TEXAS INSTRUMENTS

Improving Man's Effectiveness Through Electronics

Model 990 Computer

AMPL Microprocessor Prototyping Laboratory System Operation Guide

MANUAL NO. 946244-9701
ORIGINAL ISSUE 1 AUGUST 1977
REVISED 15 JANUARY 1978

Change 1 15 March 1978

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INSERT LATEST CHANGED PAGES DESTROY SUPERSEDED PAGES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 AMPL Microprocessor Prototyping Laboratory System (AMPL) Operation Guide (946244-9701)

Original Issue . . . . . 1 August 1977
Change 1 . . . . . 1 November 1977 (ECN 990292)
Revised . . . . . 15 January 1978 (ECN 419806)
Change 1 . . . . . 15 March 1978 (ECN 419834)

Total number of pages in this publication is 342 consisting of the following:

PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.
Cover	1	Appendix B Div	0	Appendix G Div	0
Effective Pages	1	B-1	1	G-1 - G-4	0
iii - xii	0	B-2 - B-8	0	Appendix H Div	0
1-1 - 1-6	0	B-9 - B-18	1	H-1 - H-56	0
2-1 - 2-26	0	Appendix C Div	0	Appendix I Div	0
3-1 - 3-28	0	C-1 - C-4	0	I-1 - I-4	0
4-1 - 4-40	0	Appendix D Div	0	Alphabetical Index Div	0
5-1 - 5-58	0	D-1 - D-2	0	Index-1 - Index-10	0
6-1 - 6-18	0	Appendix E Div	0	User's Response	0
7-1 - 7-16	0	E-1 - E-2	0	Business Reply	0
Appendix A Div	0	Appendix F Div	0	Cover Blank	0
A-1 - A-10	0	F-1 F-4	0	Cover	0

PREFACE

This manual describes the software of the AMPL* Microprocessor Prototyping Laboratory and the hardware installation of the emulator and buffer module and the logic-state trace data module which support the prototyping laboratory. The manual also describes the AMPL language in detail, and includes examples of the use of the laboratory to debug hardware and software.

The manual is organized into seven sections and eight appendixes including:

I General Description – Provides a general description of the prototyping laboratory and its hardware and software components. Also describes the prototype development cycle.

II System Hardware Installation – Describes the hardware configuration of the laboratory in detail, including interconnection information.

III AMPL Applications – Describes the use of the AMPL language in development of the hardware of a prototype system.

IV AMPL Microprocessor Prototyping Language – Describes the AMPL language in detail.

V System Operation – Includes operating instructions for AMPL software and description of the user commands.

VI Errors and Recovery – Describes the error and warning message formats and lists the error messages and recovery procedures.

VII Examples – Includes examples of the use of the AMPL language in developing the software of a prototype system.

A TX900 System Generation – Describes the generation of a custom TX990 for prototyping laboratory.

B DX10 System Generation – Describes the generation of a custom DX10 for prototyping laboratory.

C AMPL Grammar – The formal definition of the AMPL grammer.

D AMPL Statement Summary – A summary of the AMPL statements.

E AMPL Reserved Words – A list of the reserved words of the AMPL language.

F System Symbols – A list of the system variables of the AMPL system.

G User Commands – A summary of the AMPL user commands.

*Trademark of Texas Instruments Incorporated

H AMPL Procedure and Function Library – Descriptions of the procedures and functions of the AMPL Procedure and Function Library.

I Glossary – A glossary of terms used in the manual.

The following documents contain additional information related to the prototyping laboratory:

Title	Part Number
990 Computer Family Systems Handbook	945250-9701
Model 990 Computer TMS 9900 Microprocessor Assembly Language Programmer's Guide	943441-9701
Model 990 Computer TX990 Operating System Programmer's Guide	946259-9701
Model 990 Computer Terminal Executive Development System (TXDS) Programmer's Guide	946258-9701
Model 990 Computer DX10 Operating System Release 3 Reference Manual Vols. 1-6	946250-9701, -9702 -9703, -9704 -9705, -9706
Model 990 Computer FS990 System Installation and Operation Manual	946254-9701
Model 990 Computer DS990 System Installation and Operation Manual	946284-9701
Model 990/4 Computer System Hardware Reference Manual	945251-9701
Model 990/10 Computer System Hardware Reference Manual	945417-9701
Model 990/4 Computer System Depot Maintenance Manual	945403-9701
Model 990/10 Computer System Depot Maintenance Manual	945404-9701
Model 990 Logic-State Trace Data Module Installation and Operation	946241-9701
Model 990 Computer AMPL Logic State Trace Data Module Depot Maintenance Manual	946242-9701
Model 990 Emulator Module and TMS 9900/9980 Buffer Module Installation and Operation	946245-9701
Model 990 Computer TMS 9900/9980 Emulator/Buffer Modules Depot Maintenance Manual	946239-9701
Model 990 Computer Model FD800 Floppy Disk System Installation and Operation	945253-9701
Model 990 Computer Model DS10 Cartridge Disk System Installation and Operation	946261-9701

Title	Part Number
Model 990 Computer Model DS31/32 Disk System Installation and Operation	945260-9701
Model 990 Computer Model DS25/DS50 Disk System Installation and Operation	946231-9701
Model 990 Computer Model 911 Video Display Terminal Installation and Operation	945423-9701
Model 990 Computer Model 913 CRT Display Terminal Installation and Operation	943457-9701
Model 990 Computer Model 810 Line Printer Installation and Operation	939460-9701
Model 990 Computer TTY/EIA Terminal Interface Module Installation and Operation	946240-9701
Model 990 Computer Model 733 ASR/KSR Data Terminal Installation and Operation	945259-9701
Model 990 Computer Model 804 Card Reader Installation and Operation	945262-9701
AMPL Reference Card	946265-9701
Model 990 Computer AMPL System Tutorial	949621-9701

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

SECTION VI. ERRORS AND RECOVERY

TABLE OF CONTENTS (Continued)

APPENDIXES

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

LIST OF TABLES

SECTION I

GENERAL DESCRIPTION

1.1 INTRODUCTION

The AMPL Microprocessor Prototyping Laboratory is a powerful tool for developing microprocessor systems. The laboratory contributes significantly to the development of both software (or firmware) and a hardware prototype. A prototyping laboratory includes the following:

- A Model 990 Computer and operating system with dual floppy disks or a moving-head disk.
- A hardware Emulator with TMS 9900 Buffer Module or TMS 9980 Buffer Module.
- A hardware Logic-State Trace Data module for the prototype system.
- A set of user commands for debugging the prototype system and its software.
- AMPL Microprocessor Prototyping Language to support user commands.

The operating system supplied for the Model 990 Computer equipped with floppy disks is the Terminal Executive (TX990). It supports the Terminal Executive Development System (TXDS) that includes the Text Editor, Assembler, and Link Utility. The computer, TX990, and floppy disks also support the AMPL program which implements the user commands. Floppy disk units provide the user with a high-speed, random access storage medium on which the user can develop programs using the Text Editor, Assembler, and Link Utility. In addition, the debugging environment (including prototype programs) can be saved on and restored from diskette files.

The operating system supplied with the DS990 disk system is the Disk Executive, DX10, Release 3. It supports program development utilities including a text editor, a macro assembler, a FORTRAN compiler, a COBOL language processor, a BASIC* interpreter, and a link editor. The computer, DX10, and the disk system also support the AMPL program that implements the user commands. The disk system provides the user with a high-speed, random access storage medium on which the user can develop programs using the program development capabilities. In addition, the debugging environment (including prototype programs) can be saved on and restored from disk files.

The emulator hardware includes a buffer module for the microprocessor to be used in the user's prototype system (target system). The emulator with the appropriate buffer module replaces the microprocessor of the target system. The emulator also provides memory and control circuitry required to execute software in the target system under control of the AMPL software. The emulator memory consists of a 256-word trace memory and a 4K-word user memory. The control circuitry allows the emulator to halt processing on specified program conditions (breakpoints) and to store memory or instruction addresses in the trace memory. The host computer (Model 990 Computer in which AMPL software executes) can halt the emulated microprocessor and access and modify memory in the target system.

The trace module contains a 256 by 20-bit memory that stores up to 256 20-bit words that can be read by the host computer. The 20 bits may be connected to the address or memory bus through the emulator module or directly to test points in the target system using data probes. Four of the lines are equipped with latches that can change state on fast pulses as narrow as approximately 10 ns duration.

*Trademark of Trustees of Dartmouth College, Hanover, N.H.

Four qualifier lines select data words to be traced, and an event counter counts traced words that match a specified data word. Both the qualifiers and the data are masked to allow the user to specify any qualifiers and data bits to be used in the selection. The event counter, together with the delay counter, generates a signal that may be used to terminate the trace. Alternatively, a signal that the trace memory is full may be used to terminate the trace. Termination of the trace may either interrupt the host computer or signal the emulator to halt the microprocessor.

1.2 TYPICAL PROTOTYPE DEVELOPMENT CYCLE

Figure 1-1 shows the use of the laboratory during development of the software and integration of software into the prototype system. First, the software development capabilities executing in the host computer are used to obtain an object module. The source code is prepared and assembled using the text editor and assembler. When assembler errors are detected, the text editor is used to correct the source code and the source code is reassembled. When all errors that the assembler can detect have been corrected, the resulting object module is linked with other required object modules, if any, to obtain a linked object module. Source code for one or more modules may be corrected and reassembled if the link utility detects an error. When the linking operation detects no errors, the linked object module is ready to be loaded into memory of the target system. Alternatively, the module may be loaded into user memory or trace memory of the emulator module as appropriate.

The user may enter commands and AMPL statements to initialize trace and breakpoint conditions for testing the software and prototype system, and to start the emulator. When a breakpoint halts the emulator, the user may enter commands to inspect the results of the test. When the results are not correct, the user may correct the source code, reassemble and relink. Or the user may make corrections in the object code in target system memory. The user may also continue testing until the prototype system is thoroughly tested, and all areas of the software have been executed. When the software has been thoroughly tested, the user may execute the PROM Programmer Utility to manufacture firmware.

1.3 OPERATIONAL FEATURES

Figure 1-2 shows an AMPL configuration consisting of a trace module and an emulator module under control of a host computer connected to a target system by cables from the buffer module and from the trace module. The buffer module cables connect to the microprocessor socket in the target system, and the trace module cables connect to signals being traced in the target system.

The lab provides the following operational features:

- The user may load the target program into emulator memory or target memory or a combination of both.
- The user substitutes the emulator for the microprocessor in the target system, and thus monitors the flow of execution of the target program in the target system.
- The host processor monitors interrupts from both the emulator module and the trace module(s).
- The user may inspect and modify the contents of memory and registers in the target system, the emulator module, and the trace module(s).
- The user may save and restore the environments of debug sessions to continue the debug operation at a later time.
- The user may enter user commands to control the debug operation.

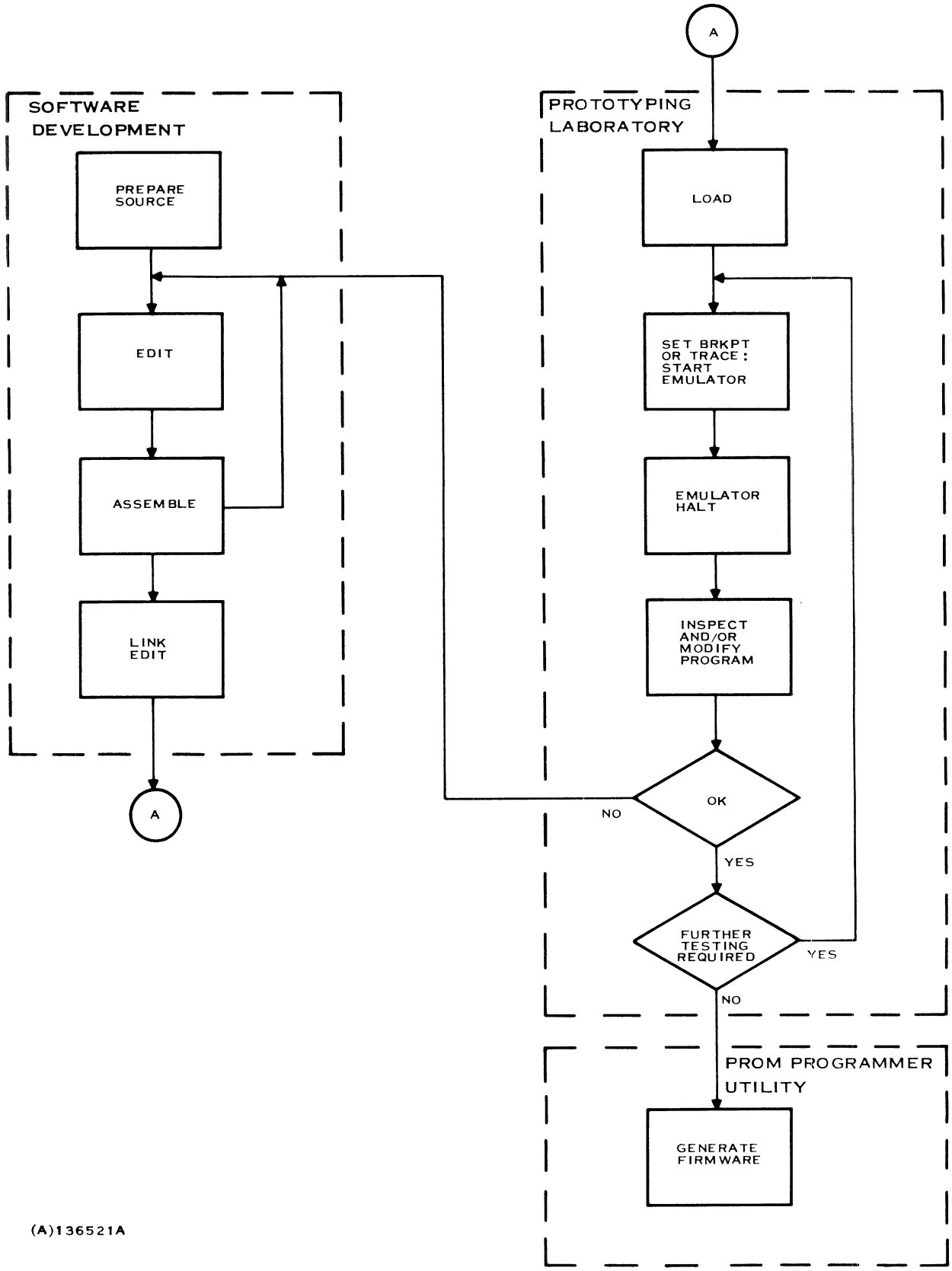


Figure 1-1. Typical Prototype Development Cycle

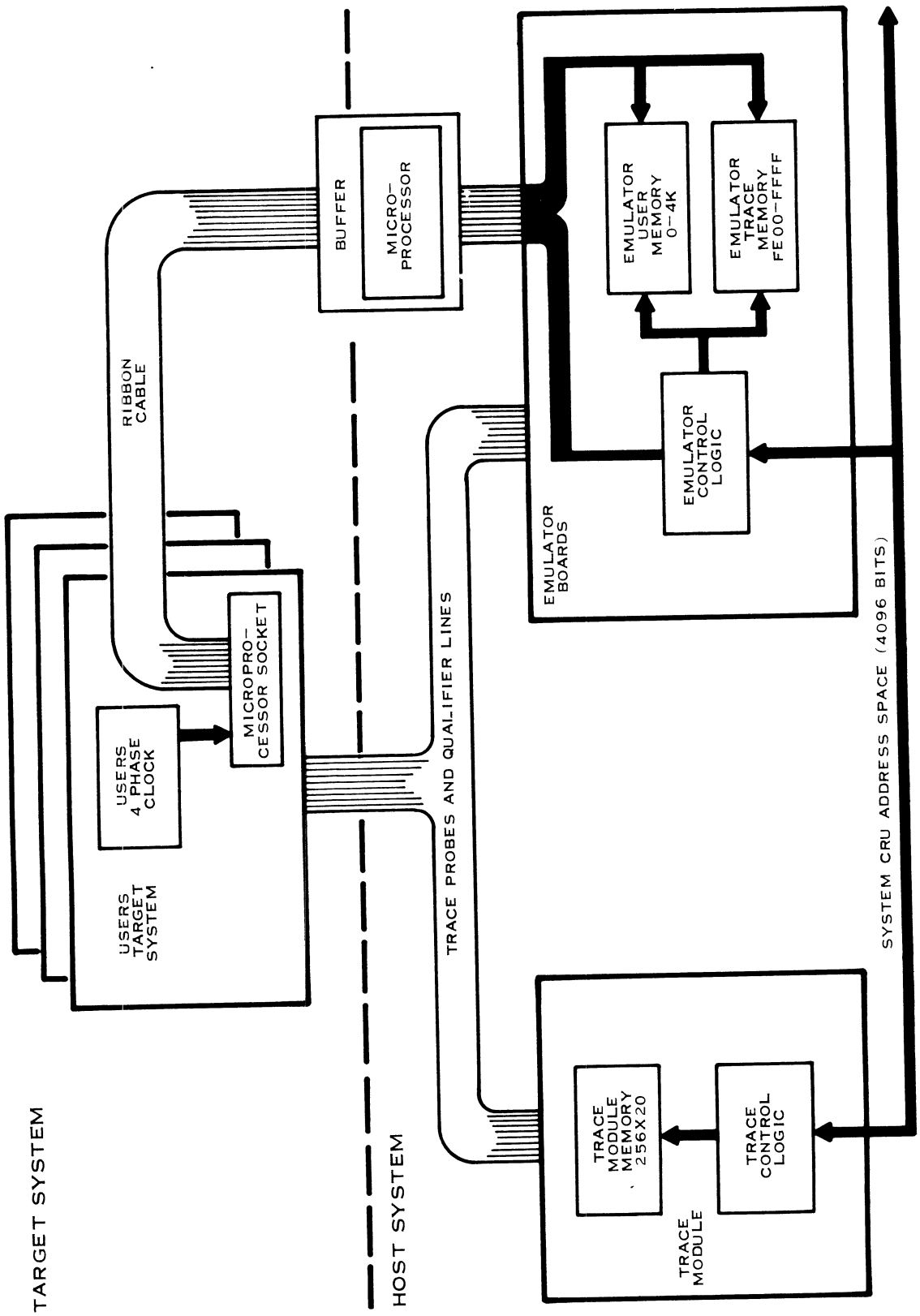


Figure 1-2. AMPL Microprocessor Prototyping Laboratory Configuration

1.4 SOFTWARE CONFIGURATION

Software required for the laboratory system is as follows:

- AMPL Software
- PROM Programmer Utilities

For a TX990 system, the additional software is:

- TX990 Operating System
- TXDS Development System
- TXMIRA Assembler
- TXEDIT Text Editor
- TXLINK Link Utility
- TXLNK Link Editor Utility

For a DX10 system, the additional software is:

- DX10 Operating System
- SDSMAC Macro Assembler
- SDSTIE Text Editor
- SDSLNK Link Editor

SECTION II

SYSTEM HARDWARE INSTALLATION

2.1 GENERAL

This section provides instructions for site planning, installation, and initial checkout of the emulator/buffer modules and the logic-state trace data module when employed with either a FS990 or DS990 system for TMS 9900 or TMS 9980 microprocessor hardware and software development.

2.1.1 **SYSTEM CONFIGURATION.** A typical AMPL Microprocessor Prototyping Laboratory is shown in figure 2-1. Several options of peripheral equipment are available for use with the system to assist with checkout and debug of the target system. In all cases the trace module and the emulator are installed in the Model 990 Computer chassis (990/4 or 990/10, or an expander chassis) and are cabled in one of several alternate configurations to suit the target system under test. In figure 2-1, the microprocessor connector from the buffer module is installed in the target system in place of the microprocessor; the other cables from the buffer module are connected to the emulator module installed in the 990 computer chassis.

2.1.2 **SYSTEM COMPONENTS.** Typical components of an AMPL Microprocessor Prototyping Laboratory are shown in figure 2-2. The trace module and the emulator module are shown installed in a Model 990 Computer chassis. Optional connections to the target system are:

1. From the emulator via the buffer module, target connect, and the interconnecting cables. Components associated with the emulator are shown in figure 2-3.

2. From the trace module via the trace data probe cable, the terminator box, and the probe leads equipped with either female connector pins or IC test clips. Components associated with the trace module are shown in figure 2-4.

A summary of the components shown in figures 2-3 and 2-4 are listed in table 2-1, along with the associated part numbers for each.

2.2 SYSTEM INSTALLATION

2.2.1 COMPUTER CHASSIS OPTIONS AND PREPARATIONS

CAUTION

Before removing or installing circuit boards and cables, be sure power is off since voltage transients can damage components parts.

The CRU base address that the software uses to address and control the prototyping laboratory system is determined by the physical location of the emulator module and the trace module in the 990 computer chassis, or in a CRU expander chassis. Since there are several different combinations of options for these installations, the user and programmer should decide upon the chassis slot locations, the CRU base address, and the interrupt level for both of these modules before installing them.

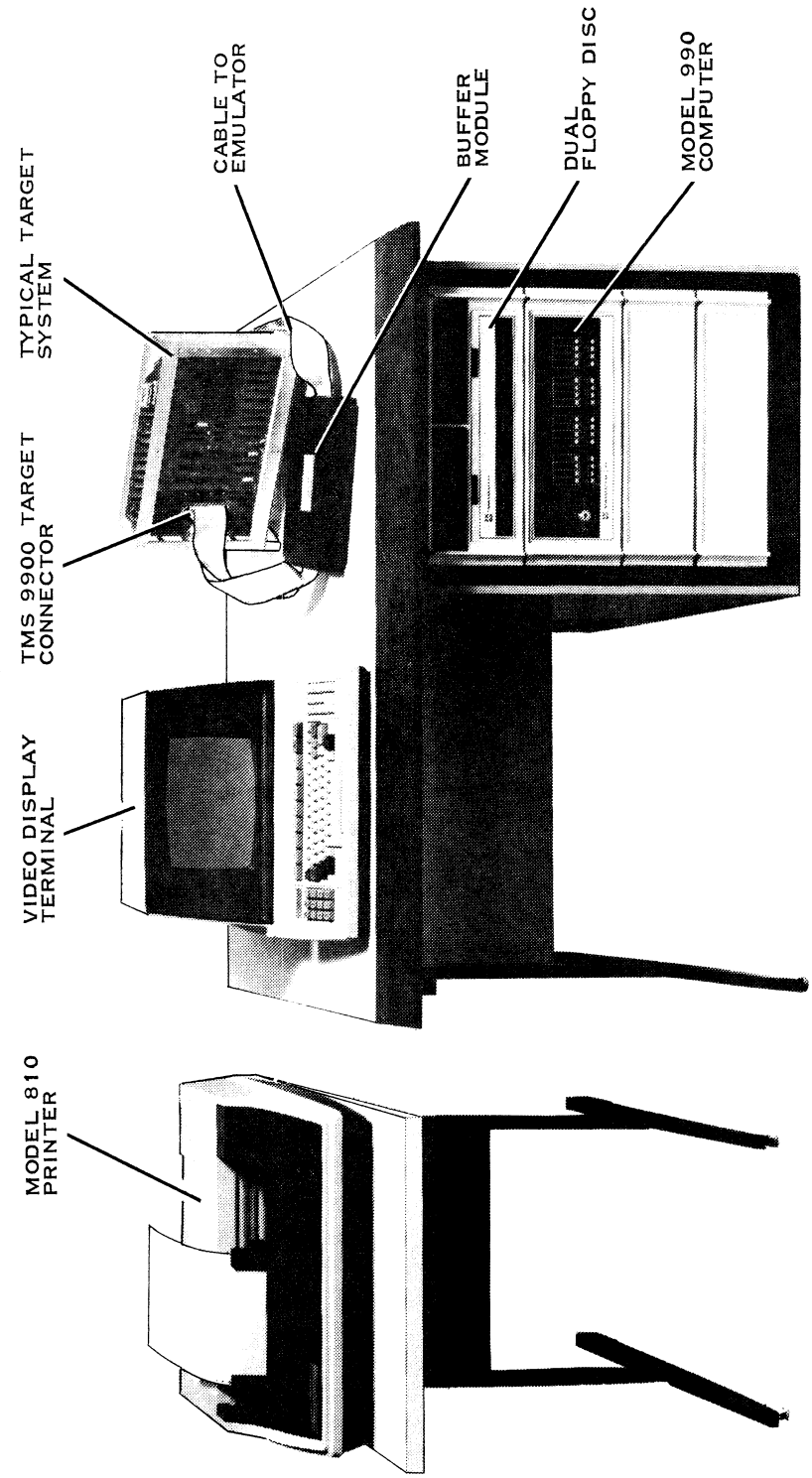
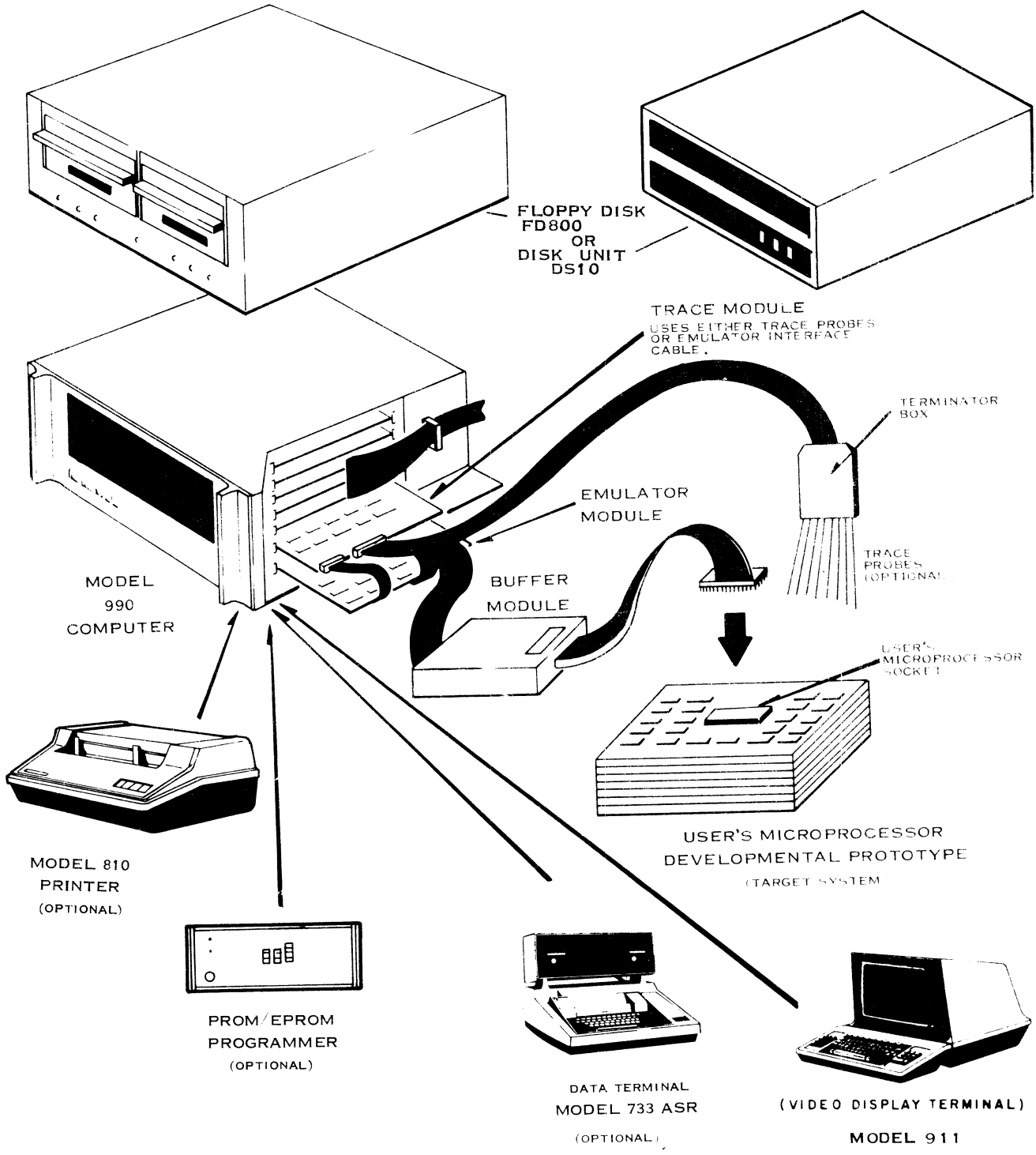


Figure 2-1. Typical AMPL Microprocessor Prototyping Laboratory



(A)137446

Figure 2-2. Typical Prototyping Laboratory Configuration

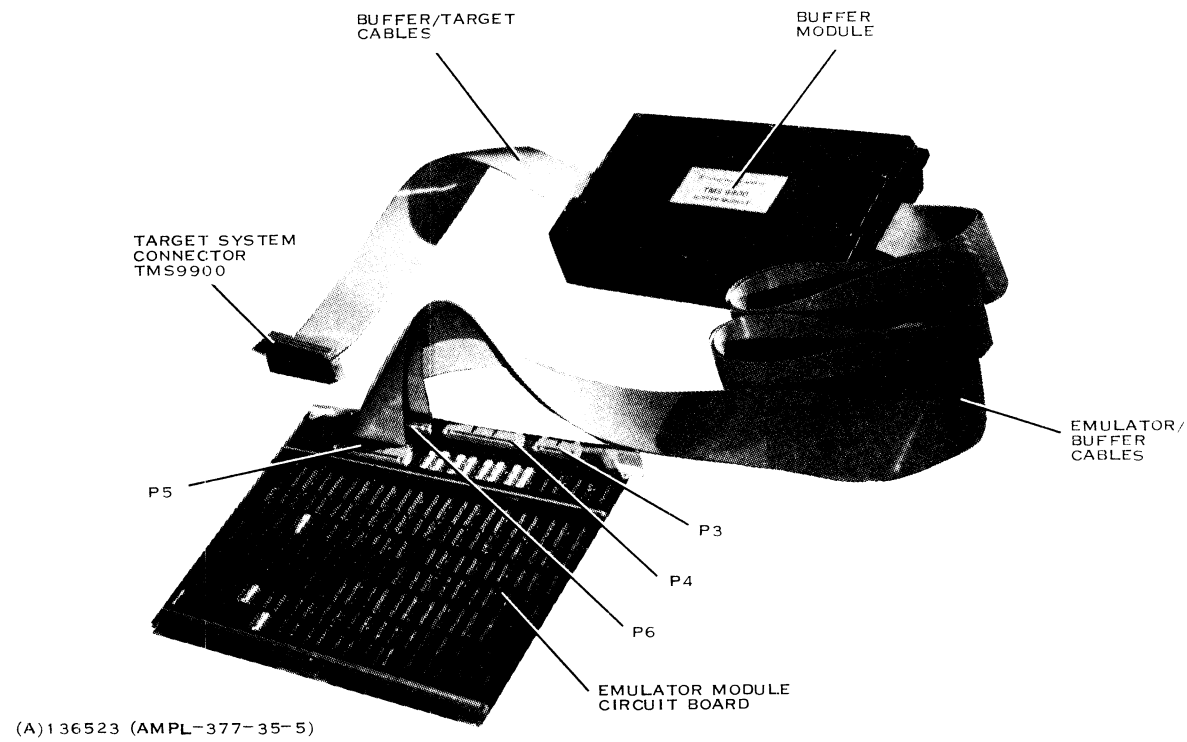


Figure 2-3. Emulator/Buffer and Interconnect Cables

There are four typical FS990 system configurations, two of which use the Model 911 Video Display Terminal (VDT) and two of which use the Model 913 VDT. Figure 2-5 shows the standard configuration, using a Model 990/4 Computer and the Model 911 VDT. Figure 2-6 shows the same peripherals and a Model 990/10 Computer. Figure 2-7 shows the typical configuration using a Model 990/4 Computer and a Model 913 VDT. Figure 2-8 shows a similar configuration using a Model 990/10 Computer with the same peripherals. If the prototyping laboratory system is received with the modules installed in the standard configurations, the CRU base addresses have been established at the factory and are compatible with the software package. Also, in that case, the correct interrupt levels have been determined and prewired in the interrupt jumper plugs for the peripherals. No alterations or modifications of the 990 computer chassis is required. Other configurations are available from Texas Instruments to modify existing FS990 systems to suit user requirements.

A DS990 system that supports the AMPL laboratory requires a Model 990/10 Computer 13-slot chassis with a disc drive system. The Model 911 VDT controller, emulator, and trace module may be installed in the main chassis as shown in figure 2-9, or in a CRU expansion chassis. The main chassis configuration for use with an expansion chassis is shown in figure 2-10, and the expansion chassis is shown in figure 2-11. A 911 VDT for another work station may be connected to the controller board, and the emulator and trace modules for the station may be installed in slots 12 and 13.

If the prototyping laboratory system is received with the modules installed in the recommended computer configuration, the CRU base addresses for the modules in the computer chassis have been established at the factory and are compatible with the software package. Also, in that case, the correct interrupt levels have been determined and prewired in the interrupt jumper plugs. The CRU base addresses of the modules in the expansion chassis are determined by the plug on the CRU expander board to which the chassis is connected, and the slots into which the modules are connected. The interrupt levels are prewired in the interrupt jumper plug.

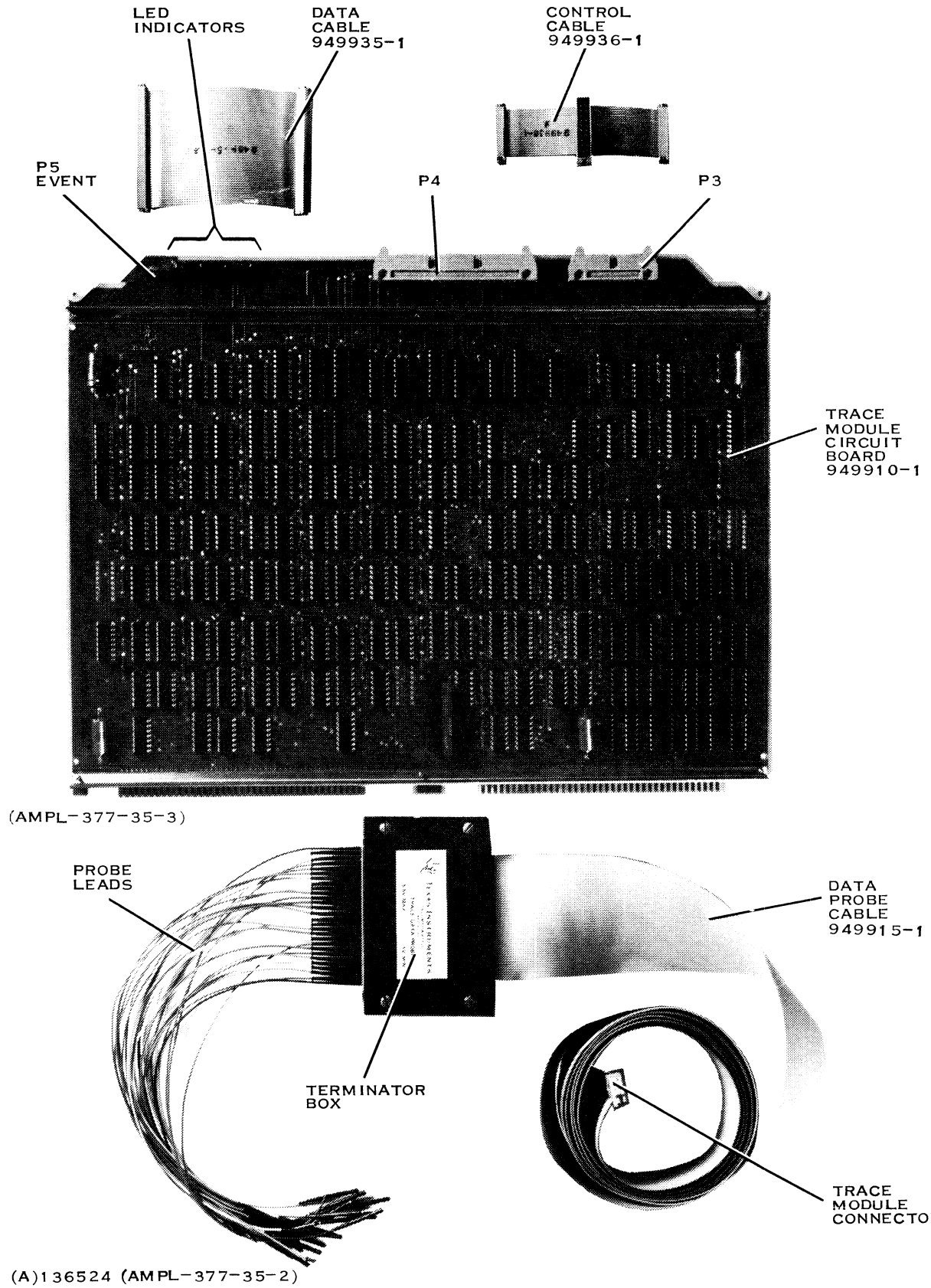


Figure 2-4. Trace Module, Trace Data Probe, and Interconnect Cables

Table 2-1. AMPL Microprocessor Prototyping Laboratory

Component	Part Number
Emulator Module	949925-0001
TMS 9900 Buffer Assembly	949937-0001
Emulator/Buffer Cable	949924-0001
Emulator/Buffer Cable	949924-0002
Buffer/Target Cable	949923-0001
Buffer/Target Cable	949923-0002
Buffer/Target Clock Cable	949945-0001
TMS 9900 Target Connector	949905-0001
TMS 9900 Buffer Module	949995-0001
TMS 9980 Buffer Assembly	949937-0002
Emulator Buffer Cable	949924-0001
Emulator/Buffer Cable	949924-0002
Buffer/Target Cable	949923-0003
Buffer/Target Cable	949923-0004
TMS 9980 Target Connector	949955-0001
TMS 9980 Buffer Module	949940-0001
Trace Module	949910-0001
Emulator/Trace Data Cable	949935-0001
Emulator/Trace Control Cable	949936-0001
Trace Data Probe	949915-0001

	P1			P2		
CHASSIS SLOT NUMBER	CRU BASE ADDRESS	CIRCUIT BOARD	INTER-RUPT LEVEL	CRU BASE ADDRESS	CIRCUIT BOARD	INTER-RUPT LEVEL
1	N/A	990/4 AU W8/K	N/A	N/A	990/4 AU W/8K	N/A
2	02E0	40K 990/4 MEMORY EXPANSION	N/A	02C0	40K 990/4 MEMORY EXPANSION	N A
3	02A0	8K 990/4 MEMORY EXPANSION (OPT.)	N/A	0280	8K 990/4 MEMORY EXPANSION (OPT.)	N/A
4	0260	SPARE	N/A	0240	SPARE	N/A
5	0220	SPARE	N/A	0200	SPARE	N A
6	01E0	SPARE	N/A	01C0	SPARE	N/A
7	01A0	SPARE	N/A	0180	SPARE	N A
8	0160	EMULATOR MODULE	6	0140	EMULATOR MODULE	6
9	0120	TRACE MODULE	6	0100	TRACE MODULE	6
10	00E0	911 VDT CONTROLLER	3	00C0	911 VDT CONTROLLER	3
11	00A0	FLOPPY DISC CONTROLLER	7	0080	FLOPPY DISC CONTROLLER	7
12	0060	LINE PRINTER (OPTIONAL)	6	0040	CARD READER (OPTIONAL)	4
13	0020	PROM PROGRAM-MER (OPTIONAL)	6	0000	733 ASR (OPTIONAL)	6

(A)137447

Figure 2-5. Model 990/4 Computer 13-Slot Chassis, Standard Configuration, TX990 with 911 VDT

CHASSIS SLOT NUMBER	P1 CRU BASE ADDRESS	P1 CIRCUIT BOARD	P1 INTER-RUPT LEVEL	P2 CRU BASE ADDRESS	P2 CIRCUIT BOARD	P2 INTER-RUPT LEVEL
1	N/A	990/10/AU2	N/A	N/A	990/10/AU2	N/A
2	02E0	990/10/AU1	N/A	02C0	990/10/AU1	N/A
3	02A0	16K MEMORY EXPANSION	N/A	0280	16K MEMORY EXPANSION	N/A
4	0260	48K MEMORY EXPANSION	N/A	0240	48K MEMORY EXPANSION	N/A
5	0220	SPARE	N/A	0200	SPARE	N/A
6	01E0	SPARE	N/A	01C0	SPARE	N/A
7	01A0	SPARE	N/A	0180	SPARE	N/A
8	0160	EMULATOR MODULE	6	0140	EMULATOR MODULE	6
9	0120	TRACE MODULE	6	0100	TRACE MODULE	6
10	00E0	911VDT CONTROLLER	3	00C0	911 VDT CONTROLLER	3
11	00A0	FLOPPY DISC CONTROLLER	7	0080	FLOPPY DISC CONTROLLER	7
12	0060	LINE PRINTER (OPTIONAL)	6	0040	CARD READER (OPTIONAL)	4
13	0020	PROM PROGRAM-MER (OPTIONAL)	6	0000	733 ASR (OPTIONAL)	6

(A)137448

Figure 2-6. Model 990/10 Computer 13-Slot Chassis Configuration for TX990 with 911 VDT

CHASSIS SLOT NUMBER	P1 CRU BASE ADDRESS	P1 CIRCUIT BOARD	P1 INTER-RUPT LEVEL	P2 CRU BASE ADDRESS	P2 CIRCUIT BOARD	P2 INTER-RUPT LEVEL
1	N/A	990/4 AU W/8K	N/A	N/A	990/4 AU W 8K	N/A
2	02E0	40K 990/4 MEMORY EXPANSION	N/A	02C0	40K 990/4 MEMORY EXPANSION	N A
3	02A0	8K 990/4 MEMORY EXPANSION (OPT.)	N/A	0280	8K 990/4 MEMORY EXPANSION (OPT.)	N A
4	0260	SPARE	N/A	0240	SPARE	N/A
5	0220	SPARE	N/A	0200	SPARE	N A
6	01E0	SPARE	N/A	01C0	SPARE	N/A
7	01A0	SPARE	N/A	0180	SPARE	N A
8	0160	EMULATOR MODULE	4	0140	EMULATOR MODULE	4
9	0120	TRACE MODULE	4	0100	TRACE MODULE	4
10	00E0	913 VDT CONTROLLER	3	00C0	913 VDT CONTROLLER	3
11	00A0	FLOPPY DISC CONTROLLER	7	0080	FLOPPY DISC CONTROLLER	7
12	0060	LINE PRINTER (OPTIONAL)	4	0040	SPARE	4
13	0020	PROM PROGRAM-MER (OPTIONAL)	N/A	0000	733 ASR (OPTIONAL)	6

(A)137449

Figure 2-7. Model 990/4 Computer 13-Slot Chassis Configuration for TX990 with 913 VDT

CHASSIS SLOT NUMBER	P1 CRU BASE ADDRESS	P1 CIRCUIT BOARD	P1 INTER-RUPT LEVEL	P2 CRU BASE ADDRESS	P2 CIRCUIT BOARD	P2 INTER-RUPT LEVEL
1	N/A	990/10/AU2	N/A	N/A	990/10/AU2	N/A
2	02E0	990/10/AU1	N/A	02C0	990/10/AU1	N/A
3	02A0	16K MEMORY EXPANSION	N/A	0280	16K MEMORY EXPANSION	N/A
4	0260	48K MEMORY EXPANSION	N/A	0240	48K MEMORY EXPANSION	N/A
5	0220	SPARE	N/A	0200	SPARE	N/A
6	01E0	SPARE	N/A	01C0	SPARE	N/A
7	01A0	SPARE	N/A	0180	SPARE	N/A
8	0160	EMULATOR MODULE	4	0140	EMULATOR MODULE	4
9	0120	TRACE MODULE	4	0100	TRACE MODULE	4
10	00E0	913 VDT CONTROLLER	3	00C0	913 VDT CONTROLLER	3
11	00A0	FLOPPY DISC CONTROLLER	7	0080	FLOPPY DISC CONTROLLER	7
12	0060	LINE PRINTER (OPTIONAL)	4	0040	CARD READER (OPTIONAL)	4
13	0020	PROM PROGRAM-MER (OPTIONAL)	N/A	0000	733 ASR (OPTIONAL)	6

(A)137450

Figure 2-8. Model 990/10 Computer 13-Slot Chassis Configuration for TX990 with 913 VDT

CHASSIS SLOT NUMBER	P1 CRU BASE ADDRESS	P1 CIRCUIT BOARD	P1 INTER-RUPT LEVEL	P2 CRU BASE ADDRESS	P2 CIRCUIT BOARD	P2 INTER-RUPT LEVEL
1	N/A	990/10/AU2	N/A	N/A	990/10/AU2	N/A
2	02E0	990/10/AU1	N/A	02C0	990/10/AU1	N/A
3	02A0	16K MEMORY EXPANSION	N/A	0280	16K MEMORY EXPANSION	N/A
4	0260	48K MEMORY EXPANSION	N/A	0240	48K MEMORY EXPANSION	N/A
5	0220	16K MEMORY EXPANSION	N/A	0200	16K MEMORY EXPANSION	N/A
6	01E0	48K MEMORY EXPANSION	N/A	01C0	48K MEMORY EXPANSION	N/A
7	01A0	DISC CONTROLLER	13	0180	DISC CONTROLLER	13
8	0160	MAG. TAPE CONTR OR TILINE COUPLER	9	0140	MAG. TAPE CONTR OR TILINE COUPLER	9
9	0120	911 VDT CONTROLLER	8	0100	911 VDT CONTROLLER	10
10	00E0	EMULATOR MODULE	12	00C0	EMULATOR MODULE	11
11	00A0	TRACE MODULE	3	0080	TRACE MODULE	7
12	0060	LINE PRINTER (OPTIONAL)	14	0040	CARD READER (OPTIONAL)	4
13	0020	PROM PROGRAM-MER (OPTIONAL)	15	0000	733 ASR (OPTIONAL)	6

(A)137451

Figure 2-9. Model 990/10 Computer 13-Slot Chassis Recommended DX10 Configuration without CRU Expansion Chassis

CHASSIS SLOT NUMBER	P1 CRU BASE ADDRESS	P1 CIRCUIT BOARD	P1 INTER-RUPT LEVEL	P2 CRU BASE ADDRESS	P2 CIRCUIT BOARD	P2 INTER-RUPT LEVEL
1	N/A	990/10 AU2	N/A	N/A	990/10 AU2	N/A
2	02E0	990/10 AU1	N/A	02C0	990/10 AU1	N/A
3	02A0	16K MEMORY EXPANSION	N/A	0280	16K MEMORY EXPANSION	N/A
4	0260	48K MEMORY EXPANSION	N/A	0240	48K MEMORY EXPANSION	N/A
5	0220	16K MEMORY EXPANSION	N/A	0200	16K MEMORY EXPANSION	N/A
6	01E0	48K MEMORY EXPANSION	N/A	01C0	48K MEMORY EXPANSION	N/A
7	01A0	DISK CONTROLLER	13	0180	DISK CONTROLLER	13
8	0160	MAG. TAPE CONTR. OR TILINE COUPLER	9	0140	MAG. TAPE CONTR. OR TILINE COUPLER	9
9	0120	911 VDT CONTROLLER	8	0100	911 VDT CONTROLLER	10
10	00E0	911 VDT CONTROLLER	12	00C0	911 VDT CONTROLLER	11
11	00A0	CRU EXPANDER	3	0080	CRU EXPANDER	7
12	0060	LINE PRINTER (OPTIONAL)	14	0040	CARD READER (OPTIONAL)	4
13	0020	PROM PROGRAM-MER (OPTIONAL)	15	0000	733 ASR (OPTIONAL)	6

(A)137452

Figure 2-10. Model 990/10 Computer 13-Slot Chassis Recommended DX10 Configuration with CRU Expansion Chassis

CHASSIS SLOT NUMBER	P1 CRU BASE ADDRESS	P1 CIRCUIT BOARD	P1 INTER-RUPT LEVEL	P2 CRU BASE ADDRESS	P2 CIRCUIT BOARD	P2 INTER-RUPT LEVEL
1	N.A	CRU BUFFER	N/A	N/A	CRU BUFFER	N/A
2	06E0	TILINE COUPLER (OPTIONAL)	N/A	06C0	TILINE COUPLER (OPTIONAL)	N.A
3	06A0	SPARE	N/A	0680	SPARE	N.A
4	0660	SPARE	N/A	0640	SPARE	N/A
5	0620	SPARE	N/A	0600	SPARE	N/A
6	05E0	SPARE	N/A	05C0	SPARE	N/A
7	05A0	DISK CONTROLLER (OPTIONAL)	13	0580	DISK CONTR. OR LINE PRINTER(OPT.)	13
8	0560	MAGNETIC TAPE CONTROLLER (OPT.)	9	0540	MAGNETIC TAPE CONTROLLER (OPT.)	9
9	0520	911 VDT CONTROLLER	8	0500	911 VDT CONTROLLER	10
10	04E0	EMULATOR MODULE	12	04C0	EMULATOR MODULE	11
11	04A0	TRACE MODULE	3	0480	TRACE MODULE	7
12	0460	EMULATOR MODULE	14	0440	EMULATOR MODULE	4
13	0420	TRACE MODULE	15	0400	TRACE MODULE	6

NOTE: THE CRU BASE ADDRESSES SHOWN APPLY TO EXPANSION CHASSIS 1 (CONNECTED TO P3 OF CRU EXPANDER BOARD). ADD $(N-1) * 400_{16}$ TO THE ABOVE ADDRESSES FOR CRU BASE ADDRESSES IN EXPANSION CHASSIS N.

