E	78	N	11-5
4F	79	O	11-6
50	80	P	11-7
51	81	Q	11-8
52	82	R	11-9
53	83	S	0-2
54	84	T	0-3
55	85	U	0-4
56	86	V	0-5
57	87	W	0-6
58	88	X	0-7
59	89	Y	0-8
5A	90	Z	0-9

Table C-2. Additional Characters

Hexadecimal Value	Decimal Value	Character	Hollerith Code	Keypunch Character
5B	91	[	12-2-8	¢
5C	92	\	0-8-2	0-8-2
5D	93	]	12-7-8	| (vertical bar)
5E	94	∧	11-7-8	¬ (logical NOT)
5F	95	–	0-5-8	– (underscore)
00	00	Null		
09	09	Tab		

C.2 DATA TERMINAL CHARACTERS

The remainder of this appendix presents a detailed summary of the characters recognized by the 733 ASR Data Terminal in accordance with 990 file and record specifications. These include the data and control characters recognized by the terminal keyboard, printer, cassette receiving input data, and cassette sending output data. In each case, the ways in which the control characters function are described.

The character sets for each I/O function are diagrammed in figure C-1 to show the character corresponding to each ASCII code value and its function.

The ASCII control characters are shown in table C-3.

C.2.1 733 **ASR TERMINAL KEYBOARD INPUT**. Refer to figure C-1.

Peripheral device: 733 ASR terminal keyboard.

Physical organization: Character, record, file.

Record and file ending characters:

End of record: CR

End of file: DC3.

Character set: As shown. Except as indicated, all characters are automatically echoed as themselves.

Control character functions:

1. BS echoes as LF,BS and deletes the last character entered in the user's buffer (CTRL H).
2. DEL echoes as LF,CR and deletes the current input record.
3. HT causes a single space to be echoed. HT is placed in the user's buffer.
4. DC3 received as the first character of a record indicates end of file and terminates the input record. DC3 is not placed in the user's buffer.
5. CR echoes as CR and is not placed in the user's buffer. CR terminates the input record.
6. LF echoes as LF and is not placed in the user's buffer.
7. Characters in the range 20_{16} to $7E_{16}$ are echoed and placed in the user's buffer.
8. The most significant bit in each character is set to zero in the user's buffer.
9. ESC aborts current output and returns a write error to the user's program.
10. All other characters are ignored.

ASCII CHARACTER SET

BITS $b_4b_5b_6b_7$ (ROW NO.)	BITS $b_1b_2b_3$ (COLUMN NO.)							
	000 (0)	001 (1)	010 (2)	011 (3)	100 (4)	101 (5)	110 (6)	111 (7)
0000 (0)	NUL	DLE	SP	0	@	P	`	p
0001 (1)	SOH	DC1	!	1	A	Q	a	q
0010 (2)	STX	DC2	"	2	B	R	b	r
0011 (3)	ETX	DC3	#	3	C	S	c	s
0100 (4)	EOT	DC4	$	4	D	T	d	t
0101 (5)	ENQ	NAK	%	5	E	U	e	u
0110 (6)	ACK	SYN	&	6	F	V	f	v
0111 (7)	BEL	ETB	'	7	G	W	g	w
1000 (8)	BS	CAN	(	8	H	X	h	x
1001 (9)	HT	EM	)	9	I	Y	i	y
1010 (10)	LF	SUB	*	:	J	Z	j	z
1011 (11)	VT	ESC	+	;	K	[	k	{
1100 (12)	FF	FS	,	<	L	\	l	¦
1101 (13)	CR	GS	−	=	M	]	m	}
1110 (14)	SO	RS	.	>	N	^	n	~
1111 (15)	SI	US	/	?	O	_	o	DEL

CHARACTERS IN BOXES ENCLOSED IN HEAVY LINES HAVE THE FUNCTIONS INDICATED BELOW. CHARACTERS IN SHADED BOXES ARE IGNORED.

CONTROL CHARACTER FUNCTIONS

CONTROL CHARACTER	KEYBOARD INPUT	PRINTER OUTPUT	CASSETTE INPUT	CASSETTE OUTPUT
BEL	−	X	X	X
BS	X	X	X	X
HT	X	X	X	X
LF	X	X	X	X
FF	−	X	X	X
CR	−	X	−	X
DC3	−	−	X	X
ETB	−	−	X	X
ESC	X	−	−	−
DEL	X	I	I	I

X − CHARACTERS WITH SPECIAL FUNCTIONS.

I − INPUT OR OUTPUT AS IS.

OTHER CHARACTERS ARE IGNORED.