(A) 137453

Figure 2-11. Model 990/10 Computer 13-Slot CRU Expansion Chassis Recommended DX10 Configuration

2.2.1.1 Interrupt Levels. If the trace and emulator modules are obtained as separate kits to be installed in an existing computer system, new interrupt level connections must be made in the computer chassis and in the expansion chassis for proper interrupt recognition by the program.

CAUTION

Ensure that ac power to the computer chassis has been disabled before beginning this procedure.

2.2.1.2 Interrupt Connections and Modifications. Wiring in the backplane of the chassis connects interrupt lines from each circuit board connector to a pair of jumper plugs located on the backplane adjacent to slot number 1, as shown in figure 2-12. Jumper wires installed in these jumper plugs connect the interrupt output lines from the circuit boards to the CRU interrupt inputs. In the expansion chassis, similar plugs connect the interrupt output lines from the circuit boards to the interrupt inputs of the CRU buffer module.

Figure 2-13 is an outline drawing of the jumper plugs for the 13-slot chassis and the 6-slot chassis. Interrupt levels are shown at the right of each connector. The 990/4 computer does not recognize interrupt levels 8 through 15. Two jumper positions are wired to each chassis interrupt line. This allows multiple interrupts to be connected to one interrupt level. To make interrupt level modifications perform the following steps:

1. Remove the circuit boards from the first five slots of the chassis to gain access to the interrupt jumper plugs.

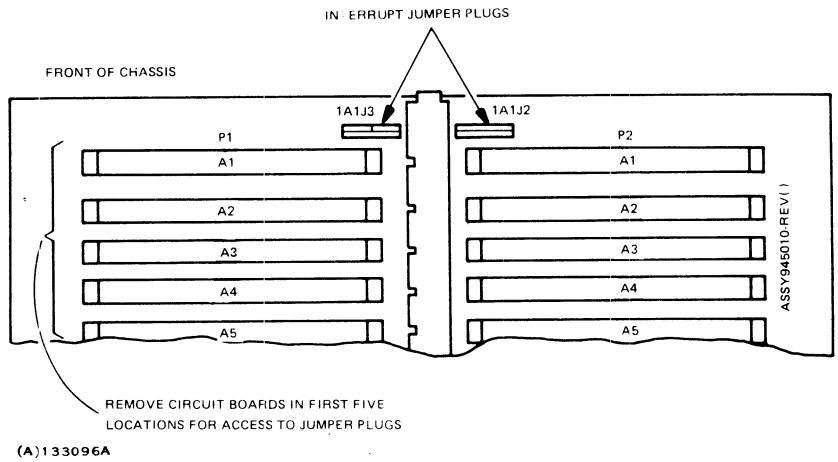


Figure 2-12. Location of Interrupt Jumper Plugs (6– and 13– Slot Chassis)

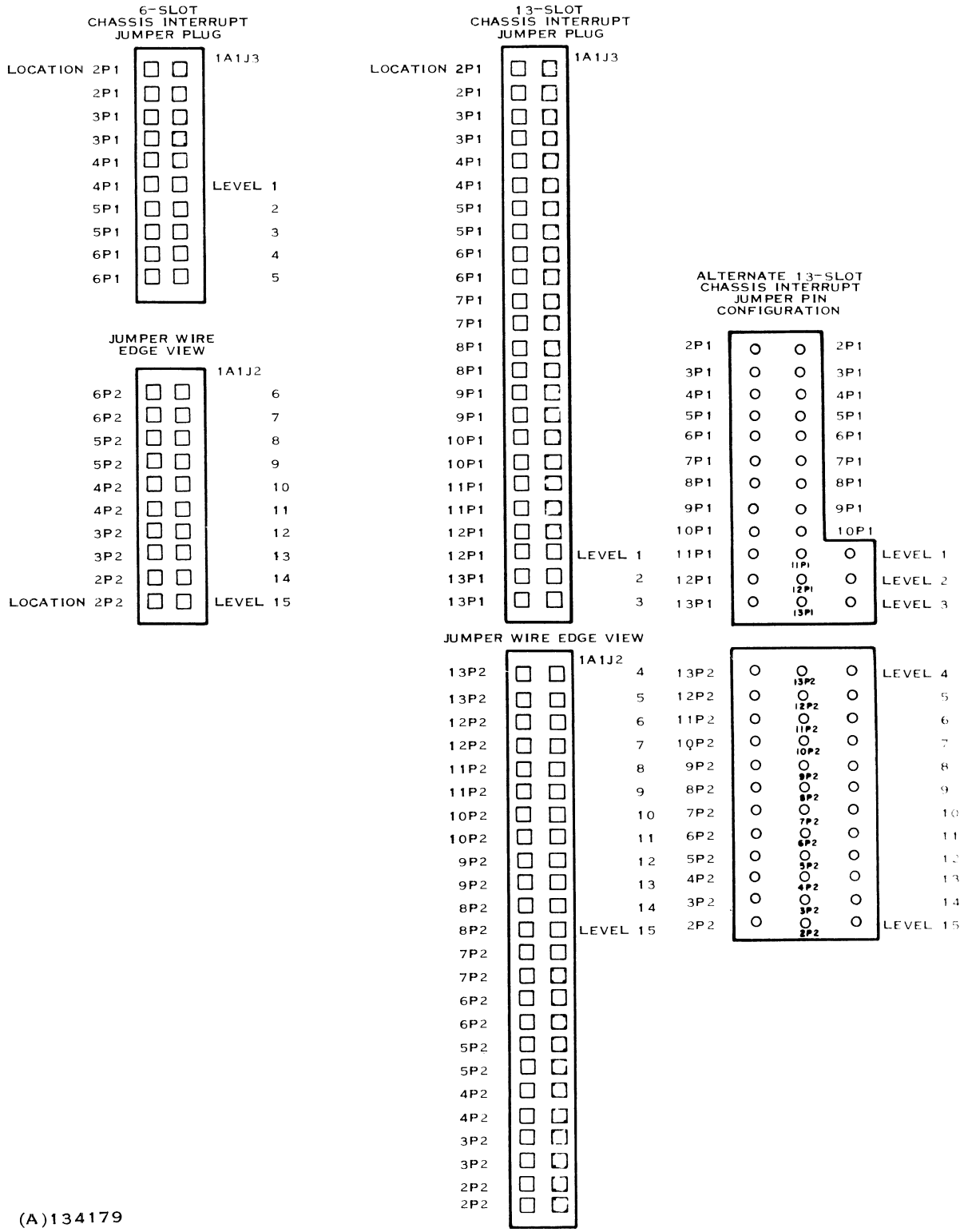


Figure 2-13. 6– and 13– Slot Chassis Interrupt Jumper Plugs

2. Use one of the following steps to connect the jumper wires.

   a. Pluggable jumpers. Remove the jumper plugs from the chassis. Insert the specially-constructed jumper wire to connect the interrupt level to the chassis slot interrupt pin.

   b. Jumper pins. Connect a jumper wire from the interrupt level pin to the chassis slot interrupt pin.

3. Reinstall the jumper plugs and then reinstall the circuit boards.

For additional information about interrupt connections, refer to the *Model 990/4 Computer System Hardware Reference Manual* or the *Model 990/10 Computer System Hardware Reference Manual.*

2.2.2 CIRCUIT BOARD INSTALLATION. When the proper locations in the computer chassis for the emulator and trace modules have been determined and the correct interrupt levels established, install the modules in the computer chassis in the following manner:

1. Ensure that computer power is off.

2. Insert the circuit board, component side up, into the selected slot so the board slides into the card guides on either side of the slot.

3. Push the board straight in until the edge connector engages the connector in the backplane. Verify that the guide slots in the circuit board mate properly with the alignment comb in the chassis.

2.2.3 CABLE CONNECTIONS. When the circuit boards have been installed in the appropriate chassis slot, install the cables for the particular configuration in which they are to be used. The AMPL Microprocessor Prototyping Laboratory can utilize one emulator/buffer as shown in figure 2-14, or one trace module as shown in figure 2-20, or both an emulator/buffer and a trace module as shown in figure 2-23. To install a plug/receptacle, be sure that the embossed arrowheads are aligned for correct orientation before mating the pair together, then align the pins, match the connectors together evenly, and press on the plug until it is firmly seated.

NOTE

Pin 1 of the connector is on the same side as the red stripe on the ribbon cable.

2.2.3.1 Installing Emulator/Buffer Cables. The emulator works in conjunction with the buffer module to replace the microprocessor and/or the program memory in the user's target system. The buffer module houses the microprocessor and completes the interface between the emulator and the target system.

Two ribbon cables are routed from the buffer module to the emulator, and three other ribbon cables (two for TMS 9980) are routed between the buffer module and the target connector. The configuration of these cable connections is shown in View A and B of figure 2-14.

Install the cables from the buffer module to the emulator as shown in the illustration. The location of the connectors on the emulator module are as shown in figure 2-15. When installing connectors, be sure the embossed arrowheads on the plug and the receptacle have the same orientation, as shown in figure 2-16. Since one of the mating connectors is slightly inset on the emulator circuit board, these connections should be made before the emulator board is fully inserted into the computer chassis

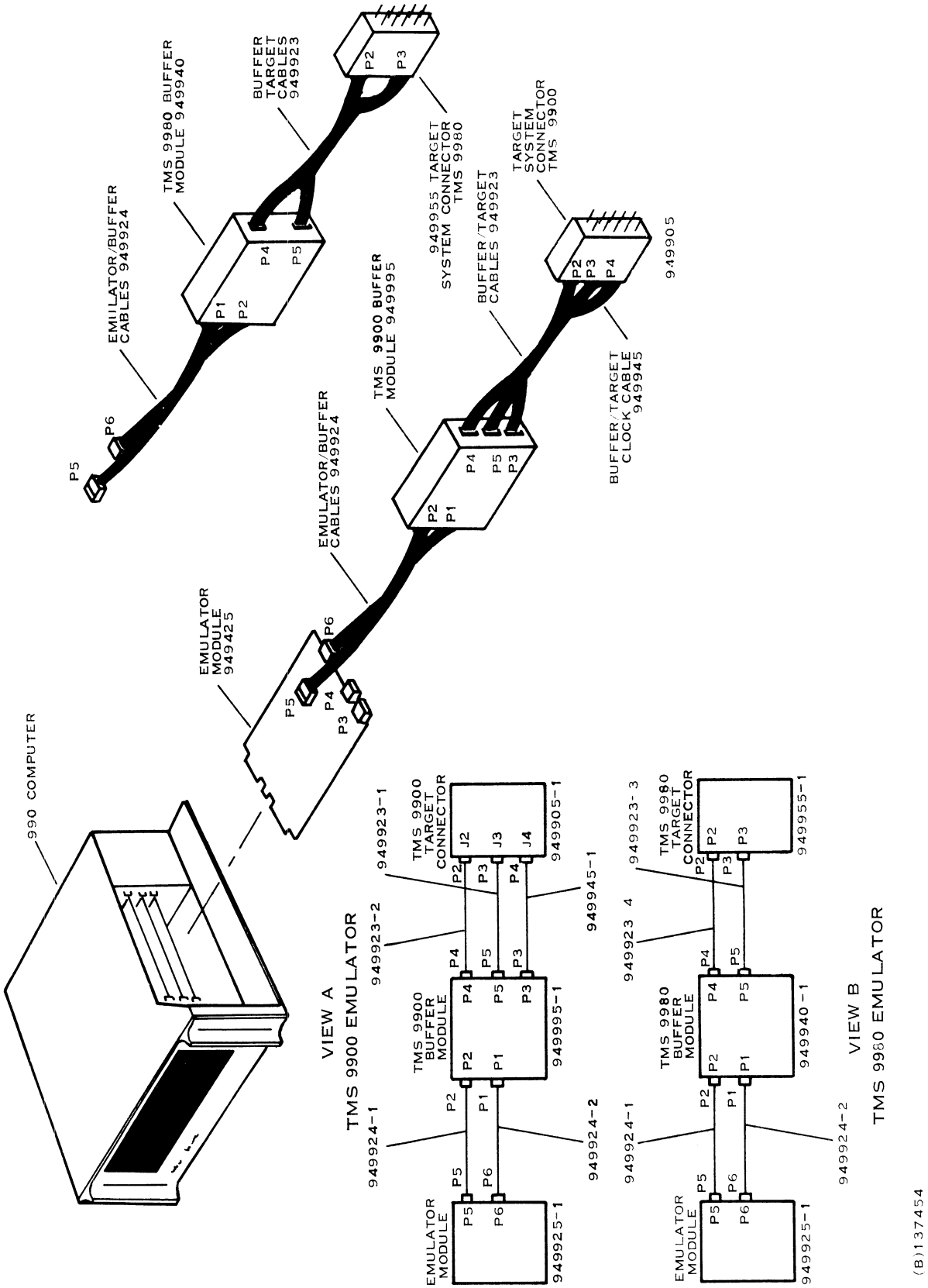


Figure 2-14. Emulator/Buffer Cabling Diagram

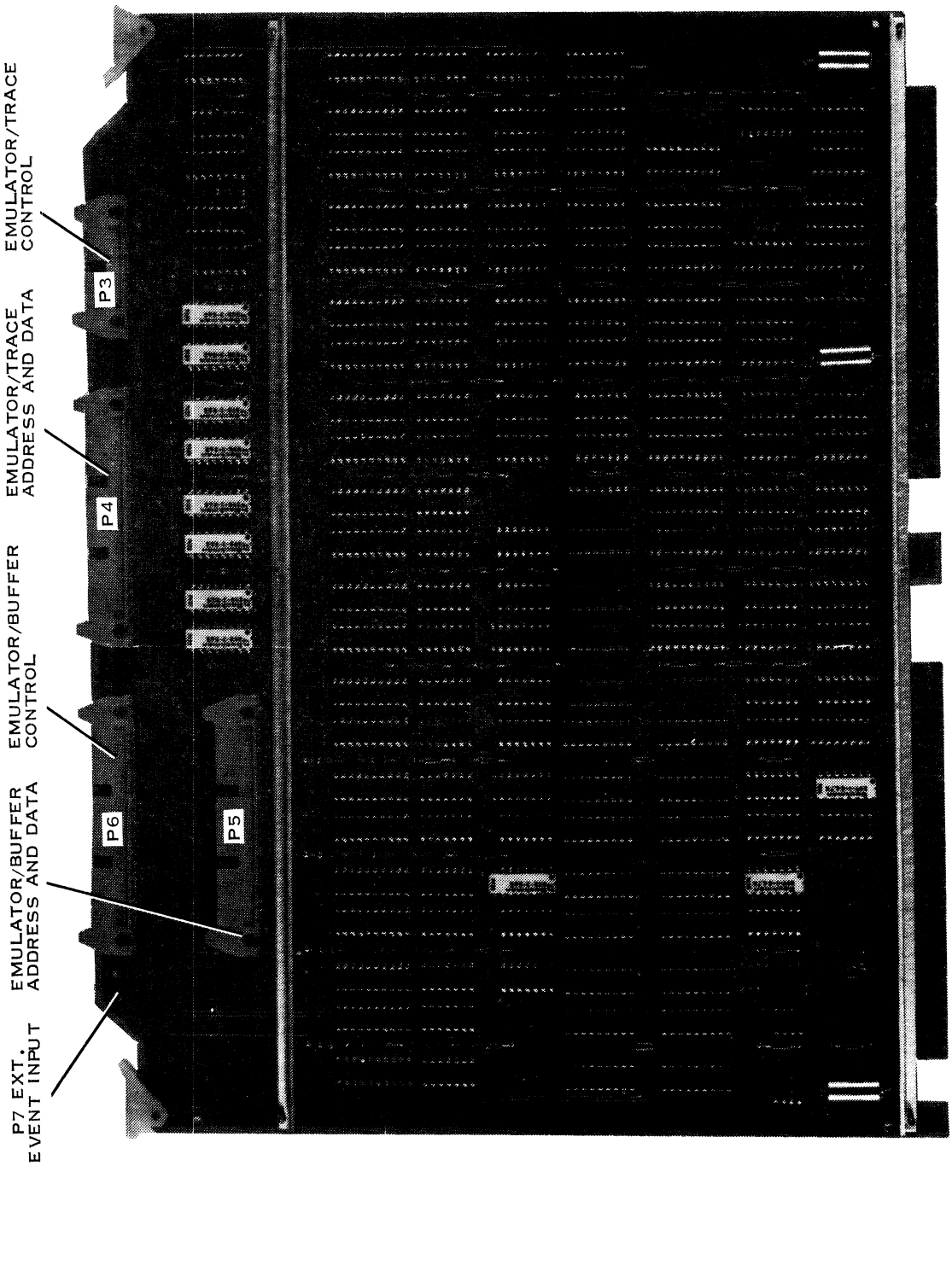


Figure 2-15. Emulator Module

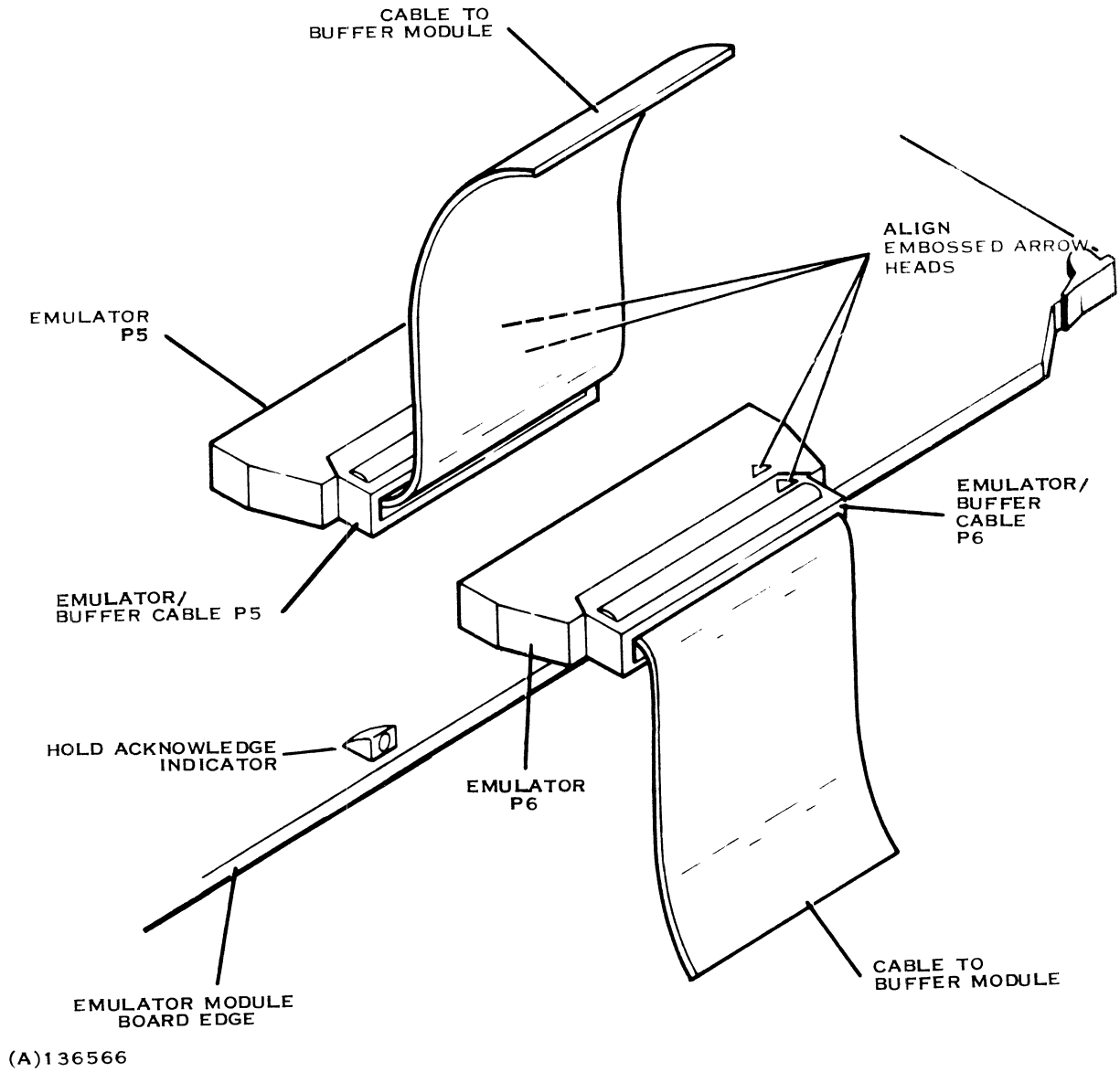


Figure 2-16. Cable Connections at Emulator

backplane connectors. Attach the emulator/buffer cable connectors, P5 and P6, to the emulator module connectors P5 and P6, respectively, then slide the emulator into the chassis backplane connectors. When the backplane connectors are fully engaged, the two plastic ejectors on the outer corners of the emulator board should settle in place against the frame.

2.2.3.2 Installing Target System Connectors. The three cables from the TMS 9900 buffer module are installed in the target system connector as shown in figure 2-17. Connect P2 to J2, P3 to J3, and P4 to J4. Be sure the orientation of the plugs is as shown in the illustration (i.e., the embossed arrowhead on the plugs is adjacent to pin 1).

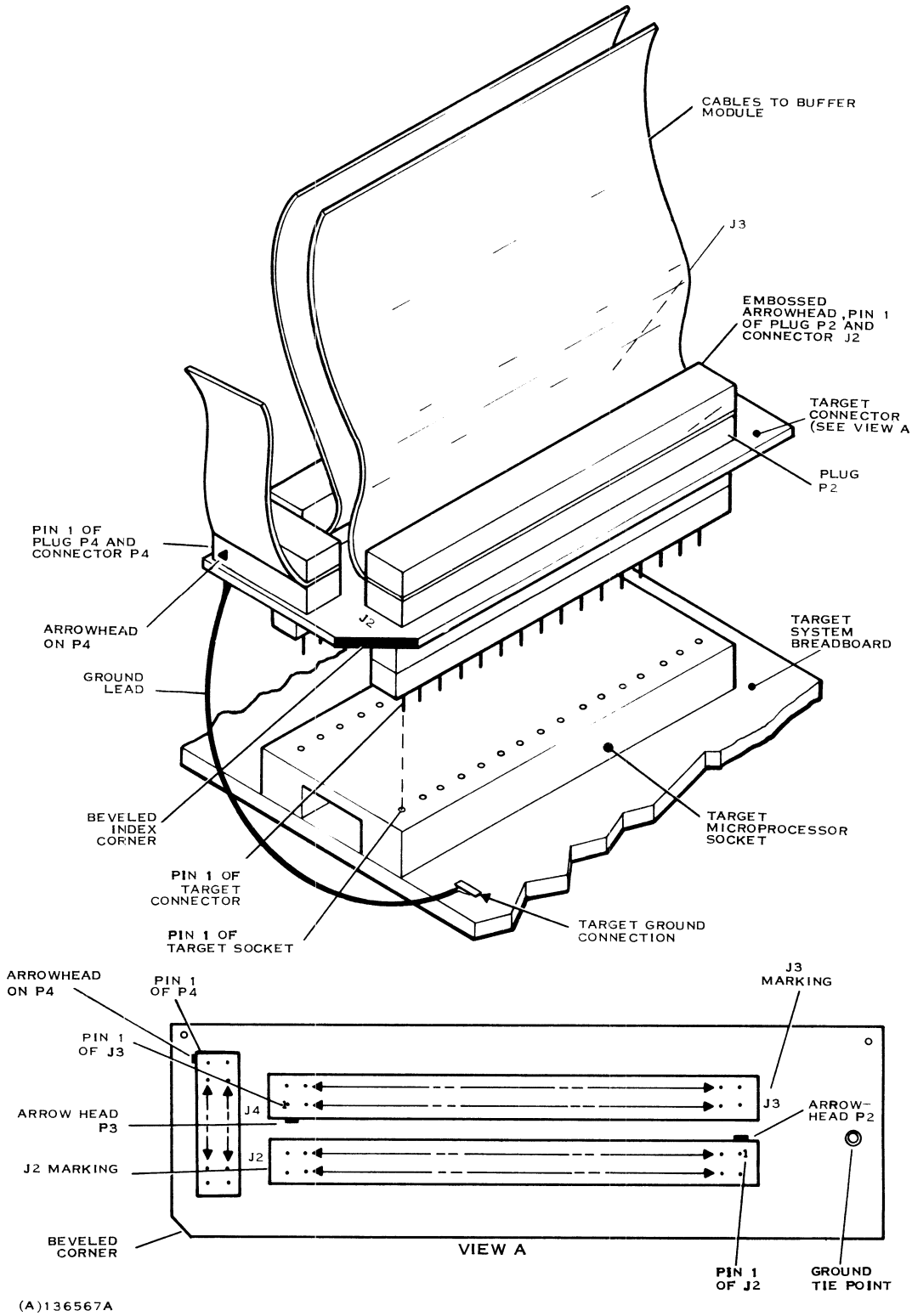


Figure 2-17. Installation of Buffer Cables and Connector at TMS 9900 Target System

The connections for the TMS 9980 Buffer are similar, as shown in figure 2-18. Connect the two cables from the buffer module to the target system connector, P2 to P2, and P3 to P3. No clock cable or separate ground connection is required. Be sure the orientation of the plugs is as shown in the illustration (i.e., the embossed arrowhead on the plugs is adjacent to pin 1.

When the system cables have been installed in the target system connector, remove the microprocessor chip from the target system breadboard and install the target connector in place of the chip. Be sure the beveled index corner of the connector (which identifies pin 1 of the microprocessor chip) has the correct orientation, as shown in figure 2-17 and 2-18.

Next, (TMS 9900 buffer only) connect the ground lead to a good target system ground, preferably near the microprocessor socket.

When all of the connections have been made, insert a slot screwdriver into the cutout of the buffer module cover and select either an INTERNAL prototyping system CLOCK or the TARGET SYS external clock (figure 2-19).

2.2.3.3 Installing Trace Module Cables. If a trace module is to be used alone, install the trace data probe as shown in figure 2-20. This general-purpose data probe is implemented on a 6-foot ribbon cable with an edge connector on one end and a terminator box on the other end. Each of the 26 signal channels in this cable (plus two ground leads) are extended from the terminator box via color-coded probe leads, as shown in figure 2-21. Each probe lead has a female connector on its outboard end, sized to fit a standard wire-wrap post. Special spring-loaded IC test clips that accommodate the probe leads are available from suppliers, and these IC test clips may be used for convenient attachment to any IC element, as shown in figure 2-22.

2.2.3.4 Connecting Emulator and Trace Module. The emulator and trace modules are cabled together as shown in figure 2-23. The interconnections consist of two, 4-inch jumper cables. The control cable is connected P3 to P3, and the data cable is connected P4 to P4. Verify that the embossed arrowheads on plugs P3 and P4 are aligned with pin 1 on connectors P3 and P4. Optionally, the jumper data cable may be omitted, if it is desired to trace data patterns directly from the target system. In this case, the trace data probe may be connected from the trace module to the target system as described in the preceding paragraph.

2.3 PERIPHERAL EQUIPMENT FOR THE PROTOTYPING LABORATORY

Many various options of peripheral equipment are available for installation in the AMPL Microprocessor Prototyping Laboratory. The most frequently used peripherals in a TX990 system are shown in figure 2-24, along with the interface board or controller and the cables for each type of equipment. Figure 2-25 shows similar information for a DX10 system.

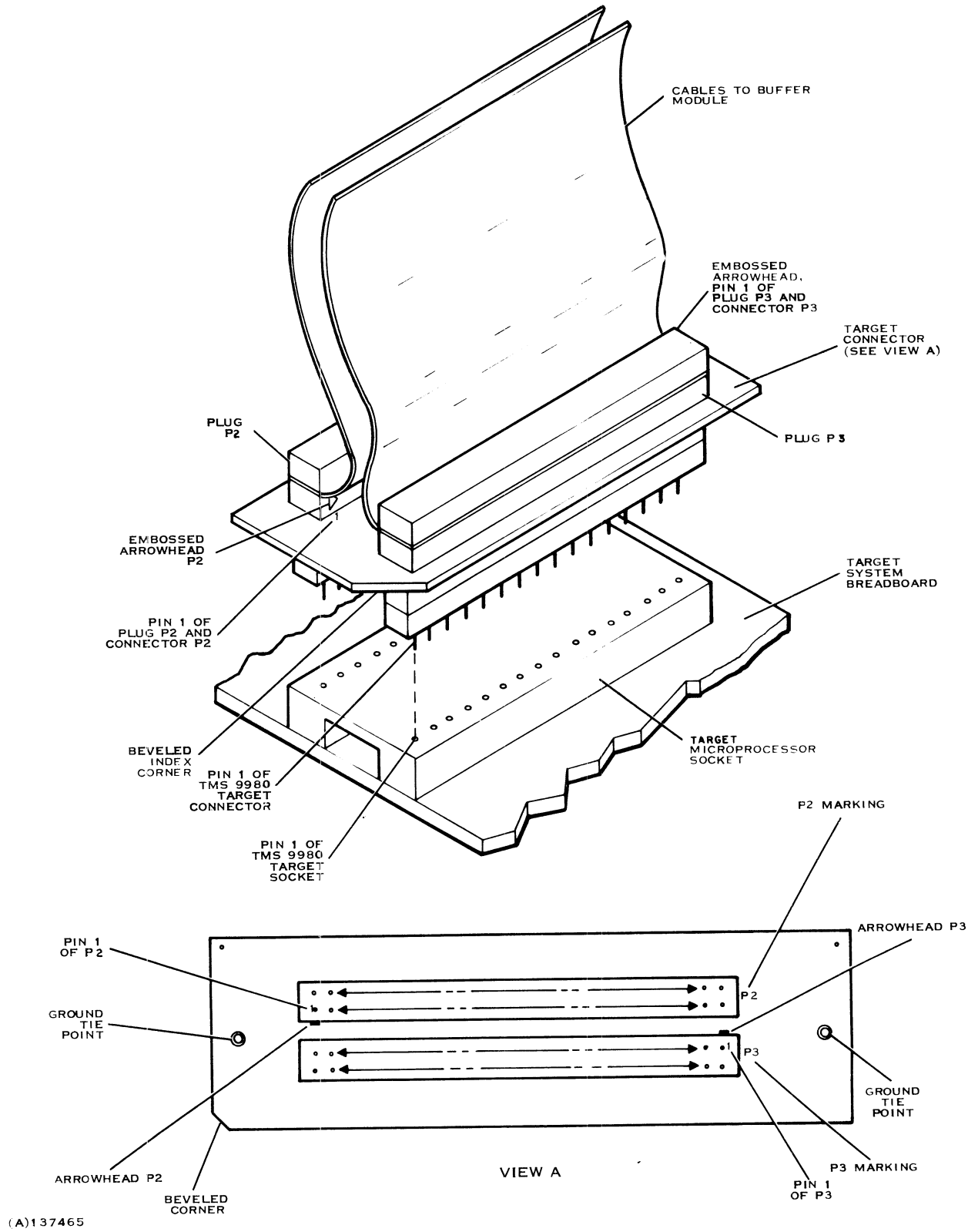


Figure 2-18. Installation of Buffer Cables and Connector at TMS 9900 Target System

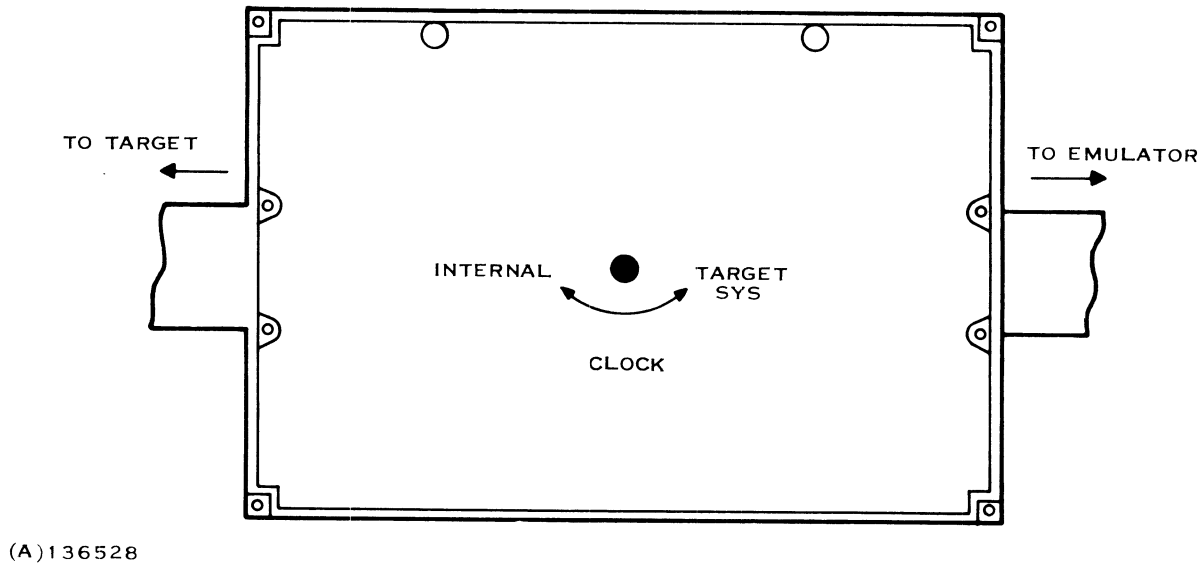


Figure 2-19. Buffer Module

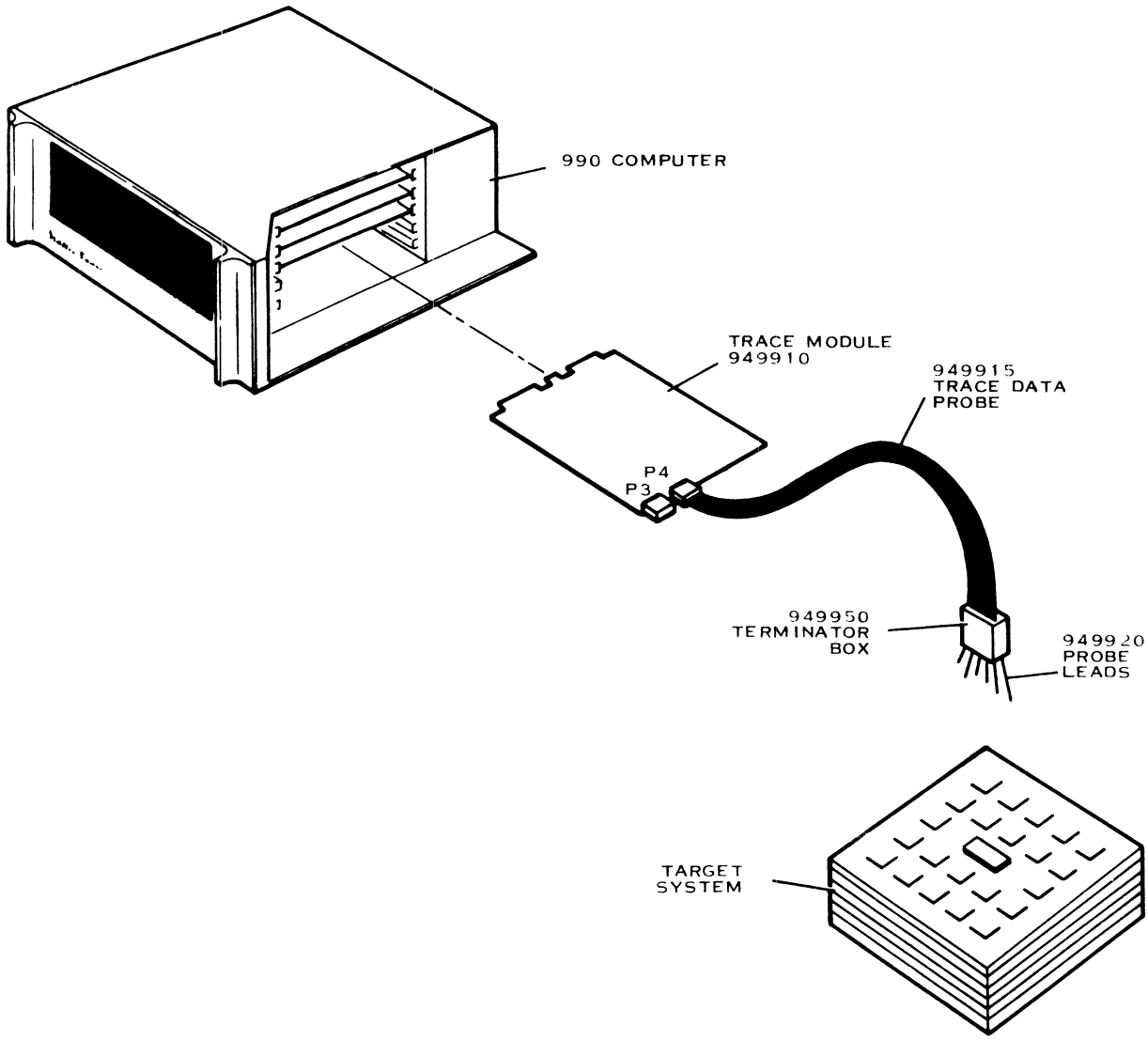


Figure 2-20. Trace Module Cabling Diagram

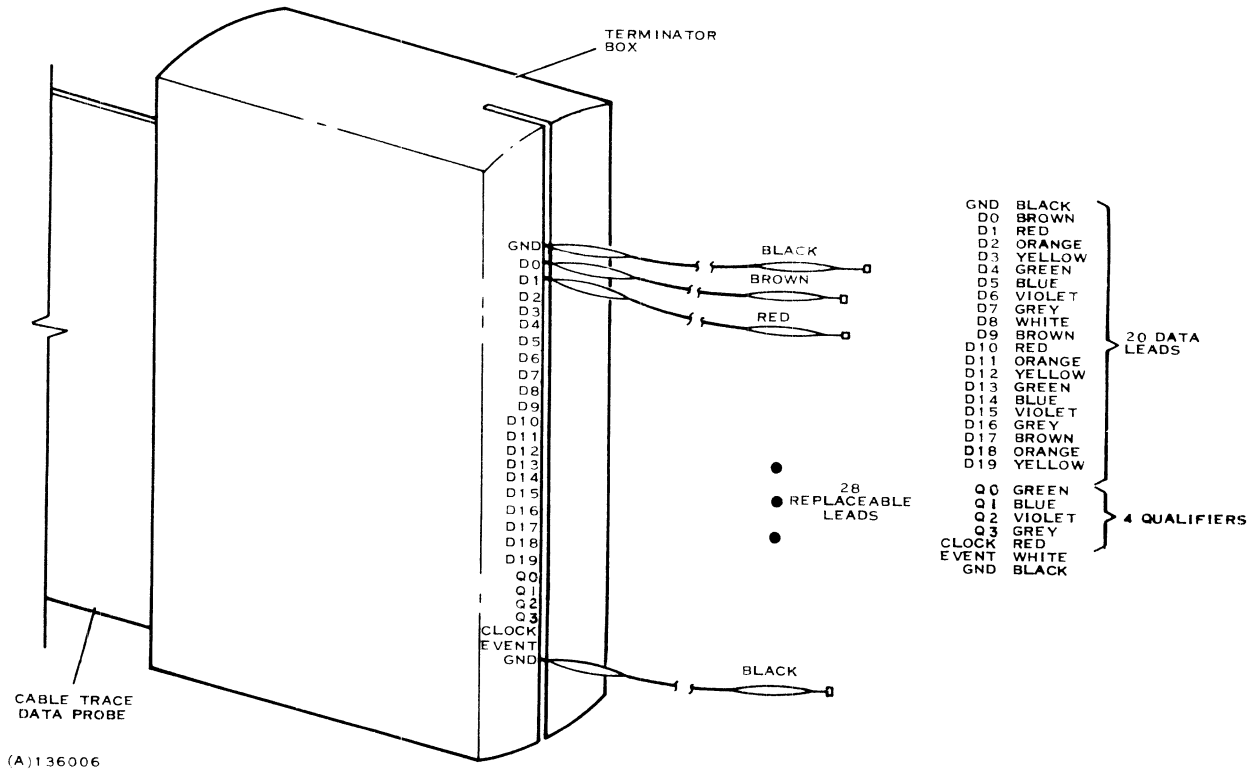


Figure 2-21. Trace Data Probe Terminator Box and Leads

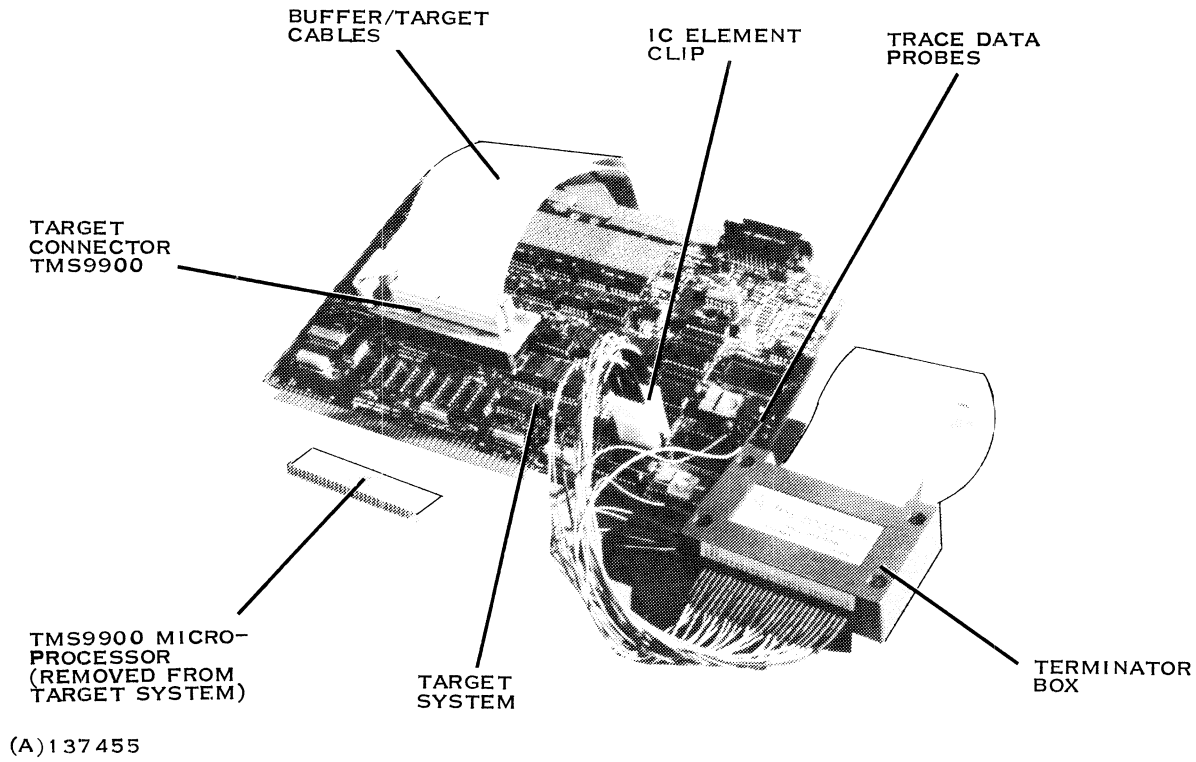


Figure 2-22. Typical Connections to Target System

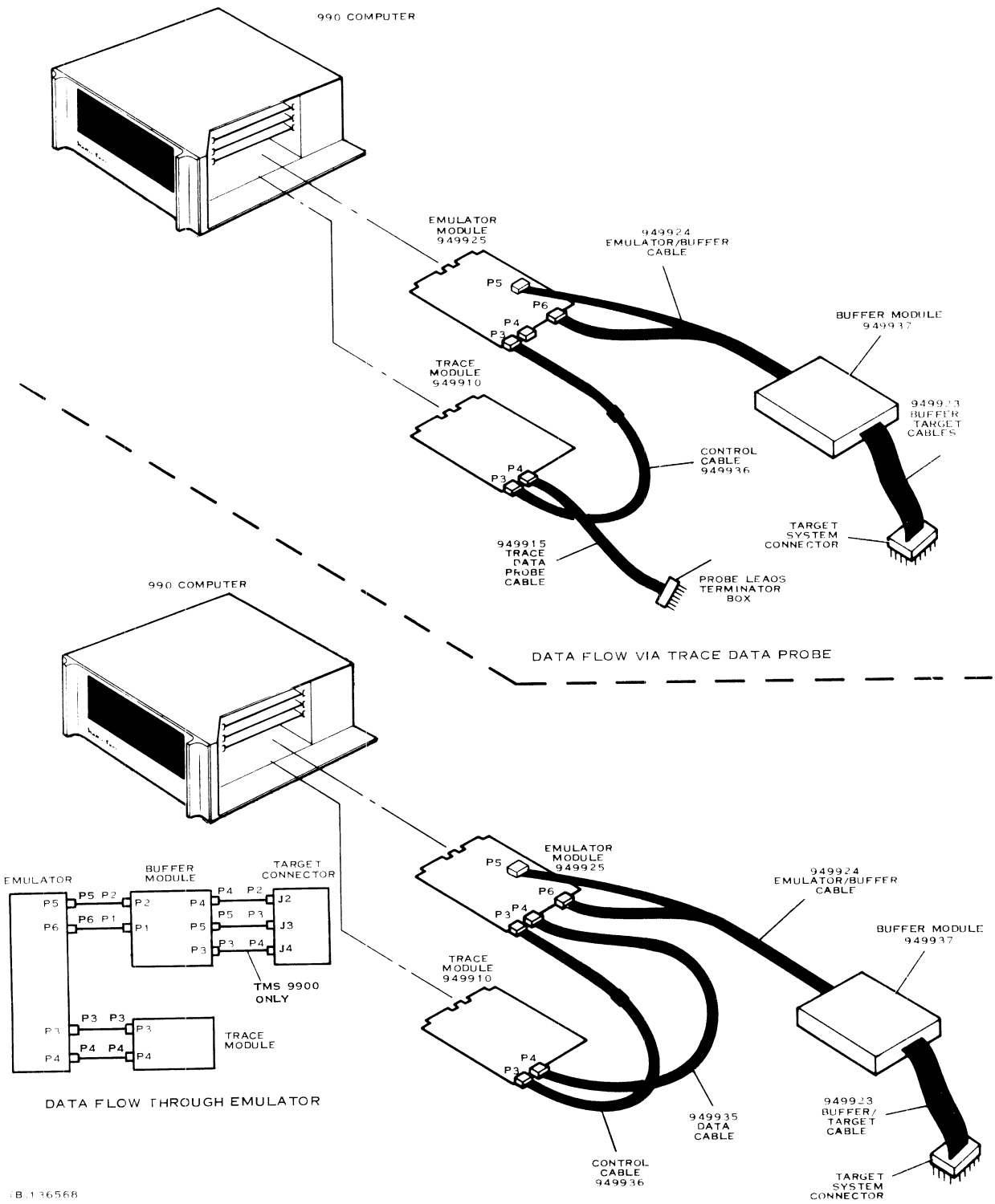


Figure 2-23. Trace Module and Emulator Module Interconnecting Cable

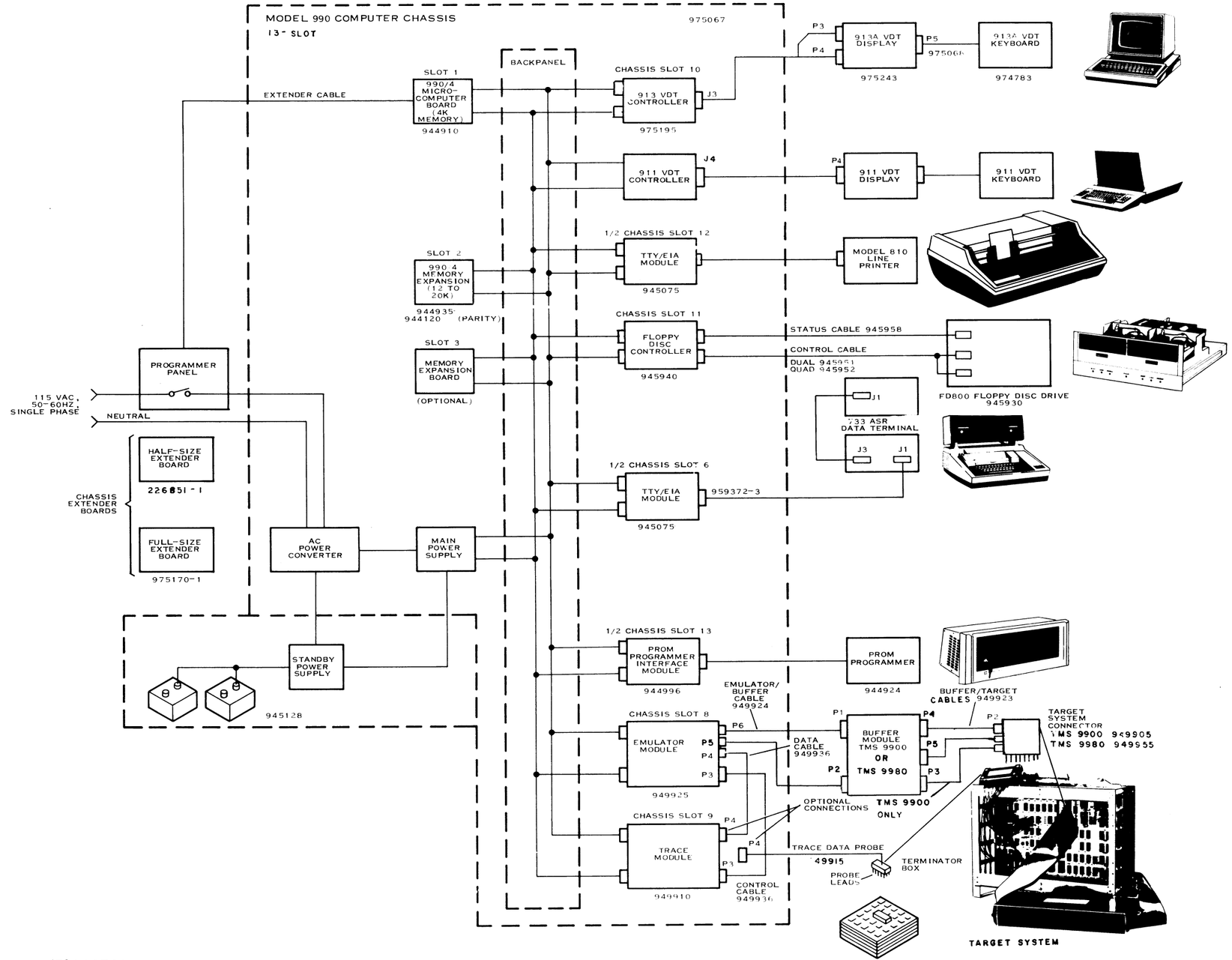


Figure 2-24. Peripheral Devices Available for AMPL Microprocessor Prototyping Laboratory (TX990)

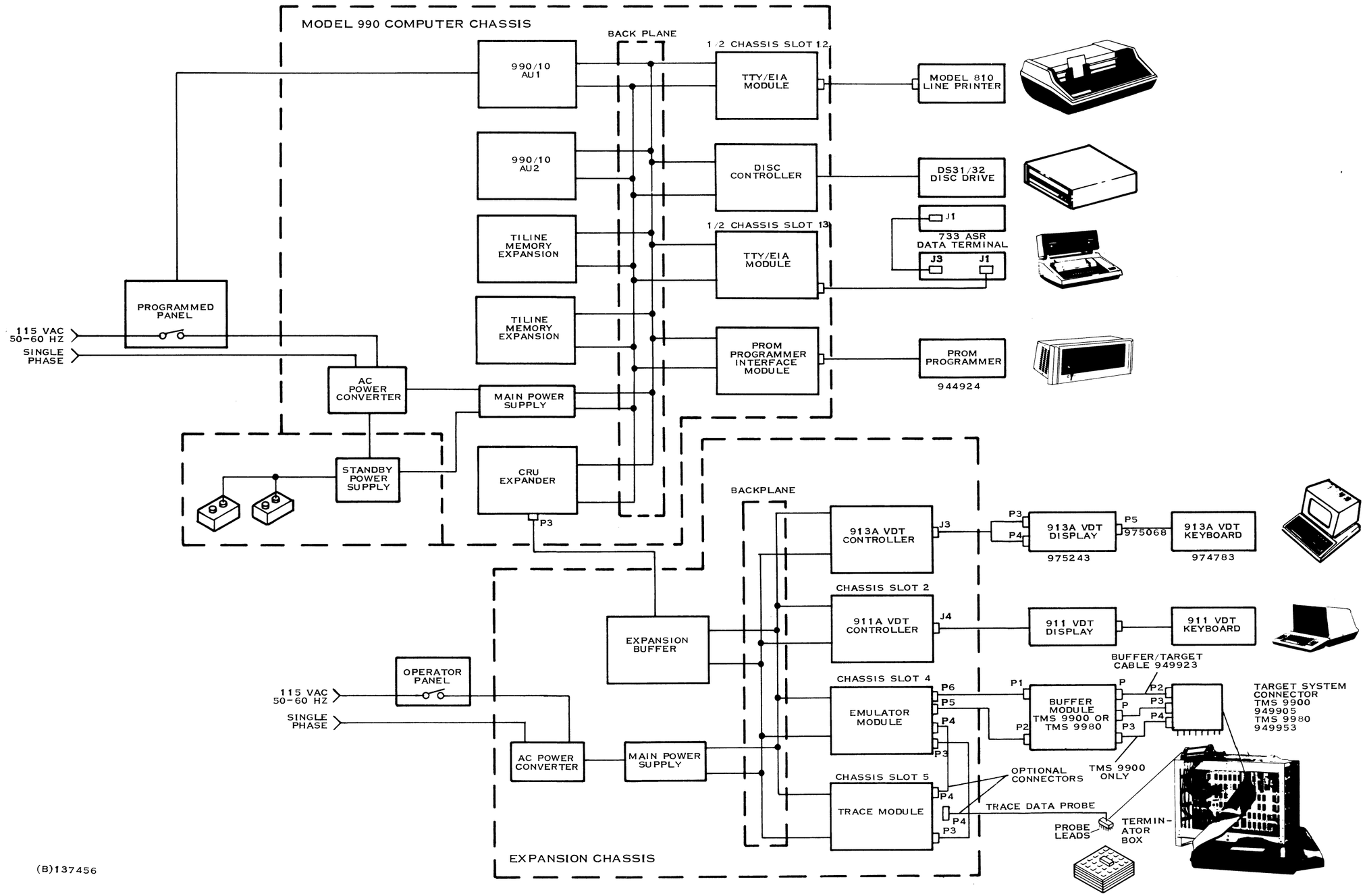


Figure 2-25. Peripheral Devices Available for AMPL Microprocessor Prototyping Laboratory (DX10)

SECTION III

AMPL APPLICATIONS

3.1 INTRODUCTION TO AMPL LANGUAGE

Other parts of this manual describe in detail the syntax and usage of each element in the AMPL language. This section briefly outlines some of the capabilities of the language and describes some actual applications to development problems.

The AMPL language is a complete test language which includes:

- integer arithmetic
- logical operators
- arithmetic (two's complement) and unsigned magnitude relation operators
- execution control and branching structures
- procedure and function declarations which facilitate design of routines and subroutines
- library procedures and functions especially designed to facilitate the gathering and display of test data
- user-controlled display formats – binary, decimal, hexadecimal, octal, ASCII, and 9900 assembly language mnemonics
- emulator/buffer control commands
- trace module control commands
- utility functions.

The AMPL language allows the user to initiate and control each test operation on an interactive basis, or to design and store extensive monitoring and testing procedures which execute without manual intervention.

The examples of AMPL applications in this section utilize the trace module control commands, the emulator control commands, and some of the library procedures.

The emulator and trace module commands are intimately related to the module hardware and the test connections (see Section II before proceeding). Briefly, the emulator and buffer connect a substitute microprocessor into the system under test (target system). The Emulator has access to all the microprocessor input and output lines, including the CRU, address bus, data bus, memory control, clock and hold lines. This gives the emulator control of the target system, and access to the Program Counter (PC), Workspace Pointer (WP), and Status Register (ST).

3.1.1 EMULATOR CONTROL COMMANDS AND VARIABLES. The emulator module must be initialized before any type of emulator access may be executed. The EINT command initializes the emulator, specifying by device name which emulator is active. The EINT command is entered as follows:

```
EINT ('EMU')          Initialize emulator EMU.
```

The example uses a TX990 device name. The proper command for a DX10 system is as follows:

```
EINT ('EM01')         Initialize emulator EM01.
```

Selection gates, controlled by a system variable EUM, allow the microprocessor access to a 8K-byte emulator user memory or to the target system memory. The emulator has a 512-byte memory which may be assigned to the microprocessor or reserved for address tracing, as specified by a system variable ETM.

The emulator has comparison logic which monitors the address bus, the Data Bus In (DBIN) line and the Instruction Acquisition (IAQ) line and gives a comparison output when they match a predetermined value. This predetermined value is controlled by the emulator compare (ECMP) command. The ECMP command has an address field which selects the address to be checked, and a type field which determines whether the compare should be issued for instruction acquisitions only, write operations only, or all accesses to the specified address. The type field can also be used to disable comparisons.

The following examples show how to use the ECMP command.

Command	Compare Condition
ECMP (ADDR, 0040)	Any access to (hex) address 0040.
ECMP (ADDR+IAQ, 0004)	Instruction acquistion access to 0004.
or	
ECMP (IAQ, 0004)	
ECMP (ADDR-DBIN, 04EE)	Write operation performed at 04EE.
or	
ECMP (-DBIN, 04EE)	
ECMP (OFF)	None. Emulator comparison disabled.

The emulator internal comparison can be selected as the triggering event for a breakpoint. This requires two instructions, one which defines the comparison as an "event" and one which selects the event as a breakpoint and specifies action after the breakpoint occurs.

The emulator event (EEVT) command defines an event. To define the emulator comparison as an event, enter EEVT (INT). If EXT is used as a modifier, an event signal connected by the user is selected as the emulator event.

The emulator breakpoint (EBRK) command selects the breakpoint condition and the action to be taken when the breakpoint condition occurs.

The breakpoint condition may be the event (EVT) defined by the EEVT command, or it may be the emulator trace buffer full (FULL) condition, or whichever occurs first. Note that the trace buffer full condition occurs when the number of samples specified in the ETRC command have been taken. This number is between 1 and 256. The OFF keyword disables the breakpoint.

The breakpoint action may be either SELF (stop emulator) or OFF (do not stop emulator). Breakpoint action OFF causes the AMPL software to interrupt the host computer.

The following examples show how to use the EBRK command:

Command	Breakpoint Condition	Breakpoint Action
EBRK (EVT, SELF)	Event	Stop Emulator
EBRK (FULL, OFF)	Trace Completion	Interrupt Host Computer
EBRK (EVT+FULL, SELF)	Event or Trace	Stop Emulator
EBRK (OFF, OFF)	None	None

Figure 3-1 is a flowchart example which shows the steps in setting up an emulator comparison breakpoint with the ECMP, EEVT, and EBRK commands.

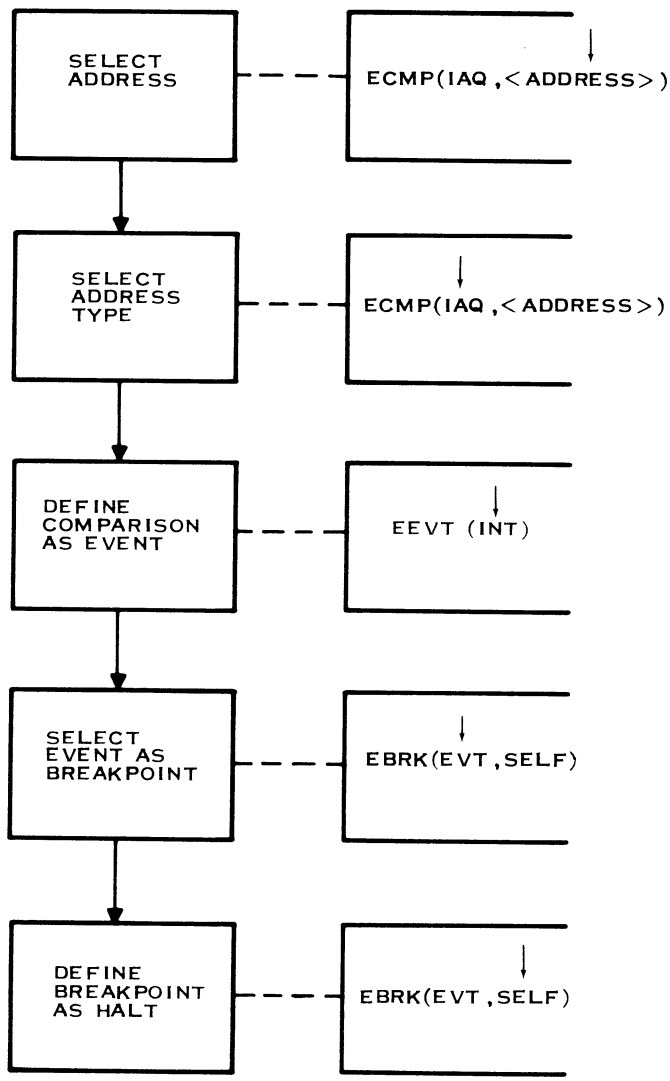


Figure 3-1. Setting Emulator Comparison Breakpoint with ECMP, EEVT, EBRK

The emulator trace command allows selection of data samples to be stored in emulator trace memory. The form of the command is similar to the ECMP command.

The first operand field of ETRC allows recording of all addresses accessed by the microprocessor (ADDR), only instruction acquisitions (ADDR+IAQ) or no trace (OFF). The second operand is a count of the number of samples desired. The count is limited by the 256-word capacity of the emulator trace memory. The third operand allows selection of internal clock from the buffer board or external clock.

The following examples show how to use the ETRC command:

Command	Trace Condition
ETRC (ADDR, 256, INT)	Trace 256 memory bus addresses using internal (target system) clock enable.
ETRC (ADDR+IAQ, 10, EXT)	Trace 10 instruction addresses using external (trace module) clock enable.
ETRC (IAQ, 50, INT)	Trace 50 instruction addresses using internal (target system) clock enable.
ETRC (OFF)	Disable trace logic.

Figure 3-2 is a flowchart which shows how to set up the ETRC command and use the emulator trace memory full as the breakpoint condition.

Note that the emulator trace memory stores memory addresses as they appear on the microprocessor address bus. It does not store the data written to or read from these addresses. The trace module, when interconnected with the emulator by control and data cables, can store the data read from or written to these addresses.

Addresses stored in trace memory are accessed as if they were elements in a single dimension array (ETB) using an index in parentheses. Index zero corresponds to the most recently stored address. Previously stored addresses are accessed using negative index values, for example, ETB(−9). An index of −1 corresponds to the address of the instruction preceding the one at the most recently stored address. More negative index values correspond to previously stored addresses, and the oldest address is accessible at the most negative index value. System variable ETBO contains the index of the oldest address stored, and ETBN contains the index of the newest address stored in the emulator trace memory.

The status of the emulator can be determined by entering EST. The resulting output is a hexadecimal display. The least significant digit gives the status as tabulated in Section V of this manual. There is a library procedure, ESTAT, which gives a plain text status printout.

The emulator trace, if any, and microprocessor operation are initiated and halted by the ERUN and EHLT commands, respectively.

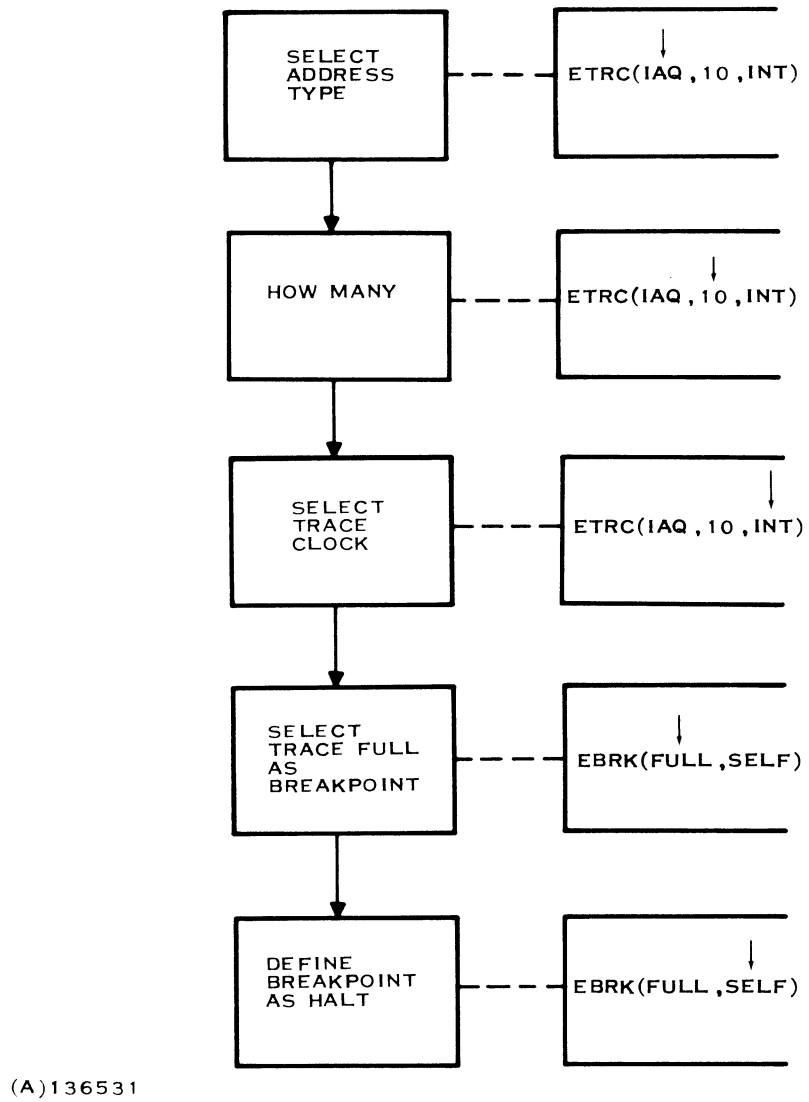


Figure 3-2. Setting Emulator Trace Breakpoint

3.1.2 **TRACE MODULE COMMANDS AND VARIABLES.** The trace module is also capable of tracing a maximum of 256 items, each consisting of up to 20 bits. The module stores the traced items in a 256 X 20 trace memory. The trace module can generate a signal when a specified number of items have been traced. The trace module also compares specified bits of each item to specified values, and generates an event signal when an equal comparison occurs. An event counter counts these signals, and starts counting delays when a specified number of events has occurred. Each item stored in trace memory after the events have been counted constitutes a delay; when the specified number of delays has occurred, the logic generates an event and delay completion signal. The breakpoint can stop the trace, signal the emulator to stop the microprocessor, or interrupt the host computer without stopping either the trace or the microprocessor.

The commands and system variables for the trace module closely parallel those which control the emulator. The TINT command must be entered first to initialize the trace module. The following is an example of a TINT command using the device name in the TX990 system supplied with the prototyping laboratory:

```
TINT ('TRA')          Initialize trace module TRA.
```

For a DX10 system, the proper command is as follows:

```
TINT ('TM01')         Initialize trace module TM01.
```

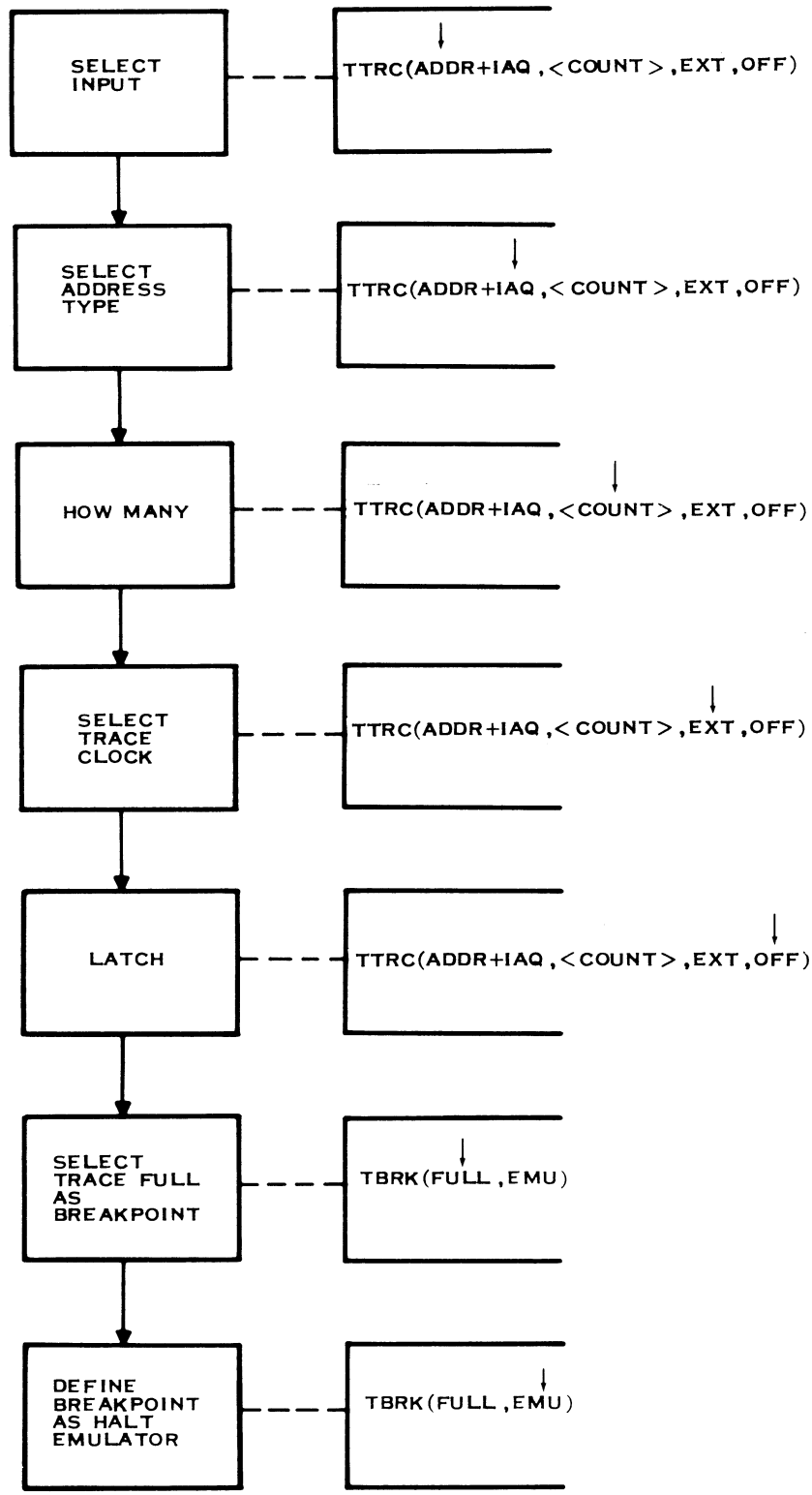
Figure 3-3 is a flowchart of the steps in setting up a trace operation in the trace module.

The TTRC command in the flowchart specifies tracing all addresses that are instruction acquire addresses (IAQ true), generating a completion signal when the count of addresses has been traced. EXT specifies the external clock, which is target system clock from the emulator. The OFF operand could have been omitted because it applies in default when no operand has been entered previously. This operand controls latching the four high-order bits to store glitches, and should be OFF when the data cable is connected to the emulator.

The following examples show how to use the TTRC command:

Command	Trace Condition
TTRC(ADDR+IAQ, 14, EXT, OFF)	Trace 14 memory bus addresses using external (target system) clock. Trace only instruction addresses (IAQ true), and trace with latches off.
TTRC (Q0+Q1, 256, INT, ON)	Trace 256 items to which probe leads are connected, using internal clock. Trace an item only when signals to which probe leads Q0 and Q1 are connected are true. Enable latches on D0 – D3.
TTRC(DATA-DBIN, 25, EXT, OFF)	Trace 25 memory data words using external (target system) clock. Trace only data that was written, and trace with latches off.
TTRC(OFF, 256, EXT, OFF)	Trace 256 words from emulator using external (target system) clock. Trace all data (qualifiers masked off) and trace with latches off.

The TBRK command specifies that the trace completion signal is to be used to generate a breakpoint, and to signal the emulator to halt at completion of the current instruction.



(A)136532

Figure 3-3. Setting Trace Breakpoint in Trace Module

The following examples show how to use the TBRK command:

Command	Breakpoint Condition	Breakpoint Action
TBRK(FULL, SELF)	Trace Completion	Stop tracing, interrupt host computer.
TBRK(EVT, EMU)	Event and Delay Completion.	Request emulator to halt.
TBRK(EVT+FULL, SELF+EMU)	Event and Delay Completion or Trace Completion.	Stop tracing and request emulator to halt.
TBRK(FULL, OFF)	Trace Completion	Interrupt host computer.

Items are stored in trace memory in a manner similar to the manner in which addresses are stored in the emulator trace memory. Items are accessed as if they are elements in a single dimension array (TTB) using an index in parentheses. Two system variables, TTBO and TTBN, contain the index values for the oldest and newest items traced, respectively.

Trace module status can be displayed by entering TST. The status value returned is a 4-digit hexadecimal number. The TSTAT library procedure prints the status in plain text.

In order to save memory space in the host computer, the library procedures are not automatically available when the AMPL system is called up. A COPY command with the file name of the library procedure may be used to bring the procedure into memory. It remains available until the symbol table is cleared by a CLR command. Each library procedure is accompanied by a short tutorial which describes the use of the procedure and the values, if any, required in the procedure call. To print the information, the user must enter a LIST command (paragraph 5.5.2) prior to entering the COPY command. Figure 3-4 is a sample of a copy statement and the resulting printout. The printout may be suppressed by a LIST (OFF) command. In the figure, the TDATA procedure has been copied.

This short section on emulator and trace module commands is intended to provide sufficient background to follow some actual examples of AMPL system use in the development environment. Detailed descriptions of the AMPL commands are contained in Section V.

3.2 PROTOTYPING LABORATORY INITIAL CHECKS

When the buffer module is connected to a target system, it is possible for the target system to interfere with proper operation of the AMPL system. The following initial checks identify these types of troubles so that they may be corrected before proceeding with further testing of the target system:

- Buffer module checkout
- Target system address and data bus checkout
- Target system clock checkout
- Target system memory checkout.

```
? LIST ('LOG')
? COPY(':TDUMPS/PRC')

? ...TITLE:          TDATA:  PRINT TRACE MODULE SAMPLES
? ...REVISION:       03/24/77
? ...
? ...ABSTRACT:
? ...     TDATA PRINTS A SPECIFIED RANGE OF SAMPLES FROM THE TRACE MODULE
? ...TRACE BUFFER.  THE SAMPLES ARE NOT INTERPRETED; ALL TWENTY BITS ARE
? ...PRINTED FOR EACH SAMPLE.
? ...
? ...USAGE:
? ...     TDATA HAS ONE REQUIRED AND ONE OPTIONAL ARGUMENT.  THE FIRST
? ...ARGUMENT IS THE STARTING INDEX TO BE USED.  THE OPTIONAL SECOND
? ...ARGUMENT IS THE ENDING INDEX.  IF IT IS OMITTED, THE ENDING INDEX
? ...IS THE STARTING INDEX PLUS 29, GIVING 30 SAMPLES, IF THERE ARE
? ...THAT MANY IN THE TRACE BUFFER.  TDATA MAY BE CALLED WHENEVER THE
? ...TRACE MODULE IS HALTED.
? ...
? 'DEFINING TDATA' NL
? ...
? PROC TDATA(1,2) BEGIN
1?   IF ARG 0 EQ 2
1?        THEN LOC 1 = ARG 2                    ...ENDING INDEX WAS GIVEN
1?        ELSE LOC 1 = ARG 1 + 29;              ...NOT GIVEN
1?   IF LOC 1 GT TTBN
1?        THEN LOC 1 = TTBN;                    ...LIMIT ENDING INDEX
1?   LOC 2 = 0;                                 ...SET NEWLINE FLAG
1?   ' INDEX     HIGH   LOW            HIGH   LOW';
1?   '             HIGH   LOW';
1?   WHILE ARG 1 LE LOC 1
1?        DO BEGIN
2?             IF LOC 2 MOD 3 EQ 0
2?                  THEN BEGIN
3?                       ARG 1:ND '/ '         ...START A NEW LINE
3?                       END
2?                  ELSE '    '                ...TAB TO NEXT COLUMN
2?             LOC 2 = LOC 2 + 1;              ...STEP THE FLAG COUNTER
2?             TTBH(ARG 1):H;                  ...DISPLAY SAMPLE
2?             TTB(ARG 1):HA;
2?             ARG 1 = ARG 1 + 1;
2?        END;
1? END
```

(A)136533

Figure 3-4. Sample Library Procedure

3.2.1 BUFFER MODULE CHECKOUT. The buffer module checkout procedure should be executed before connecting the AMPL system to a target system to verify that the buffer module is operating satisfactorily. The procedure may also be executed at any time there is a doubt of the proper operation of the buffer module. Perform the following steps:

1. Install the emulator and buffer modules and connect the cables as described in Section II. If the trace module is to be used in testing the target system, connect the trace module as required. Do not connect the target connector to the target system.

2. Set the CLOCK switch on the buffer module to the INTERNAL position.

3. Apply power to the AMPL system.

4. Load the AMPL program as described in paragraph 5.2.

NOTE

The emulator Halt Acknowledge indicator (figure 2-16) should be on at this point. If not, perform maintenance on the emulator module before continuing the test.

5. Enter the EINT command to select the emulator. The following is an example of an EINT command. Substitute the device name of the emulator if EMU or EMDI is not the device name.

? EINT ('EMU')	Select emulator EMU (TX990).
?EINT ('EM01')	Select emulator EM01 (DX10).

6. Enter the following statement to map emulator user memory into the address space:

7. Enter the following command to make library procedure ESTAT available:

? COPY ('.S#SYSLIB.AMPL$LIB.STAT') For DX10.

8. Call library procedure ESTAT. The following is an example of the call and the resulting display:

```
   ? ESTAT:
EMULATOR IS NOT EXECUTING.
EMULATOR TRACE BUFFER IS NOT FULL.
EVENT CONDITIONS ARE NOT SATISFIED.
    35  BREAKPOINTS COUNTED.
    1Ø  ADDRESSES TRACED:   -9  ...   Ø
```

9. The emulator should not be executing at this point. If the emulator is executing perform maintenance on the emulator before proceeding.

10. Enter the following function definition to test target memory:

```
 ? FUNC MEMTST(3,1) BEGIN
1?   LOC 1 = ARG 1
1?   WHILE LOC 1 LOE ARG 2 DO
1?     BEGIN
2?     @LOC 1 = ARG 3
2?     LOC 1 = LOC 1 + 2
2?     END
1?   LOC 1 = ARG 1
1?   WHILE LOC 1 LOE ARG 2 DO
1?     BEGIN
2?     IF @LOC 1 NE ARG 3 THEN RETURN LOC 1
2?     LOC 1 = LOC 1 + 2
2?     END
1?     RETURN -1
1? END
```

11. Enter the following function call to execute function MEMTST. The example shows the return that is displayed by a successful test. If a hexadecimal address is displayed instead of the −1, the test has failed, and maintenance should be performed on the emulator before continuing.

```
? MEMTST(Ø,>1FFE, >1ØFF)
  -1
```

12. Enter the following statement to set the program counter of the target system:

```
? PC = Ø1ØØ
```

13. Enter the following command to start the emulator:

```
? ERUN;
```

14. Call procedure ESTAT again. The following is an example of the call and the resulting display:

```
   ? ESTAT;
EMULATOR IS EXECUTING.
EMULATOR TRACE BUFFER IS NOT FULL.
EVENT CONDITIONS ARE NOT SATISFIED.
 >ØØØØ  BREAKPOINTS COUNTED.
```

15. If the emulator is not running, perform maintenance on the emulator before proceeding. Otherwise, enter the following command:

```
? EHLT;
```

16. Call procedure ESTAT again. The following is an example of the call and the resulting display:

```
   ? ESTAT;
EMULATOR IS NOT EXECUTING.
EMULATOR TRACE BUFFER IS NOT FULL.
EVENT CONDITIONS ARE NOT SATISFIED.
 >0000 BREAKPOINTS COUNTED.
    0  ADDRESSES TRACED.
```

If the display shows that the emulator is still executing, or if there have been error messages other than those caused by entering a command or statement incorrectly, there is a problem with the emulator or buffer module which must be corrected before performing any further tests.