(A)133111

Figure C-1. 733 ASR Terminal Character Set

Table C-3. ASCII Control Characters

Control Character	Description
ACK	Acknowledge
BEL	Bell
BS	Backspace
CAN	Cancel
CR	Carriage return
DC1 = X-ON	Device control 1
DC2 = TAPE	Device control 2
DC3 = X-OFF	Device control 3
DC4 = TAPE	Device control 4 (stop)
DEL* = RUB OUT	Delete
DLE	Data link escape
EM	End of medium
ENQ = WRU	Inquiry
EOT	End of transmission
ESC	Escape
ETB	End of transmission block
ETX	End of text
FF	Form feed
FS	File separator
GS	Group separator
HT	Horizontal tabulation
LF	Line feed
NAK	Negative acknowledge
NUL	Null
RS	Record separator
SI	Shift in
SO	Shift out
SOH	Start of heading
STX	Start of text
SUB	Substitute
SYN	Synchronous idle
US	Unit separator
VT	Vertical tabulation

*Not strictly a control character

C.2.2 733 **ASR TERMINAL PRINTER.** Refer to figure C-1.

Peripheral device: 733 ASR terminal printer.

Physical organization: Record, file.

Record and file ending characters:

End of record: Depletion of character count.

End of file: Not applicable.

Character set: As shown.

Control character functions:

1. HT prints as a space.
2. FF prints as eight LFs.
3. CR prints as CR.
4. LF prints as LF.
5. Characters in the range 20_{16} to $7E_{16}$ are printed as is.
6. BEL is output as BEL.
7. BS is output as BS.
8. All other characters are ignored.

C.2.3 733 ASR TERMINAL CASSETTE INPUT. Refer to figure C-1.

Peripheral device: 733 ASR terminal cassette input (ASCII, direct).

Physical organization: Record, file.

Record and file ending characters:

End of record: CR.

End of file: DC3.

Character set: As shown.

Control character functions:

1. HT and FF as well as characters in the range 20_{16} to $7E_{16}$ are stored in the user's buffer.
2. ETB is translated to CR and stored in the user's buffer.
3. CR indicates end of record. CR is not placed in the user's buffer.
4. DC3 received as the first valid character of a record indicates end of file. When DC3 is read, the block is restarted by performing a block forward. End of file status is returned after completion of the block. DC3 is not placed in the user's buffer.
5. BEL and BS are input unchanged.
6. The sequences LF, DEL or DEL at the beginning of a record are ignored if present. The first character following such a sequence is considered the first valid character in the record.

7. In direct mode, the contents of a physical block on tape are transferred to the user's buffer without conversion. Parity bits are reset.

C.2.4 733 ASR TERMINAL CASSETTE OUTPUT. Refer to figure C-1.

Peripheral device: 733 ASR terminal cassette output (ASCII, direct).

Physical organization: Record, file.

Record and file ending characters:

End of record: Depletion of character count (83 characters maximum).

End of file: DC3.

Character set: As shown.

Control character functions:

1. HT, FF and characters in the range 20_{16} to $7E_{16}$ are output as is.

2. CR in the user's buffer is translated to ETB and output.

3. End of block character sequence is CR, LF, DC4, DEL. These characters are automatically output to control the cassette and are not user data characters.

4. BEL and BS are output unchanged.

5. End of file character sequence is DC3, CR, DC4, DEL.

6. DC3 is allowed within a record for compatibility with stand-alone software. It may not be written, however, as the first data character in the record.

APPENDIX D

COMMAND AND DIRECTIVE SUMMARY

D.1 GENERAL

This appendix contains summaries of the commands, directives and pseudo-instructions available to the system user. They include the following:

- Monitor keyboard commands
- Text editor commands
- Assembler directives
- Assembler pseudo-instructions

The assembly language machine instructions are summarized in the *Model 990 Computer TMS9900 Microprocessor Assembly Language Programmer's Guide*, Manual No. 943441-9701.

D.2 MONITOR KEYBOARD COMMANDS

Table D-1 lists the monitor keyboard commands with a brief description of the purpose of each command, the syntax, and a paragraph reference to a detailed discussion of the command. The syntax is presented in abbreviated form. The separator between parameters – a blank or comma – is not shown, and the user must remember that distinct separators must be included to indicate the position of omitted parameters.

The parameters used in table D-1 are explained in the following list:

bias	Base memory address for relocatable code
bit quant	Number of bits to be changed
char string	Character string describing trace options
CRU addr	CRU word address
CRU end addr	Ending CRU address
CRU start addr	Starting CRU address
device	Name of I/O device – LOG, DUM, CS1 or CS2
end addr	Ending memory address
ending index no.	Index number of ending element (breakpoint, snapshot, or trace region)
ending reg	Register number of ending workspace register

entry point	Entry point of program
format index	Trace format index number
index no.	Index number of breakpoint, snapshot, or trace region
instr count	Maximum number of instructions to be executed
luno	Logical unit number of I/O device
mask value	Hexadecimal number value to be ANDed with another value
mem addr	Memory address
P	Indicator specifying that end-of-module tag character and end-of-file marker will not be written on tape
program name	Name of program – alphanumeric character string
ref cnt	Reference count – pass number on which a breakpoint is taken
search value	Hexadecimal word or byte for which a search is made
snapshot no.	Number of a previously defined snapshot
start addr	Starting memory address
starting index no.	Index number of starting element (breakpoint, snapshot, or trace region)
starting reg	Register number of starting workspace register
step control	Indicator that specifies single instruction execution or continuous execution
value	Hexadecimal number value
var	Variable address to be traced

For all optional parameters, default values are provided.