The following error message is typical of the error messages that indicate a buffer module problem:

```
*** ERROR     205  0700  0001
    EMULATOR DSR ERROR;
         02XX  =  ILLEGAL OPERATION,
         05XX  =  MEMORY WRITE ERROR,
         06XX  =  OPERATION TIMED OUT,
         07XX  =  DEVICE ERROR.
```

For a more thorough test of the emulator and buffer, perform the procedure in paragraph 5.3 to execute the hardware demonstration test. The hardware demonstration test also tests the trace module as an option.

3.2.2 **TARGET SYSTEM ADDRESS AND DATA BUS CHECKOUT.** When the target connector is plugged into a target system, faults on the address bus or data bus can cause improper buffer module operation. To detect this type of fault, perform the following steps:

1. Verify that power to the target system is off, and plug the target connector into the target system. If the value of EUM has been altered, repeat step 6 of paragraph 3.2.1.

2. Apply power to the target system.

3. Enter the following function call to execute function MEMTST (if the buffer module checkout has not been performed since the AMPL program was loaded, perform steps 5 and 10 of paragraph 3.2.1 before performing this step):

```
? MEMTST(0,>1FFE,>10FF)
  -1
```

4. If the return from function MEMTST is not −1 as in the example, check the address bus and the data bus on the target system before proceeding. Otherwise, enter the following statement to set the program counter of the target system:

```
? PC = 0100
```

5. Enter the following command to start the emulator:

```
? ERUN;
```

6. Call procedure ESTAT again. The following is an example of the call and the resulting display:

```
   ? ESTAT;
EMULATOR IS EXECUTING.
EMULATOR TRACE BUFFER IS NOT FULL.
EVENT CONDITIONS ARE NOT SATISFIED.
 >0000 BREAKPOINTS COUNTED.
```

7. If the emulator is not running, perform checks on the target system before proceeding. Otherwise, enter the following command:

```
? EHLT
```

8. Call procedure ESTAT again. The following is an example of the call and the resulting display:

```
   ? ESTAT;
EMULATOR IS NOT EXECUTING.
EMULATOR TRACE BUFFER IS NOT FULL.
EVENT CONDITIONS ARE NOT SATISFIED.
 >0000 BREAKPOINTS COUNTED.
     0 ADDRESSES TRACED
```

If the display shows that the emulator is still executing, or if there have been error messages other than those caused by entering a command or statement incorrectly, there is a problem with the target system; a cross, short, ground, or other improper connection on the target system prevents proper operation of the emulator or proper memory access. Correct the problem before attempting any further tests. Error messages shown in paragraph 3.2.1 are typical.

3.2.3 TARGET SYSTEM CLOCK CHECKOUT. After verifying that the emulator works properly when connected to the target system, perform the following steps:

1. Set the CLOCK switch on the buffer module to the TARGET SYS position.

2. Enter the EINT command as in step 5 of paragraph 3.2.1 to select the emulator. An EINT command must be issued following each change of CLOCK switch setting.

3. Enter the following function call to execute function MEMTST. The example shows the return that is displayed by a successful test. If a hexadecimal address is displayed instead of the −1, the test has failed, and the target system clock circuitry should be checked before continuing.

```
? MEMTST(0,>1FFE,>10FF)
  -1
```

4. Enter the following statement to set the program counter of the target system:

```
? PC = 0100
```

5. Enter the following command to start the emulator:

```
? ERUN;
```

6. Call procedure ESTAT. The following is an example of the call and the resulting display:

```
   ? ESTAT;
EMULATOR IS EXECUTING.
EMULATOR TRACE BUFFER IS NOT FULL.
EVENT CONDITIONS ARE NOT SATISFIED.
  0000 BREAKPOINTS COUNTED.
```

7. If the emulator is not running, check the target system clock circuitry before continuing. Otherwise, enter the following command:

```
? EHLT;
```

8. Call procedure ESTAT again. The following is an example of the call and the resulting display:

```
   ? ESTAT;
EMULATOR IS NOT EXECUTING.
EMULATOR TRACE BUFFER IS NOT FULL.
EVENT CONDITIONS ARE NOT SATISFIED.
 >0000 BREAKPOINTS COUNTED.
    0  ADDRESSES TRACED
```

If the display shows that the emulator is still executing, or if there have been error messages other than those caused by entering a command or statement incorrectly, there is a problem with the target system clock. Correct the problem before attempting any further tests. Error messages shown in paragraph 3.2.1 are typical.

3.2.4 TARGET SYSTEM MEMORY CHECKOUT. After verifying that the emulator works properly when connected to the target system and clocked by the target system clock, verify target system memory by performing the following steps:

1. Enter the following statement to map target system memory into addresses 0 through $1FFF_{16}$:

```
? EMU = 0
```

2. Enter a function call to execute function MEMTST. The arguments of the example shown assume that addresses 0 through $1FFF_{16}$ are available as Random Access Memory (RAM) in the target system. The first argument should be the lowest address available and the second argument should be the highest address available within the range of 0 to $FFFF_{16}$. The example shows the return that is displayed by a successful test. If a hexadecimal address is displayed instead of the −1, the test has failed, and target system memory should be checked before continuing.

```
? MEMTST(0,>1FFE,>10FF)
  -1
```

3. Enter the following statement to set the program counter of the target system. The value shown is consistent with the arguments of the call to MEMTST in the preceding example. Use a value within the range of the first two arguments of the call to MEMTST.

```
? PC = 0100
```

4. Enter the following command to start the emulator:

```
? ERUN;
```

5. Call procedure ESTAT. The following is an example of the call and the resulting display:

```
   ? ESTAT;
EMULATOR IS EXECUTING.
EMULATOR TRACE BUFFER IS NOT FULL.
EVENT CONDITIONS ARE NOT SATISFIED.
 >0000  BREAKPOINTS COUNTED.
```

6. If the emulator is not running, check the target system memory before proceeding. Otherwise, enter the following command:

```
? EHLT;
```

7. Call procedure ESTAT again. The following is an example of the call and the resulting display:

```
   ? ESTAT;
EMULATOR IS NOT EXECUTING.
EMULATOR TRACE BUFFER IS NOT FULL.
EVENT CONDITIONS ARE NOT SATISFIED.
     0 BREAKPOINTS COUNTED.
     0  ADDRESSES TRACED
```

If the display shows that the emulator is still executing, or if there have been error messages other than those caused by entering a command or statement incorrectly, there is a problem with target system memory. Correct the problem before attempting any further tests of the target system. Error messages shown in paragraph 3.2.1 are typical.

When the preceding tests of the target system have been performed successfully, the interface between the target system and the buffer module is working properly. Proceed with further tests of the target system.

3.3 PROTOTYPING LABORATORY APPLICATION EXAMPLE

A real-life example of prototyping laboratory application to development problems may help bring together many of the AMPL concepts in this book. This particular application uses only a very small part of the AMPL capability to rapidly solve a problem which resists analysis by conventional diagnostic program, oscilloscope, and logic analyzer techniques.

The target system is a peripheral controller under development for use with the 990 family of computers. The controller uses a TMS 9900 microprocessor and a 2K ROM program to control data transfer to and from a bulk storage device. An onboard RAM provides a scratch pad for the TMS 9900 and a data buffer between the 990 computer and the storage device. This data buffer is managed as a "software FIFO" (first in, first out) buffer by the TMS 9900 microprocessor. The peripheral controller is installed in a 990 computer, and is exercised by a diagnostic program running in the 990 computer.

An intermittent failure in data transfer and device control occurs as the diagnostic tests are performed. This intermittent failure occurs once in 30 or 40 minutes of continuous operation. The diagnostic test printouts indicate a data buffer underflow, which does not pinpoint the problem, even with a conventional logic analyzer.

The solution to the problem is complicated by the real-time operating environment required by the peripheral device. It is not possible to insert breakpoints into the controller program and still maintain control of the peripheral device. For the same reason, it is not possible to single-step through the controller program.

The random nature of the failure indicates the possibility of a noise problem. An oscilloscope is used to trace noise sources and noise sources are suppressed. The result of this effort is a noise-free board with an intermittent failure. The problem is solved by the AMPL prototyping laboratory as described in the following paragraphs.

The trace and emulator modules are connected together in the emulator control and data mode (figure 5-1), and the target connector is plugged into the TMS 9900 socket on the peripheral controller under test.

The TDATA, TEDUMP, TSTAT, and ESTAT procedures are copied from the AMPL system diskette, using the procedures of Appendix H. The "canned" procedures and functions of Appendix H add significantly to ease of operation.

The fault is accompanied by an indication of data buffer underflow, so it seems reasonable to install a breakpoint when underflow occurs, and then to examine the events leading up to the underflow.

Memory location $10D2_{16}$ in the peripheral controller is used as a counter which keeps track of the number of words in the "software FIFO". During a write operation to the bulk storage device, this counter should increment as data is loaded into the FIFO, and decrement as data is transferred out. Under normal control by the read/write program, the counter contents should never go negative.

The emulator is set up to monitor and to give a compare each time an access is performed to FIFO counter address $10D2_{16}$, such as:

```
? ECMP (ADDR,>1ØD2)
```

The trace module is set up to trace data written into that address as follows:

```
? TTRC (DATA+EMU-DBIN, 256,EXT)
```

This command traces data when the emulator compare is true (+EMU) and a write operation (-DBIN) is performed. The emulator compare was previously set to occur at address $10D2_{16}$.

The TTRC command is verified by entering:

```
?TTRC;
```

The response is:

```
+DATA-DBIN+EMU
     256
+EXT
+ØFF
```

This verifies that the TTRC command was accepted and displays the current values of the optional arguments which were not modified by the TTRC command. In the preceding case, these are the default values.

The select trace event command TEVT is checked by entering:

```
?TEVT;
```

The response is:

```
        1
        Ø
+NORM+EACH+INT
```

These are the default parameters and are correct as is. They specify that 1 event will stop the trace, that there will be no delay, triggering will be on the true level, each equal comparison will be counted, and the internal trace module comparison will be used as the event.

The trace module will be set to give a compare when all the trace conditions are met, and the FIFO counter goes negative. This command is entered as follows:

```
?TCMP (DATA+EMU-DBIN,ØFFFF,ØFFFF)
```

The first 3 conditions are the same as used in the TTRC command; that is, when the emulator compares (at address $10D2_{16}$), and a write operation is performed. The second entry is the value $FFFF_{16}$. The mask (third entry, 0FFFF) examines the least significant 16 bits of the 20-bit trace word. If address $10D2_{16}$ contains $FFFF_{16}$, the FIFO counter has underflowed (gone negative).

The breakpoints are set up with the EBRK and TBRK commands as follows:

```
?EBRK (OFF,SELF)
?TBRK (EVT,SELF)
```

and the emulator and trace modules are started with:

```
?TRUN;
?ERUN;
```

Wait for a failure of the peripheral controller. Status of the trace can be checked periodically in the TST variable with the TSTAT procedure. The TSTAT procedure gives a plain language status printout, rather than a hexadecimal code, as shown below:

```
?TSTAT;
 TRACE MODULE IS NOT TRACING
 TRACE BUFFER IS FULL
 EVENT CONDITIONS ARE SATISFIED
  ØØØ1 BREAKPOINTS COUNTED
   256 SAMPLES IN BUFFER: -255 ... Ø
  ØØØ1 EVENTS COUNTED
```

This status printout indicates that the event occurred. A trace dump procedure (TDATA) can be used to print all 256 data values in the trace buffer, as shown in figure 3-5. The INDEX column shows the index into the trace buffer, with the oldest sample printed first. Three samples are printed in each row. The values of interest are the hexadecimal values printed in the LOW columns. Each of these values represents a 16-bit FIFO counter value written into location $10D2_{16}$.

The FIFO counter is at 13_{16} when sample -255 is taken. The counter increments to a maximum value of 50_{16} (sample -194) as the FIFO fills, and decrements to 0 (sample -114) as the FIFO empties. The pattern is shown in figure 3-6. An unexplained reset, from 31_{16} to 0 occurs at sample -1, and the underflow occurs on the next access.

This test with the prototyping laboratory shows that the underflow of the FIFO counter is, in this case, apparently caused by a "phantom reset" of the FIFO counter in the middle of a data transfer.

Now investigation centers on the possible causes of this "phantom reset". Somehow, some instruction is writing a zero value into address $10D2_{16}$. It might be a CLR instruction, a MOV, or a number of other instructions.

Reading the program listing shows several instructions capable of clearing address $10D2_{16}$. These instructions are part of the normal program, but it may be that a code containing a clear instruction is being executed out of sequence.

Set up the emulator module to trace addresses and the trace module to trace data as follows:

```
?ETRC (ADDR,256,EXT)
?TTRC (DATA,256,EXT)
```

Define the emulator event as the internal compare, and the emulator breakpoint as the emulator event, as follows:

```
?EEVT (INT)
?EBRK (EVT,SELF)
```

Set up the emulator compare to occur when the suspected instruction is executed. This is done with an ECMP command:

```
?ECMP (IAQ,>hex address of suspected instruction)
```

Start the trace with the TRUN and ERUN commands, and wait for the failure to occur. If the failure occurs, but the emulator compare does not, the suspect instruction did not cause the failure. If the emulator compare does occur and the failure does not, the instruction has executed without causing the failure. The trace is repeated until the failure occurs, and if no compare accompanies the failure, a new instruction is selected for examination.

If the failure is accompanied by a compare, the trace and emulator buffers are dumped via a TEDUMP procedure. The TEDUMP procedure prints out the emulator trace indexes, the address, the corresponding trace buffer index and the instruction, as shown in figure 3-7.

```
? TDATA(TTBO,TTBN)
INDEX      HIGH   LOW              HIGH   LOW              HIGH   LOW
 -255  /  >0001  >0013  ..        >0001  >0014  ..        >0001  >0015  ..
 -252  /  >0001  >0016  ..        >0001  >0017  ..        >0001  >0018  ..
 -249  /  >0001  >0019  ..        >0001  >001A  ..        >0001  >001B  ..
 -246  /  >0001  >001C  ..        >0001  >001D  ..        >0001  >001E  ..
 -243  /  >0001  >001F  ..        >0001  >0020  .         >0001  >0021  .!
 -240  /  >0001  >0022  ."        >0001  >0023  .#        >0001  >0024  .$
 -237  /  >0001  >0025  .%        >0001  >0026  .&        >0001  >0027  .'
 -234  /  >0001  >0028  .(        >0001  >0029  .)        >0001  >002A  .*
 -231  /  >0001  >002B  .+        >0001  >002C  .,        >0001  >002D  .-
 -228  /  >0001  >002E  ..        >0001  >002F  ./        >0001  >0030  .0
 -225  /  >0001  >0031  .1        >0001  >0032  .2        >0001  >0033  .3
 -222  /  >0001  >0034  .4        >0001  >0035  .5        >0001  >0036  .6
 -219  /  >0001  >0037  .7        >0001  >0038  .8        >0001  >0039  .9
 -216  /  >0001  >003A  .:        >0001  >003B  .;        >0001  >003C  .<
 -213  /  >0001  >003D  .=        >0001  >003E  .>        >0001  >003F  .?
 -210  /  >0001  >0040  .@        >0001  >0041  .A        >0001  >0042  .B
 -207  /  >0001  >0043  .C        >0001  >0044  .D        >0001  >0045  .E
 -204  /  >0001  >0046  .F        >0001  >0047  .G        >0001  >0048  .H
 -201  /  >0001  >0049  .I        >0001  >004A  .J        >0001  >004B  .K
 -198  /  >0001  >004C  .L        >0001  >004D  .M        >0001  >004E  .N   <- BUFFER STARTS EMPTYING
 -195  /  >0001  >004F  .O        >0001  >0050  .P        >0001  >004F  .O
 -192  /  >0001  >004E  .N        >0001  >004D  .M        >0001  >004C  .L
 -189  /  >0001  >004B  .K        >0001  >004A  .J        >0001  >0049  .I
 -186  /  >0001  >0048  .H        >0001  >0047  .G        >0001  >0046  .F
 -183  /  >0001  >0045  .E        >0001  >0044  .D        >0001  >0043  .C
 -180  /  >0001  >0042  .B        >0001  >0041  .A        >0001  >0040  .@
 -177  /  >0001  >003F  .?        >0001  >003E  .>        >0001  >003D  .=
 -174  /  >0001  >003C  .<        >0001  >003B  .;        >0001  >003A  .:
 -171  /  >0001  >0039  .9        >0001  >0038  .8        >0001  >0037  .7
 -168  /  >0001  >0036  .6        >0001  >0035  .5        >0001  >0034  .4
 -165  /  >0001  >0033  .3        >0001  >0032  .2        >0001  >0031  .1
 -162  /  >0001  >0030  .0        >0001  >002F  ./        >0001  >002E  ..
 -159  /  >0001  >002D  .-        >0001  >002C  .,        >0001  >002B  .+
 -156  /  >0001  >002A  .*        >0001  >0029  .)        >0001  >0028  .(
 -153  /  >0001  >0027  .'        >0001  >0026  .&        >0001  >0025  .%
 -150  /  >0001  >0024  .$        >0001  >0023  .#        >0001  >0022  ."
 -147  /  >0001  >0021  .!        >0001  >0020  .         >0001  >001F  ..
 -144  /  >0001  >001E  ..        >0001  >001D  ..        >0001  >001C  ..
 -141  /  >0001  >001B  ..        >0001  >001A  ..        >0001  >0019  ..
 -138  /  >0001  >0018  ..        >0001  >0017  ..        >0001  >0016  ..
 -135  /  >0001  >0015  ..        >0001  >0014  ..        >0001  >0013  ..
 -132  /  >0001  >0012  ..        >0001  >0011  ..        >0001  >0010  ..
 -129  /  >0001  >000F  ..        >0001  >000E  ..        >0001  >000D  ..
 -126  /  >0001  >000C  ..        >0001  >000B  ..        >0001  >000A  ..
 -123  /  >0001  >0009  ..        >0001  >0008  ..        >0001  >0007  ..
 -120  /  >0001  >0006  ..        >0001  >0005  ..        >0001  >0004  ..
 -117  /  >0001  >0003  ..        >0001  >0002  ..        >0001  >0001  ..   <- BUFFER REFILL
 -114  /  >0001  >0000  ..        >0001  >0000  ..        >0001  >0001  .!
 -111  /  >0001  >0002  ..        >0001  >0003  ..        >0001  >0004  ..
                  ^ NORMAL BUFFER EMPTY                    ^ NORMAL BUFFER EMPTY
                                                           { BUFFER FULL
```

(B)136534 (1/2)

Figure 3-5. TDATA Printout for Prototyping Lab Example (Sheet 1 of 2)

```
-105  / >0001  >0008  ..      >0001  >0009  ..      >0001  >000A  ..
-102  / >0001  >000B  ..      >0001  >000C  ..      >0001  >000D  ..
 -99  / >0001  >000E  ..      >0001  >000F  ..      >0001  >0010  ..
 -96  / >0001  >0011  ..      >0001  >0012  ..      >0001  >0013  ..
 -93  / >0001  >0014  ..      >0001  >0015  ..      >0001  >0016  ..
 -90  / >0001  >0017  ..      >0001  >0018  ..      >0001  >0019  ..
 -87  / >0001  >001A  ..      >0001  >001B  ..      >0001  >001C  ..
 -84  / >0001  >001D  ..      >0001  >001E  ..      >0001  >001F  ..
 -81  / >0001  >0020  .       >0001  >0021  .!      >0001  >0022  ."
 -78  / >0001  >0023  .#      >0001  >0024  .$      >0001  >0025  .%
 -75  / >0001  >0026  .&      >0001  >0027  .'      >0001  >0028  .(
 -72  / >0001  >0029  .)      >0001  >002A  .*      >0001  >002B  .+
 -69  / >0001  >002C  .,      >0001  >002D  .-      >0001  >002E  ..
 -66  / >0001  >002F  ./      >0001  >0030  .0      >0001  >0031  .1
 -63  / >0001  >0032  .2      >0001  >0033  .3      >0001  >0034  .4
 -60  / >0001  >0035  .5      >0001  >0036  .6      >0001  >0037  .7
 -57  / >0001  >0038  .8      >0001  >0039  .9      >0001  >003A  .:
 -54  / >0001  >003B  .;      >0001  >003C  .<      >0001  >003D  .=
 -51  / >0001  >003E  .>      >0001  >003F  .?      >0001  >0040  .@
 -48  / >0001  >0041  .A      >0001  >0042  .B      >0001  >0043  .C
 -45  / >0001  >0044  .D      >0001  >0045  .E      >0001  >0046  .F
 -42  / >0001  >0047  .G      >0001  >0048  .H      >0001  >0049  .I
 -39  / >0001  >004A  .J      >0001  >004B  .K      >0001  >004C  .L
 -36  / >0001  >004D  .M      >0001  >004E  .N      >0001  >004F  .O
 -33  / >0001  >0050  .P      >0001  >004F  .O      >0001  >004E  .N
 -30  / >0001  >004D  .M      >0001  >004C  .L      >0001  >004B  .K
 -27  / >0001  >004A  .J      >0001  >0049  .I      >0001  >0048  .H   <- BUFFER STARTS EMPTYING
 -24  / >0001  >0047  .G      >0001  >0046  .F      >0001  >0045  .E
 -21  / >0001  >0044  .D      >0001  >0043  .C      >0001  >0042  .B
 -18  / >0001  >0041  .A      >0001  >0040  .@      >0001  >003F  .?
 -15  / >0001  >003E  .>      >0001  >003D  .=      >0001  >003C  .<
 -12  / >0001  >003B  .;      >0001  >003A  .:      >0001  >0039  .9
  -9  / >0001  >0038  .8      >0001  >0037  .7      >0001  >0036  .6
  -6  / >0001  >0035  .5      >0001  >0034  .4      >0001  >0033  .3
  -3  / >0001  >0032  .2      >0001  >0031  .1      >0001  >0000  ..
   0  / >0001  >FFFF  ..
?

                    UNDERFLOW        BUFFER FULL      DISCONTINUITY
                                                      BUFFER STARTS EMPTYING

(A)136534 (2/2)
```

Figure 3-5. TDATA Printout for Prototyping Lab Example (Sheet 2 of 2)

The 256 sample trace has been shortened in the figure. The emulator compare occurs at emulator index -3. Prior history is checked by reading up the printout from index 0 toward -256. At -38, a level 7 interrupt (address $001C_{16}$) is reported. However, the peripheral controller is not supposed to have an interrupt on level 7. The lowest priority interrupt should be a 4 millisecond timer at level 4. The interrupt priority assignments are at fault in this case. The valid level 4 interrupt is not being serviced in time. It is causing an interrupt request, but by the time the priority encoder is tested, the valid interrupt has disappeared, leaving a default hexadecimal 7 on the priority encoder output. The code associated with level 7 interrupts is causing a FIFO underflow indication, and masking the real problem, occasional failure to service the timing interrupt.

A reassignment of interrupt priorities assures that the timing interrupt could not be masked for an excessive length of time, and completely cures the problem.

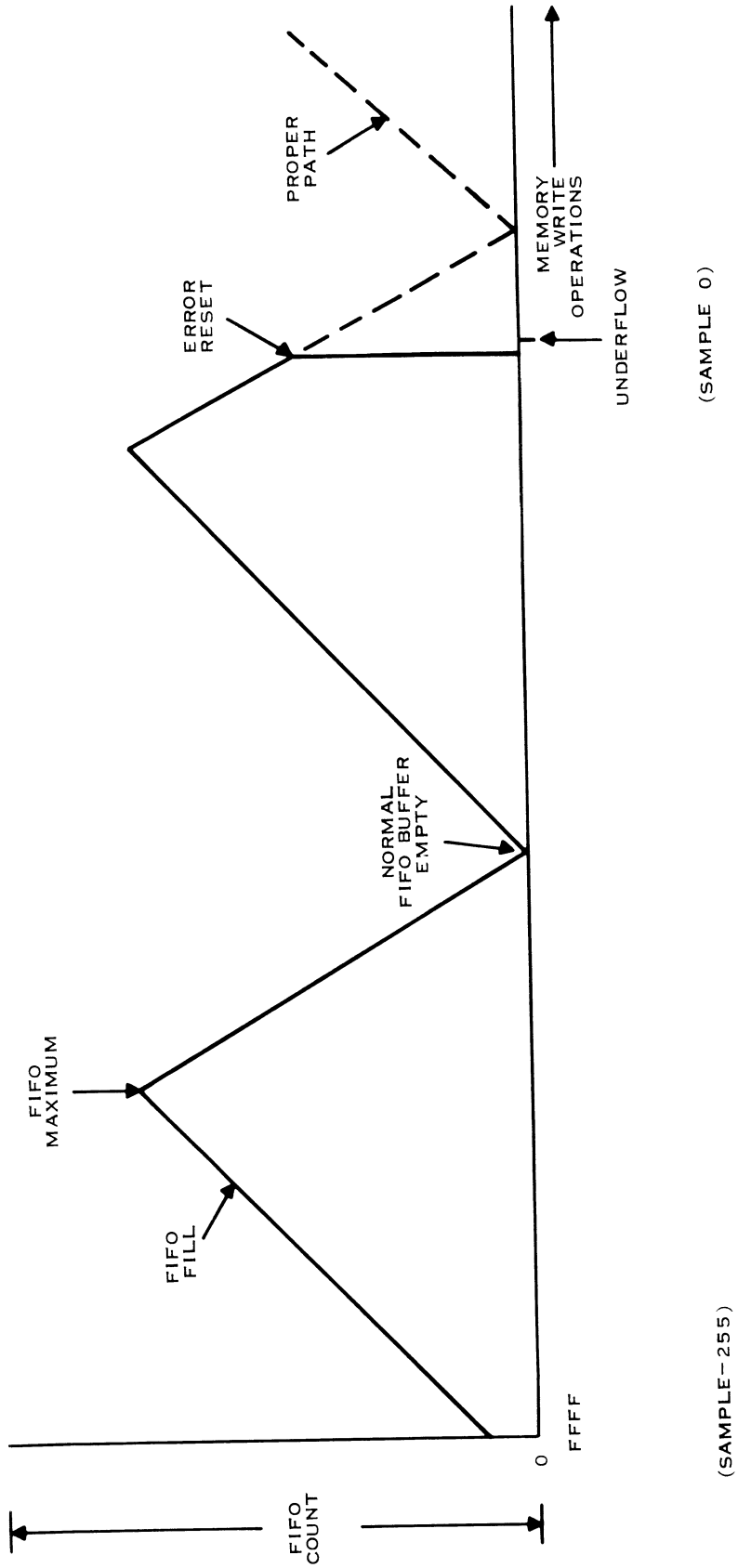


Figure 3-6. Observed FIFO Counter Operation

```
  ? TSTAT
TRACE MODULE IS TRACING.
TRACE BUFFER IS FULL.
EVENT CONDITIONS ARE SATISFIED.
 >0000  BREAKPOINTS COUNTED.
  ? ESTAT
EMULATOR IS NOT EXECUTING.
EMULATOR TRACE BUFFER IS FULL.
EVENT CONDITIONS ARE SATISFIED.
 >0001  BREAKPOINTS COUNTED.
   256  ADDRESSES TRACED:  -255  ...      0

? TEDUMP(ETBO,ETBN)
```

EMU INDEX	ECMP?	ADDR	TRA INDEX	DATA(HEX,ASC)	INTERPRETATION
-255		>102A	-55	>0555 .U	WRITE >0555
-254		>0C0E	-54	>1F05 ..	TB >0005
-253		>1038	-53	>00A0 .	READ >00A0
-252		>0C10	-52	>160B ..	JNE $+>0018
-251		>0C12	-51	>3105 1.	LDCR R5,4
-250		>102A	-50	>0555 .U	READ >0555
-44		>1036	156	>0BB2 .2	READ >0BB2
-43		>0BB2	157	>0380 ..	READ >0380
-42		>0BB2	158	>0380 ..	RTWP
-41		>103E	159	>C406 D.	READ >C406
-40		>103C	160	>0366 .F	READ >0366
-39		>103A	161	>1000 ..	READ >1000
INTERRUPT OCCURS → -38		>001C	162	>10A0 .	READ >10A0
-37		>10BE	163	>C406 D.	WRITE >C406
-36		>10BC	164	>0366 .F	WRITE >0366
-35		>10BA	165	>1000 ..	WRITE >1000
-34		>001E	166	>01CA .J	READ >01CA
-33		>01CA	167	>0205 ..	LI R5,>0000
-32		>01CC	168	>0047 .G	READ >0047
-31		>10AA	169	>0047 .G	WRITE >0047
-30		>01CE	170	>0460 .`	B @>0000
-29		>01D0	171	>015C ./	READ >015C
-28		>015C	172	>06C5 .E	READ >06C5
-27		>015C	173	>06C5 .E	SWPB R5
-26		>10AA	174	>0047 .G	READ >0047
-25		>10AA	175	>4700 G.	WRITE >4700
-24		>015E	176	>D805 X.	MOVB R5,@>0000
-23		>10AA	177	>4700 G.	READ >4700
-22		>0160	178	>10E3 .C	READ >10E3
-21		>10E2	179	>0800 ..	READ >0800
-20		>10E2	180	>0847 .G	WRITE >0847
-19		>0162	181	>5820 X	SZCB @>0000,@>0000

(A)136536 (1/2)

Figure 3-7. TEDUMP Printout for AMPL Example (Sheet 1 of 2)

```
 -18          >0164     182     >0029  .)        READ  >0029  .)
 -17          >0028     183     >5022  P"        READ  >5022  P"
 -16          >0166     184     >10E2  .B        READ  >10E2  .B
 -15          >10E2     185     >0847  .G        READ  >0847  .G
 -14          >10E2     186     >0847  .G        WRITE >0847  .G
 -13          >0168     187     >F820  .      SOCB @>0000,@>0000
 -12          >016A     188     >002E  ..        READ  >002E  ..
 -11          >002E     189     >D000  P.        READ  >D000  P.
 -10          >016C     190     >10E2  .B        READ  >10E2  .B
  -9          >10E2     191     >0847  .G        READ  >0847  .G
  -8          >10E2     192     >D847  XG        WRITE >D847  XG
  -7          >016E     193     >04E0  .`     CLR  @>0000
  -6          >0170     194     >10E0  .@        READ  >10E0  .@
  -5          >10E0     195     >7000  P.        READ  >7000  P.
  -4          >10E0     196     >0000  ..        WRITE >0000  ..
  -3    *ECMP >0172     197     >04E0  .`     CLR  @>0000
  -2          >0174     198     >10D2  .R        READ  >10D2  .R
  -1          >10D2     199     >0031  .1        READ  >0031  .1
   0          >10D2     200     >000   ..        WRITE >0000  ..
?

'A)136536 (2/2)
```

Figure 3-7. TEDUMP Printout for AMPL Example (Sheet 2 of 2)

3.4 TRACE PROBE EXAMPLE

The trace module may be used independently of the emulator for general tracing operations. For example, a tape cassette controller is experiencing data readback failures during diagnostic-testing. The diagnostic shows that a hexadecimal "05" is being read back occasionally from a record of all "04" characters. The problem appears to be in the data, rather than in the operation of the peripheral controller program.

Figure 3-8 is a simplified diagram of the controller "front end". If the error is present at the FIFO output, the error must be ahead of (or in) the FIFO. This would eliminate the program and all the logic downstream from the FIFO.

To determine if the error is present at the FIFO output, attach a DIP test adapter onto the 3341 FIFO device and attach the trace probe leads to the adapter pins corresponding to FIFO1, 2, 3, and 4. Any 4 of the 20 trace data leads could be used but they should represent one hex character, with LSB to MSB order preserved. For this example, the four least significant leads, D16-D19, are used. The CLOCK lead is attached to the shift out clock pin. The GND lead is connected to logic ground.

The other end of the trace probe cable is connected to the trace module, replacing the emulator/trace data cable.

The emulator will only be used to supply a TMS9900 microprocessor to the target system.

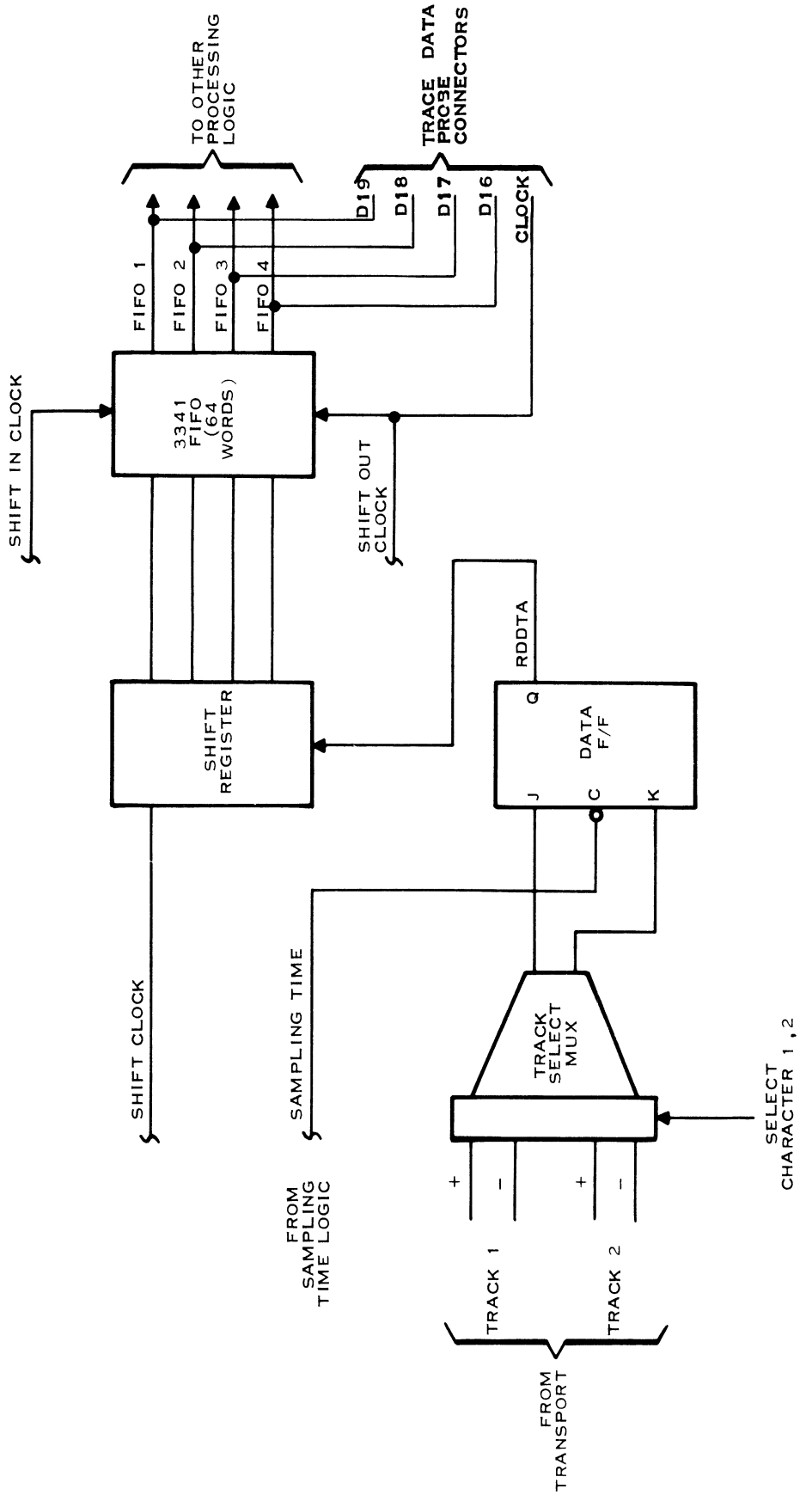


Figure 3-8. Target System Front End

Set up the trace as follows:

```
?TTRC (ØFF,126,+EXT,+ØFF)
```

The qualifiers, Q0-Q3, are not required, *126* samples are to be stored, and the rising edge of the external (FIFO shift out) clock is used. The latch mode is disabled.

Set up a comparison to provide a breakpoint when the error occurs, as follows:

```
? TCMP(ØFF,5,ØF)
```

Specify 1 test event, and delay of 128 as follows:

```
?TEVT (1,128)
```

This puts the event in the middle of the 256-word trace buffer, so data on either side of the event may be examined.

The trace breakpoint is defined as:

```
?TBRK (EVT,EMU)
```

and the trace is initiated by:

```
?TRUN;
?ERUN;
```

After the target system read operation is started, the status is checked and the trace module trace buffer is printed out by TDATA. The printout, figure 3-9, shows that the faulty data is appearing at the FIFO output, so the fault is ahead of this point. The program and about 90% of the logic have been exonerated by this quick test, which takes longer to describe than to perform.

3.5 TMS 9980 EXAMPLES

The application examples are for target systems that use the TMS 9900 microprocessor. They may be used on a target system that uses a TMS 9980 microprocessor, but the addresses and data stored in the memories of the emulator are different. To understand the difference, consider the difference in the microprocessors.

The most significant difference between the microprocessors is that the TMS 9900 has a 16-bit data bus, and accesses a word of memory at each memory access, and the TMS 9980 has an 8-bit data bus and accesses a byte of memory at each memory access. The TMS 9980 performs two memory accesses for each word required. In the emulator (in which memory addresses are traced) a trace operation stores the addresses of each byte stored. When the trace module is tracing data supplied by the emulator, the data is supplied on a 16-bit bus. During a memory access of an even address, the most significant byte of the bus contains the data being transferred to or from memory, and the least significant byte contains random data. During a memory access of an odd address, the least significant byte of the bus contains the data being transferred to or from memory, and the most significant byte retains its previous contents. Thus the entire word as it is stored in memory is present on the 16-bit data bus during a memory access to an odd address.

```
TCMP(OFF,5,OF)

  ? TRUN

  ? ERUN

  ? TSTAT
TRACE MODULE IS NOT TRACING.
TRACE BUFFER IS FULL.
EVENT CONDITIONS ARE SATISFIED.
 >0000  BREAKPOINTS COUNTED.
   256  SAMPLES IN BUFFER:  -127  ...   128
 >0001  EVENTS COUNTED.
 ? TDATA(TTBO,TTBN)
INDEX      HIGH   LOW              HIGH   LOW              HIGH   LOW
 -127   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
 -124   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
 -121   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
 -118   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
 -115   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
 -112   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
 -109   / >0000  >0004  ..        >0000  >0000  ..        >0000  >0000  ..
 -106   / >0000  >000E  ..        >0000  >000A  ..        >0000  >000A  ..
 -103   / >0000  >000C  ..        >0000  >0000  ..        >0000  >0000  ..
 -100   / >0000  >0003  ..        >0000  >0000  ..        >0000  >000C  ..
  -97   / >0000  >000A  ..        >0000  >000A  ..        >0000  >0008  ..
  -94   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
  -91   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
    I                                                                  I
  -22   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
  -19   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
  -16   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
  -13   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
  -10   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0000  ..
   -7   / >0000  >0000  ..        >0000  >000E  ..        >0000  >000A  ..
   -4   / >0000  >000A  ..        >0000  >000C  ..        >0000  >0000  ..
   -1   / >0000  >0000  ..        >0000  >0005  ..        >0000  >0000  ..
    2   / >0000  >000C  ..        >0000  >000A  ..        >0000  >000A  ..
    5   / >0000  >0008  ..        >0000  >0004  ..        >0000  >0004  ..
    8   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
   11   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
   14   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
   17   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
   20   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
   23   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
    I                                                                  I
  116   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
  119   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
  122   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
  125   / >0000  >0004  ..        >0000  >0004  ..        >0000  >0004  ..
  128   / >0000  >0004  ..
?

(A)136538
```

Figure 3-9. Trace Probe Example

When the examples as stated for the TMS 9900 microprocessor are executed in the TMS 9980 microprocessor, the emulator trace memory contains redundant addresses and the trace module memory contains meaningful information in alternate words. Qualifier Q0 of the trace module is connected to the least significant address bit, true on even memory addresses. By using qualifier DATA-Q0 in the TTRC commands, the trace module memory stores only alternate words, when the data stored consists of valid memory words. This data corresponds to the data traced in the TMS 9900 operation. By specifying EXT in the ETRC command, the clock for tracing in the emulator is enabled by the trace module clock (qualified by Q0), and odd addresses are traced. Thus the capabilities of the AMPL Microprocessor Prototyping Laboratory are as effective in developing TMS 9980 systems, but the commands are somewhat different. Details of the differences in the commands are described in Section V.

SECTION IV

AMPL MICROPROCESSOR PROTOTYPING LANGUAGE

4.1 INTRODUCTION

Statements entered by the user to control the emulator, trace module, and target system are written in the AMPL Microprocessor Prototyping Language. AMPL elements are used as operands of the user commands described in Section V.

Statements entered at the operator console are checked for syntax and translated into an intermediate code. The intermediate code is processed by the AMPL Interpreter, which performs the operation specified by the statement.

4.2 LANGUAGE ELEMENTS

The characters of the AMPL character set are combined to form the following language elements:

- Constants
- Symbols

4.2.1 CHARACTER SET. The set of ASCII characters recognized by AMPL consists of the 26 letters (A-Z), the ten numerals (0-9), and the following special characters: space, ", ', (,), *, /, +, -, =, ., , , . ., ?, :, ::, ;, <, >, !, @, #.

NOTE

AMPL accepts only upper case characters. If a 911 VDT is being used as the console, push the key labeled UPPER CASE LOCK. This key is in the upper left corner of the keyboard.

These special characters are used as follows:

space	To separate keywords and symbols
"	To enclose character constants
'	To enclose character strings
( )	To enclose argument lists and subexpressions
*/+-	To indicate arithmetic operations
=	To assign a value to a variable
.	To identify load module symbols

,	To separate elements in argument lists
..	To identify the following characters as a comment
?	To request user input, and to specify a display and modify operation
:	To concatenate a format specification to an expression to be displayed
::	To separate the expression from the statement in a CASE statement
;	To terminate a display and modify operation, and (optionally) to separate statements within a procedure, function, or compound statement
<	To identify the following digits as a binary constant
>	To identify the following digits as a hexadecimal constant
!	To identify the following digits as an octal constant
@	To indicate an indirect target memory access
#	To enclose single source code lines for assembly

4.2.2 **CONSTANTS.** The user may specify a constant in any of the following formats:

- Decimal
- Hexadecimal
- Octal
- Binary
- Character
- Instruction

The expressions required in AMPL statements may include one or more constants, alone or combined with other elements by means of arithmetic or logical operators.

4.2.2.1 **Decimal Integer Constants.** A decimal integer constant is entered as a string of decimal digits (0 through 9), the first of which may not be 0. The range of values of decimal integers is 1 through 32767. The following are valid decimal constants:

Constant	Value
1000	1000 or $3E8_{16}$
23	23 or 17_{16}

The following is an invalid decimal constant:

0250	First digit is zero.

4.2.2.2 Hexadecimal Integer Constants. A hexadecimal integer constant is entered as a string of hexadecimal digits (0 - 9, A - F). Either the first digit must be zero or the string must be preceded by a greater-than character (>). The range of values of hexadecimal integers is 0 through $FFFF_{16}$. The following are valid hexadecimal constants:

Constant	Value
0F	F_{16} or 15
0250	250_{16} or 592
0	0
>100	100_{16} or 256
>28	28_{16} or 40
>F3	$F3_{16}$ or 243

4.2.2.3 Octal Integer Constants. An octal integer constant is entered as a string of octal digits (0 - 7) preceded by an exclamation point (!). The range of values of octal integers is zero through 177777_8. The following are valid octal constants:

Constant	Value
!173	173_8 or $7B_{16}$ or 123
!2620	2620_8 or 590_{16} or 1424
!14	14_8 or C_{16} or 12

4.2.2.4 Binary Integer Constants. A binary integer constant is entered as a string of binary digits (0 or 1) preceded by a less-than character (<). The range of values of binary integers is 0 through 1111111111111111_2. The following are valid binary constants:

Constant	Value
<101101	101101_2 or $2D_{16}$ or 45
<100	100_2 or 4
<010011010010	10011010010_2 or $4D2_{16}$ or 1234

4.2.2.5 **Character Constants.** A character constant is entered as a string of one or two characters enclosed in quotation marks. The characters are represented internally as eight-bit ASCII characters with the leading bit set to zero. When a character constant consists of one character, the character is right-justified and leading zeros are placed in the internal representation. The following are valid character constants:

Constant	Value
"AB"	4142_{16}
"C"	0043_{16}
"#/"	$232F_{16}$

4.2.2.6 **Instruction Constants.** An instruction constant is entered as a source code instruction line enclosed in pound signs (#). The source code line consists only of an operation field and an operand field. No label field is entered, and comments are not entered within the pound signs. (Comments may be entered as in any AMPL statement as described in paragraph 4.2.6.) Only instruction operation codes and pseudo-instruction operation codes RT and NOP may be used. Workspace registers must be specified as R0 through R15, and the operand field of jump instructions must be program-counter relative; i.e., $+4, $-2, etc.

The operand field of an instruction constant may not contain a label or a user symbol. It may contain a load module symbol. A load module symbol (paragraph 4.2.3.3) is a symbol in a program that has been loaded into target system memory.

An instruction constant may be used in an assign statement (paragraph 4.6.2) or in the display and modify mode (paragraph 4.6.3.5). In the first case, the internal form of an instruction constant is a 16-bit word containing the machine instruction (or the first word of the machine instruction) derived from the source code instruction. Two- and three-word instructions require that the literal value or address or addresses be entered as integer constants or variables, even though the value or address has been specified in the source code. The AMPL single line assembler requires the operands to correctly assemble the addressing modes in the first instruction word, but does not assemble second and third words of instructions.

When an instruction constant is used in the display and modify mode the complete instruction (one, two or three 16-bit words) is assembled and placed in target system memory. Examples of use of instruction constants in this mode are shown in paragraph 4.6.3.5.

The following are examples of instruction constants used other than in the display and modify mode:

`#MOV *RØ+,*R1#`	A move instruction that moves the word at the address in workspace register 0 to the address in workspace register 1, and increments workspace register 0 by two. The value of the constant is $C470_{16}$.

`#LI RØ, >FFFF#`	A load immediate instruction that places a value in workspace register 0. The value of the instruction constant is 0200_{16}. The value to be placed in workspace register 0 is the contents of the word that follows the instruction constant.
`#MOV @Ø1ØØ,@Ø3ØØ(R1)#`	A move instruction that moves the word at location 100_{16} to location 300_{16} indexed by workspace register R1. The value of the instruction constant is $C860_{16}$. The actual addresses for the instruction are the contents of the two words that follow the address constant.
`#JMP $-4#`	A jump instruction. The value of the instruction constant is $10FD_{16}$.

The following are examples of invalid instruction constants:

`#JMP`	`Ø1Ø4#`	Invalid. Jump instructions may not specify addresses.
`#LOOP`	`MOV *RØ+,*R1`	Invalid. The label field may not be used.
`#DEF`	`BUFF,MSG1#`	Invalid. Assembler directives may not be used.

4.2.3 **SYMBOLS.** The symbols used in AMPL may be of any of the following types:

- User symbols
- System symbols
- Load module symbols

The expressions required in AMPL statements may include one or more symbols, alone or combined with other elements by means of arithmetic or logical operators.

4.2.3.1 **User Symbols.** A user symbol is a string of alphanumeric characters, the first of which must be alphabetic. If more than six characters are entered, the first six characters are stored internally as the symbol and the remaining characters are ignored. The software prints a warning message. The user defines a symbol to represent a location within a program, a mask, a counter, a constant, a variable, a switch, or a flag. The following are valid user symbols:

`LBL1`	
`STOPHERE`	Internal representation of this symbol consists of STOPHE (first six characters).

The following are not valid user symbols for the reasons indicated:

`1LOC`	Begins with a numeral.
`A!BC`	Contains a special character.

User symbols are global symbols; i.e., they continue to apply until the symbol table is cleared. User symbols must be unique with respect to each other and with respect to system symbols and reserved words (Appendixes D and E).

4.2.3.2 **System Symbols.** A system symbol is a string of up to four alphanumeric characters that is predefined in AMPL software. System symbols are assigned to the following:

- Workspace registers and PC, WP and ST of the target system
- Emulator bits
- Trace module constants
- System values and masks
- Names of user commands
- Keywords used as operands of user commands.

The user may not delete system symbols, nor add symbols to the set of system symbols. The user may alter the values of some system symbols. The system symbols, which are reserved words, are listed in Appendix E. Use of specific system symbols is described with the AMPL statements and user commands to which the symbols apply.

4.2.3.3 **Load Module Symbols.** Symbols from a source program may be defined when the object module is loaded, and may be used in AMPL statements. Optionally, the following types of source code symbols may be defined:

- The module identifier (operand of the IDT directive)
- External definitions (operands of DEF directives)
- Unresolved external references (operands of REF directives).

These load module symbols may be used until symbols from another module are loaded, or until the symbol table is cleared by command. The assembly language allows up to eight characters for the module identifier; the AMPL loader truncates the identifier to the first six characters. When using a module identifier in an AMPL statement, enter a period following the identifier as follows:

```
MYPROG.
SINTST.
```

When using either an external definition or an unresolved external reference in an AMPL statement, precede the definition or reference with a period, as follows:

```
.START
.INBUFF
```

4.2.4 **ARRAYS.** The AMPL language supports one- and two-dimensional arrays (paragraph 4.6.1) consisting of groups of elements that are referenced by the array name and one or two expressions in parentheses. Each element of an array may be assigned a value and may be used as a variable in an AMPL expression. A one-dimensioned array is a group of elements arranged in a row or list and specified by their position in the row. A two-dimensioned array is a group of elements arranged in rows and columns and specified by a row and column position. The following are valid references to elements of arrays:

NUM(4)	The fourth element of array NUM.
VAL(I)	An element of array VAL specified by the value of variable I.
TABLE(4,6)	The element of array TABLE in row 4 of column 6.
INPUT(A,B)	The element of array INPUT in the row specified by the value of A of the column specified by the value of B.

4.2.5 **CHARACTER STRINGS.** A character string is written as a string of up to 64 characters enclosed in single quotes. When a single quote is required as a character of the character string, two consecutive single quotes are entered to specify the single quote. The characters are represented internally as eight-bit ASCII characters with the leading bit set to zero. When a character string is entered as a statement, it is displayed as a message or comment. This is particularly useful when statements are written on a file to be entered with a COPY command (paragraph 5.7.14). The following are valid character strings:

```
'MAIN'

'INFILE'

':OUTFIL/OBJ'
```

4.2.6 **COMMENTS.** A comment may either be added following an AMPL statement or entered as a separate AMPL statement. In either case, two consecutive periods introduce the comment, and a carriage return terminates the comment. The following are examples of valid comments:

```
..  TEST PROGRAM LOOP 1
ECMP (IAQ,>1ØØ).. SET BREAKPOINT FOR INSTRUCTION ACQUISITION AT ADDRESS >1ØØ
```

4.3 NOTATION

The notational conventions used in the syntax definitions in the manual are as follows:

- Words shown in capital letters are reserved words (keywords). Reserved words and special characters must be entered as shown.
- Words enclosed in angle brackets represent constants, symbols, expressions, or character strings supplied by the user.
- Items enclosed in brackets ([ ]) are optional.
- Alternative items are enclosed in braces ({ }). One must be chosen.
- The ellipsis (. . .) indicates that the preceding item may be repeated.
- A slashed lower case b (b̸) represents a required space character.

In the examples, the b̸ has the same meaning as in syntax definitions. Responses by AMPL software are underlined.

4.4 FORMAT

AMPL software prints a question mark as a prompting character whenever it is ready to receive an AMPL statement or command:

```
?
```

In statements and commands, each reserved word, constant, or symbol must be followed by one or more spaces to separate these AMPL elements. In this document, carriage return means the operation associated with keys marked RETURN or NEW LINE on various devices. The user must enter a carriage return to terminate each line.

When a statement requires more than one line, either because of the length of the statement or to show the statement structure more clearly, a compound statement must be used.

4.5 EXPRESSIONS

An expression consists of a constant, a symbol, a subexpression, an array element, or a series of constants, symbols, subexpressions and/or array elements separated by arithmetic, logical, or relational operators. Each constant, symbol, subexpression or array element may be preceded by any of the unary operators.

4.5.1 SUBEXPRESSIONS. A subexpression is an expression enclosed in parentheses. A subexpression is evaluated before evaluating the expression that contains the subexpression. Subexpressions may contain other subexpressions; the innermost subexpression is evaluated first.

4.5.2 ARITHMETIC OPERATORS. The arithmetic operators of AMPL perform integer arithmetic. The operators are:

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
MOD	Remainder

The following are expressions using arithmetic operators:

`2 * (START + 4)`	The product of 2 times the sum of START plus 4
`LCN1/2`	The quotient of LCN1 divided by 2
`LCN1␣MOD␣2`	The remainder obtained by dividing LCN1 by 2
`TABEND-TABST`	The difference of TABEND minus TABST

4.5.3 LOGICAL OPERATORS. The following logical operators are used in AMPL expressions:

Operator	Operation
AND	Logical product
OR	Logical sum

The following are examples of expressions using logical operators:

```
CHARIN␣AND␣MASK

LCHAR␣OR␣RCHAR
```

4.5.4 **RELATIONAL OPERATORS.** The following relational operators are used in AMPL expressions:

Operator	Relation
EQ	Equal to
NE	Not equal to
LT	Arithmetic less than
LE	Arithmetic less than or equal to
GT	Arithmetic greater than
GE	Arithmetic greater than or equal to
HI	Logical higher than
HIE	Logical higher than or equal to
LO	Logical lower than
LOE	Logical lower than or equal to

The following are examples of expressions using relational operators:

```
DAY$GE$340

HOUR$LT$12

COUNT1$HIE$COUNT2
```

4.5.5 **UNARY OPERATORS.** The following unary operators are used in AMPL expressions:

Operator	Operation
+	Plus
-	Negation
NOT	One's complement
@	Indirect

More than one unary operator may be used with a symbol, constant, or subexpression, and the operators may appear in any order. The operations are described in the following paragraphs.

4.5.5.1 **Plus.** The unary operator + performs no operation on the constant, symbol, or subexpression to which it applies. It is included to provide uniformity.

4.5.5.2 Negation. The unary operator – specifies negation of the value of the symbol, constant, or subexpression to which it applies. The negation is equivalent to the result of subtracting the value from zero. The following are examples of negation:

-248	The result is -248.
->FE	The result is $FF02_{16}$ (two's complement) of $-FE_{16}$, -254.

4.5.5.3 One's Complement. The unary operator NOT specifies the one's complement of the value of the symbol, constant, or subexpression. The one's complement is equivalent to substituting zeros for ones and ones for zeros in the binary representation of the value. The following are examples of one's complements:

NOT>F1F0	The result is $0E0F_{16}$.
NOTFIVE	Assuming symbol FIVE has been assigned the value 5, the result is $FFFA_{16}$.

4.5.5.4 Indirect. The unary operator @ accesses a location in target system memory using the value of the symbol, constant, or subexpression as an address. The @ operator specifies "the contents of". The following are examples of indirects:

@>100	The result is the value in location 100_{16} of target memory.
@START	The result is the value in a location in target system memory. The value assigned to user symbol START is the target system memory address.
@@WRKSPC	The result is the contents of the contents of a location in target system memory. Assuming that WRKSPC has been assigned the value of the workspace address, the result is the contents of the location at the address in workspace register 0.

Indirect Addressing Example. Assuming that LCN1 has been assigned the value 0106_{16}, and that target system memory contains the following values:

Target System Memory Address (Hexadecimal)	Contents (Hexadecimal)
0106	010C
0108	0001
010A	0002
010C	0003
010E	0004

The values of the following expressions are:

@LCN1	$010C_{16}$
@@LCN1	0003_{16}
@(LCN1+4)	0002_{16}

Target System Memory Addressing. The emulator contains a 256-word trace memory and a 4K-word user memory. These memories are accessible at addresses in the target memory address space under control of logic in the emulator. Figure 4-1 illustrates the mapping of the address space to emulator or target system memory. System variable ETM maps addresses $FE00_{16}$ through $FFFF_{16}$ into emulator trace memory when it is set to one, and into target system memory when it is set to zero. System variable EUM maps addresses 0000_{16} through $1FFF_{16}$ into emulator user memory when it is set to one, and into target system memory when it is set to zero. References to target memory in this document may be either target memory or emulator memory depending on the address and the current values of system variables EUM and ETM.

4.5.6 EXPRESSION EVALUATION. Table 4-1 shows the hierarchy of the operators used in AMPL expressions. In evaluating an expression, level 7 operations are performed first, followed by operations at successively lower levels until the entire expression has been evaluated. Operations at the same level are performed in left to right order.

The following is an example of an expression:

```
42 + 5 * 8 / 4
```

The hierarchical levels of the operations are 5, 6, and 6, from left to right. The leftmost level 6 operation, 5 * 8, is evaluted first, and the result is 40. The other level 6 operation becomes 40 / 4, or 10. The expression is now 42 + 10, or 52. Had the expression been:

```
(42 + 5) * 8 / 4
```

the subexpression (42 + 5) or 47 would have been evaluated first. The multiplication would be next, 47 * 8, or 376. The division would be last, 376 / 4, or 94.

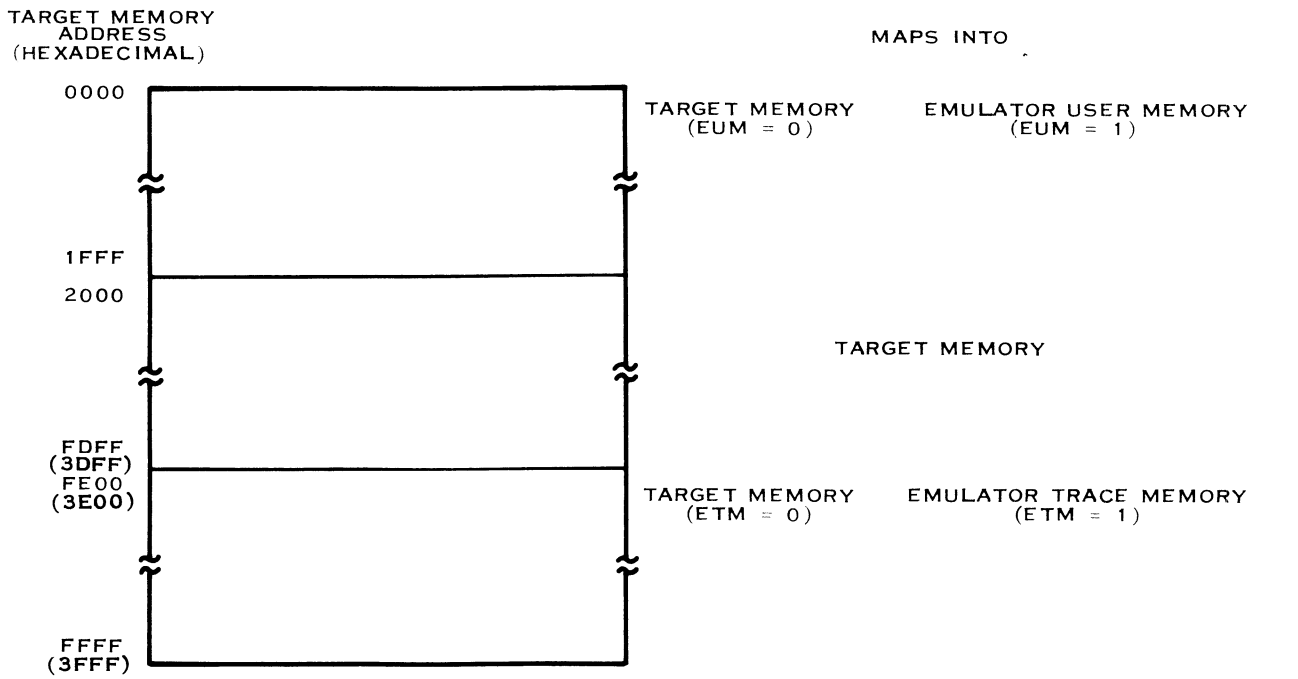


Figure 4-1. Target Memory Address Mapping

Table 4-1. Hierarchy of Operations in Expressions

Hierarchical Priority	Operator	Description
7	@	Indirect
7	+	Unary plus
7	-	Unary minus (negation)
6	*	Multiplication
6	/	Division
5	-	Subtraction
5	+	Addition
4	LT	Arithmetic less than
4	LE	Arithmetic less than or equal
4	EQ	Equal
4	NE	Not equal
4	GE	Arithmetic greater than or equal
4	GT	Arithmetic greater than
4	HI	Logical high
4	LO	Logical low
4	HIE	Logical high or equal
4	LOE	Logical low or equal
3	NOT	One's complement
2	AND	Logical product
2	OR	Logical sum

In evaluating an expression, the carry that can occur during a + or - operation is stored in system variable MDR. The carry occurs when there is a carry out of the sign bit position of the 16-bit result of the operation. MDR is set to 0 during either operation when no carry occurs, and to 1 when a carry occurs. To use the carry (as in double-precision arithmetic, for example) the user must test the contents of MDR following each + or - operation, because the current contents of MDR is the state of the carry bit resulting from the most recent operation.

4.6 STATEMENTS

The AMPL language supports the following statements:

- ARRAY statements
- Assign statements
- Display statements
- IF statements
- CASE statements

- WHILE statements
- REPEAT statements
- FOR statements
- Compound statements
- ESCAPE statements
- NULL statements

4.6.1 **ARRAY STATEMENTS.** An ARRAY statement declares one or more arrays. The syntax for an ARRAY statement is as follows:

```
?ARRAY∅<array name>(<expr>[,<expr>])[,<array name>(<expr>[,<expr>])] ...
```

An array name is a user symbol that is to be the name of an array. It is followed by one or two expressions within parentheses. One expression is entered for a one-dimensioned array. The value of the expression is the number of elements in the array. Two expressions separated by a comma are entered for a two-dimensioned array. The value of the first expression is the number of rows of elements in the array. The value of the second element is the number of columns of elements in the array. Several arrays may be declared in an ARRAY statement. The length of the line is the limit of the number of arrays that may be defined in one statement.

An array must be declared in an ARRAY statement before any element of the array may be referenced. Declaring an array causes the AMPL software to reserve space for the array and assigns the value of zero to every element in the array. Once declared, an array is global, applying to all programs being tested, until the symbol table is cleared. The space assigned to an array may be made available for other purposes by deleting the array, but the array name may only be used as the name of an array of the same number of dimensions until the symbol table is cleared. Following deletion, a one-dimensioned array may be declared again with a different number of elements, or a two-dimensioned array may be declared again with different numbers of rows and columns.

The following are examples of ARRAY statements:

`? ARRAY NUM (7)`	Declares array NUM consisting of seven elements.
`? ARRAY TABLE (5,1∅)`	Declares array TABLE consisting of 50 elements arranged in five rows and ten columns.
`? ARRAY VAL(8),INPUT (6, COLS)`	Declares array VAL consisting of eight elements, and array INPUT consisting of 6 rows. The value of variable COLS is the number of columns in array INPUT.

4.6.2 **ASSIGN STATEMENTS.** An assign statement assigns a value to a user variable or a system variable. The syntax for this type of assign statement is as follows:

```
?<symbol>=<expression>
```

The interpreter evaluates the expression and assigns the value to the symbol, a user symbol or a system symbol. An assign statement defines a user symbol when it assigns a value to a user symbol that has not been previously defined. An assign statement that assigns a value to a user symbol that has been defined alters the value of the symbol. In either case, the value of the expression is assigned to the symbol. The following are examples of this type of assign statement:

`? LCN = >1Ø4`	Assigns the value 104_{16} to user symbol LCN. If LCN is not defined, also defines LCN as a user symbol.
`? PC = LCN + 4`	Assigns the sum of the value of LCN plus 4 to system variable PC (target system program counter).

Another form of the assign statement assigns a value to a target system memory location. The syntax for this form of the assign statement is as follows:

```
? @<symbol>=<expression>
```

The @ sign identifies the value of the symbol as an indirect address. The symbol may be a user symbol or a system symbol, and may be preceded by any of the unary operators. The following are examples of this type of assign statement:

`? @LCN = 23`	Assigns the value 23 to the target system memory location specified in LCN.
`? @WP = 548`	Assigns the value 548 to the target system memory location contained in WP. This location is workspace register 0.

A constant may be used to specify the target system memory address to which a value is assigned. The syntax for this type of assign statement is as follows:

```
? @<constant>=<expression>
```

The @ sign identifies the constant as an indirect address. The following are examples of this form of assign statement:

`? @Ø1ØØ = #INC  RØ#`	Assign the value corresponding to the increment workspace register 0 instruction to target system memory location 100_{16}.
`? @>EFØØ = 256`	Assigns the value 256 to target system memory location $EF00_{16}$.

An expression within parentheses may be used to specify a target system memory address to which a value is assigned. The syntax for this type of assign statement is as follows:

```
? @(<expr 1>)=<expr2>
```

The @ sign identifies the value of the expression within parentheses as an indirect address. The following are examples of this type of assign statement:

`? @(LCN + 8) = @LCN`	Place the contents of target memory location LCN in target memory location LCN + 8.
`? @(WP + 16) = 345`	Place the value 345 into target system memory location corresponding to the sum of the contents of the workspace pointer register plus 16. This location is workspace register 8.
`?@(START+4) = #MOV *RØ+,*R1#`	Assign the value of an instruction constant to location START + 4 in target system memory.

An assign statement may be used to assign a value to every element of an array by entering the array name followed by the equal sign and the value to be assigned to the elements of the array. The following are examples of this type of assignment statement:

`?NUM = 25`	Assign the value 25 (19_{16}) to all elements of array NUM.
`?TABLE = Ø1ØØ`	Assign the value 100_{16} to all elements of array TABLE.

The assign statement may be used to assign a value to an element of an array by entering the array name with one or two expressions in parentheses to specify the element of the array. The following are examples of this type of assign statement:

`?NUM(3) = ØD`	Assign the value D_{16} to the third element of array NUM.
`?TABLE(3,2)=Ø4ØØ`	Assign the value 400_{16} to the element at row 3 of column 2 of array TABLE.

4.6.3 DISPLAY STATEMENTS. A display statement displays the value of an expression. The syntax for a display statement using default display options is as follows:

```
?<expression>
```

The value of the expression determines the variable or location to be displayed. The following are examples of display statements using default options:

```
   ?LCN1
LCN1 = >0104
```
Prints value of user variable LCN1 as a hexadecimal number (default).

```
   ?PC
PC   = >010C
```
Prints value of system variable PC (target system Program Counter).

```
   ?>200 +  46
>022E
```
Prints the value of the expression.

```
   ?@>200
>343A
```
Prints value in location 200_{16} of target system memory.

```
    ?#JNE  $-8#
>16FA
```
Display the value of an instruction constant.

```
    ?NUM(2)
>0019
```
Display the value of element 2 of array NUM.

4.6.3.1 **Format Specification.** The display statement may include a format specification for the display. The form of the display statement shown in the preceding paragraph uses the default value. The following is the syntax for a display statement using format specification characters:

```
?<expression>:<format>...
```

The expression has the same significance as in the syntax previously described. The colon indicates that one or more format specification characters follow.

The format specification characters are listed in table 4-2. An E specifies printing the displayed symbol followed by an equal sign. When it is used, E should precede any other format character because characters are effective in the order in which they are entered. When the expression to be displayed is an expression other than a single symbol, format character E is ignored.

Format character B specifies a display in binary format, consisting of a less than character (<) followed by 16 binary digits (0 or 1) and two spaces. When the B is followed by an optional digit 1 through 9, that number of binary digits (starting with the least significant digit) is displayed. The specified number of digits is preceded by a less than character and followed by two spaces. When the B is followed by the digit 0, the display consists of a less than character and 16 binary digits with no spaces separating the display from the following display.

Format character D specifies a display in decimal format, consisting of a space or minus sign, one to five decimal digits (0 through 9), and two spaces. Leading zeros are replaced by spaces. AMPL software converts the contents of the word as a two's complement value.

When the D is followed by an optional digit 1 through 5, that number of decimal digits (starting with the least significant digit) is displayed. The specified number of digits is preceded by a space or a minus sign and followed by two spaces. When the D is followed by a digit 6 through 9, the normal display occurs. When the D is followed by a 0, the normal display occurs but the two trailing spaces are omitted; no spaces separate the display from the following display.

Table 4-2. Format Specification Characters

Character	Type of Display	Note
E	Symbol and equal sign	
B	Binary	1
D	Decimal	1
U	Unsigned decimal	1
H	Hexadecimal	1
O	Octal	1
A	ASCII	
N	Line feed and carriage return	2
X	Space	2
I	Instruction	
S	Symbolic address	

Notes: 1. May optionally be followed by a single digit 0-9 to specify the number of digits to be displayed.

2. May optionally be followed by a single digit 1-9 to specify the number of repetitions of the operation, or a single 0 to specify ten repetitions.

Format character U specifies a display in unsigned decimal format, consisting of one to five decimal digits and two spaces. Leading zeros are replaced by spaces, and the word is converted as an unsigned magnitude value. The U may be followed by a digit, which has the same significance as for decimal displays.

Format character H specifies a display in hexadecimal format, consisting of a greater than character (>) followed by four hexadecimal digits (0 through 9, A through F) and two spaces. The H may be followed by a single digit. When the digit is 1 through 4, the display consists of the greater than character followed by the specified number of digits and two spaces. When the digit is 5 through 9, the normal display occurs. A 0 results in a display consisting of a greater than character and four hexadecimal digits, with no spaces separating the display from the following display.

Format character O specifies a display in octal format, consisting of an exclamation point (!) followed by six octal digits (0 through 7) and two spaces. When the O is followed by a digit 1 through 6, that number of octal digits (starting with the least significant digit) is displayed. The digits are preceded by an exclamation point and followed by two spaces. When the O is followed by a digit 7 through 9, the normal display occurs. When the O is followed by a 0, the normal display occurs but the two trailing spaces are omitted; no spaces separate the display from the following display.

Format character A specifies a display in ASCII format. The 16-bit value of the expression is translated into two 8-bit ASCII characters. A period is printed for the contents of any byte that does not contain a printable ASCII character.

Format character N causes a line feed and carriage return between displays. When the optional digit follows the N, the digit specifies a number of line feed and carriage return operations to be performed. The digit 0 specifies ten operations.

Similarly, format character X provides an additional space between displays and may be followed by a decimal digit to provide up to ten spaces (a 0 specifies ten spaces).

The I format character specifies the instruction display, which decodes the value as a machine instruction and displays the mnemonic operation code and operands as they could be specified in an assembly language statement. Further explanation and examples of the instruction display are included in the next paragraph.

The S format character specifies a display of a symbolic address when applicable. The symbolic addresses are the module identifier, external definitions, external references, and addresses relative to the module identifier. The module identifier, external definitions, and unresolved external references are optionally loaded with the program.

The value of an expression to be displayed in the S mode is compared to the load module symbols, and the symbol having the same value is printed. When the value of the expression is not equal to the value of any load module symbol, the module identifier is printed followed by a plus sign and the hexadecimal address relative to the module identifier value (load address). In any of the following instances, the S display is replaced by an H display because the S display is not applicable:

- No module has been loaded.
- Module was loaded without defining load module symbols.
- Load module symbols have been deleted.
- Expression value is below lowest address or above highest address loaded.

The initial default value of the format specification is EH, which specifies displaying the symbol, the equal sign, and the value is hexadecimal format. The user may enter one or more format characters following the colon. When more than one format character follows the colon, the characters apply in the order in which they are entered. The following are examples of display statements with several format characters:

```
   ?>4154:ANDHONB
AT
16724  >4154  !0404524
<0100000101010100

   ?PC:EHX2EBX4ED
PC  = >010E    PC    = <0000000100001110     PC   =   270

   ?0100:D303B9
256   !400  <100000000

   ?>FFFF:DU
  -1  65535

   ?0104:DUS                Assuming load module MYPROG is loaded at location 100₁₆.
  260    260  MYPROG.+ 0004
```

4.6.3.2 Instruction Display. The I format character causes the AMPL software to decode the value of the expression as a machine instruction. When the value decodes to an illegal operation code, AMPL software interprets the value as data, and lists a DATA directive with the hexadecimal equivalent of the value as the operand. When the value decodes to a two- or three-word instruction, AMPL software supplies one or two words of zeros and displays the result. The following are examples of display statements with I format characters:

```
   ?>C000:I
MOV  R0,R0

   ?>CFFF:I
MOV  *R15+,*R15+

   ?0E3:I
DATA  >00E3

   ?>C802:I
MOV  R2,@>0000
```

The software computes and stores the source and destination addresses as it decodes a value when the I character has been specified. System variable SRC contains the source address, and system variable DST contains the destination address. When the instruction does not have a source address, SRC is set to $FFFF_{16}$, and when the instruction does not have a destination address, DST is set to $FFFF_{16}$. The contents of variables SRC and DST are derived from the contents of the active workspace of the target system, and will only be valid when the workspace contents are valid. When displaying an instruction independently of emulation of the program, the user must check that the workspace registers contain the values that they will contain when the program is executed. The following example displays a value in the instruction mode, and then displays system variables SRC and DST:

```
   ?>C7CF:I
MOV  R15,*R15
   ?SRC:H
>FFF8
   ?DST:H
>23FE
```

Memory address $FFF8_{16}$ is workspace register 15, the source address. Workspace register 15 contains $23FE_{16}$, the destination address.

AMPL software accumulates the total numbers of TMS 9900 clock cycles and TMS 9900 memory accesses for each instruction decoded in the instruction mode. The totals are set to zero following each instruction that modifies the program counter (jump or branch). System variable TIME is a switch that contols the display of these totals along with the display of each instruction. When TIME is set to one (using an assign statement), two values are displayed to the right of the line that displays the instruction. The leftmost of these values contains the total of clock cycles, and the rightmost value contains the total of memory accesses. When a jump or branch is decoded, the totals are cleared following the display and an asterisk is displayed to the right of the values. The clock cycle total may be accessed as system variable CC and the memory access total may be accessed as system variable MC. The totals may be cleared with assign statements. The totals may be translated into time values by

multiplying the clock count total by the clock cycle time for the target system and the memory access total by the memory access time for the portion of target memory in which the access occurs. There are several cases for which the number of clock cycles and memory access cycles cannot be stated exactly. The number of clock cycles for a shift instruction with a count of zero is a function of the contents of workspace register 0, which is dynamic. The software uses the maximum clock cycle count (52) in this case. Also, the conditional jump instructions require more clock cycles when the condition is true than when the condition is false. The following are examples of statements to set TIME on, clear the totals, and display timing data:

```
   ?TIME = ON
   ?CC = 0
   ?MC = 0
   ?>CFFF:I
MOV  *R15+,*R15+                    30    8
   ?>C000:I
MOV  R0,R0                          44    12
   ?>1000:I
JMP  $+>0002                        54    13 *
```

The formula for translating the timing figures to total TMS 9900 execution times is as follows:

$$T = t_{c(\phi)} \quad (C + W * M)$$

in which:

T = Execution Time

$t_{c(\phi)}$ = Clock cycle time, typically 0.333 us.

C = Number of clock cycles (value of CC, left column of example)

W = Number of wait states per memory access (depends on type of memory)

M = Number of memory accesses (value of MC, right column of example)

For the set of instructions in the preceding example, assuming W = 0, the TMS 9900 execution time is:

$$T = 0.333 * (54 + (0 * 13) = 0.333 * 54 = 17.982 \text{ us.}$$

The formula for translating the timing figures to total TMS 9980 execution times is as follows:

$$T = t_c(\phi) (C + (W + 1) 2 * M)$$

The symbols in the formula have the same meanings as in the TMS 9900 formula. For the set of instructions in the preceding example, assuming W = 0, the TMS 9980 execution time is:

$$T = 0.333 * (54 + (0 + 1) * 2 * 13) = 26.64 \ \mu s$$

4.6.3.3 Changing Default Format Specifications. The display statement examples in the preceding paragraph include display specifications that apply only to the statement in which they appear. To change the default value the user may enter a G between the colon and the first character of the new format specification. The following are examples of statements that change the default format specification:

```
   ?PC:GHN
>34DE
```

Change the default format specification to display the value in hexadecimal format and provide a carriage return.

```
   ?PC:GEH
PC    = >34DE
```

Change the default value, restoring the initial format specification.

4.6.3.4 Displaying Target Memory. An option of the display statement may be used to display the contents of a range of locations in target system memory. The following is the syntax for the display statement option:

```
?<addr>ØTOØ<addr>[:<format>...]?[:<format>...]
```

Each addr operand may be any valid expression; its value is interpreted as an address in target memory. The first format specification applies to the display of the address; when it is omitted, the default format specification applies. The question mark (?) causes the display of the value of the expression to be followed by a display of the contents of the target system memory location corresponding to the value. The second format specification applies to the contents of the target system memory location; when it is omitted, the default format specification applies.

The initial default format specification for either the value or the contents is H (for hexadecimal display). The user may change the default by entering a G and a new default format specification for either the value or the contents. The following is an example of a display of target system memory locations:

```
   ?START = >20A6  ..ASSIGN STARTING ADDRESS TO START
   ?FINI = >20AE   ..ASSIGN ENDING ADDRESS TO FINI
   ?STARTØTOØFINI?
>20A6 / >0420
>20A8 / >22A4
>20AA / >C807
>20AC / >21A6
>20AE / >C808
```

Any valid expression can be used in place of START and FINI; a similar display can be obtained using system symbols. Assuming that the program counter contains $20A6_{16}$, the following example displays the same data with the I format character:

```
   ?PCØTOØPC+8?:I
>20A6 / BLWP @>22A4
>20AA / MOV  R7,@>21A6
>20AE / MOV  R8,@>21A8
```

4.6.3.5 Display and Modify. When used in a display statement having a single expression, the question mark allows the user to alter the contents of the target system memory location. Alternatively, the user may specify another location to be displayed. The following is the syntax for a display and modify statement:

```
? addr [:<format>...]?[:<format>...]
```

The addr operand and the format specification have the same significance as in the syntax previously described. The difference is that following the display of the requested location, the user may enter one of the command characters listed in table 4-3 or an expression in response to the question mark printed by the software. The following is an example of a display and modify statement which alters the contents of the target system memory location:

Table 4-3. Display and Modify Command Characters

Character	Meaning
= <expr>	Evaluate the expression following the equal sign and place the value in the location that was displayed. The expression is restricted to consist of a variable, a constant, or a combination of variables and or constants separated by arithmetic operators + or - only.
@	Display the location corresponding to the displayed contents. Displayed value is used as an indirect address.
+	Set the step mode to plus and display and modify the next word, or the next instruction when the I format character is in effect.
-	Set the step mode to minus and display and modify the preceding word.
Carriage Return	When the step mode is plus, display and modify the next word, or the next instruction when the I format character is in effect. When the step mode is minus, display and modify the preceding word.
:<format>...	Alter the format specification for the contents of the target memory location.
;	Terminate the display and modify operation.

Note: When an expression is entered (not preceded by a command character), AMPL software evaluates the expression and displays the target system memory location that corresponds to that value. The user may modify that location as if it had been entered in a display and modify statement. The expression may be a system symbol; e.g., SRC or DST when the I format character applies.

```
   ?PC?:H
>20A6 / >0420  ? = >0460
>20A6 / >0460  ?
```

The result of this operation is to place 0460_{16} in location $20A6_{16}$ of target system memory.

Alternatively, the user could have entered an instruction constant, as follows:

```
   ?PC?:H
20A6 / 0420  ? = #B        @22A4#
20A6 / 0460  ?
```

The instruction constant shown in the preceding example is that of a two-word instruction. Entered in this mode, an instruction constant provides all words for multi-word instructions. Entering a carriage return after the system displays the question mark causes a display of the second word, as follows:

```
20A8 / >22A4 ?
```

Alternatively, the user could have entered a + instead of the carriage return. The + sets the step mode to plus and accesses the next word. The step mode is plus initially, however, and a carriage return also accesses the next word. The + is required when the step mode has been set to minus, and it is desired to access the next word. The following example shows the result of entering the plus sign:

```
>20A6 / >0460  ? +
>20A8 / >22A4  ?
```

The user may display the contents of address $22A4_{16}$ by entering an @. The following example shows the result of entering an @:

```
>20A8 / >22A4   ? @
>20A4 / >22D8   ?
```

The user may change the format specification that applies to the target memory contents by entering a colon and a new format specification. The new specification does not alter the default values; it only applies to the current display. The following is an example:

```
>22A4 /  22D8   ?  :I
>22A4 / COC  *R8,R13   ?
```

The user may display and modify any location in target system memory by entering an expression the value of which is a location in target memory. The expression may only contain variables, constants, or question marks. Only addition and subtraction operations may be used. In this context, the question mark signifies the current target system memory location. The following is an example of entering an expression:

```
>22A4 / COC  *R8,R13   ? ? + >A
>22AE / >06A0  ?
```

User and system symbols are expressions also. When in the instruction mode it is frequently useful to display the source address, as follows:

```
>22AE / >06A0  ? :I
>22AE / BL  @>0074  ? SRC
>0074 / >04C0  ?
```

The user may terminate the display and modify operation by entering a semicolon, as in the following example:

```
>0074 / >04C0   ? ;
```

Using instruction constants in this mode is very effective because all words of the instruction are placed in target system memory. However the user must be aware that entering an instruction constant that supplies more words than the instruction that was previously stored at the location requires the user to enter the rest of the program again. Similarly, entering an instruction that supplies fewer words than the instruction it replaces requires the user to enter a NOP instruction or two, or to enter the rest of the program again. The following is an example of the problem:

```
   ? 0652?:I
>0652 / MOV R7,@>0104 ? = #MOV R7, R12#  . . NEW INSTRUCTION USES ONLY ONE WORD
>0652 / MOV R7, R12  ?
>0654 / DATA >0104  ? = #MOV R8,@0106#   . . REENTER NEXT INSTRUCTION
>0654 / MOV R8,@>0106  ?
>0658 / DATA >0106  ? = #MOV @040C,@040E# . . OLD INSTRUCTION USED TWO WORDS
>0658 / MOV @>040C,@>040E   ?
>065E / BLWP @>0206  ?  ;      . . BACK IN PROPER SEQUENCE-RETURN TO AMPL
```

4.6.4 IF STATEMENTS. The IF statement contains one or two statements, one of which is executed depending on the result of evaluating an expression. The syntax for the statement is as follows:

```
?IF<expression>ØTHENØ<statement>[ØELSEØ<statement>]
```

The expression is evaluated as a logical expression, yielding a true or false value. When the arithmetic value of an expression is zero, the logical value is false. The logical value of the expression is true when the arithmetic value is not zero. Execution of the statement depends on the evaluation, as shown in figure 4-2. When the logical value of the expression is true, the statement following the word THEN is executed, and another statement is requested. When the logical value of the expression is false, the statement following the word THEN is not executed. If the optional statement following the word ELSE has been included, this statement is executed; otherwise, another statement is requested.

The statements within the IF statement may be any of the AMPL statements described in this section, including other IF statements. The statements may also be any of the user commands described in Section V.

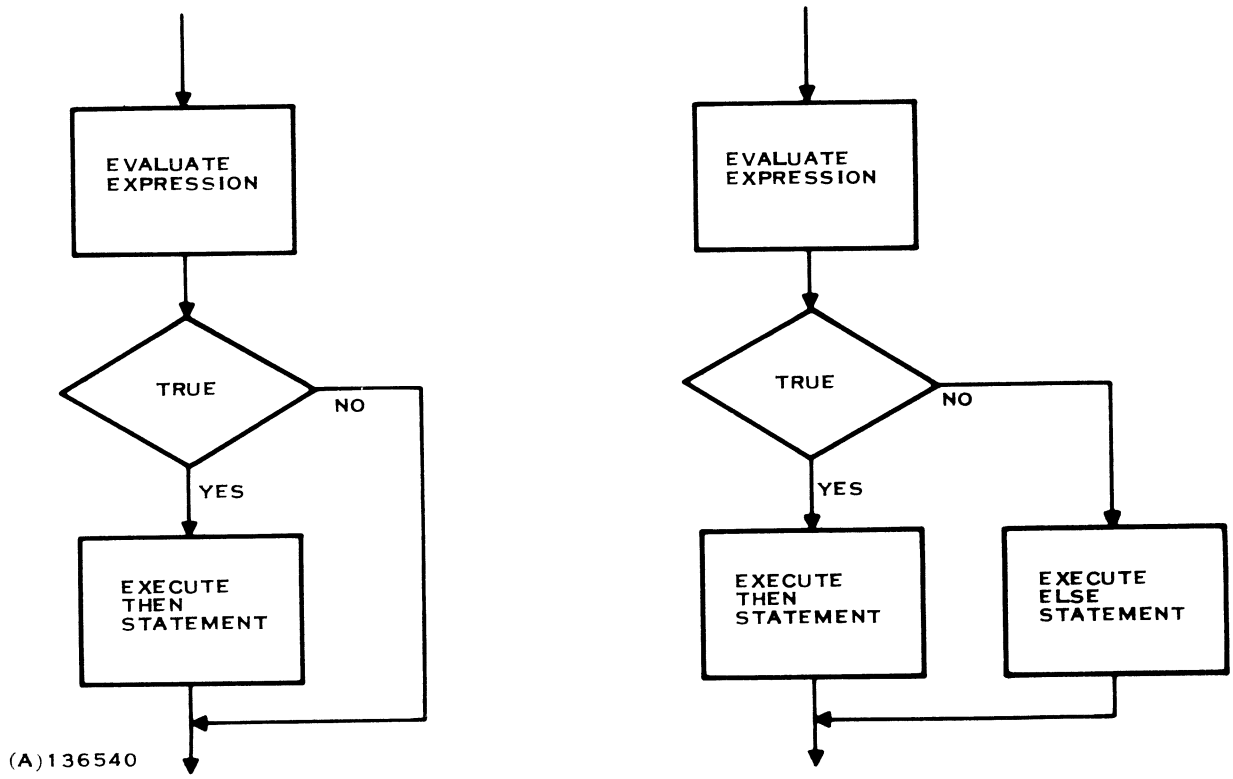


Figure 4-2. IF Statement Execution

The following is an example of an IF statement:

```
?IF␣PC␣EQ␣>1␀6␣THEN␣R␀ = >AAAA
```

The example statement executes the assign statement that sets R0 to $AAAA_{16}$ when PC contains 106_{16}. Otherwise, the assignment statement is not executed.