Table D-1. Monitor Keyboard Commands

Mnemonic Name	Description and Function	Syntax	Paragraph
AL	Assign LUNO. Assigns the LUNO to the specified device for subsequent I/O.	AL <luno> <device>	3.4.2
CB	Clear Breakpoint. Specifies a series of breakpoints to be disabled.	CB [<starting index no.>] [<ending index no.>]	3.4.24
CP	Clear Write Protect Region. Clears the protect register and removes protection from the write-protected region.	CP	3.4.32
CR	Clear Trace Region. Disables the specified regions. Execution of code within the region is with the hardware SIE instead of the software interpreter.	CR [<starting index no.>] [<ending index no.>]	3.4.27
CS	Clear Snapshot. Disables the display of the specified snapshots.	CS [<starting index no.>] [<ending index no.>]	3.4.22
DP	Dump in Absolute Format. Dumps specified memory to LUNO 7 in absolute format.	DP <start addr> <end addr> [<entry point>] [<program name>] [P]	3.4.7
EX	Execute User Program Directly. Transfers control directly to the user's program with PC, WP and ST registers as displayed by the IR command.	EX	3.4.10
FB	Find Byte. Scans memory under mask to find occurrences of the specified value.	FB [<start addr>] [<end addr>] <search value> [<mask value>]	3.4.28
FW	Find Word. Scans memory under mask to find occurrences of the specified value.	FW [<start addr>] [<end addr>] <search value> [<mask value>]	3.4.29
HA	Hexadecimal Arithmetic. Displays sum and difference of two hexadecimal numbers. The display is in both hexadecimal and decimal, as a 2's complement number. Arithmetic is modulo 2^{16}.	HA [<value>] [<value>]	3.4.30
IC	Inspect CRU Input Lines. Displays the specified range of the CRU input lines on the printer.	IC [<CRU start addr>] [<CRU end addr>]	3.4.19
IM	Inspect Memory. Dumps the specified memory range to the printer.	IM [<start addr>] [<end addr>]	3.4.13
IR	Inspect Registers. Displays the current contents of the user's PC, WP and ST registers on the printer.	IR	3.4.15
IS	Inspect Snapshot. Dumps the registers and memory range associated with the specified snapshot to the data terminal. If the snapshot has not been defined, no action is taken.	IS [<starting index no.>] [<ending index no.>]	3.4.21
IW	Inspect Workspace Registers. Displays the specified workspace registers on the printer. The workspace is that given in the WP register by the IR command.	IW [<starting reg>] [<ending reg>]	3.4.17

Table D-1. Monitor Keyboard Commands (Continued)

Mnemonic Name	Description and Function	Syntax	Paragraph
LA	Load Program in Compressed Absolute Format. Loads the memory data sequence dumped in compressed absolute format. Requires the Absolute Dump/Absolute Load overlay.	LA [<luno>]	3.4.9
LL	Link and Load Program. Links user program modules and loads them into memory. Requires Link and Load overlay.	LL	3.4.6
LP	Load Program. Loads program from specified device into memory and performs any necessary relocation.	LP [<luno>] [<bias>]	3.4.3
LU	Load Program in Compressed Absolute Format with Upfront Loader. Loads the absolute memory image with the upfront loader.	LU [<luno>] [<bias>]	3.4.8
MC	Modify CRU Register. Displays the contents of the specified CRU input lines and accepts data to change the corresponding output lines. All data is right-justified in a 16-bit field.	MC [<CRU addr>] [<bit quant>]	3.4.18
MM	Modify Memory. Displays the contents of the specified memory location and accepts an input to change it.	MM [<mem addr>]	3.4.12
MR	Modify Registers. Displays the contents of the user's PC, WP and ST registers and accepts an input value to change each register.	MR	3.4.14
MW	Modify Workspace Registers. Displays the specified register of the workspace displayed in the IR command and accepts an input value to be used to change it.	MW [<starting reg>]	3.4.16
OV	Load Overlay. (1) Disables commands currently in transient area. (2) Loads overlay into transient area. (3) Enables new commands of overlay in the transient area.	OV [<luno>]	3.4.4
PL	Load PROM Programmer. Loads PROM Programmer software module into memory.	PL <luno> <bias>	3.4.5
RU	Execute User Program under SIE or Trace. Requires the Instruction Trace overlay for trace.	RU [<instr count>]	3.4.11
SB	Set Breakpoint. Sets a software breakpoint at the specified location for use with the RU command. Breakpoints occur before instruction execution.	SB <index no.> <mem addr> [<ref cnt>] [<snapshot no.>]	3.4.23

Table D-1. Monitor Keyboard Commands (Continued)

Mnemonic Name	Description and Function	Syntax	Paragraph
SP	Set Write Protect Region. Sets the write protected region to the specified address bounds.	SP <start addr> <end addr>	3.4.31
SR	Set Trace Region. Defines a memory region to be executed with the software interpreter under the RU command. Requires the Instruction Trace overlay.	SR <index no.> <start addr> <end addr> <format index> [<step control>] [<var>] [<var>] [<var>]	3.4.26
SS	Set Snapshot. Defines a display of registers and memory which may be displayed in response to a breakpoint or an IS command.	SS <index no.> [<starting reg>] [<ending reg>] [<start addr>] [<end addr>]	3.4.20
ST	Set Trace Definition. Specifies items to be displayed by the trace interpreter. Requires the Instruction Trace overlay.	ST <format index> <char string>	3.4.25

The following symbols and conventions are used in defining the syntax of the monitor keyboard commands:

- Angle brackets (< >) enclose items supplied by the user.
- Brackets ([ ]) enclose optional items.

D.3 TEXT EDITOR COMMANDS

The text editor commands are summarized in table D-2. The syntax of each command and a paragraph reference to a detailed discussion of the command are shown.

The following symbols and conventions are used in defining the syntax of text editor commands:

- Angle brackets (< >) enclose items supplied by the user.
- Brackets ([ ]) enclose optional items.
- Braces ({ }) enclose two or more items of which one must be chosen.
- Items in capital letters and punctuation marks must be entered as shown.

The syntax definitions and examples do not show spaces between the characters of the two-character commands, between the command and operands, or between operands. Spaces may be entered at these points if desired.

D.4 ASSEMBLER DIRECTIVES

The assembler directives for the Model 990 Assembly Language are listed in table D-3. All directives may include a comment field following the operand field. Those directives that do not require an operand field may have a comment field following the operator field. Those directives that have optional operand fields (RORG and END) may have comment fields only when they have operand fields.

The following symbols and conventions are used in defining the syntax of assembler directives:

- Angle brackets (< >) enclose items supplied by the user.
- Brackets ([ ]) enclose optional items.
- An ellipsis (. . .) indicates that the preceding item may be repeated.
- Braces ({ }) enclose two or more items of which one must be chosen.

The following words are used in defining the items used in assembler directives:

- symbol - a symbol
- label - a symbol used in the label field
- string - a character string of a length defined for each directive
- expr - an expression
- wd expr - well-defined expression

Table D-2. Text Editor Commands

Name	Use	Syntax	Paragraph
SL	To restore printing of line numbers	SL	4.5.2.1
SN	To omit line numbers from command printouts	SN	4.5.2.2
SP	To set right margin for command printouts	SP <s>	4.5.2.3
SM	To set left and right limits for scan of F command	SM <s>, <t>	4.5.2.4
D	To move pointer down, and read additional lines when required	D [<n>]	4.5.3.1
U	To move pointer up	U [<n>]	4.5.3.2
T	To move pointer to top of buffer	T	4.5.3.3
B	To move pointer to bottom of buffer	B	4.5.3.4
C	To delete lines and enter lines at that point in buffer	$C\left\{\begin{array}{l}<s>-<t>\\ {[+]}\ [<n>]\\ -<n>\end{array}\right\}$	4.5.4.1
I	To insert lines in buffer	I [<k>]	4.5.4.2
M	To move a block of lines to a specified point in buffer	$M\left\{\begin{array}{l}<s>-<t>,\ [<r>]\\ {[+]}\ <n>,<r>\\ -<n>,<r>\end{array}\right\}$	4.5.4.3
R	To remove a block of lines from the buffer	$R\left\{\begin{array}{l}<s>-<t>\\ {[+]}\ [<n>]\\ -<n>\end{array}\right\}$	4.5.4.4
F	To scan a block of lines from the buffer to locate lines having a specified character string	$F\left\{\begin{array}{l}<s>-<t>\\ {[+]}<n>\\ -<n>\end{array}\right\}\left\{\begin{array}{l}L\\ F\end{array}\right\}<d1><string1><d1>\left\{\begin{array}{c}[P]\\ <d2>[<string2>]<d2>[V]\ [P]\end{array}\right\}$	4.5.4.5
L	To identify the first and last lines in the buffer	L	4.5.5.1
P	To print specified lines from the buffer	$P\left\{\begin{array}{l}<s>-<t>\\ {[+]}\ [<n>]\\ -<n>\end{array}\right\}$	4.5.5.2
K	To write lines from the buffer on the output device	K [<n>]	4.5.6.1
Q	To write, or complete, the output file	Q [<s>]	4.5.6.2
E	To terminate execution without completing the output file.	E	4.5.6.3