The following example shows an IF statement with the optional ELSE statement:

```
?IF␣PC␣NE␣>1␀␀␣THEN␣@PC:HI␣ELSE␣WP␣TO␣WP+32:H?:HD
```

In the example, the software displays the instruction at the address in the program counter in hexadecimal and instruction format when the program counter contains a value other than 100_{16}. When the program counter contains 100_{16}, the software displays the contents of the target system workspace in hexadecimal and decimal format.

4.6.5 CASE STATEMENTS. The CASE statement provides selective execution of a statement within a group of statements depending upon the value of an expression. The syntax for the CASE statement is as follows:

```
?CASE␣<expr>␣OF␣<expr>::<stmt>;[<expr>::<stmt>;]...[␣ELSE␣<stmt>]␣END
```

The expression following the reserved word CASE is evaluated to use in selecting the statement to be executed. Reserved word OF is followed by a group of expression and statement pairs separated by two colons. When the value of the expression in one of these pairs is equal to the value of the first expression, the statement of the pair is executed. When the value of the first expression is not equal to the value of the expression in any pair, and the ELSE reserved word is included, the statement following reserved word ELSE is executed. When the optional ELSE statement is omitted, and the value of the first expression is not equal to the value of the expression in any pair, the CASE statement terminates.

The expression and statement pairs and the ELSE statement may be entered on separate lines; a carriage return may be entered at any point following the reserved word OF. The following is an example of a CASE statement:

```
 ?CASE␣COUNT␣MOD␣4␣OF
1?   2::COUNT:X9D;
1?   3::COUNT:X9X9D;
1?   Ø::COUNT:X9X9X9D;
1?     ELSE␣COUNT:D
1?END
```

Display COUNT in a position on a line according to the remainder when COUNT is divided by 4.

The expressions may be entered in any order. An expression equal to 1 could be entered with the decimal display of COUNT instead of the ELSE statement shown. It is not necessary that an expression and statement be provided for each possible value of the first expression; normally, the ELSE statement is executed for more than one value, as in the following example:

```
 ?CASE␣VALUE␣OF
1?   25::FLAG = ON;
1?   3Ø::FLAG = ON;
1?   ELSE␣FLAG = OFF
1?   END
```

Set FLAG to one if VALUE is equal to 25 or 30; set FLAG to zero for other values.

4.6.6 WHILE STATEMENTS. The WHILE statement consists of an expression which is evaluated as a logical expression and a statement that is executed conditionally. The syntax for the WHILE statement is as follows:

```
?WHILE␣<expression>␣DO␣<statement>
```

The expression is evaluted first, as shown in figure 4-3. When the value of the expression is false (zero), the remainder of the statement is ignored, and the software requests another statement. When the value of the expression is true (nonzero), the statement following the word DO is executed and the expression is reevaluated. As long as the expression remains true, the execution of the statement and the reevaluation of the expression are repeated. When the value of the expression becomes false, the statement is not executed and the software requests another statement.

The statement following the word DO may be any of the AMPL statements described in this section, including another WHILE statement. The statement may also be any of the user commands described in Section V. However, execution of the statement should change the value of the expression; otherwise, the operation may never terminate. The following is an example of a WHILE statement:

```
?WHILE␣COUNT␣LE␣1ØØ␣DO␣COUNT = COUNT + 2
```

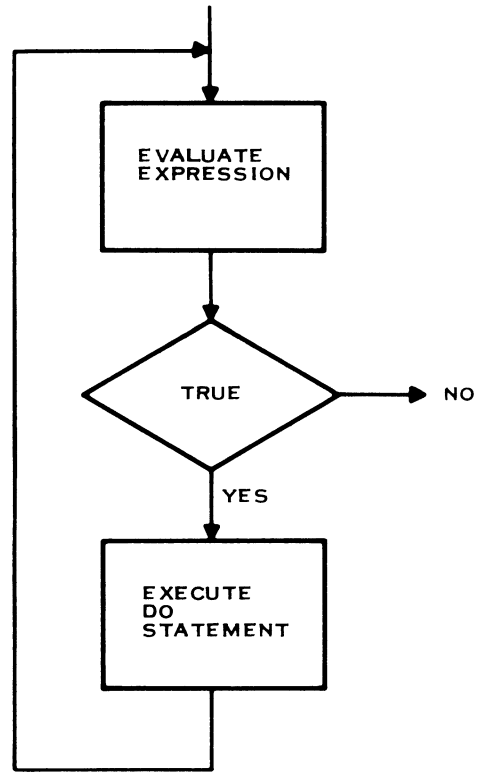


Figure 4-3. WHILE Statement Execution

If COUNT were greater than 100, initially, it would remain unaltered. If COUNT were less than or equal to 100 initially, it would be incremented by 2 repeatedly until it became greater than 100. The final value of COUNT is either 101, 102, or its initial value (greater than 100).

4.6.7 **REPEAT STATEMENTS.** The REPEAT statement consists of a statement that is executed at least once, and an expression which is evaluated as a logical expression. Subsequent execution is conditioned on the logical value of the expression. The syntax for the REPEAT statement is as follows:

```
?REPEAT␣<statement>␣UNTIL␣<expression>
```

The statement is executed and the expression is evaluated, as shown in figure 4-4. When the value of the expression is true (nonzero), the AMPL software requests another statement. When the value of the expression is false (zero), the execution of the statement and the evaluation of the expression is repeated. Execution of the statement continues until the value of the expression is true.

The statement following the word REPEAT may be any of the AMPL statements described in this section, including another REPEAT statement. The statement may also be any of the user commands described in Section V. However, execution of the statement should change the value of the expression; otherwise, the operation may never terminate.

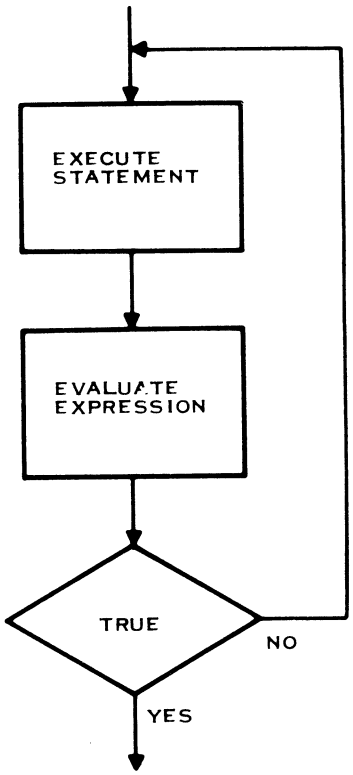


Figure 4-4. REPEAT Statement Execution

The following is an example of a REPEAT statement:

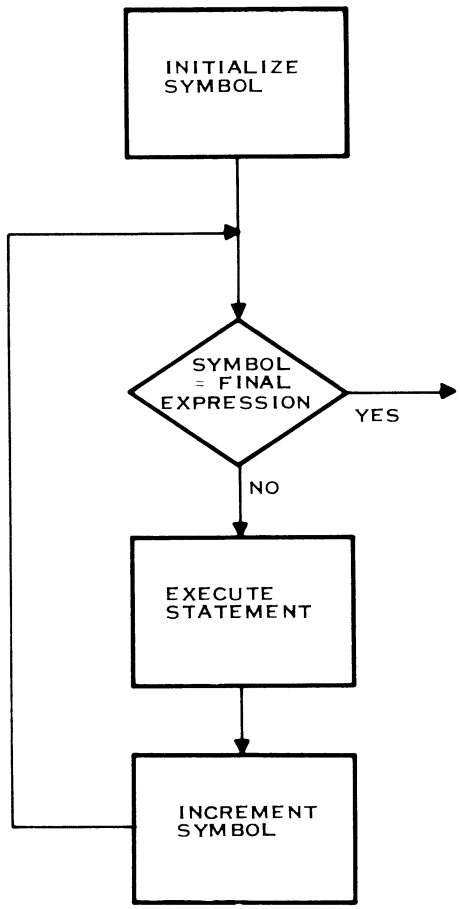
```
?REPEAT␣IF␣TALLY␣LE␣0␣THEN␣TALLY = 24␣ELSE␣TALLY = TALLY - 1␣UNTIL␣TALLY␣EQ␣0
```

The IF statement within the REPEAT statement is executed at least once. When TALLY has a negative or zero value, TALLY is set to 24. Otherwise, TALLY is decremented by one. The expression TALLY EQ 0 is evaluated. The IF statement is repeated, decrementing TALLY until TALLY equals zero (TALLY EQ 0 is true).

4.6.8 **FOR STATEMENTS.** A FOR statement assigns a value to a symbol and uses that symbol to control repeated execution of a statement. The syntax of the FOR statement is as follows:

```
?FOR␣<symbol>=<expr>␣TO␣<expr>[␣BY␣<expr>]␣DO␣<stmt>
```

The symbol may be a user symbol or a system variable. It is assigned the value of the expression that follows the equal sign, as shown in figure 4-5. Next, the software compares the value of the symbol to the value of the expression following the reserved word TO, and executes the statement following the reserved word DO if the values are not equal. The software then increments the value of the symbol by the value of the expression that follows the reserved word BY, or by one when the BY expression is omitted. The software repeats the comparison and execution of the statement until the value of the symbol equals the value of the expression that follows the reserved word TO, and then terminates the operation.



(A)136590

Figure 4-5. Execution of FOR Statement

The statement following the word DO may be any of the AMPL statements described in this section, including another FOR statement. The statement may also be any of the user commands described in Section V. When the BY expression is included, the difference of the terminal expression minus the initial expression must be an exact multiple of the increment expression; otherwise, the symbol will never be equal to the terminal value.

The following is an example of a FOR statement to initialize values in column one of a two-dimensioned array:

```
?FOR INDEX = 1 TO 5 DO MATRIX(INDEX,1)=INDEX
```

Execution of the example statement places values 1 to 5 in rows 1 to 5 of column 1 of array MATRIX. The following is an example of a FOR statement that uses a BY expression:

```
?FOR N = 0 TO 8 BY 2 DO @(BUFF+N)=0
```

This example statement stores zeros in four words of target memory starting at address BUFF.

4.6.9 **COMPOUND STATEMENTS.** A compound statement is a sequence of AMPL statements treated as a single statement. The syntax for a compound statement is as follows:

```
?BEGINb<statement>[[;]<statement>]...bEND
```

The statements following the word BEGIN may be any of the AMPL statements described in this section, including other compound statements. The statements may also be user commands described in Section V.

A compound statement may be used at any point to organize statements into a block of statements that are executed after the reserved word END is entered. A compound statement must be used when a statement cannot be entered on one line. The AMPL structure statements (IF, CASE, WHILE, REPEAT, and FOR) provide control over execution of one or more statements. By using compound statements within the structure statements a block of statements is executed rather than a single statement as shown in the preceding examples.

The following is an example of a compound statement:

```
 ?BEGIN
1?   CC = Ø;
1?   MC = Ø;
1?   TIME = ON;
1?   PCbTObPC+2Ø?:I;
1?   TIME = OFF;
1?END
```

Execution of the statement begins when the END statement is entered. The compound statement in the example clears the clock and memory cycle counts, sets the system variable TIME ON, displays a set of instructions, sets TIME OFF, and terminates, without user intervention.

The following is an example of the use of a compound statement with an IF statement:

```
 ?IFbPCbEQb>1Ø6bTHENbBEGIN
1?    CC = Ø
1?    MC = Ø
1?    PCbTObPC+>1ØØ?:I
1?ENDbELSEbPC:EH
```

This example tests system variable PC and executes a compound statement when PC is equal to 106_{16}. The compound statement contains two assign statements and a display statement. The assign statements clear the clock cycle and memory cycle counts. The display statement displays the contents of target system memory locations in instruction format. When PC is not equal to 106_{16}, the program counter contents is displayed in hexadecimal format.

Compound statements may be used within a CASE statement, as in the following example:

```
 ?CASE␣LINE␣OF
1?   1Ø::LINE:U              ..DISPLAY LINE NUMBER ON
1?   2Ø::LINE:U              ..LINES 1Ø AND 2Ø
1?   3Ø::BEGIN
2?        LINE:UN5           ..DISPLAY LINE NUMBER 3Ø
2?        LINE = 1           ..AND START NEW PAGE
2?        PAGE = PAGE + 1
2?        'PAGE'; PAGE:U3N
2?       END
1?   ELSE␣'     '            ..BLANK LINE NUMBER COLUMN ON OTHER LINES
1?   END
```

The preceding example assumes that the CASE statement is executed repeatedly along with a statement that displays a line of data and a statement that increments the line number. In that environment, the statement displays a line number on every 10th line and starts a new page on every 30th line.

Similarly, a WHILE statement may include a compound statement, as in the following example:

```
 ?MAIN = >1ØØ  ... INITIALIZE SYMBOL
 ?N = Ø        ... INITIALIZE COUNT
 ?WHILE␣N␣LT␣8Ø␣DØ␣BEGIN
1?  @(MAIN + N) = >2Ø2Ø;
1?  N = N + 2
1?END
```

The example stores ASCII spaces (20_{16}) in 80 consecutive bytes in target system memory. The first location is 100_{16}.

A REPEAT statement may also include a compound statement, as in the following example:

```
 ?MAIN = >1ØØ    .. INITIALIZE SYMBOL
 ?BUFF = >32Ø    .. INITIALIZE ADDRESS
 ?N = Ø          .. INITIALIZE COUNT
 ?REPEAT␣BEGIN
1?  @(MAIN + N) = @(BUFF + N)
1?  N = N + 2
1?END␣UNTIL␣N␣EQ␣1Ø
```

The example copies 5 words of target system memory from an area starting at location BUFF to an area starting at the location addressed by variable MAIN.

The following is an example of a FOR statement that includes a compound statement:

```
 ?FOR␣N = Ø␣TO␣1Ø␣BY␣2␣DO␣BEGIN
1?    IF␣@(DATA+N)␣AND␣ØØØ1␣THEN␣@(DATA+N):H ..DISPLAY ODD NUMBERS IN TARGET MEMORY
1?    ELSE␣'EVEN NUMBER'                      ..ADDRESSES DATA THROUGH DATA + 1Ø.
1?    @(DATA+N) = @(BUFF+N)                   ..DISPLAY "EVEN NUMBER" FOR EVEN NUMBERS
1?    END                                     ..MOVE VALUES FROM BUFF THROUGH BUFF + 1Ø
 ?                                            ..INTO DATA THROUGH DATA + 1Ø.
```

The example tests the contents of 10 target system memory locations starting at address DATA for odd numbers and displays the contents if odd, or a message if even. Then the example moves the contents of the corresponding location in an area beginning at address BUFF into the location.

4.6.10 ESCAPE STATEMENTS. The ESCAPE statement causes termination of execution of the statements in a loop. The loop structures supported by AMPL software are the WHILE, REPEAT, and FOR statements. The syntax of the ESCAPE statement is as follows:

```
?ESCAPE
```

When an ESCAPE statement within a loop structure is executed, control passes to the statement that would have been executed at normal completion of the loop operation. When an ESCAPE statement is entered other than in a WHILE, REPEAT, or FOR statement, the AMPL softwa.. issues an error message.

The following are examples of ESCAPE statements:

```
 ?WHILE Ø N Ø LE Ø M Ø DO Ø BEGIN
1?    IF Ø DATA+N Ø HIE Ø >1FFF Ø THEN Ø ESCAPE    ..TEST FOR ADDRESS ABOVE 1FFF16.
1?    @(DATA+N) = Ø                               ..STORE ZERO IN ADDRESS
1?    N = N+2
1? END

 ?REPEAT Ø BEGIN
1?  IF Ø @(BUFF+N) Ø EQ Ø >414 Ø THEN Ø ESCAPE    ..TEST FOR AA IN BUFFER
1?  N = N+2                                       ..INCREMENT N
1?  PTR = BUFF + N                                ..STORE NEXT ADDRESS IN PTR
1?END Ø UNTIL Ø N=M

 ?FOR Ø CNT = N Ø TO Ø M Ø DO Ø BEGIN
1?    IF Ø ADDR Ø HIE Ø FFFE Ø THEN Ø ESCAPE      ..TEST FOR ADDRESS ABOVE MEMORY ADDRESS SPACE
1?    @ADDR =  FFFF                               ..STORE -1 INTO ADDR
1?    ADDR = ADDR+2                               ..INCREMENT ADDR
1?END
```

In each of the examples, the ESCAPE statement terminates the loop when a specified condition other than the condition of the loop occurs.

4.6.11 NULL STATEMENTS. The NULL statement is a no operation statement. The syntax of the NULL statement is as follows:

```
?NULL
```

The NULL statement may be used at any time a no operation statement is appropriate. For example, in a WHILE or REPEAT statement in which the expression depends on a signal from hardware to terminate the operation, a NULL statement may be used following DO or REPEAT, respectively.

4.7 PROCEDURES AND FUNCTIONS

AMPL supports procedures and functions written by the user. Procedures and functions are subroutines available for use in AMPL statements. A procedure is called by a procedure call statement and may define one or more arguments to be passed from the calling statement. A procedure consists of a definition statement that includes a single executable statement, typically a compound statement. A function call is used as a variable, the value of which is the value returned by the function. A function may define one or more arguments to be passed from the function call. A function consists of a definition statement that includes a single executable statement, typically a compound statement. A function should contain at least one return statement, either as the only statement, or as a statement within the compound statement.

The statement of a procedure or function is not executed at the time the procedure or function is defined, but is executed each time the procedure or function is called. It is especially important to realize that an ARRAY statement in a procedure or function is not executed until the procedure or function is called. Any reference to an element of such an array is valid only after the procedure or function that contains the ARRAY statement has been called.

4.7.1 PROCEDURE DEFINITION STATEMENT. The procedure definition (PROC) statement defines the statement that follows as a procedure and specifies the required number of arguments and optionally the number of words of local storage for the procedure. The syntax for the PROC statement is as follows:

```
?PROC␣<procedure name>[(<arguments>[,<local storage>])]<statement>
```

The procedure name operand is the name used to call the procedure; it must meet the requirements for user symbols listed in paragraph 4.2.3.1, and must be unique with respect to other procedure names, function names, array names, and user symbols. The arguments operand is an integer constant having a value from zero through 255 that specifies the minimum number of arguments required by the procedure. When the procedure requires local storage, the local storage operand specifies the number of words of local storage required. The local storage operand is a positive integer constant less than 65,536. When the local storage operand is omitted, no local storage is provided for the procedure. When both operands are omitted, the value of zero is used for both operands.

The following are examples of PROC statements:

`?PROC␣DSPLY (4,1Ø) BEGIN`	Defines the compound statement as procedure DSPLY that requires at least four arguments and ten words of local storage
`?PROC␣NAME (Ø) 'PROGRAM NAME IS'`	Defines procedure NAME to print a message that identifies the program name.

4.7.2 FUNCTION DEFINITION STATEMENT. The function definition (FUNC) statement defines the statement that follows as a function, and specifies the required number of arguments and optionally the number of words of local storage for the function. The syntax for the FUNC statement is as follows:

```
?FUNC␣<function name>[(<arguments>[,<local storage>])]<statement>
```

The function name operand is the name used in the function call; it must meet the requirements for user symbols listed in paragraph 4.2.3.1, and must be unique with respect to other function names, procedure names, array names, and user symbols. The arguments operand is an integer constant having a value from zero through 255 that specifies the minimum number of arguments required by the function. When the function requires local storage, the local storage operand specifies the number of words of local storage required. The local storage operand is a positive integer constant less than 65,536. When the local storage operand is omitted, no local storage is provided for the function. When both operands are omitted, the value of zero is used for both operands.

The following are examples of FUNC statements:

?FUNC LSUM (3,2) BEGIN	Defines the compound statement as function LSUM that requires at least three arguments and two words of local storage.
?FUNC ADDIT(2) BEGIN	Defines the compound statement as function ADDIT that requires at least two arguments and no local storage.

4.7.3 ARGUMENTS. The arguments for procedures and functions are placed in the call statement or function call. At least as many as are specified in the definition must be supplied. The unary operator ARG is used within the procedure or function to determine the number of arguments supplied in the call, and to access the arguments. The constant, variable, or sub-expression to which the operator ARG applies must evaluate to a positive integer value or zero. When the value is zero, the result is the number of arguments supplied in the call; when the value is a positive value, the operator and its operand access the argument corresponding to the value; i.e., ARG 1 accesses the first argument (leftmost argument) in the call, ARG 2 the second, etc. The value of the operand of the ARG operator must not exceed the value of ARG 0. The ARG operator may be used to access an argument either for reading or writing, unless the value of the operand is zero; the value of ARG 0 may not be altered. A value stored in an argument is local to the procedure or function, and is not returned to the calling statement.

The priority of unary operator ARG with respect to the operators listed in table 4-1 is 7 (highest priority). The following are examples of the use of unary operator ARG:

ARG 3	Accesses the third argument from the left in the set of arguments of a call to a procedure or function.
ARG N	When N has a value greater than zero, and equal to or less than the number of arguments, accesses the corresponding argument. When the value of N is negative or greater than the number of arguments, the expression is invalid. When the value of N is zero, accesses the number of arguments entered.

4.7.4 **LOCAL STORAGE.** A PROC or FUNC statement that has the optional local storage operand causes the software to reserve memory for local storage for the procedure or function. Local storage is provided each time the procedure or function is called, even though the call is nested within another call, and becomes inaccessible when the procedure or function terminates. Data cannot be passed between procedures and functions or between a procedure or function and the calling statement through local storage.

The words of local storage are referenced by the unary operator LOC. The constant, variable, or subexpression to which the operator LOC applies must evaluate to an integer value greater than or equal to zero, and less than or equal to the number of words of local storage provided. When the operand evaluates to zero, the result is the number of words of local storage provided. When the value is a positive value, the operator and its operand access the word in local storage corresponding to the value; i.e., LOC 1 accesses the first word, LOC 2 the second, etc. The LOC operator may be used to access a word for either reading or writing, except that the value of LOC 0 may not be altered.

The priority of unary operator LOC with respect to the operators listed in table 4-1 is 7 (highest priority). The following are examples of the use of unary operator LOC:

LOC Ø4	Accesses the fourth word of local storage for the procedure or function.
LOC Ø(NUM+3)	When subexpression NUM+3 has a value greater than zero and less than or equal to the number of words of local storage, accesses the word of local storage that corresponds to the value. When the value of NUM+3 is negative or greater than the number of words of local storage, the expression is invalid. When the value of NUM+3 is zero, accesses the number of words of local storage allocated.

4.7.5 **RETURN STATEMENT.** The RETURN statement terminates the execution of a function or a procedure. The syntax for a RETURN statement is as follows:

```
?RETURN[Ø<expression>]
```

The expression is evaluated, and when the statement appears in a function, the value is assigned to the function call. When the expression is omitted, and the RETURN statement is in a function, the value of zero is returned. When the RETURN statement is in a procedure, the expression is evaluated but the value is not returned.

The RETURN statement is valid in a function, either as the only statement in the function, or as a part of the compound statement of the function. The RETURN statement is also valid in the compound statement of a procedure. The following is an example of a function that contains only a RETURN statement:

```
?FUNCØSUM(2) RETURNØARGØ1 + ARGØ2
```

This function returns the sum of the arguments as the value of the function call. Other examples of RETURN statements appear in examples in paragraph 4.7.7.

A function that returns a value other than zero to the function call must contain a RETURN statement. Execution of a RETURN statement during execution of a procedure or function terminates execution of the procedure or function, and returns control to the calling statement. An implicit RETURN with an operand of 0 occurs following the executable statement of a procedure or function. A function that contains no explicit RETURN statement terminates at that point and returns a value of zero for the function call.

4.7.6 CALLS TO PROCEDURES AND FUNCTIONS. A call to a procedure is an AMPL statement consisting of the procedure name followed by arguments, if any, enclosed in parentheses. The syntax of a procedure call is as follows:

```
?<procedure name>[(<argument>[,<argument>]...)]
```

The procedure name is a name that has previously been entered as the first operand of a PROC statement. The user must enter the number of arguments specified in the PROC statement, and may enter as many more as desired. Arguments must be entered in the sequence required by the procedure. Each argument is an expression.

A procedure call may be entered whenever the software requests a statement, or within an IF, WHILE, REPEAT, or compound statement. A call to a procedure may be included in the same procedure (recursive call), in another procedure, or in a function.

The following is an example of a procedure call:

`?DSPL1 (LBL1,LBL1+8)`	Calls procedure DSPL1 specifying two arguments, LBL1 and LBL1+8. ARG Ø within the procedure returns 2, ARG 1 accesses LBL1, and ARG 2 accesses LBL1+8.

A call to a function is used as a variable in an expression. The value returned by the function becomes the value of the variable. The function call consists of the function name followed by arguments, if any, enclosed in parentheses. The syntax of a function call is as follows:

```
<function name>[(<argument>[,<argument>]...)]
```

The function name is a name that has previously been entered as the first operand of a FUNC statement. The user must enter the number of arguments specified in the FUNC statement and may enter as many more as desired. Arguments must be entered in the sequence required by the function. Each argument is an expression. The operand of the RETURN statement that terminates execution of the function becomes the value of the function call.

A function call may be used in any AMPL expression. A function call may be used in the same function (recursive call), in another function, or in a procedure. The following are examples of AMPL statements that contain function calls:

`?STADD = SUM(BEGIN,OFSET)`	Variable STADD is set to the value of the SUM function of BEGIN and OFSET.
`?SUM(BEGIN,OFSET) + 1Ø:HD`	Displays the sum of the SUM function of BEGIN and OFSET and 1Ø in hexadecimal and decimal format.

4.7.7 **PROCEDURE AND FUNCTION EXAMPLES.** The following is an example of a procedure definition:

```
 ?PROCØDT (Ø,2) BEGIN   .. DUMP EMULATOR TRACE BUFFER (Ø ARGUMENTS, 2 LOCAL VARIABLES)
1?   LOCØ1 = ETBO        .. GET INDEX OF OLDEST STORED VALUE
1?   LOCØ2 = ETBN        .. GET INDEX OF NEWEST STORED VALUE
1?   WHILEØLOCØ1ØLEØLOCØ2ØDO
1?     BEGIN
2?     ETB(LOCØ1):H      ..DISPLAY TRACED VALUE
2?     @ETB(LOCØ1):HIN   ..DISPLAY CONTENTS  OF TRACED VALUE
2?     LOCØ1 = LOCØ1 + 1..INCREMENT INDEX
2?     END
1?   RETURN              ..THIS STATEMENT MAY BE OMITTED
1?END
```

Procedure DT displays the contents of the emulator trace memory. On the assumption that the memory locations contain program counter values, the procedure also displays the contents of these addresses. The procedure uses a command (ETB) and two system variables (ETBO and ETBN) which are described in Section V. The command reads a value from the trace memory, and the system variables contain the lower and upper index limits that define the portion of trace memory into which trace values have been stored.

The PROC statement defines procedure DT with no arguments and two words of local storage. The procedure consists of a compound statement that contains two assign statements, a WHILE statement, and a RETURN statement. The first assign statement assigns the index value of the oldest value in trace memory (system variable ETBO) to LOC 1. The second assign statement assigns the index value of the newest value in memory (system variable ETBN) to LOC 2. The WHILE statement executes a compound statement repeatedly until local storage word 1 is greater than local storage word 2. The compound statement contains two display statements and an assign statement. The first display statement displays the contents of a trace memory location; the second display statement displays the contents of the displayed address in hexadecimal and instruction format. The assign statement increments the contents of local storage word 1.

The WHILE statement terminates when the values stored in trace memory have all been displayed. The RETURN statement terminates execution of the procedure. If the RETURN statement is omitted, the implied RETURN provided at the end of every procedure terminates the procedure. The call for the procedure is as follows:

```
?DT;
```

The call for a procedure or function can include more arguments than the required number specified in the PROC or FUNC statement. Many procedures and functions, such as the preceding example, would ignore any arguments (or additional arguments). The following is an example of a procedure that requires one argument, but displays results (execution times) for as many arguments as the user enters.

```
 ?PROCØTIMEN (1,1) BEGIN
1?  LOCØ1 = 1
1?  TIME = OFF
1?  WHILEØLOCØ1ØLEØARGØ0ØDO
1?    BEGIN
2?      MC = Ø
2?      CC = Ø
2?      @ARG (LOCØ1):IXXXXX
2?      333 * (CC + MC):DN
2?      LOCØ1 = LOCØ1 + 1
2?    END
1?END
```

The example computes execution times of TMS 9900 instructions (other than branch or jump), assuming a clock cycle time of 333 ns and a single wait cycle. The calling statement includes the locations to be displayed for which time is to be computed.

Procedure TIMEN contains a compound statement consisting of two assign statements and a WHILE statement. The first assign statement assigns the value of 1 to LOC 1 to be used as an index to the arguments. The second assign statement sets system variable TIME OFF to inhibit printing of the values of CC and MC in the display. The WHILE statement causes a compound statement to be executed for each argument. Within this compound statement, assign statements clear MC and CC, and display statements display the contents of the memory location corresponding to the argument in instruction format, and the computed execution time in decimal format. The last statement in the compound statement increments the index to access the next argument. The following are examples of statements that call procedure TIMEN:

```
?TIMEN(START)

?TIMEN (MAIN+1Ø,MAIN+24,COMP)
```

The following is an example of a function definition:

```
 ?FUNCØMEMTST (3,1) BEGIN .. MEMORY TESTER (3 ARGUMENTS, 1 LOCAL VARIABLE
1?  LOCØ1 = ARGØ1          .. SET LOCAL STORAGE WORD TO STARTING ADDRESS ARGUMENT
1?  WHILEØLOCØ1ØLOEØARGØ2ØDO  .. WRITE PHASE
1?    BEGIN
2?    @LOCØ1 = ARGØ3
2?    LOCØ1 = LOCØ1 + 2
2?    END
1?  LOCØ1 = ARGØ1          ..SET LOCAL STORAGE WORD TO STARTING ADDRESS ARGUMENT
1?  WHILEØLOCØ1ØLOEØARGØ2ØDO  .. READ AND COMPARE PHASE
1?    BEGIN
2?    IFØ@LOCØ1ØNEØARGØ3ØTHENØRETURNØLOCØ1 .. ERROR EXIT
2?    LOCØ1 = LOCØ1 + 2
2?    END
1?    RETURNØ-1    .. NORMAL EXIT
1?END
```

CAUTION

If the second argument for this function is equal to $FFFF_{16}$ or if the first argument is an even address and the second argument is $FFFE_{16}$, the function will not terminate properly, and will destroy the contents of low order memory if allowed to execute indefinitely. Incrementing $FFFE_{16}$ by 2 results in 0000_{16}, which is lower than either $FFFE_{16}$ or $FFFF_{16}$.

Function MEMTST tests an area of memory as specified in three arguments. The first argument is the starting address in target system memory; the second argument is the ending address. The third argument is the test data to be used in the test.

The FUNC statement defines function MEMTST with three arguments and a word of local storage. The function consists of a compound statement that contains an assign statement and a WHILE statement for the write phase, an assign statement and a WHILE statement for the read and compare phase, and a RETURN statement. The assign statement for the write phase sets the local storage word to the starting address in the first argument. The WHILE statement executes a compound statement repeatedly until the address in the local storage word is greater than the ending address in argument 2. The compound statement consists of two assign statements. The first writes the test data word into the target system memory location specified by the local storage word. The second assign statement increments the contents of the local storage word by two. The WHILE statement terminates when all words in the test area have been written. The assign statement for the read and compare phase sets the local storage word to the starting address. The WHILE statement executes a compound statement repeatedly until the address in the local storage word is greater than the ending address in argument 2. The compound statement consists of an IF statement and an assign statement. The IF statement executes a RETURN statement if the contents of memory are not equal to the test data in the third argument; this constitutes a memory error, and the address in the local storage word (the address that contains an error) is returned. When the contents of the memory location are correct, the assign statement is executed, incrementing the contents of the local storage word by two. The WHILE statement terminates when all words in the test area have been read and found to be correct. The RETURN statement provides a normal return, returning a -1 to identify the normal return.

The following are examples of AMPL statements that call MEMTST:

```
?TMTST = MEMTST (START,FINISH, >A5A5)  ..  TEST MEMORY
?IF␣TMTST␣NE␣-1␣THEN␣TMTST:D           ..  DISPLAY ERROR ADDRESS IF ERROR RETURN

 ?BEGIN
1?  TSTDAT = >A5A5  ..INITIALIZE DATA FOR TEST
1?  TSTAD = >␣1␣␣   ..INITIALIZE STARTING ADDRESS
1?  TSTEND = >1␣␣␣  ..INITIALIZE ENDING ADDRESS
1?  WHILE␣TSTAD␣LO␣TSTEND␣DO␣BEGIN
2?    TSTMEM = MEMTST (TSTAD,TSTEND,TSTDAT)  .. TEST MEMORY
2?    IF␣TSTMEM␣EQ␣-1␣THEN
2?       TSTAD = TSTEND .. TEST COMPLETE
2?       ELSE␣BEGIN
3?         TSTMEM:H     .. DISPLAY ERROR ADDRESS
3?         TSTAD = TSTMEM + 2  .. CONTINUE TEST BEGINNING AT
3?       END                   .. ADDRESS FOLLOWING ERROR
2?  END
1?END
```

SECTION V

SYSTEM OPERATION

5.1 INTRODUCTION

Prior to loading and operating the AMPL system, connect the target system to the host system through the emulator/buffer module, the logic state trace data module, or both, as described in the installation and operation manuals for these modules.

Operation of the system consists of loading the operating system, starting the AMPL program, and entering AMPL statements and commands. This section includes the loading procedures for TX990 and for DX10 and descriptions of the user commands. User commands are described in the following categories:

- Target System Program Loading and Saving Commands
- Utility Commands
- CRU Commands
- Data Input Commands
- Emulator Operation Commands
- Trace Module Operation Commands

5.2 LOADING AND STARTING THE SYSTEM

Loading and starting the system differs for TX990 and DX10 systems. The following paragraphs describe the procedures for the two operating systems.

5.2.1 TX990. To load the AMPL system, perform the following steps:

1. Place the AMPL system diskette in either floppy disk unit and place the unit in the Ready mode.

NOTE

Place the system disc in floppy disc unit 1 (left-hand unit), if possible. When the system disc is in unit 2 a limited set of error messages is printed (paragraph 6.2).

2. Press the HALT/SIE switch on the programmer panel of the computer.
3. Press the RESET switch on the programmer panel.
4. Press the LOAD switch on the programmer panel.
5. The computer should load TX990 from the diskette and print a message similar to the following when the loading has completed:

```
TX990 SYSTEM
MEMORY SIZE (WORDS): 24576        AVAILABLE: 12923
```

6. When a Model 911 VDT is used as system console, verify that the UPPER CASE LOCK key is in the upper case position for proper entry to TXDS and to the AMPL program.

7. Enter an exclamation point (!) to activate the Terminal Executive Development System (TXDS). The computer prints a message similar to the following:

```
TXDS 936215 **        1/0        0: 0

PROGRAM:
```

8. Place FS990 TXDS parts diskette in the other floppy disk unit (normally unit 2) and enter the following information to execute utility SYSUTL to initialize time and date. (If time and date information is not required, steps 8 and 9 may be omitted.) The following example shows entering data corresponding to 3:45 p.m., March 11, 1977:

```
PROGRAM: DSC2:SYSUTL/SYS
INPUT:
OUTPUT:
OPTIONS: ID,1977,3,11,15,45.
```

9. SYSUTL executes the command, prints the following message, and returns control to TXDS:

```
TX990 SYSTEM UTILITY        937544**

15:45:00 MAR 11, 1977
```

10. After executing the ID command, SYSUTL terminates. Control returns to TXDS, which prints a message similar to the following:

```
TXDS        936215 **        70/77        15:45

PROGRAM:
```

11. Enter the program name, followed by an asterisk. The asterisk specifies that the entry of input, output and options is not required. Enter the name as follows:

```
PROGRAM:  :AMPL/SYS *
```

12. The program begins execution, and prints a question mark requesting entry of a user command or an AMPL statement. The program is initialized to provide space for 30 user symbols. The user may require more than 30 symbols, or may not require that many. When fewer than 30 user symbols are required, specifying a smaller number allows more memory for procedures and functions. When deciding on symbol table requirements, allow space for module names and procedure and function names. The user may enter a CLR command with a positive integer operand to specify user symbol table size. The following is an example of a CLR command:

```
? CLR (25)        Reserve space for 25 symbols in the user symbol
                  table.
```

13. Enter commands and statements as required. The program prints a question mark requesting entry at the completion of each command or statement:

```
?
```

5.2.2 DX10. To load the AMPL system, perform the following steps:

1. Place the disk cartridge containing the DX10 system and the AMPL program on system disk unit 0, and place the unit in ready with write protection disabled.

2. At the programmer panel of the computer, press the HALT/SIE switch.

3. Press the RESET switch on the programmer panel.

4. Press the LOAD switch on the programmer panel.

5. When a Model 911 VDT is to be used as AMPL terminal, verify that the UPPER CASE LOCK key is in the upper case position for proper entry of System Command Interpreter commands and AMPL commands and statements.

6. Press the blank orange key (upper right of keyboard) on the Model 911 VDT.

7. Enter an exclamation point (!). The system displays a pair of brackets ([ ]) requesting entry of a command.

8. Enter IS for an initialize system command. The system displays the following:

   INITIALIZE SYSTEM

   INITIALIZE SYSTEM LOG?:

9. Enter YES. The system displays the following request:

   YEAR:

10. Enter the current year; e.g., 1977. The system displays the following request:

    MONTH:

11. Enter the number of the month; e.g., 9 for September. The system displays the following request:

    DAY:

12. Enter the day of the month; e.g., 13. The system displays the following request:

    HOUR:

13. Enter the hour (adding 12 to P.M. hours); e.g., 17 for 5 P.M. The system displays the following message:

    MINUTE:

14. Enter the minute; e.g., 35. The system displays the following request:

    ATTENTION DEVICE:

15. Enter the device name of a terminal on which the system displays attention messages, or ME if the messages are to be displayed on the AMPL terminal. The system displays the following request:

    LOGGING DEVICE:

16. Enter DUMY or the device name of a device to display system log messages. The system displays the following request:

    FILES?:

17. Enter YES. The system displays the following request:

    PRIMARY FILE:

18. Enter 0. The system displays the following request:

    INITIAL ALLOCATION:

19. Enter 100. The system displays a pair of brackets requesting another command.

20. Enter AMPL to activate the AMPL program. The system displays the following:

    AMPL MICROPROCESSOR PROTOTYPING LAB
    USER MEM (K) = 8

21. Either enter another number instead of 8 and press the RETURN key, or press the RETURN key to specify 8. The number is the size in 1024 (1K) word blocks of memory required for the AMPL system data and work area. The user may specify less than 8 when the work to be done does not require many procedures or functions and other users and other tasks require system resources. The user must specify more than 8 when the work to be done requires many long functions and procedures. The system displays the following (with the applicable version number):

    AMPL 3.0

    ?

22. If a symbol table size less than or greater than 30 symbols is required, enter a CLR command similar to the following to specify a symbol table size. When few symbols are required, a small symbol table provides more memory for procedures and functions. When deciding on symbol table requirements allow space for load module symbols and for procedure and function names. The CLR command is as follows:

? CLR (45)	Reserve space for 45 symbols in the user symbol table.

23. The AMPL system displays a question mark (?) to request a command or statement at the completion of execution of each command or statement. Enter commands and statements as required.

5.3 HARDWARE DEMONSTRATION TEST

The AMPL software includes a hardware demonstration test that verifies the installation of the emulator and trace modules. It should be executed following the initial loading of the AMPL program. The hardware demonstration test is not a diagnostic test of the emulator and trace modules. However, it may be executed at any time you desire to exercise the emulator and trace modules. Perform the following steps to execute the test immediately after loading and starting the AMPL program:

1. Enter a COPY command to load the test. Use the applicable one of the following examples, followed by a carriage return:

   ? COPY (':AMPHDT/PRC') For a TX990 system, or

   ? COPY ('.S$SYSLIB.AMPL$LIB.AMPHDT') For a DX10 system.

2. When the AMPL program has loaded the procedures for the test, it displays operating instructions. Follow the instructions in each display, entering the following as requested.

   =? C

3. When C is entered following the third group of instructions, the test begins. The first test consists of displaying instructions for entering the EINT command to initialize the emulator. Use the applicable one of the following examples:

   ? EINT ('EMU') TX990 system

   ? EINT ('EM01') DX10 system

4. The AMPL software responds by displaying a question mark when the initialization completes satisfactorily. The system displays an error message when initialization fails. A typical error that might occur during an initialization attempt displays the following message:

```
*** ERROR      205>0700 >0001
    EMULATOR DSR ERROR:
        >02XX = ILLEGAL OPERATION
        >06XX = OPERATION TIMED OUT
        >07XX = DEVICE ERROR
```

   If any error message is displayed, verify the connections to the emulator, and verify that the emulator module is firmly seated in the proper slot. Then repeat the command.

5. When the emulator has been successfully initialized, enter the following command to resume the test:

   ? AMPHDT;

6. The second test displays the following title, and clears and sets the two system variables that map target system memory addresses into emulator memory. Normally the test completes with no further message.

   -EMULATOR MEMORY SELECTION TEST-

Failure of the test is indicated by an error message; typically, the error message shown in step 4 is displayed. If an error message is displayed, verify that the microprocessor connector of the buffer is not connected to anything, or is not contacting any electrically conductive material. Should the microprocessor connector be connected to a target system, both the emulator and the target system are being tested. Disconnect the microprocessor connector to isolate the problem.

7. After successfully completing the second test, the software begins the third test by displaying the following title:

-EMULATOR USER MEMORY TEST (4K WORDS)-

The test consists of writing the memory address into the addressed location, beginning at address 0. The test then reads the tested address, and compares the contents to the address. When the comparison is equal, the test continues by adding 32 to the tested address, and repeating the test, until the range of addresses, 0 through $1FE0_{16}$, has been tested. When the comparison is not equal, the test displays the following title:

*** ERROR: EMULATOR USER MEMORY ERROR

The test may be repeated by entering the following command. If the failure persists, refer the problem to maintenance personnel.

? AMPHDT;

8. After successfully completing the third test, the software begins the fourth test by displaying the following title:

-EMULATOR TRACE BUFFER MEMORY TEST (256 WORDS)-

The test is similar to the user memory test, except that all even addresses from $FE00_{16}$ through $FFFE_{16}$ are tested. If the test software detects an error, it displays the following message:

*** ERROR: EMULATOR TRACE MEMORY FAILURE

Recover from the error as described in step 7.