Table D-3. Assembler Directives

Directive	Syntax	Force Word Boundary	Note
Page Title	[<label>] TITL <string>	NA	
Program Identifier	[<label>] IDT <string>	NA	
External Definition	[<label>] DEF <symbol>[,<symbol>] . . .	NA	
External Reference	[<label>] REF <symbol>[,<symbol>] . . .	NA	
Absolute Origin	[<label>] AORG <wd expr>	No	
Relocatable Origin	[<label>] RORG [<expr>]	No	1, 3
Dummy Origin	[<label>] DORG <wd expr>	No	
Block Starting with Symbol	[<label>] BSS <wd expr>	No	
Block Ending with Symbol	[<label>] BES <wd expr>	No	
Initialize Word	[<label>] DATA <expr>[,<expr>] . . .	Yes	
Initialize Text	[<label>] TEXT [-] <string>	No	2
Define Extended Operation	[<label>] DXOP <symbol>,<term>	NA	
Define Assembly-Time Constant	<label> EQU <expr>	NA	3
Word Boundary	[<label>] EVEN	Yes	
No Source List	[<label>] UNL	NA	
List Source	[<label>] LIST	NA	
Page Eject	[<label>] PAGE	NA	
Initialize Byte	[<label>] BYTE <wd expr> [,<wd expr>]...	No	
Program End	[<label>] END [<symbol>]	NA	4

NOTES

1. The expression must be relocatable.
2. The minus sign causes the assembler to negate the rightmost character.
3. Symbols in expressions must have been previously defined.
4. Symbol must have been previously defined.
5. Keywords are XREF, OBJ, SYMT, NOLIST, and TEXT.

- term - a term
- operation - mnemonic operation code, macro name, or previously defined operation or extended operation

D.5 ASSEMBLER PSEUDO-INSTRUCTIONS

Model 990 Assembly Language pseudo-instructions are listed in table D-4. The pseudo-instructions, which have no operand fields, have optional comment fields. The symbols and conventions are the same as in the assembler directive syntax.

Table D-4. Assembler Pseudo-Instructions

Pseudo-Instruction	Syntax	Hexadecimal Operation Code
No Operation	[<label>] NOP	1000
Return	[<label>] RT	045B

D.6 PROGRAMMER AND MEMORY DUMP COMMANDS

Table D-5 lists the PROM programmer and memory dump commands and subcommands with a brief description of the purpose of each command or subcommand, the syntax, and a paragraph reference to a detailed discussion of the command or subcommand. The syntax is presented in abbreviated form. The separator between parameters – a blank or comma – is not shown, and the user must remember that distinct separators must be included to indicate the position of omitted parameters.

The parameters used in table D-5 are explained in the following list:

base addr — CRU base address for the PROM programming module interface card chassis slot

bit — Bit position of the starting bit of a memory or PROM/ROM bit string

char string 1 — Name of first record of PROM or memory control information

char string 2 — Name of second record of PROM or memory control information

compare — Value that specifies whether a bit string comparison is to be made

dmn — Initial bit displacement that determines the starting address in a memory data configuration

drn — Initial bit displacement that determines the starting address in a PROM/ROM data configuration

duty cycle — Percentage of the time that the programming pulse is on when programming a PROM

end addr — Address of the last word in the memory block, or the address of the last byte to be dumped

high or low — Value that specifies either high or low logic level output conditions

imn	Bit increment that determines bit string addresses in a memory data configuration
irn	Bit increment that determines bit string addresses in a PROM/ROM data configuration
level n	Memory or PROM/ROM mapping level
lower bound	Address of the first byte or word in a memory or PROM/ROM block
mem disp	Value that specifies whether memory bit strings and addresses are to be displayed
mmn	Number of bit strings used in the programming cycle in a memory data configuration
mrn	Number of bit strings used in the programming cycle in a PROM/ROM data configuration
pgmable bits	Number of bits that can be programmed simultaneously
prom disp	Value that specifies whether PROM or ROM bit strings and addresses are to be displayed
pwl	Pulse width used for PROM programming
retries	Number of times programming is to be retried
start addr	Address of the first word in the memory block, or the address of the first byte to be dumped
subcommand	Subcommand that follows a command
transfer	Value that specifies the data transfer option
upper bound	Address of the last byte or word in a memory or PROM/ROM block
width	Number of bits per word, or number of bits per bit string

Table D-5. PROM Programmer and Memory Dump Commands and Subcommands

Mnemonic Name	Description and Function	Syntax	Paragraph
C	Compare BNPF Format on Cassette to Memory. Verifies that the correct data has been written on tape.	DB C	8.3.2.2
C	Compare HIGH/LOW Format on Cassette to Memory. Verifies that the correct data has been written on tape.	HL C <start addr> <end addr> [<bit>]	9.3.2.2
CS	Set CRU Interface Base Address. Defines the PROM Programmer Module CRU base address.	PP CS <base addr>	7.5.3.3
D	Dump Memory to Cassette in BNPF Format. Converts memory data to BNPF format and writes it to tape.	DB D <start addr> <end addr>	8.3.2.1
D	Dump in HIGH/LOW Format. Converts memory data to HIGH/LOW format and writes it to tape.	HL D <start addr> <end addr> [<bit>]	9.3.2.1
DB	Perform BNPF Operation. Causes a BNPF dump, load or data comparison.	DB <subcommand>	8.3.1
GO	Go. Initiates the programming cycle.	PP GO	7.5.3.5
HL	Perform HIGH/LOW Operation. Causes a HIGH/LOW dump or data comparison.	HL <subcommand>	9.3.1
L	Load BNPF-Formatted Data Module in Memory. Reads a BNPF-formatted data module, converts the data to hexadecimal, and stores the data in memory.	DB L	8.3.2.3
MB	Define Memory Bounds. Specifies the lower and upper address bounds of programming data in memory.	PP MB <lower bound> <upper bound>	7.5.3.1
MI	Define Memory Data Configuration Mapping Parameters. Defines the control information used to determine the addresses of bit strings.	PP MI <level n> [<imn>] [<mmn>] [<dmn>]	7.5.3.6
PP	PROM Programmer. Controls the PROM programming process.	PP <subcommand>	7.5.2
PS	PROM Programmer Standard. Searches the Standard Control Information Cassette for the specified records.	PS <char string 1> [<char string 2>]	7.5.1
RB	Define PROM/ROM Bounds. Specifies the lower and upper address bounds of programming data in ROM or PROM.	PP RB <lower bound> <upper bound>	7.5.3.2
RC	Define PROM/ROM Characteristics. Define physical hardware characteristics needed for data transfer.	PP RC <width> <high or low> <pwl> [<retries>] [<duty cycle>] [<pgmable bits>]	7.5.3.8

Table D-5. PROM Programmer and Memory Dump Commands and Subcommands (Continued)

Mnemonic Name	Description and Function	Syntax	Paragraph
RI	Define PROM/ROM Data Configuration Mapping Parameters. Defines control information needed to determine the addresses of bit strings.	PP RI <level n> [<irn>] [<mrn>] [<drn>]	7.5.3.7
SW	Define String Width. Define the bit string widths.	PP SW <width>	7.5.3.9
TS	Set Toggles. Sets numeric parameters that specify actions to be taken.	PP TS [<mem disp>] [<prom disp>] [<transfer>] [<compare>]	7.5.3.4

APPENDIX E

ERROR MESSAGES

MX01	Unrecoverable I/O error
MX02	Invalid parameter in Assign LUNO command
MX03	Command not resident in the transient area
MX04	Attempt to execute in trace mode when trace not resident
MX06	Invalid memory address or instruction
MS01	Invalid command
MS05	Required parameter missing
MP00	Parameter specification error
DP00	Invalid hexadecimal number input
DP03	Parameter value is greater than the allowed maximum
DP04	Snapshot is already defined
DP10	Invalid trace region index
DP12	CRU bit width parameter invalid
DP13	Invalid range of registers or memory addresses
DP20	Breakpoint specification error
DP23	Syntax error in trace format character string
DP26	Invalid trace format index number
LD00	Invalid tag or I/O error
LD01	Invalid load LUNO
LL01	Invalid load sequence
LL02	Invalid load code
LL03	Missing end statement

LL04	Load address error
LL05	Previous load module error
LL06	Checksum error - retry
PP01	Required parameter missing
PP02	Value out of range
PP03	Values required to match do not match
PP04	Bad address or record not found
PP05	Hardware malfunction
PP06	PROM Programming Module off-line

APPENDIX F

MEMORY AND PROM MAPPING

Bit strings are fetched from and stored into the 990 memory and PROM under control of memory and PROM mapping parameters. These parameters are used to evaluate a mathematical expression which determines the beginning bit address of a bit string.

The mapping parameters are:

IM_1, IM_2, IM_3 IR_1, IR_2, IR_3 — The increment values associated with each term of the polynomial (in bits)

MM_1, MM_2, MM_3 MR_1, MR_2, MR_3 — The maximum multiplier for the increment for each term of the polynomial

DM_1, DM_2, DM_3 DR_1, DR_2, DR_3 — The initial displacement associated with each term of the polynomial (in bits)

BMA,BCA — Beginning memory (byte) address and beginning chip (physical word) address

RWW — PROM/ROM physical word width

Note that the condition $\prod_{i=1}^{3} MM_i = \prod_{i=1}^{3} MR_i$ must be met, namely

that the algorithm will map an identical number of bit strings in memory as it will in PROM/ROM.