9. After successfully completing the fourth test, the software begins the fifth test by displaying the following title:

-EMULATOR RUN TEST-

The test writes a simple program into target memory, consisting of four instructions, the last of which is an IDLE instruction. The test then sets up the emulator to execute with no tracing or breakpoint, and issues a command to start the emulator. The program in memory executes, placing the microprocessor in the idle mode. The test verifies that the microprocessor is executing an IDLE instruction. If the status of the microprocessor indicates that the microprocessor is not executing, or is not executing an IDLE instruction, the test program displays the following message:

*** ERROR: EMULATOR EXECUTION FAILURE

If the test fails, refer the problem to maintenance personnel. Repeat the test by entering the following:

? AMPHDT;

10. After successfully completing the fifth test, the software begins the sixth test by displaying the following title:

-EMULATOR HALT TEST-

The test executes an emulator command to halt the emulator, and verifies that the emulator halts. If the status of the microprocessor indicates that the emulator did not halt, the test displays the following message:

*** ERROR: EMULATOR FAILED TO HALT PROPERLY

Recover from the error as described in step 9.

11. After successfully completing the sixth test, the software begins the seventh test by displaying the following title:

-EMULATOR TRACE TEST-

The test sets up a trace of the program in target memory, and a breakpoint at completion of the trace operation. The test then re-executes the program in target memory, and verifies that the trace operation traced properly and halted the operation. When the trace operation fails, the test displays the following message:

*** ERROR: EMULATOR FAILED TRACE TEST

Recover from the error as described in step 9.

12. After successfully completing the seventh test, the software begins the eighth test by displaying the following title:

-EMULATOR ADDRESS COMPARISON TEST-

The test sets up a breakpoint at an instruction address, and re-executes the program in target memory. The test then verifies that the breakpoint stopped the emulator at the proper point. When the breakpoint operation fails, the test displays the following message:

*** ERROR: EMULATOR FAILED PC BREAKPOINT

Recover from the error as described in step 9.

13. At this point, emulator tests have been completed, and the test displays the following message:

*** EMULATOR PASSES HARDWARE DEMONSTRATION TEST ***
DO YOU WISH TO TEST A LOGIC STATE TRACE MODULE? (Y=YES, N=NO)

If the system being tested includes a trace module, enter Y; otherwise, enter N, and skip to step 20.

14. When the user requests testing a trace module, the test displays instructions for entering the TINT command to initialize the trace module. Use the applicable one of the following examples:

TINT ('TRA')	TX990 system
TINT ('TM01')	DX10 system

The software responds by displaying a question mark when the initialization is complete. The software displays an error message when intialization fails. A typical error that might occur during an initialization attempt displays the following message:

```
*** ERROR      219 0700    0001
    TRACE MODULE DSR ERROR:
        02XX = ILLEGAL WHILE TRACING,
        06XX = OPERATION ABORTED
        07XX = TRACE MODULE DEVICE ERROR
```

If any error message is displayed, verify the connections to the trace module, and verify that the trace module is firmly seated in the proper slot. Then repeat the command.

When the trace module has been successfully initialized, enter the following command to resume the test:

```
? AMPHDT;
```

15. The software begins the first trace module test by displaying the following title:

-TRACE MODULE INTERNAL CLOCK TEST-

The test sets up the emulator to execute with no breakpoints or tracing, and sets up the trace module to trace using the internal clock, and to halt after tracing 10 items. The test then re-executes the program in target memory, and verifies that the trace operation halted the trace module. When the status of the trace module indicates that tracing did not stop or that something other than completion of the trace operation halted the trace module, the test displays the following message:

*** ERROR: TRACE MODULE FAILED INTERNAL CLOCK TEST

Recover from the error as described in step 9.

16. After successfully completing the first trace module test, the software begins the second test by displaying the following title:

-TRACE MODULE DATA TRACE TEST-

The test sets up the trace module to trace data using external (emulator) clock. The test also sets the trace module to halt the emulator at the completion of the instruction that completes the trace operation. The test then re-executes the program in target memory, and verifies that the completion of the trace operation halted the emulator and the trace module, and that the trace operation correctly traced the data. When the status of the emulator and trace modules indicates that the test failed, the test displays the following message:

*** ERROR: TRACE MODULE FAILED DATA TRACE

Recover from the error as described in step 9.

17. After successfully completing the second trace module test, the software begins the third test by displaying the following title:

    -TRACE MODULE EVENT TEST-

    The test sets up the trace module to breakpoint on the first instruction of the program in target memory, and re-executes that program. The test then verifies that both the emulator and trace modules halted, and that the breakpoint halted the modules. When the status of the emulator and trace modules indicates that the breakpoint failed, the test displays the following message:

    *** ERROR: TRACE MODULE EVENT FAILURE

    Recover from the error as described in step 9.

18. After successfully completing the third trace module test, the software begins the fourth test by displaying the following title:

    -TRACE MODULE: TRACE DELAY TEST-

    The test uses the same breakpoint set up for the preceding test, but sets up a delay of halting the emulator and trace modules following the breakpoint. The test then re-executes the program in target memory, and verifies that the breakpoint executed properly after the specified delay. When the status of the emulator and trace modules indicates that the breakpoint or delay failed, the test displays the following message:

    *** ERROR: TRACE MODULE DELAY FAILURE

    Recover from the error as described in step 9.

19. After successfully completing the fourth trace module test, the software displays the following message:

    *** TRACE MODULE PASSES HARDWARE DEMONSTRATION TEST ***

20. The test then displays the concluding message, as follows:

    *** CONCLUSION OF AMPL HARDWARE DEMONSTRATION TEST ***

5.4 RECOVERY PROCEDURE

The buffer modules contain a switch that connects the clock source for the microprocessor to the clock in the emulator or to the clock in the target system. Setting the CLOCK switch to a different position during a debugging session requires a recovery procedure to be performed before continuing the test. A similar recovery procedure is required when the power to the target system is disconnected and then restored. When the TMS 9900 buffer module is in use, observe the following rules:

- Avoid altering the CLOCK switch setting or interrupting power to the target system while the emulator is running. Should this occur inadvertently, restart as described in paragraph 5.2.

- After switching clock sources or restoring power with the emulator halted, enter an EINT command (paragraph 5.10.1) and restore contents of target system memory.

When the TMS 9980 buffer module is in use, recovery is the same as described for the TMS 9900 buffer except that a restart is not necessary. An EINT command must be entered to recover in all cases, and target system memory contents must be restored.

5.5 ENTERING COMMANDS

The user may enter commands and AMPL statements interchangeably. However, when a compound statement contains a CLR, RSTR, EDIT, or COPY command, execution of the compound statement terminates at completion of the CLR, RSTR, EDIT, or COPY command. Any statement or command following any of these commands within a compound statement is not executed.

Any command or statement may be terminated by a semicolon to promote clarity. There are several instances in which semicolons are required to prevent ambiguity. It is good practice to place a semicolon following each statement within a CASE statement. This prevents the expression that follows from being interpreted as a continuation of the statement. It is also a good practice to place a semicolon following each command that has no operands; otherwise an expression within parentheses that follows the command is taken as a group of operands and ignored.

Execution of any command can be terminated and control returned to the AMPL program by pressing a key on the system console. When the system console is a 913 VDT, press the HELP key; when the console is the 911 VDT, press the CMD key; when the console is the 733 ASR, hold the CNTRL key while pressing the X key.

System variables YR, DAY, HR, MIN, and SEC may be used to add time and date information in the listings of AMPL statements, commands and results. These may be displayed but not assigned new values. The following example shows the use of these variables:

?HR:D2 14	Hour of the day. Value shown corresponds to 2 P.M.
?MIN:D2 43	Minutes past the hour.
?SEC:D2 23	Seconds past the minute.
?DAY:D3 133	Day of the year. Value shown corresponds to May 13.
?YR:D4 1977	Year.

5.6 PROGRAM COMMANDS

AMPL software supports two commands to load and save programs in target memory. The LOAD command loads a relocatable object module or an absolute module from a diskette file into target system memory. Optionally, the command defines load module symbols. The DUMP command stores a program from target system memory on a diskette file in absolute format.

5.6.1 LOAD COMMAND. The LOAD command loads a relocatable module in either standard or compressed object format or an absolute module into target system memory. It performs relocation functions for relocatable object modules using the bias value specified in the command. The syntax for a LOAD command is as follows:

?LOAD('<access>'[,<bias>[, [IDT] [+REF] [+DEF]]])

The access name is a character string that contains the name of the module to be loaded. The name of the module is a name that is acceptable to the operating system; e.g., a TX990 file name, or a DX10 pathname or synonym. The bias is the address in target memory into which the first word of a relocatable object module is loaded. When the bias operand is omitted and a relocatable module is to be loaded, the default value of $A0_{16}$ is used. When an absolute module is to be loaded, it is loaded at the same locations from which it was stored, and the bias operand is ignored.

The third operand consists of one or more of the keywords IDT, REF, and DEF (separated by plus signs) or the keyword OFF. When the operand is omitted, or when all keywords are entered, the load module identifiers, external definitions, and unresolved external references are defined in the symbol table as load module symbols. Keyword IDT causes the load module identifier to be defined; keyword REF causes unresolved external references to be defined; and keyword DEF causes external definitions to be defined. One, two, or all of these keywords may be used, or the keyword OFF may be used to load a module without defining any load module symbols.

The user may display the values of any of the three types of load module symbols. The value of an unresolved external reference is the target memory address of the end of a linked list of target memory locations that require the value of the external reference. Procedure RESOLV (Appendix G) may be called to assign a value to these locations, and the external reference may also be assigned that value. However, the user must supply the address of the end of the linked list, i.e., the initial value of the reference, to procedure RESOLV. The user may not alter the values of external definitions or load module identifiers.

The following are examples of LOAD commands using TX990 file names:

?LOAD (':MEMOBJ/OBJ',>FE00)	Load object file MEMOBJ/OBJ into target system memory starting at address $FE00_{16}$, defining all load module symbols.
?LOAD (':MEMABS/ABS')	Load absolute file MEMABS/ABS into target system memory at address supplied in the file.
?LOAD (':MYPROG/OBJ', 0100,IDT)	Load an object file having a synonym of FILE1 MYPROG/OBJ into target system memory starting at address 100_{16}, defining load module identifiers only.

The following are examples of LOAD commands using DX10 pathnames and synonyms:

? LOAD ('VAMPL.OBJ.MEMOBJ',>FE00)	Load object file VAMPL.OBJ.MEMOBJ into target system memory starting at address $FE00_{16}$, defining all load module symbols.
? LOAD ('VAMPL.ABS.MEMABS')	Load absolute file VAMPL.ABS. MEMABS into target system memory at address supplied in the file.
? LOAD ('FILE1',0100,IDT)	Load an ojbect file having the synonym FILE1 into target system memory starting at address 100_{16}, defining load module identifiers only.

The LOAD command does not support image files or segmentation tags. If the object code contains a tag other than 0 through F or I, an error occurs when the loader attempts to process the tag. The object module may be link edited by either SDSLINK or TXSLNK to obtain a module that may be loaded by the LOAD command. However, no load module symbols are defined unless the SYMT option is used both in assembly and link editing, and then only the module identifier is defined.

5.6.2 DUMP COMMAND. The DUMP command stores a program on a specified file from specified locations in target system memory. Optionally, the command also stores a specified entry point. The syntax for a DUMP command is as follows:

?DUMP ('<access name>',<start>,<end>[,<entry>])

The access name is a character string that contains the name of the file on which the program is to be stored. The name of the file is a name that is acceptable to the operating system; e.g., a TX990 file name, or a DX10 pathname or synonym. When the files does not exist, AMPL software creates a file. When the file contains data, the program replaces the previous contents of the file. The start operand is the address of the first word to be stored, and the end operand is the address of the last word to be stored. The entry operand is the address of the entry point of the stored program.

The DUMP command stores a program in absolute format. When the entry operand is specified, the value is placed in the program counter of the emulator when the program is loaded. Files written by the DUMP command may be reloaded by the LOAD command which restores the program in target memory for further testing and debugging.

The following is an example of a DUMP command using a TX990 pathname:

? DUMP (':MEMABS/ABS',>FE00,>FE46,>FE26)

Store target memory contents beginning at address $FE00_{16}$ and ending at address $FE46_{16}$ on file MEMABS/ABS, and store $FE26_{16}$ as the entry point for the program.

The following is an example of a DUMP command using a DX10 pathname:

? DUMP ('VAMPL.ABS.MEMABS',>FE00,>FE46,>FE26)

Stores target memory contents beginning at address $FE00_{16}$ and ending at address $FE46_{16}$ on file VAMPL.ABS.MEMABS, and store $FE26_{16}$ as the entry point for the program.

5.7 UTILITY COMMANDS

AMPL software supports the following utility commands:

- Define Console (CNSL)
- Define Listing Device (LIST)
- Multiply (MPY)
- Divide (DIV)

- Display Register Contents (DR)
- Display User Symbol Table (USYM)
- Display Load Module Symbol Table (MSYM)
- Delete Load Module Symbol Table (MDEL)
- Display System Symbol Table (SSYM)
- Save Test Environment (SAVE)
- Clear Test Environment (CLR)
- Restore Test Environment (RSTR)
- Enter Text Editor (EDIT)
- Copy Input (COPY)
- Delete Procedure (DELE)
- Delay AMPL Execution (WAIT)
- Output New Line (NL)
- Verify (VRFY)
- Terminate AMPL Program (EXIT)

5.7.1 DEFINE CONSOLE COMMAND. The Define Console command (CNSL) defines an alternate device to replace the device currently being used as system console, or switches the output to the console on and off. The syntax for the CNSL command is as follows:

$$\underline{?}\text{CNSL}\left(\begin{Bmatrix}\text{'<device name>'}\\ \text{OFF}\\ \text{ON}\end{Bmatrix}\right)$$

The device name is a character string that contains the device name (as defined for the operating system) of a console device to replace the current system console. The keyword OFF causes output to the system console to be disabled. The keyword ON causes output to the system console to be reenabled following entry of a CNSL command with the OFF keyword. When a CNSL command is entered with a device name operand, the named device displays the question mark requesting a new command at the completion of this command. When a CNSL command is entered with the keyword ON the previously defined system console displays the question mark at the completion of this comand. The following are examples of CNSL commands:

? CNSL ('ASR')	The 733 ASR replaces the system console (TX990).
? CNSL ('ST07')	The terminal ST07 replaces the system-assigned console terminal (DX10).

Execution of a CNSL command in a DX10 system does not affect the synonyms assigned to the AMPL terminal. The synonyms defined at the terminal from which AMPL was called continue to apply.

5.7.2 DEFINE LISTING DEVICE COMMAND. The Define Listing Device command (LIST) defines a supplementary listing device or file, or stops or starts supplementary listing. While listing is on, all data displayed on the system console is also displayed on the listing device or written to the listing file. The following is the syntax for the LIST command:

```
?LIST( {'<access name>'} )
       {    OFF      }
       {    ON       }
```

The operand is a character string that contains a pathname or either of the reserved words OFF or ON. A TX990 pathname may be that of a device or file; a DX10 pathname may be a device name, a file pathname, or a synonym. The pathname is that of a device or file to display or store the supplementary listing. When the pathname is that of a file that does not exist, AMPL software creates the file. When the file already contains data, the data in the file is replaced by the new data. When OFF is entered, supplementary output is terminated. Reserved word ON is only valid when reserved word OFF has been entered in a previous LIST command, and restores supplementary output to the device or file that was previously defined for supplementary output. A LIST command with a new pathname may be entered at any time; the device or file previously defined is closed and the newly defined device or file is opened and output begins.

The following is an example of a series of LIST commands using TX990 pathnames:

?LIST(':OFILE/LST')	Write supplementary output on file DSC:OFILE/LST.
?LIST('LP')	Print supplementary output on Line Printer instead of writing to DSC:OFILE/LST.
?LIST(':OFILE/LST')	Write supplementary output on file DSC:OFILE/LST replacing previously written output. Terminate output to Line Printer.
?LIST(OFF)	Terminate supplementary output, displaying output on system console only.
?LIST(ON)	Resume supplementary output, adding it to file DSC:OFILE/LST, following previously written output.

The following is an example of a series of LIST commands using DX10 pathnames:

?LIST('VAMPL.LST.OFILE')	Write supplementary output on file VAMPL.LST.OFILE.
?LIST('LP01')	Print supplementary output on Line Printer instead of writing to VAMPL.LST.OFILE.
?LIST('VAMPL.LST.OFILE')	Write supplementary output on file VAMPL.LST.OFILE replacing previously written output. Terminate output to Line Printer.

?LIST(OFF)	Terminate supplementary output, displaying output on system console only.
?LIST(ON)	Resume supplementary output, adding it to file VAMPL.LST.OFILE following previously written output.

5.7.3 MULTIPLY COMMAND. The Multiply command (MPY) is a system function that multiplies two 16-bit unsigned numbers and returns the least significant 16 bits of the product. The most significant 16 bits of the product are stored in system variable MDR. The syntax of the MPY command is as follows:

?MPY(<multiplicand>,<multiplier>)

The operands are expressions, the 16-bit values of which are multipled as unsigned numbers. The following is an example of an MPY command:

```
    ? MPY (24,1Ø)
MPY  = >ØØFØ
```

Multiply 24 times 10.

```
    ? MDR
MDR  = >ØØØØ
```

Obtain the most significant half of the product. The complete product is $000000F0_{16}$, or 240.

Alternatively, the command and its operands may be used as an expression in an AMPL statement. The value of the expression is the 16 low-order bits of the product. Variable MDR contains the 16 high-order bits of the product. The following is an example of an AMPL statement that contains an MPY command:

```
?REPEATØN = N + 1ØUNTILØMPY (CNT,N)ØGTØ1ØØØ
```

Increment N by 1 until the least significant half of the product is greater than 1000. Unless the product of the initial value of CNT times N + 1 is greater than 65,535 MDR contains zero.

5.7.4 DIVIDE COMMAND. The Divide command (DIV) is a system function that divides a 32-bit unsigned number by a 16-bit unsigned number, and returns the 16-bit quotient. The most significant 16 bits of the dividend must be placed in system variable MDR prior to executing the command. The remainder is in MDR following the operation. The syntax of the DIV command is as follows:

?DIV(<divisor>,<dividend>)

The operands are expressions. The 16-bit value of the dividend operand is combined with the value of MDR and is divided by the 16-bit value of the divisor operand. The following is an example of an assign statement that places the high-order portion of the dividend in MDR, and a DIV command:

```
    ? MDR = Ø1
    ? DIV (59,Ø)
DIV  = >Ø456
```

Divide 10000_{16} (65,536) by 59.

```
    ? MDR
MDR  = >ØØ2E
```

Display remainder.

Alternatively, the command and its operands may be used as an expression in an AMPL statement. The value of the expression is the quotient, and variable MDR contains the remainder. The following is an example of an AMPL statement that contains DIV command:

? DISP = DEST - PC + 2	Initialize DISP.
? MDR = Ø	Set MDR to most significant half of dividend.
? DISP = DIV(2,DISP)	Compute displacement for jump to DEST when DEST is greater than contents of PC.
? IFØMDRØNEØOØTHENØ'ERROR'	Display ERROR if DEST has an odd value.

5.7.5 DISPLAY REGISTER COMMAND. The Display Register command (DR) displays the contents of the target system program counter, workspace pointer, and status register, the contents of the address in the program counter, and the contents of workspace registers 0 through 15 of the current target system workspace. The contents are displayed in hexadecimal format and the contents of the address in the program counter are also displayed in instruction format. The syntax for the command is as follows:

?DR;

The following is an example of the DR command showing the resulting display:

```
? DR;
R0   = >0030 R8  = >0000 PC  = >012E /  >0601 DEC R1
R1   = >0F98 R9  = >0000 WP  = >0134
R2   = >FFFF R10   >0000 ST  = >D000
R3   = >FFFF R11 = >010E
R4   = >FFFF R12 = >1FE0
R5   = >FFFF R13 = >0000
R6   = >0000 R14 = >0000
R7   = >0000 R15 = >0000
```

5.7.6 DISPLAY USER SYMBOL TABLE COMMAND. The Display User Symbol Table command (USYM) displays a list of all user symbols, the symbol types, and the symbol values. The display includes a count of the symbols also. The syntax for the command is as follows:

?USYM;

The AMPL software prints one of the following symbol types after each symbol:

VAR	Variable
PROC	Procedure name
FUNC	Function name
ARRAY	Array name

The software prints the symbol value after the symbol type. For a variable, the value is the value of the variable. For a procedure name, a function name, or an array name, the value is the host memory address of the procedure, function, or array, respectively.

5.7.7 **DISPLAY LOAD MODULE SYMBOL TABLE COMMAND.** The Display Load Module Symbol Table command (MSYM) displays a list of all load module symbols, the symbol types, and the symbol values. The display includes a count of the symbols also. The syntax for the command is as follows:

```
?MYSM;
```

The AMPL software prints one of the following symbol types after each symbol:

IDT	Module identifier
DEF	External definition
REF	Unresolved external reference

The software prints the symbol value after the symbol type. For the module identifier, the value is the target system memory address into which the module was loaded. For an external definition, the value is the target system memory address corresponding to the defined symbol. For an unresolved external reference, the value is either the most-recently assigned value, or the initial value, if no value has been assigned. The initial value is the target system memory address of the last location of a linked list of addresses into which the reference should be loaded.

The following is an example of the MSYM command showing the resulting display:

```
   ?MSYM;
MAIN    IDT     >Ø1ØØ
START   DEF     >Ø12Ø
BUFF    DEF     >Ø43Ø
```

5.7.8 **DELETE LOAD MODULE SYMBOL TABLE COMMAND.** The Delete Load Module Symbol Table command (MDEL) deletes all load module symbols in the load module symbol table. The syntax for the command is as follows:

```
?MDEL;
```

It is possible to load a module without deleting the load module symbols of a previously loaded module by using the OFF option in the LOAD command. The user may clear the load module symbol table without clearing the other symbol tables by entering this command.

The following is an example of the MDEL command followed by an MSYM command:

```
?MDEL;
?MSYM;
0 ENTRIES IN TABLE
```

5.7.9 **DISPLAY SYSTEM SYMBOL TABLE COMMAND.** The Display System Symbol Table command (SSYM) displays a list of system symbols, the symbol types, and the symbol values. The display includes a count of the symbols, also. The syntax for the command is as follows:

```
?SSYM;
```

The AMPL software prints one of the following symbol types after each symbol:

VAR	Variable
PROC	Procedure name
FUNC	Function name
ARRAY	Array name

The software prints the symbol value after the symbol type. For a variable, the value is the value of the variable. For a procedure name, a function name, or an array name, the value is the host memory address of the procedure, function, or array, respectively.

5.7.10 SAVE TEST ENVIRONMENT COMMAND. The Save Test Environment command (SAVE) stores the test environment and a program from specified locations in target system memory on a specified file. The test environment includes a user symbol table, and user procedures and functions. Optionally, the command also stores the specified entry point. The syntax for the SAVE command is as follows:

?SAVE('<access name>'[,<start>,<end>[,<entry>]])

The access name is a character string that contains the name of the file on which the test environment and program are to be stored. The name of the file is a name that is acceptable to the operating system; e.g., a TX990 file name, or a DX10 pathname or synonym. When the file does not exist, AMPL software creates the file. When the file contains data, the test environment and program replaces the previous contents of the file. The start operand is the address of the first word to be stored, and the end operand is the address of the last word to be stored. The entry operand is the address of the entry point of the stored program. When the start and end operands are omitted, the program is not saved; only the test environment is saved. When the entry operand is omitted, the RSTR command does not alter the contents of the program counter. The file written by the SAVE command may be read by the RSTR command to restore a test environment and a target system program to resume a test.

The test environment stored by a SAVE command consists of currently defined user symbols, current default format specifications, user procedures, and user functions. No device assignments or device status are stored. The following is an example of a SAVE command using a TX990 file name:

```
?SAVE (':MYDBUG/SAV',>FEØØ,>FE46,>FE26)
```

Store test environment on file DSC:MYDBUG/SAV. Also store the contents of target memory starting at address $FE00_{16}$ and ending at address $FE46_{16}$ and store $FE26_{16}$ as the entry point.

The following is an example of a SAVE command using a DX10 file name:

```
?SAVE ('VAMPL,SAV,MYDBUG,>FEØØ,>FE46,>FE26)
```

Store test environment on file VAMPL.SAV. MYBUG. Also store the contents of target memory starting at address $FE00_{16}$ and ending at address $FE46_{16}$, and store $FE26_{16}$ as the entry point.

5.7.11 CLEAR TEST ENVIRONMENT COMMAND. The Clear Test Environment command (CLR) clears the test environment, and optionally reserves space for a user symbol table for a specified number of symbols. The syntax for the CLR command is as follows:

?CLR [(<symbols>)]

The CLR command deletes all user symbols, user procedures, and user functions. The optional symbols operand is an expression having a positive integer value (greater than zero) which is the number of user symbols that may be defined in the user symbol table. A CLR command allows redefining a test environment. Following a SAVE command, a CLR command prepares the working storage for the start of another test. The user may change the size of the user symbol table with the optional operand. Unless a previous CLR command has changed the symbol table, the capacity of the user symbol table is 30 symbols. The user may reduce the size to provide more memory space for procedures and functions if few symbols are required. The user may increase the size to allow more symbols to be defined. In deciding how large the table should be, the user should notice that LOAD commands define module names as user symbols, PROC and FUNC statements define procedure and function names that are placed in the table, and assign statements define user symbols unless the symbol has previously been defined.

When a CLR command is entered in a compound statement, and the statement executed, completion of execution of the CLR command terminates the compound statement. Any statements within the compound statement following the CLR statement are not executed.

The following is an example of a CLR command:

?CLR (19)	Clear the user symbol table, and delete any user procedures and functions. Provide a user symbol table having a capacity of 19 user symbols.

5.7.12 RESTORE TEST ENVIRONMENT COMMAND. The Restore Test Environment command (RSTR) restores the test environment in the host computer memory and a program in target system memory. The test environment and program are obtained from a file written by a SAVE command. When the SAVE command included an entry point operand, the RSTR command places the entry point in the target system program counter. The following is the syntax for the RSTR command:

?RSTR('<access name>')

The access name is a character string that contains a TX990 file name, or a DX10 pathname or synonym. When the RSTR command completes, the user symbol table, user procedures, and user functions are in the host computer memory in the state stored by the SAVE command, and the program defined in the SAVE command has been loaded into target system memory. Device assignments made in CNSL, LIST, or COPY commands prior to the RSTR command remain in effect.

When an RSTR command is entered in a compound statement and the statement is executed, completion of execution of the RSTR command terminates the compound statement. Any statements within the compound statement following the RSTR command are not executed.

It is possible for there to be more than one TX990 operating system in which the AMPL program executes; systems in which the program occupies different locations in memory. The SAVE and RSTR commands use absolute memory addresses. An RSTR command will not correctly restore the test environment when the SAVE command is executed by the AMPL program in different areas of memory.

In a DX10 system, memory mapping allows the system to place the AMPL program anywhere in memory, and the restriction of the SAVE and RSTR commands to the same system environment does not apply. However, the memory size parameter (paragraph 5.2.2) in effect when the RSTR command is executed must be large enough to accommodate the data stored by the SAVE command.

The following is am example of a RSTR command using a TX990 file name:

```
?RSTR (':MYDBUG/SAV')
```

Restore lab test environment and program at point at which file DSC:MYDBUG/SAV was written.

The following is an example of an RSTR command using a DX10 pathname:

```
?RSTR (VAMPL.SAV.MYDBUG)
```

Restore lab test environment and program at point at which file VAMPL.SAV.MYDBUG was written.

5.7.13 ENTER TEXT EDITOR COMMAND. The Enter Text Editor command (EDIT) saves the test environment and terminates the AMPL program. The command also activates the Text Editor, and returns control to the AMPL program at the end of the editing operation. Then the command restores the stored test environment and resumes the test. The command is supported only in a TX990 system. The following is the syntax for the EDIT command:

```
?EDIT [('<access name>')]
```

The access name is a character string that contains the name of a file. The operand is a TX990 file name of the file to be edited. When the access name is omitted, the file name is DSC:AMPL000/PRC.

The command stores the test environment on file DSC:AMPL000/SAV and transfers control to the program in a file named DSC:TXEDIT/SYS or DSC2:TXEDIT/SYS (on floppy disk unit 1 or 2, respectively). The parameters passed to TXEDIT are the file name (the operand of the command) or DSC:AMPL000/PRC as the file to be edited, file DSC:AMPL000/SCR as the scratch file, and M9000 as the memory size parameter. This allows about 150 lines of code in the editor's buffer. When editing is terminated, the data entered is in the scratch file. TXEDIT displays the message:

```
TEXT IN SCRATCH FILE
TRANSFER TO INPUT?
```

The user should enter a Y for yes to transfer the data to the file named in the EDIT command or to file DSC:AMPL000/PRC. If the user enters N for no, the data may only be accessed on file DSC:AMPL000/SCR, where it will be replaced during the next editing performed with an EDIT command. Control transfers to the program in file DSC:AMPL/SYS (on the left hand floppy disk unit), and the environment is restored from file DSC:AMPL000/SAV.

If the transfer to TXEDIT or back is unsuccessful, the system displays the following message:

```
BATCH ABORTED
```

The user may reactivate AMPL and enter a RSTR command that specifies file DSC:AMPL000/SAV to recover from the error.

The following are examples of EDIT commands:

? EDIT;	Edit file DSC:AMPL000/PRC. Invalid in a DX10 system.
? EDIT ('DSC2:MYPROJ/PRC')	Edit file DSC2:MYPROJ/PRC. Invalid in a DX10 system.

5.7.14 COPY INPUT COMMAND. The Copy Input command (COPY) changes the input source to a specified file or device. The AMPL software reads commands and statements from the specified file or device until it reads an end-of-file, then resumes input from the system console. The syntax for the COPY command is as follows:

?COPY [('<access name>')]

The access name is a character string that contains the name of a file or device. The name of the file or device is a name that is acceptable to the operating system; e.g., a TX990 file or device name, or a DX10 device name or file pathname or synonym. When the operand is omitted in a TX990 system, the COPY command copies the most recently edited file (edited by the Text Editor during execution of an EDIT command). When an EDIT command has not been executed since the AMPL program was activated, an error message is issued. The operand is required in a DX10 system.

When commands and statements are read as specified in a COPY command, they are not displayed on the system console. When a LIST command is in effect, they are written to the list device or file.

A copy file may contain a COPY command that transfers the input to another device or file. This provides chaining, rather than nesting; reading of statements from a file continues until either another COPY statement or an end-of-file is read. When an end-of-file is read, the system console resumes input. Any statements or commands that follow a COPY command on a copy file or in a compound statement are not executed. There is no limit to the number of copy files that may be chained in this way.

The following is an example of a COPY command using a TX990 file name:

?COPY (':AMPLFN/CPY')	Execute the commands and statements on file DSC:AMPLFN/CPY and return input to the system console at end-of-file.

The following is an example of a COPY command using a DX10 pathname:

?COPY ('VAMPL.CPY.AMPLFN')	Execute the commands and statements on file VAMPL.CPY.AMPLFN and return input to the AMPL terminal at end-of-file.

5.7.15 DELETE COMMAND. The Delete command (DELE) deletes the specified user procedure, function, or array from host memory. The syntax for the DELE command is as follows:

?DELE('<name>',['<name>']. . .)

The operands are character strings that contain the names of user procedures, functions, or arrays to be deleted. Deleting a procedure, function, or array makes its memory space available for redefinition of the same procedure, function, or array, or for other purposes.

Executing a DELE command does not delete the name in the user symbol table. The name of a deleted procedure may only be used in a PROC statement as a procedure name, and the name of a deleted function may only be used in a FUNC statement as a function name. The name of a deleted array may only be used in an ARRAY statement as the name of an array with the same number of dimensions. To make these names available for other purposes, the user must execute a CLR command to clear the symbol table.

The following is an example of a DELE command:

```
? DELE ('TIMEN', 'MEMTST', 'NUM')        Delete procedure TIMEN, function
                                         MEMTST, and array NUM.
```

5.7.16 DELAY AMPL EXECUTION COMMAND. The Delay AMPL Execution command (WAIT) delays execution of AMPL software for a specified number of 50 ms periods. The syntax of the WAIT command is as follows:

```
?WAIT(<number>)
```

The operand is an expression that specifies the number of 50 ms periods of delay. During the delay period, execution of AMPL software is suspended. At the end of the delay period the AMPL software resumes execution and displays a question mark to request another command or statement. An operand of 20 provides a delay of one second. The WAIT command is useful when displaying data on the screen of a VDT. When displaying data continuously the data may not remain on the screen long enough to be read by the user. The user may place a WAIT command following each display statement to provide enough time to read the display. The WAIT command is not affected by pressing the key that terminates execution of other commands as described in paragraph 5.3.

The following is an example of a WAIT command:

```
? WAIT(200)                              Delay 10 seconds.
?    . . TEN SECONDS LATER
```

5.7.17 OUTPUT NEW LINE COMMAND. The Output New Line command (NL) outputs a carriage return and line feed to the system console and supplementary listing device, when a supplementary listing device is active. The following is the syntax for the NL command:

```
?NL;
```

No operand is required. The NL command is intended for use in a procedure, function, or compound statement. It is equivalent to a display statement using the N format specification and displaying spaces. The following is an example of an NL command:

```
? NL;
```

5.7.18 VERIFY COMMAND. The Verify command (VRFY) verifies a program in target system memory by comparing the contents of the specified file with the contents of memory starting at the specified bias address. The software prints the contents of the addresses that do not contain the values in the file. The syntax for a VRFY command is as follows:

```
?VRFY('<access name>[,<bias>])
```

The access name is a character string that contains a TX990 file name, or a DX10 pathname or synonym. The bias operand is the address in memory at which the program to be verified was loaded. When the specified file contains an absolute program, the software ignores the bias operand. When the file contains a relocatable program and the bias operand is omitted, the default value of $A0_{16}$ is used.

The VRFY command prints the address, file contents, and memory contents of every address that contains data different from that in the file. The command may be used to verify the loading of a program, or to identify locations within a program that have been altered from the values loaded. Data areas that are reserved by the program using directives such as BSS and BSE cannot be verified with the VRFY command. When the contents of every address agree with the contents of the file, no message is printed.

The following is an example of a VRFY command using a TX990 file name and the messages that identify locations that have been altered:

```
    ?VRFY('DSC:SINTST/OBJ',>1000)
>0114 / >0000  >000E
>015A / >0000  >3C00
>015C / >0000  >6ED8
>015E / >0000  >4004
```

Verify contents of locations loaded from file DSC:SINTST/OBJ at location 1000_{16}.

The following is an example of a VRFY command using a DX10 pathname:

```
? VRFY('VAMPL.OBJ.SINTST')
```

Verify contents of locations loaded from file VAMPL.OBJ.SINTST at location 100_{16}. Absence of a message indicates that no location was altered.

5.7.19 TERMINATE AMPL PROGRAM COMMAND. The Terminate AMPL Program command (EXIT) terminates the program and returns control to the operating system. The syntax for the EXIT command is as follows:

```
?EXIT ('<access name>' ,<start>,<end> ,<entry> )
```

The operands apply only to DX10; if they are entered for TX990, they are ignored. The access name is a character string that contains the pathname of synonym of a file on which the test environment and program are to be stored. When the file does not exist, AMPL software creates the file. When the file contains data, the test environment and program replaces the previous contents of the file. The start operand is the address of the first word of target system memory to be stored, and the end operand is the address of the last word to be stored. The entry operand is the address of the entry point of the stored program. When the EXIT command is entered without operands, the AMPL program terminates. When the operands are included (in a DX10 system), the program performs a SAVE command, assigns the access name operand as the value of synonym $AMPL$E, and terminates. Refer to paragraph 5.7.10 for the effect of omitting the start and end operands or the entry operand. The use of a synonym allows the AMPL SCI command (which activates the AMPL program) to obtain the pathname of the save file and restore the environment and program when the AMPL program is next activated. Use of synonym $AMPL$E for any other purpose, assigning another value to it, or deteting it will prevent proper restoration of the test environment when the AMPL program is next activated.

To resume execution of the AMPL program before loading another program into host memory (when executing under TX990), it is not necessary to load the program again. The following is an example of an EXIT command followed by the entry that restarts the program:

?EXIT;	Terminate the AMPL program.
PROGRAM: >10*	Restart the AMPL program.

When another program has been loaded into host system memory, the AMPL Program must be restarted as follows (under TX990):

```
PROGRAM:  :AMPL/SYS*
```

When executing under DX10, perform the procedure in paragraph 5.2.2 beginning at step 20 to restart the AMPL program. When the EXIT command specified a save file, a start address, and an end address, the test environment and program are restored as the AMPL program begins execution. If synonym $AMPL$E has been altered, the results are unpredictable.

The following is an example of an EXIT command with operands supported by DX10:

?EXIT ('VAMPL.SAV.MYDBUG',>FE00,>FE46,>FE26)	Terminate the AMPL program, storing the test environment on file VAMPL.SAV.MYDBUG. Also store the contents of target memory at address $FE00_{16}$ and ending at address $FE46_{16}$, and store $FE26_{16}$ as the entry point.

The capability of storing the test environment with an EXIT command provides a capability under DX10 similar to that of the EDIT command under TX990. After the execution of the EXIT command, the user may activate the Text Editor to write or edit a file. When the user restarts the AMPL program, the program resumes at the point at which the EXIT command was executed.

5.8 CRU COMMANDS

AMPL software supports two sets of CRU commands. One set, CRUR and CRUW, reads from and writes to a CRU device of the target system. System variable CRUB contains the CRU base address for the read and write operations. The other set, HCRR and HCRW, reads from and writes to a CRU device of the host system. System variable HCRB contains the CRU base address for the read and write operations for the host system.

The user may check out any device on the CRU interface of the target system using CRUR and CRUW. When the target system is a controller or similar device that interfaces with a Model 990 Computer via the CRU interface, it can be connected to the host system, and HCRR and HCRW can be used to verify the CRU interface.

5.8.1 CRU READ COMMAND. The CRU Read command (CRUR) is a system function that reads a specified number of bits from a specified address in the target system CRU. System variable CRUB contains the CRU base address and the command returns the value right-justified in a 16-bit word. The syntax for the command is as follows:

```
?CRUR(<displacement>,<length>)
```

The displacement operand is an expression, the value of which represents a number of CRU lines (either positive or negative) by which the most significant bit read is displaced from the base address in system variable CRUB. The length operand is an expression having a positive integer value in the range of 1 through 16, which represents the number of bits to be read.

The software multiplies the value of the displacement operand by two and adds the product to the base address algebraically. The software then reads the number of bits specified by the length operand, and returns the value right-justified in a 16-bit word.

The following is an example of assigning a base address value and a CRUR command:

?CRUB = >0100	Assign value of 100_{16} as CRU base address.
?CRUR (8,8)	Read 8 bits from target system CRU at address derived from base address of 100_{16} plus 10_{16} (CRU address 88_{16}).

Alternatively, the command and its operands may be used as an expression in an AMPL statement. The value of the expression is the data read from the target system CRU. The following is an example of a CRUR command used in this way, assuming that CRUB remains unaltered from the preceding example:

?CRUOUT = CRUR (0,8)	Read 8 bits from target system CRU and assign the value to user symbol CROUT. The CRU address is derived from CRU base address 100_{16} (CRU address 80_{16}).

5.8.2 CRU WRITE COMMAND. The CRU Write command (CRUW) writes a specified value into a specified number of bits of the target system CRU at a specified address. System variable CRUB contains the CRU base address. The following is the syntax for the command:

?CRUW(<displacement>,<length>,<value>)

The displacement operand is an expression, the value of which represents a number of CRU lines (either positive or negative) by which the most significant bit written is displaced from the base address in system variable CRUB. The length operand is an expression having a positive integer value in the range of 1 through 16, which represents the number of bits to be written. The value operand is an expression, the value of which is written.

The software multiplies the value of the displacement operand by two and adds the product to the base address algebraically. The software then writes the number of bits specified by the value of the length operand. The rightmost bits of the value operand are written to the CRU. The following is an example of the CRUW command, which assumes that CRUB remains unaltered from the preceding example:

?CRUW (16,4,>A)	Write the value 1010_2 into the target system CRU at address derived from base address of 100_{16} plus 20_{16} (CRU address 90_{16}).

5.8.3 HOST CRU READ COMMAND. The Host CRU Read command (HCRR) is a system function that reads a specified number of bits from a specified address in the host system CRU. System variable HCRB contains the CRU base address and the command returns the value right-justified in a 16-bit word. The syntax for the command is as follows:

?HCRR(<displacement>,<length>)

The displacement operand is an expression, the value of which represents a number of CRU lines (either positive or negative) by which the most significant bit read is displaced from the base address in system variable HCRB. The length operand is an expression having a positive integer value in the range of 1 through 16, which represents the number of bits to be read.

The software multiplies the value of the displacement operand by two and adds the product to the base address algebraically. The software then reads the number of bits specified by the length operand, and returns the value right-justified in a 16-bit word.

The following is an example of assigning a base value and an HCRR command:

?HCRB = >0200	Assign value of 200_{16} as CRU base address.
?HCRR(0,8)	Read 8 bits from host system CRU at address derived from base address of 200_{16} plus 0 (CRU address 100_{16}).

Alternatively, the command and its operands may be used as an expression in an AMPL statement. The value of the expression is the data read from the host system CRU. The following is an example of an HCRR command used in this way, assuming that HCRB remains unaltered from the preceding example:

?NUCHAR = HCRR(16,8)	Read 8 bits from target system CRU and assign the value to user symbol NUCHAR. The CRU address is derived from the base address of 200_{16} plus 20_{16} (CRU address 110_{16}).

5.8.4 HOST CRU WRITE COMMAND. The Host CRU Write command (HCRW) writes a specified value into a specified number of bits of the host system CRU at a specified address. System variable HCRB contains the CRU base address. The following is the syntax for the command:

?HCRW(<displacement>,<length>,<value>)

The displacement operand is an expression, the value of which represents a number of CRU lines (either positive or negative) by which the most significant bit written is displaced from the base address in system variable HCRB. The length operand is an expression having a positive integer value in the range of 1 through 16, which represents the number of bits to be written. The value operand is an expression, the value of which is written.

The software multiplies the value of the displacement operand by two and adds the product to the base address algebraically. The software then writes the number of bits specified by the value of the length operand. The rightmost bits of the value operand are written to the CRU. The following is an example of the CRUW command, which assumes that HCRB remains unaltered from the preceding example:

?HCRW (8,8,"BC")	Write the ASCII representation of C into the host system CRU at the address derived from base address of 200_{16} plus 10_{16} (CRU address 108_{16}).

CAUTION

Use extreme care that the CRU base address in HCRB is correct before executing an HCRW command. If an HCRW command is executed with a value in HCRB that causes the command to address one or more lines to the host system peripheral devices, normal operation of the system is impaired.

5.9 DATA INPUT COMMANDS

The AMPL software supports reading a file of data into host or target memory. The file may be prepared using the Text Editor, and must contain valid AMPL expressions separated by spaces. The expressions in the file may contain variables and/or constants separated by arithmetic operators + and - only. The Data Input commands are OPEN, READ, and CLSE. Another command, EOF, also applies to read operations.