Let $n = \prod_{i=1}^{3} MM_i - 1 = \prod_{i=1}^{3} MR_i - 1.$

Let $K = 0,1,2 \ldots ,n$

Then compute

$$CM_1 = k \text{ modulo } MM_1 \qquad CR_1 = k \text{ modulo } MR_1$$

$$CM_2 = \text{int } (\frac{K}{MM_1}) \qquad CR_2 = \text{int } (\frac{K}{MR_1})$$

$$CM_3 = \text{int } (\frac{K}{MM_1 \cdot MM_2}) \qquad CR = \text{int } (\frac{K}{MR_1 \cdot MR_2})$$

Then beginning memory bit address of string

$$BMBA = 8 \cdot BMA + \sum_{i=1}^{3} (DM_i + CM_i \cdot IM_i)$$

and beginning PROM/ROM bit address of string

$$BRBA = RWW \cdot BCA + \sum_{i=1}^{3} (DR_i + CR_i \cdot IR_i)$$

This algorithm may be expressed in FORTRAN in two different ways. The first encodes the algorithm directly.

```
      IMPLICIT  INTEGER  (A-Z)
      N = MM1 * MM2 * MM3
C
C                  GENERATE  ALL  BEGINNING  BIT  ADDRESSES
C
      DO      IO    KK  = 1,N
              K  =  KK-1
              CMI  =  MOD (K,MM1)
              CM2  =  K/MM1
              CM3  =  K/MM1  *  MM2)
              BMBA  =  8  *  BMA  +  DM1  + CM1  *  IM1  +  DM2  +  CM2  *  IM2  +  DM3  +  CM3
                          *  IM3
10            CONTINUE
```

The second method utilizes nested DO-loops to avoid the calculations of CM1, CM2, and CM3.

```
              IMPLICIT INTEGER (A-Z)
C
C                  GENERATE ALL BEGINNING BIT ADDRESSES
C
      DO      30   I = 1, MM3
              CM3 = I - 1
              DO 20    J = 1, MM2
                CM2 = J - 1
                DO   10   L - 1, MM1
                     CM1 = L - 1
                     BMBA = 8 * BMA + DM1 + CM1 * IM1 + DM2 + CM2 * IM2 + DM3
                            + CM3 * IM3
10                     CONTINUE
20                 CONTINUE
30            CONTINUE
```

A similar mechanism would generate all beginning ROM bit addresses.

APPENDIX G

ADDITIONAL USER TABLES

Additional information related to PROM programming is presented in tables G-1 through G-5.

Table G-1. Pulse Widths

Number	Multiplier	Pulse Width (ms)
1	2	0.5
2	4	1.0
3	8	2.0
4	16	4.0
5	32	8.0
6	64	16.0

The number (x) is used as an exponent to get the multiplier, which is 2^X. Hardware uses the multiplier to produce the corresponding pulse width.

Table G-2. Minimum, Standard and Maximum Pulse Widths and Duty Cycles

PROM Types	Pulse Width (ms)			Duty Cycle		
	Minimum	Standard	Maximum	Minimum	Standard	Maximum
TTL						
188A, S188, S288, S287, S387, S470, S471, S472, S473	1	2	20	–––	25%	35%
EPROMs						
2704, 2708	0.1	0.1	1	–––	50%	50%

Note: TTL PROM types have the prefix SN74.

Table G-3. Memory Configurations

PROM Type	Bit String Width	Length (Words)	Initial Bit	Configuration in Consecutive Strings Across (A) or Down (D) Memory
MS287-0	4	256	0	D
MS287-4	4	256	4	D
MS287-8	4	256	8	D
MS287-C	4	256	C	D
MS287A	4	64	0	A
MS288-0	8	32	0	D
MS288-8	8	32	8	D
MS288A	8	16	0	A
MS471-0	8	256	0	D
MS471-8	8	256	8	D
MS471A	8	128	0	A
MS472-0	8	512	0	D
MS472-8	8	512	8	D
MS472A	8	256	0	A
ME2704-0	8	512	0	D
ME2704-8	8	512	8	D
ME2704A	8	256	0	A
ME2708-0	8	1024	0	D
ME2708-8	8	1024	8	D
ME2708A	8	512	0	A

Note: TTL PROM types have the prefix SN74.

Table G-4. PROM Configurations

PROM Type	PROM Word Width	Length (Words)
S288	8	32
S287	4	256
S471	8	256
S472	8	512
E2704	8	512
E2708	8	1024

Note: TTL PROM types have the prefix SN74.

Table G-5. Standard Control Information Cassette Data Configurations

Memory (M) or ROM (R)	PROM Type	Bit String Width	Mapping Parameters									PROM Characteristics					Programmable String Width
			Increments (Hexadecimal)			Displacements (Hexadecimal)			Maxima (Hexadecimal)			PROM Word Width	Program 0's or 1's	Pulse Width	Retries	Duty Cycle (Hexadecimal)	
			L1	L2	L3	L1	L2	L3	L1	L2	L3						
M	MS288-0	8	10	0	0	0	0	0	20	0	0	–	–	–	–	–	–
M	MS288-8	8	10	0	0	8	0	0	20	0	0	–	–	–	–	–	–
M	MS288A	8	8	0	0	0	0	0	20	0	0	–	–	–	–	–	–
R	S288	8	8	0	0	0	0	0	20	0	0	8	1	2	0	19	1
M	MS287-0	4	10	0	0	0	0	0	100	0	0	–	–	–	–	–	–
M	MS287-4	4	10	0	0	4	0	0	100	0	0	–	–	–	–	–	–
M	MS287-8	4	10	0	0	8	0	0	100	0	0	–	–	–	–	–	–
M	MS287-C	4	10	0	0	C	0	0	100	0	0	–	–	–	–	–	–
M	MS287A	4	4	0	0	0	0	0	100	0	0	–	–	–	–	–	–
R	S287	4	4	0	0	0	0	0	100	0	0	4	0	2	0	19	1
M	MS471-0	8	10	0	0	0	0	0	100	0	0	–	–	–	–	–	–
M	MS471-8	8	10	0	0	8	0	0	100	0	0	–	–	–	–	–	–
M	MS471A	8	8	0	0	0	0	0	100	0	0	–	–	–	–	–	–
R	S471	8	8	0	0	0	0	0	100	0	0	8	1	2	0	19	1
M	MS472-0	8	10	0	0	0	0	0	200	0	0	–	–	–	–	–	–
M	MS472-8	8	10	0	0	8	0	0	200	0	0	–	–	–	–	–	–
M	MS472A	8	8	0	0	0	0	0	200	0	0	–	–	–	–	–	–
R	S472	8	8	0	0	0	0	0	200	0	0	8	1	2	0	19	1
M	ME2704-0	8	10	0	0	0	0	0	200	C8	0	–	–	–	–	–	–
M	ME2704-8	8	10	0	0	8	0	0	200	C8	0	–	–	–	–	–	–
M	ME2704A	8	8	0	0	0	0	0	200	C8	0	–	–	–	–	–	–
R	E2704	8	8	0	0	0	0	0	200	C8	0	8	0	1	0	32	8
M	ME2708-0	8	10	0	0	0	0	0	400	C8	0	–	–	–	–	–	–
M	ME2708-8	8	10	0	0	8	0	0	400	C8	0	–	–	–	–	–	–
M	ME2708A	8	8	0	0	0	0	0	400	C8	0	–	–	–	–	–	–
R	E2708	8	8	0	0	0	0	0	400	C8	0	8	0	1	0	32	8

Notes: The prefix SN74 is omitted from TTL PROM types. L1, L2 and L3 represent Level 1, Level 2 and Level 3.

ALPHABETICAL INDEX

ALPHABETICAL INDEX

INTRODUCTION

The following index lists key words and concepts from the subject material of the manual together with the area(s) in the manual that supply major coverage of the listed concept. The numbers along the right side of the listing reference the following manual areas:

- Sections - References to Sections of the manual appear as "Section x" with the symbol x representing any numeric quantity.

- Appendixes - References to Appendixes of the manual appear as "Appendix y" with the symbol y representing any capital letter.

- Paragraphs - References to paragraphs of the manual appear as a series of alphanumeric or numeric characters punctuated with decimal points. Only the first character of the string may be a letter; all subsequent characters are numbers. The first character refers to the section or appendix of the manual in which the paragraph is found.

- Tables - References to tables in the manual are represented by the capital letter T followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the table). The second character is followed by a dash (-) and a number:

  Tx-yy

- Figures - References to figures in the manual are represented by the capital letter F followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the figure). The second character is followed by a dash (-) and a number:

  Fx-yy

- Other entries in the Index - References to other entries in the index are preceded by the word "See" followed by the referenced entry.

USER'S RESPONSE SHEET

Manual Title: Model 990 Computer Prototyping System Operation Guide (945255-9701)

Manual Date: 1 May 1976 Date of This Letter:

User's Name: Telephone:

Company: Office/Department:

Street Address:

City/State/Zip Code:

Please list any discrepancy found in this manual by page, paragraph, figure, or table number in the following space. If there are any other suggestions that you wish to make, feel free to include them. Thank you.

Location in Manual Comment/Suggestion

CUT ALONG LINE

NO POSTAGE NECESSARY IF MAILED IN U.S.A.
FOLD ON TWO LINES (LOCATED ON REVERSE SIDE), TAPE AND MAIL

FOLD

FIRST CLASS
PERMIT NO. 7284
DALLAS, TEXAS

BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY

TEXAS INSTRUMENTS INCORPORATED
DIGITAL SYSTEMS DIVISION

P.O. BOX 2909 · AUSTIN, TEXAS 78769

ATTN: TECHNICAL PUBLICATIONS
MS 2146

FOLD

TEXAS INSTRUMENTS
INCORPORATED
DIGITAL SYSTEMS DIVISION
POST OFFICE BOX 2909 AUSTIN, TEXAS 78769