5.9.1 OPEN COMMAND. The OPEN command specifies the device to be used for data input. The syntax for the OPEN command is as follows:

?OPEN('<access name>')

The operand is a character string that contains a pathname (as defined for the operating system) of a data file or a device name of a data input device. An OPEN command must be executed prior to reading data from a file or device other than the AMPL terminal or if reading is to be resumed from a file or device other than the AMPL terminal after a Close operation.

The following are examples of OPEN commands using TX990 file and device names:

? OPEN(':INPUT/DAT')	Open file :INPUT/DAT for data input.
? OPEN('ASR')	Open 733 ASR terminal (not system console) for data input. System console does not require entry of an OPEN command.

The following are examples of OPEN commands using DX10 pathnames and device names:

? OPEN('VAMPL.DAT.INPUT')	Open file VAMPL.DAT.INPUT for data input.
? OPEN('ST09')	Open terminal other than AMPL terminal for data input. AMPL terminal does not require entry of an OPEN command.

5.9.2 READ COMMAND. The READ command is a system function that returns the value of the next expression from the data input file or device specified in the current OPEN command, or from the AMPL terminal if an OPEN command is not in effect. The syntax for the READ command is as follows:

? READ;

The READ command may be entered as an AMPL statement to display the value of an expression read from a file or device. Each record may contain only one expression. The typical use of the command is in an assign statement, to assign the value of the expression to a variable or to a target system location. The following are examples of assign statements using READ commands:

?DATIN = READ;	Assign the next constant in the file to DATIN.
?@(START+48) = READ;	Assign the next constant in the file to target system memory location START +48.

When a READ command is executed with no OPEN command in effect (either no OPEN command has been entered or a CLSE command has been entered since the most recent OPEN command), the software requests the user to enter an expression as follows:

=?

The user enters an expression followed by a carriage return. The expression must be either a constant, a variable, or a series of constants and/or variables connected by arithmetic operator + or -. Other operators are invalid and result in an error message. Spaces may be entered as desired. The following are examples of expressions that may be entered:

=? >23C0	The value of the expression is $23C0_{16}$.
=? MYVAR + 0A0	The value of the expression is the sum of the value of variable MYVAR plus $A0_{16}$.
=? MYVAR - 4	The value of the expression is the difference of the value of variable MYVAR minus 4.

When an OPEN command specifies a terminal and a READ command is entered, the AMPL program does not display an equal sign and a question mark as in the preceding examples. The user may enter expressions in the same manner as if the prompt had been displayed.

5.9.3 EOF COMMAND. The EOF command is a system function that returns the state of the end-of-file flag for the file or device. The syntax for the command is as follows:

?EOF;

The EOF command performs a read operation to obtain the current state of the flag, and returns a nonzero value (true) when end-of-file has been set, or a zero value (false) when end-of-file has not been set. A subsequent READ command returns the data read in the read operation performed by an EOF command. When a READ command follows one or more EOF commands in the order of execution, the READ command returns the value obtained by the immediately preceding EOF command; when more than one EOF command is executed prior to a READ command, data is lost.

The following is an example of the EOF command used in an IF statement:

```
?IF␣NOT␣EOF␣THEN␣DATA = READ
```

5.9.4 CLOSE COMMAND. The Close command (CLSE) terminates data input. The syntax for the CLSE command is as follows:

?CLSE;

A CLSE command may be used to terminate data input from a file or device in order to open a different file or device for data input. The user may close a file and then open the file in order to rewind the file and input the data again.

5.10 EMULATOR OPERATION COMMANDS

The emulator replaces the microprocessor of the target system and provides the interface between the host system and the target system. This interface allows the host system to:

- Start the microprocessor
- Stop the microprocessor
- Display the current status of the emulator
- Stop the microprocessor on an external signal
- Stop the microprocessor when it accesses a specified address
- Stop the microprocessor when it writes into a specified address
- Stop the microprocessor when it executes the instruction at a specified address
- Store a specified number of memory addresses accessed and stop the microprocessor when the addresses have all been stored
- Store a specified number of addresses of instructions executed and stop the microprocessor when the addresses have all been stored
- Substitute trace memory for the 512 high-order addresses of target system memory
- Substitute user memory for target system memory, addresses 0000_{16} through $1FFF_{16}$.

The host computer communicates with the emulator through the CRU to control the emulator and ultimately the target system. AMPL software supports the following commands to control the emulator:

- Initialize Emulator (EINT)
- Define Breakpoint Conditions (EBRK)
- Select Event (EEVT)
- Initialize Compare Logic (ECMP)
- Initialize Trace Logic (ETRC)
- Start Microprocessor (ERUN)

- Stop Microprocessor (EHLT)
- Read Trace Memory (ETB)

There are several system variables that relate to emulator operation. Where a system variable relates directly to a command, the variable is discussed with the command. System variables ETM and EUM control the mapping of the target system memory, and relate to all target system operations. These variables may be displayed or changed. The following are examples of displaying ETM and EUM:

```
    ?ETM:B
<0000000000000001
```

Display system variable ETM in binary mode. Value of 1 indicates that target system memory addresses $FE00_{16}$ through $FFFF_{16}$ (TMS 9900) or $3E00_{16}$ through $3FFF_{16}$ (TMS 9980) are mapped into Emulator trace memory (figure 4-1).

```
    ?EUM:B
<0000000000000001
```

Display system variable EUM in binary mode. Value of 1 indicates that target system memory addresses 0 through $1FFF_{16}$ are mapped into Emulator user memory (figure 4-1).

To change the values of ETM and EUM to zero, mapping all target system memory addresses into target system memory, use assign statements as follows:

```
?ETM = 0
?EUM = 0
```

Set system variable ETM to zero.
Set system variable EUM to zero.

System variable EMT identifies the type of microprocessor being emulated. This variable may be displayed and tested, but may not be altered by the user. System variable EMT has a value of zero when a TMS 9980 buffer is connected to the emulator, and a value of one when a TMS 9900 buffer is connected to the emulator. The following are examples of the use of system variable EMT:

```
  ? EMT:B1
1
```

Display the value of system variable EMT. The result indicates that a TMS 9900 microprocessor is being emulated.

```
  ? IF EMT THEN 'TMS 9900' ELSE 'TMS 9980'
TMS 9980
```

Display message appropriate to type of microprocessor being emulated.

5.10.1 INITIALIZE EMULATOR COMMAND. The Initialize Emulator command (EINT) selects and initializes an emulator to execute subsequent emulator control commands. The syntax for the command is as follows:

```
?EINT('<device name>')
```

The operand is the device name assigned during system generation. In the TX990 system supplied by Texas Instruments, the device name is EMU. For a DX10 system, the device name is EM01 (for the first emulator). The EINT command must be entered prior to accessing target memory. Once the command has been entered, it need not be entered again unless the AMPL program is reloaded or unless there is more than one emulator in the system. When there is more than one emulator, an EINT command must be entered prior to using a different emulator.

The following are examples of EINT commands using TX990 device names:

?EINT('EMU')	Initialize EMU as the active emulator.
?EINT('EMU2')	Initialize EMU2 as the active emulator.

The following are examples of EINT commands using DX10 device names:

? EINT('EM01')	Initialize EM01 as the active emulator.
? EINT('EM02')	Initialize EM02 as the active emulator.

5.10.2 DEFINE BREAKPOINT CONDITIONS COMMAND. The Define Breakpoint Conditions command (EBRK) defines the conditions for breakpoints, and the action the emulator takes when a breakpoint occurs. The syntax for the command is as follows:

```
?EBRK[( EVT       [, SELF ])]
      ( FULL      [, OFF  ])
      ( EVT+FULL  [, SELF ])
      ( OFF       [, OFF  ])
```

The first operand specifies breakpoint conditions. When this operand is the keyword EVT, a breakpoint occurs on an event as defined by the EEVT command. When the operand is the keyword FULL, a breakpoint occurs when a specified number of trace addresses have been stored or when the trace memory overflows. When the breakpoint condition operand is the keyword EVT+FULL, a breakpoint occurs either when an event occurs or when the specified number of trace addresses have been stored. When the operand is the keyword OFF, no breakpoint occurs.

The second operand specifies breakpoint action. When the operand is the keyword SELF, a breakpoint consists of stopping the microprocessor (target system) at the completion of the currently executing instruction. When the operand is the keyword OFF, a breakpoint consists of sending a breakpoint signal to the trace module and incrementing the count in system variable ENI without stopping the microprocessor. When the breakpoint action operand is omitted, whichever keyword was entered most recently in an EBRK command remains in effect.

The EBRK command may be entered with no operands to display the breakpoint conditions and action in effect. When this is done prior to entering an EBRK command with one or more operands, the initial breakpoint conditions and action are displayed, as follows:

```
?EBRK;
+OFF +OFF
+OFF
```

This indicates that the two breakpoint conditions are off and that the breakpoint action is also off. The first keyword on the first line is either +EVT or +OFF, and means that the breakpoint on an event is on or off, respectively. The second keyword on that line is either +FULL or +OFF, and means that the breakpoint on completion of a trace operation is either on or off, respectively. The keyword on the second line is either SELF or OFF and means that the breakpoint action of stopping the microprocessor is either on or off, respectively.

The following are examples of EBRK commands:

? EBRK(EVT)	Enable breakpoint on event only. The event is defined by an EEVT command. Breakpoint action remains unchanged.
? EBRK(FULL)	Enable breakpoint on completion of trace only. Breakpoint action remains unchanged.
? EBRK(EVT+FULL)	Enable breakpoint on event or on completion of trace. Breakpoint action remains unchanged.
? EBRK(OFF)	Disable all breakpoints. Breakpoint action (which would apply by default in a subsequent EBRK command) remains unaltered.
? EBRK(EVT,SELF)	Stop the microprocessor when an event occurs.
? EBRK(FULL,OFF)	Breakpoint on trace completion. Breakpoint results in signal to trace module and is counted, but does not stop the target system.

5.10.2.1 System Variable ENI. System variable ENI maintains a count of emulator interrupts. An emulator interrupt occurs at each breakpoint while the breakpoint action operand is set to OFF. The count is set to zero when the target system is started. The user may display the contents of ENI, but may not assign a value to this variable. The following is an example of a display of ENI:

? ENI:D 25	Display the value of system variable ENI.

5.10.3 SELECT EVENT COMMAND. The Select Event command (EEVT) selects the event for breakpoints. The syntax for the EEVT command is as follows:

?EEVT[({INT / EXT})]

When the user enters the keyword INT, the event that causes a breakpoint is the internal comparison circuitry of the emulator. When the user enters the keyword EXT, an external signal is the event that causes the breakpoint. The signal is connected to a terminal (P7) on the emulator module.

The user may enter an EEVT command with no operand to display the event selection in effect. When this is done prior to entering an EEVT command with an operand, the initial event selection is displayed as follows:

```
?EEVT;
+INT
```

This indicates that the internal comparison circuitry of the emulator triggers a breakpoint. When the keyword EXT is displayed, the external breakpoint signal triggers a breakpoint.

When the EVT keyword of the EBRK command is in effect, an EEVT command that specifies keyword EXT sets up a breakpoint on the external signal. When the external signal connected to P7 goes low the breakpoint is effective. An EEVT command that specifies keyword INT sets up a breakpoint on an internal comparison defined in an ECMP command.

The following are examples of EEVT commands:

? EEVT (INT)	Select internal comparison signal as event.
? EEVT (EXT)	Select external signal as event.

5.10.4 INITIALIZE COMPARE LOGIC COMMAND. The Initialize Compare Logic command (ECMP) initializes the emulator comparison logic to compare a specified type of address to a specified address. The syntax for the ECMP command is as follows:

```
?ECMP[( ADDR          )[,<address>])]
      ( [ADDR]-DBIN   )
      ( [ADDR+]IAQ    )
      ( OFF           )
```

The first operand is the address type (qualifier) operand. When this operand is the keyword ADDR, all addresses on the address bus (for read, write, or instruction acquisition accesses) are compared to a specified value. When the operand is the keyword ADDR-DBIN the address on the address bus is compared only when DBIN is false (write cycles only). When the address type operand is the keyword ADDR+IAQ, the address on the address bus is compared only when IAQ (Instruction Acquisition) is true (program counter address). When the operand is the keyword OFF, comparison is inhibited.

The second operand, the address operand, is the value to which the address bus value is compared. When the target system uses a TMS 9900 microprocessor, the second operand may be any value in the range of 0 through $FFFF_{16}$. When the target system uses a TMS 9980 microprocessor the range of valid addresses is 0 through $3FFF_{16}$. No event can occur if the second operand is greater than $3FFF_{16}$.

The second operand has no significance when the first operand is the keyword OFF. When the address operand is omitted, the address in the most recent ECMP command is used in the comparison. The user may enter an ECMP command with no operands to display the address type and address value in effect. When this is done prior to entering an ECMP command with operands, the initial address type and value are displayed as follows:

```
? ECMP;
OFF
>0000
```

This indicates that the comparison logic is disabled and that the address operand is 0. The keyword on the first line may be any of the four keywords used as the address type operand and identifies the type of address being compared. The second line displays the current address operand.

For the comparison specified in an ECMP command to result in a breakpoint, an EEVT command must set event to INT and an EBRK command must set the breakpoint condition to EVT or to EVT+FULL.

The following are examples of ECMP commands:

? ECMP (ADDR,>FF46)	Compare addresses of all memory accesses to $FF46_{16}$.
? ECMP (-DBIN,>FF00)	Compare addresses of all write memory accesses to $FF00_{16}$.
? ECMP (IAQ)	Compare addresses of all instruction acquisition memory accesses to the most recently entered address operand ($FF00_{16}$, if the preceding example is the most recently entered ECMP command).
? ECMP (OFF)	Inhibit comparison.

5.10.5 INITIALIZE TRACE LOGIC COMMAND. The Initialize Trace Logic command (ETRC) specifies the type of address to be stored, the number of addresses to be stored, and the clock for storing the addresses. The syntax for the ETRC command is as follows:

$$\text{?ETRC } (\left\{\begin{matrix}\text{ADDR}\\ [\text{ADDR+}]\,\text{IAQ}\\ \text{OFF}\end{matrix}\right\}[,\langle\text{count}\rangle[,\left\{\begin{matrix}\text{INT}\\ \text{EXT}\end{matrix}\right\}]])$$

The first operand specifies the type of addresses to be stored. When the operand is the keyword ADDR, all addresses on the address bus (for read, write or instruction acquisition accesses) are stored. When the operand is the keyword ADDR+IAQ or the keyword IAQ, addresses on the address bus are stored only when IAQ (Instruction Acquisition) is true (program counter addresses). When the address type operand is the keyword OFF, no addresses are stored.

The second operand is the count operand, which may be any valid expression having a value in the range of 1 through 256. The count operand has meaning only when an EBRK command with a condition operand of FULL or EVT+FULL and an action operand of SELF is in effect. Under these conditions, the count operand specifies the number of values to be stored during the trace operation. When breakpoint conditions and action is effect do not halt the target system, addresses are stored until the target system is halted in some other manner, overwriting previously stored addresses when the 256-word trace memory becomes full. When the count operand is omitted, the most recently entered count operand applies.

The third operand selects the clock enable signal for storing the addresses. When the operand is the keyword INT, the operation is clocked by the internal target system clock. When the operand is the keyword EXT, the trace operation is clocked by an external enable signal, normally from the trace module. When the clock operand is omitted, the most recent clock selection applies.

A trace operation consists of storing the specified type of addresses. When the breakpoint condition in effect is either FULL or EVT+FULL and the breakpoint action in effect is SELF, the operation completes a breakpoint occurs when the number of addresses specified as the count has been stored. The stored data is accessible using the ETB command. The data is stored in the trace memory of the emulator module. System variable ETM should be set to zero during a trace operation. If ETM is set to one and if the target system accesses one or more addresses above $FE00_{16}$, interference between the target system and the trace operation occurs.

Whenever tracing is enabled, addresses are stored in trace memory. When the breakpoint action operand of the EBRK command in effect is OFF, completion of a trace operation or overflow of trace memory does not halt the target system. If the target system runs long enough, all 256 words of trace memory will have been filled with stored addresses when the target system halts. The trace memory contains up to 256 traced addresses or the most recently stored 256 traced addresses when it is halted.

The user may enter an ETRC command with no operands to display the address type, count, and clock operands in effect. When this is done prior to entering an ETRC command with operands, the initial address type, count, and clock are displayed as follows:

```
    ?ETRC;
OFF
    256
+INT
```

This indicates that the trace logic is disabled, the trace count is set to 256, and the trace is internally clocked. The keyword on the first line may be any of the three keywords used as the address type operand and identifies the type of addresses traced. The value on the second line is the count of addresses to be traced. The keyword on the third line is either +INT or +EXT, and indicates internal or external clock, respectively.

The following are examples of ETRC commands:

? ETRC (ADDR,10,INT)	Trace ten addresses, storing an address at each memory access using internal clock.
? ETRC (OFF)	Turn trace off.

In using the ETRC command in a TMS 9980 system the count operand may be odd, but an even number of addresses is always traced. This is because the TMS 9980 accesses two bytes for each word accessed, and tracing does not stop until the currently executing instruction completes execution. The following example traces one address in a TMS 9900 system, and two addresses in a TMS 9980 system:

? ETRC (IAQ,1,INT)	Trace one instruction, storing an address at each memory access of the instruction word. If the FULL and SELF options of the EBRK command are in effect, this results in execution of a single instruction each time the microprocessor is started.

In a TMS 9900 system, emulator trace memory contains the instruction address at the completion of the trace operation. In a TMS 9980 system, emulator trace memory contains the addresses of both bytes of the instruction (or of the first word of the instruction). Storing of the address of the even byte can be inhibited when the emulator and trace modules are set up to trace simultaneously, the clock operand of the ETRC command is EXT, and the trace module is set up to trace only when the address is odd. paragraph 5.9.5 describes the TTRC command for setting up tracing in the trace module.

5.10.6 START MICROPROCESSOR COMMAND. The Start Microprocessor command (ERUN) clears the Hold signal to the emulator, starting the microprocessor. Execution begins at the address in system variable PC, with the target system workspace pointer set to the value in system variable WP and the target system status register set to the value in system variable ST. The syntax for the command is as follows:

```
?ERUN;
```

In many cases, a program would have been loaded into the target system memory. If an entry point were not specified in the source code, the user would assign the entry address to variable PC. Similarly, system variables WP and ST are set to the required values. The user could define a breakpoint using an EBRK command, and EEVT command, and an ECMP command. Alternatively, the user could define a breakpoint on completion of a trace operation using an EBRK command and an ETRC command, or define both breakpoints. Or the user may allow the program to execute to its normal termination or stop it with an EHLT command.

The following is an example of a set of statements and commands to load and execute a program consisting of a single instruction:

`?@>100 = >10FF`	Single instruction loop into address 100_{16}.
`?PC = >100`	Set PC.
`?WP = >200`	Set WP.
`?ST = 0`	Set ST.
`?ERUN`	Start target system.

5.10.6.1 System Variable EST. The software maintains the current status of the emulator in system variable EST. The user may display the value of EST at any time, but may not assign a value to this variable. The following is an example of a display of EST:

```
   ? EST;
EST = >0005
```

Display status of emulator in system variable EST. Result indicates that the target system is running and that the event logic has generated a breakpoint.

The status value is returned in the least significant digit, as shown in table 5-1. System variable EST may also be used with a mask value to test for a specific status condition, as in the following example:

```
? IF@EST@AND@1@THEN@ 'EMULATOR IS RUNNING'
```

Print message if least significant bit of system variable EST is true.

A mask value of 1 tests the status bit that is true when the emulator is running. A mask value of 2 tests the status bit that is true when a trace overflow breakpoint has occurred. A mask value of 4 tests the status bit that is true when an event breakpoint has occurred. A mask value of 8 tests the status bit that is true when an Idle instruction is being executed.

5.10.7 STOP MICROPROCESSOR COMMAND. The Stop Microprocessor command (EHLT) sends the Hold signal to the emulator, halting the microprocessor. The syntax for the EHLT command is as follows:

```
?EHLT;
```

Table 5-1. Emulator Status

Contents of Least Significant Digit of System Variable EST (Hexadecimal)	Status
0	Target system is halted. Neither an event or trace overflow has caused a breakpoint.
1	Target system is running and is not executing an Idle instruction. Neither an event or trace overflow has caused a breakpoint.
2	Target system is halted, a trace overflow breakpoint has occurred, and an event breakpoint has not occurred.
3	Target system is running and is not executing an Idle instruction. A trace overflow breakpoint has occurred and an event breakpoint has not occurred.
4	Target system is halted. An event breakpoint has occurred, and a trace overflow breakpoint has not occurred.
5	Target system is running and is not executing an Idle instruction. An event breakpoint has occurred, and a trace overflow breakpoint has not occurred.
6	Target system is halted and both an event and a trace overflow breakpoint have occurred.
7	Target system is running and is not executing an Idle instruction. Both an event and a trace overflow breakpoint have occurred.
9	Target system is running, executing an Idle instruction. Neither an event or trace overflow has caused a breakpoint.
B	Target system is running, executing an Idle instruction. A trace overflow breakpoint has occurred, and an event breakpoint has not occurred.
D	Target system is running, executing an Idle instruction. An event breakpoint has occurred, and a trace overflow breakpoint has not occurred.
F	Target system is running, executing an Idle instruction. Both an event and a trace overflow breakpoint have occurred.

When an expected breakpoint is never reached, or when no breakpoint is in effect, the microprocessor may be stopped by executing an EHLT command. The command displays the emulator status (system variable EST). Since the status is read after the microprocessor is halted, only the even numbers in table 5-1 apply. The following is an example of an EHLT command:

? EHLT >0002	Halt the microprocessor. The displayed status indicates that the target system is halted, an event breakpoint has not occurred, and a trace overflow breakpoint has occurred.

5.10.8 READ TRACE MEMORY COMMAND. The Read Trace Memory command (ETB) is a system function that reads the specified value from the emulator trace memory. The syntax for the command is as follows:

?ETB(<index>)

The index operand may be any valid expression. When the value of the expression is a valid index to the emulator trace memory, the command prints the value stored in the word corresponding to the index value. The limits of valid index values can be determined by displaying the values of two system variables. System variable ETBN contains the index value corresponding to the most recently stored word. ETBN contains 0 at the completion of a trace operation. System variable ETBO contains the index value corresponding to the first word stored (oldest word).

The emulator stores values in trace memory when tracing is initialized by an ETRC command, and the target system is started. The last value stored during the trace operation is accessible using an index value of 0. The values stored prior to the completion of the trace operation are accessed using negative index values, with the most negative index value corresponding to the value first stored.

TMS 9900 addresses are 16-bit addresses, and occupy all 16 bits of the trace memory words. TMS 9980 addresses are 14-bit addresses; the trace memory stores these addresses in the least significant 14 bits of trace memory words, with zeros in the two most significant bits. The TMS 9980 accesses two bytes for each word accessed by the TMS 9900. Unless the trace module is used to inhibit tracing both addresses, the trace memory in the emulator contains an even and an odd address for each word accessed.

Alternatively, an ETB command and its operand may be used as a variable in an expression in an AMPL statement. The value stored in trace memory corresponding to the value of the operand becomes the value of the command when used as a variable.

The user should enter an ETB command immediately following the trace operation. If an ERUN command is entered following the trace operation and before the ETB command, traced values will be lost unless the trace is turned off before starting the microprocessor. Also, if system variable ETM is set to one following the trace, and if an ERUN command is entered, contents of the trace memory may be altered if the microprocessor accesses target system memory addresses $FE00_{16}$ through $FFFF_{16}$ (TMS 9900) or $3E00_{16}$ through $3FFF_{16}$, $7E00_{16}$ through $7FFF_{16}$, $BE00_{16}$ through $BFFF_{16}$, or $FE00_{16}$ through $FFFF_{16}$ (TMS 9980).

The following are examples of ETB commands:

```
    ? ETB (Ø)
>1056
```

Display the most recently stored address.

```
    ? ETB (-1)
>1054
```

Display the address stored prior to the address displayed in the preceding example.

The following is an example of a series of AMPL statements that use the ETB command as a variable:

```
    ? N = ETBO
    ? WHILEØNØLEØETBNØDOØBEGIN
1?      ETB(N):HN
1?      N = N+1
1? END
```

Initialize variable N to the index of the oldest item stored. Display the stored value and increment N. Perform display until all traced values have been displayed.

When the user enters an index value greater than the value of ETBN or less than the value of ETBO, the software prints an error message.

5.10.8.1 System Variable ETBO. System variable ETBO is set to the index value of the oldest value stored in memory by the emulator module. ETBO may not be assigned a value by the user, but may be displayed whenever the emulator is not running. The value of ETBO is the smallest (most negative) value that may be used as an index in an ETB command. When this value is 1 and the value of ETBN is 0, no values have been stored and an ETB command cannot be executed. The following is an example of a display statement to display ETBO:

```
    ? ETBO
ETBO= >FFFE
```

Display the index to the oldest address stored in emulator trace memory.

5.10.8.2 System Variable ETBN. System variable ETBN is set to the index value of the most recently stored value in memory by the emulator module. ETBN may not be assigned a value by the user, but may be displayed whenever the emulator is not running. The value of ETBN is zero, the largest (most positive) value that may be used as an index in an ETB command. The following is an example of a display statement to display ETBN:

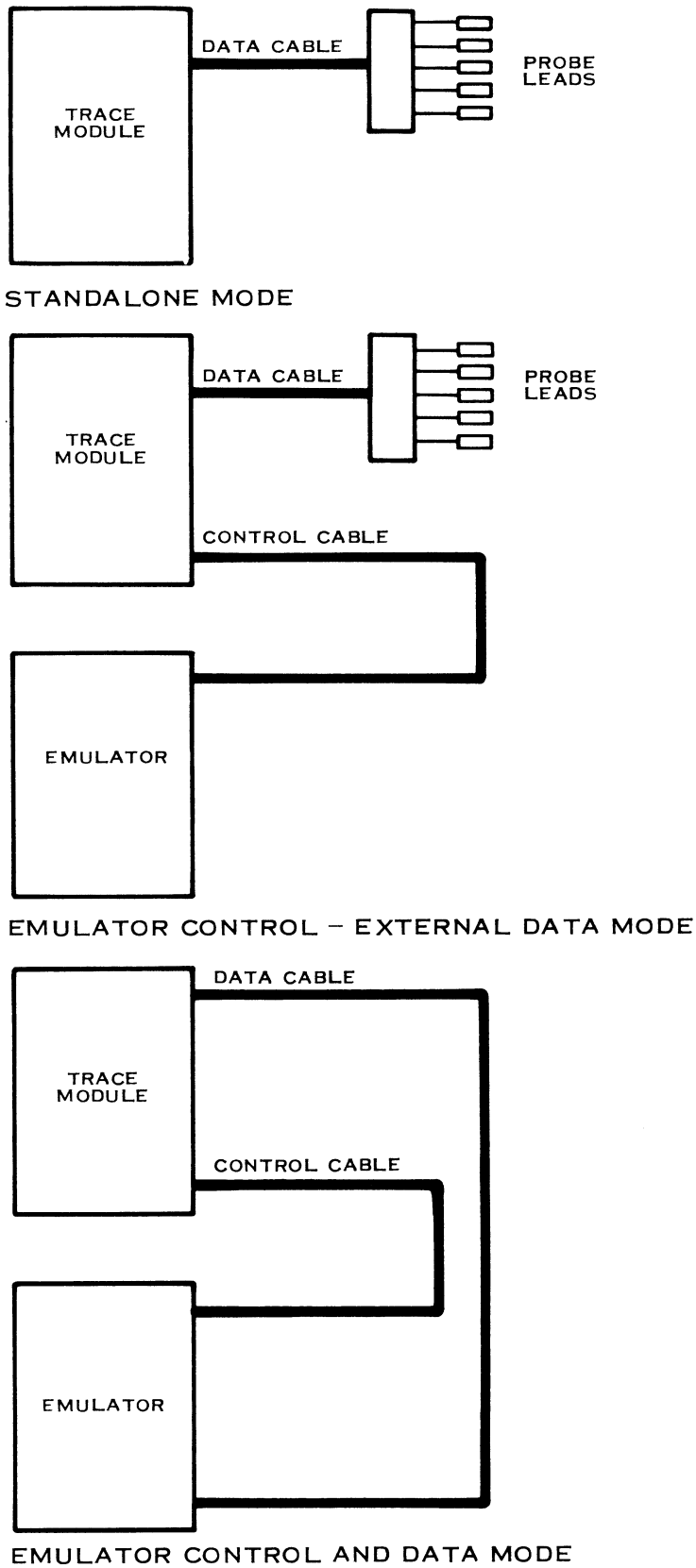
```
    ? ETBN
ETBN= >ØØØØ
```

Display the index to the newest address stored in emulator trace memory.

5.11 TRACE MODULE OPERATION COMMANDS

The trace module stores and analyzes up to 256 20-bit values under control of the host computer and optionally in conjunction with the emulator module. The clock used for storing and analyzing traced values may be either an internal 10 MHz clock or an external clock up to 10 MHz. Four qualifiers may be used to select a clock as a trigger for storing and analyzing a value. Analyzing a value consists of comparing selected bits of the value to corresponding bits of a value stored in a comparison register.

The trace module may be connected in any of three modes as shown in figure 5-1. In the stand-alone mode, the data cable with probe leads is connected to the trace module and no control cable is connected. This allows tracing of up to 20 logic signals to which probe leads are connected, under control of up to four qualifiers (logic signals to which other probe leads are connected). Two additional probe leads may be connected to an external clock and an external event signal.



(A)136543

Figure 5-1. Trace Module Connections

In the emulator control-external data mode, the data cable with probe leads is connected to the trace module to provide data and control input as in the stand-alone mode. A control cable connects the trace module to the emulator module to synchronize tracing. In this mode, tracing does not begin until the target system is started by the emulator, and the trace module may request the emulator to halt the target system at the completion of the current instruction.

In the emulator control and data mode, both the input data cable and the control cable connect the trace module to the emulator module. This allows tracing of either memory addresses or memory data using qualifiers and external clock from the emulator module. When the target system uses a TMS 9900 microprocessor, only three qualifiers are used. When the target system uses a TMS 9980 microprocessor, all four qualifiers are used. The qualifiers are connected to the following emulator signals:

- The inversion of the least significant address bit, true when an even address is on the address bus (TMS 9980); true at all times (TMS 9900).
- Instruction Acquisition (IAQ), true during a memory cycle that reads an instruction from memory.
- Data Bus In (DBIN), true during memory read cycles.
- Emulator Event (EMU), true during memory cycles in which comparison logic in the emulator indicates an equal comparison.

When the data cable connects the trace module to the emulator module, the external event signal is connected to the emulator event signal in the emulator, and the external clock signal is connected to the emulator clock.

In the stand-alone mode, operation of the trace module begins when a Start Trace command is executed. Otherwise, tracing begins when the target system is started following execution of a Start Trace command.

The input data is stored on each selected clock. When the breakpoint is defined, tracing may be stopped at the occurrence of a breakpoint. Otherwise, the user may stop tracing by executing a Stop Trace command, or the emulator may halt the trace module (except in the stand-alone mode). Storing of data continues until the module halts, overlaying previously stored data after 256 values have been stored. Thus, when the module halts, the trace memory contains the 256 most recently stored values.

When the trace module is connected in the emulator control and data mode, either addresses or data may be stored in the trace module. When the target system uses a TMS 9900 microprocessor, the addresses and the data are both 16-bit words. When the target system uses a TMS 9980 microprocessor, the address is a 14-bit address and the data is accessed in 8-bit bytes. The 14-bit address is placed on the address bus in the 14 least significant bits and the two most significant bits are at logic zero. The 8-bit data bytes are combined to form a 16-bit word under control of the least significant bit of the address. On an even address (least significant bit equals zero) the 8 data bits are placed in the most significant half of the data bus. On an odd address (least significant bit equals one) the 8 data bits are placed in the least significant half of the data bus, and the most significant half retains its previous contents. Thus during memory accesses at even addresses the data bus only contains the most significant half of the word. During memory accesses at odd addresses the data bus contains the word at the next lower even address.

A trace operation may be defined in terms of a number of values to be stored, and a breakpoint may be specified to occur when the trace operation completes.

The comparison logic of the trace module compares selected bits of the data being stored with the data in the comparison register. An event occurs when the result of the comparison is equal. Alternatively, an external signal identifies an event. The trace module contains both an event counter and a delay counter, each of which is set to a value supplied by the user. The event counter decrements when an event occurs. When the event counter counts down to zero, the delay counter begins decrementing each time a value is stored in trace memory. When the delay count reaches zero, the event and delay completion occurs, which may optionally cause a breakpoint.

When a breakpoint occurs at completion of a trace operation or of an event and delay, the trace module may interrupt the host computer without stopping the trace or requesting the emulator to stop. Optionally, the breakpoint may interrupt the host computer and stop the trace but not request the emulator to stop. Another option is that of requesting the emulator to stop the target system at completion of the currently executing instruction without interrupting the host computer. Tracing continues until the emulator stops the target system in this case. A fourth option requests the emulator to stop the target system at completion of the currently executing instruction and also stops the trace. Only the first two options apply to the stand-alone mode.

When tracing is stopped by the emulator following a request from the trace module, it is possible for as many as eight memory clock cycles to occur in the emulator between the time the request is issued and the time the trace stops. If the trace module is following the emulator clock, there could be eight values stored during this time depending on the qualifiers specified. When the trace module is internally clocked, the number of values stored during this time is a function of the emulator clock period, the qualifiers, and the currently executing instruction. When the sum of the number of additional clock cycles plus the specified delay exceeds 255, the trace memory is filled with values stored after the last event and previously stored values are lost.

The host computer communicates with the trace module through the CRU to define the addresses and options for the trace. The AMPL software supports the following commands to control the trace module:

- Initialize Trace Module (TINT)
- Define Trace Breakpoints (TBRK)
- Select Trace Event (TEVT)
- Initialize Trace Compare Logic (TCMP)
- Initialize Trace Module Trace Logic (TTRC)
- Start Trace (TRUN)
- Stop Trace (THLT)
- Read Low Order Trace Module Memory (TTB)
- Read High Order Trace Module Memory (TTBH)

5.11.1 INITIALIZE TRACE MODULE COMMAND. The Initialize Trace Module command (TINT) selects and initializes a trace module to execute subsequent trace module control commands. The syntax for the command is as follows:

```
?TINT('<device name>')
```

The operand is the device name assigned during system generation. In the TX990 system supplied by Texas Instruments the device name is TRA. For a DX10 system, the device name is TM01 (for the first trace module). Once the command has been entered it need not be entered again unless the AMPL program is reloaded or unless there is more than one trace module in the system. When there is more than one trace module, a TINT command must be entered prior to using a different trace module.

The following are examples of TINT commands using TX990 device names:

?TINT('TRA')	Initialize TRA as the active trace module.
?TINT('TRA2')	Initialize TRA2 as the active trace module.

The following are examples of TINT commands using DX10 device names:

? TINT('TM01')	Initialize TM01 as the active trace module.
? TINT('TM02')	Initialize TM02 as the active trace module.

5.11.2 DEFINE TRACE BREAKPOINT COMMAND. The Define Trace Breakpoint command (TBRK) defines the conditions that cause a breakpoint and the action that the breakpoint causes. The syntax for the TBRK command is as follows:

```
?TBRK[( EVT       [, SELF      ])]
      ( FULL      [, EMU       ])
      (EVT+FULL   [,SELF+EMU   ])
      ( OFF       [, OFF       ])
```

The first operand specifies breakpoint conditions. When the keyword EVT is entered, a breakpoint consists of an event and delay completion. When the keyword FULL is entered, a breakpoint consists of a trace completion. When the keyword EVT+FULL is entered, a breakpoint may be either an event and delay completion or a trace completion. When the keyword OFF is entered, neither completion causes a breakpoint.

The second operand specifies a breakpoint action. When the keyword SELF is entered, a breakpoint causes the trace operation to stop and an interrupt to be sent to the host computer. When the keyword EMU is entered, a breakpoint causes the trace module to request the emulator to halt at the completion of the currently executing instruction. When the keyword SELF+EMU is entered, a breakpoint causes the trace operation to stop and requests the emulator to halt at the completion of the currently executing instruction. When the keyword OFF is entered, a breakpoint causes the trace module to interrupt the host computer. When the breakpoint action operand is omitted, the most recently entered breakpoint action operand of a TBRK command continues to apply.

All four keywords that may be used as the breakpoint condition operand apply to all connection modes. However, when the keyword OFF is entered and the trace module is connected in the stand-alone mode, a THLT command must be entered to stop the trace. When the keyword OFF is entered in either of the other connection modes, tracing stops whenever the emulator stops. When the EVT or EVT+FULL keyword is entered, this command operates in conjunction with a TEVT and TCMP command to breakpoint after a specified delay following a specified event. When the FULL or EVT+FULL keyword is entered, this command operates in conjunction with a TTRC command to breakpoint when a specified number of values have been stored in trace memory. System variable TNE (described in a subsequent paragraph) maintains a count of events.

Keywords EMU and SELF+EMU are not relevant as the breakpoint action operand in the stand-alone connection mode. All breakpoint action keywords are relevant in the other two connection modes. When keywords EMU and SELF+EMU are used, a breakpoint causes the trace module to request the emulator to halt the target system. Unless the SELF keyword of an EBRK command is in effect as the breakpoint action operation for the emulator, the emulator will not halt the target system. When the OFF keyword has been entered, the trace module interrupts the host computer when a breakpoint occurs.

The TBRK command may be entered with no operands to display the breakpoint conditions and action in effect. When this is done prior to entering a TBRK command with one or more operands, the initial breakpoint conditions and action are displayed, as follows:

```
? TBRK;
+OFF +OFF
+OFF +OFF
```

This indicates that the two breakpoint conditions are off, and the two breakpoint actions are off. The first keyword on the first line is either +EVT or +OFF and means that an event and delay completion breakpoint is on or off, respectively. The second keyword on that line is either +FULL or +OFF, and means that the breakpoint on completion of a trace is either on or off, respectively. The first keyword on the second line is either +SELF or +OFF, and means that the stop trace breakpoint action is either on or off, respectively. The second keyword on the second line is either +EMU or +OFF, and means that the send request to emulator breakpoint action is either on or off, respectively.

The following are examples of TBRK commands:

? TBRK (EVT+FULL,SELF+EMU)	Define a breakpoint to occur on completion of an event and delay or of a trace operation, and to stop the trace operation immediately and request the emulator to stop at completion of the current instruction.
? TBRK (FULL)	Define a breakpoint to occur on completion of a trace operation only. Breakpoint action is that defined in previous command.
? TBRK (EVT,OFF)	Define breakpoint to occur on completion of an event and delay operation only, and to interrupt the host computer without stopping the trace.
? TBRK (OFF)	Inhibit breakpoints.

5.11.2.1 System Variable TNE. System variable TNE is set to zero when the trace is started, and counts events. TNE may not be assigned a value by the user, but may be displayed whenever the trace module is not running. The following is an example of a display statement to display TNE:

```
   ? TNE
>0004
```

Display the count of events that have occurred since the most recent TRUN command.

Counting of events is independent of the count specified for a breakpoint, and does not stop until the trace stops. When a delay is specified, events that occur during the delay period cause the final event count to exceed the count specified as a breakpoint. The count is maintained as an unsigned number.

5.11.2.2 System Variable TNI. System variable TNI is set to zero when the trace is started, and counts interrupts. TNI may not be assigned a value by the user, but may be displayed at any time. The following is an example of a display statement to display TNI:

```
   ? TNI
>0010
```

Display the count of interrupts that have occurred since the most recent TRUN command.

The count in system variable TNI is maintained as an unsigned number.

5.11.3 SELECT TRACE EVENT COMMAND. The Select Trace Event command (TEVT) selects an event by specifying an event count, a delay count, and the event mode. The syntax for the TEVT command is as follows:

```
?TEVT[(<events>[,<delays>[,[NORM] + [EACH] + [INT]]])]
      [INV]  [EDGE]  [EXT]
```

The first operand specifies the number of events. The operand may be any valid expression having a value of zero through 65,535 to be placed in the event counter. Since the maximum positive value that can be expressed as a decimal number is 32,767, values greater than this must be entered as hexadecimal, octal, or binary values. A value of zero provides an event count of 65,536.

An event may be the equal comparison result of an internal comparison specified in a TCMP command (paragraph 5.9.4) or an external signal to which the EVENT probe lead is connected, determined by the mode operand described in a subsequent paragraph. The user specifies a number of events to be counted before initiating the delay period. Thus the first event or a specific subsequent event is the first condition for a potential breakpoint.

The second operand provides the delay value. It may be any expression having a value of zero through 255 to be placed in the delay counter. When the delay operand is omitted, the most recently entered delay operand continues to apply.

The delay count specifies a number of items to be traced after the event count has been satisfied. The potential breakpoint occurs after the specified items have been traced. The user specifies a delay to obtain a trace of desired items immediately following the event.

The third operand selects the event mode. The operand consists of a single keyword, the sum of two keywords, or the sum of three keywords. Keyword NORM specifies decrementing the event counter when the selected event signal is true. Keyword INV specifies decrementing the counter on a false signal. When neither NORM nor INV is specified, a true signal decrements the count. Keyword EACH specifies decrementing the counter at each selected event signal. Keyword EDGE specifies decrementing the counter only when the specified value of the specified event signal is different from the value at the previous clock. When neither EACH nor EDGE is specified, the count is decremented at each signal. Keyword INT selects the equal comparison result of the internal comparison logic as the event. Keyword EXT selects the external event signal as the event. In the stand-alone and emulator control (external data connection modes), the external event signal is connected by the user to an appropriate logic signal. In the emulator control and data connection mode, the external event is the emulator event signal. When neither INT or EXT is specified, the internal event is selected. When the entire operand is omitted, the most recently entered event mode operand continues to apply.

The TEVT command may be entered with no operands to display the event count, delay count, and event mode in effect. When this is done prior to entering a TEVT command with one or more operands, the initial event count, delay count, and event mode are displayed, as follows:

```
? TEVT;
    1
    0
+NORM+EACH+INT
```

This indicates that the event count is one, the delay count is zero, and the internal event signal true constitutes an event at each clock. The event count is displayed on the first line and the delay count is displayed on the second line. The event mode is displayed on the third line. Table 5-2 shows the eight possible modes and the meaning of each.

The operands of a TEVT command specify a potential breakpoint after a delay specified as a number of trace memory stores following an event that satisfies the event count.

The third operand defines the conditions for an event. A TBRK command that specifies EVT or EVT+FULL must be in effect for an actual breakpoint to occur.

The following are examples of TEVT commands:

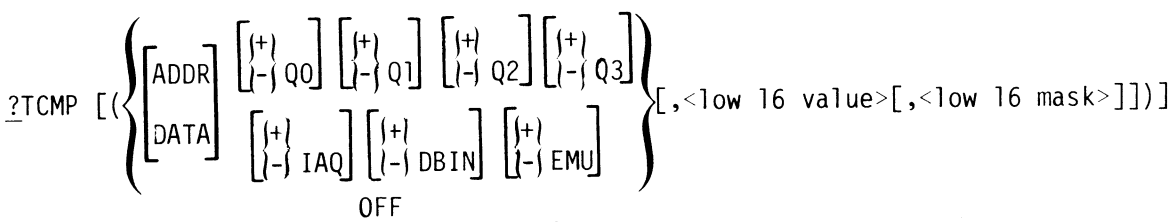
? TEVT (0,0,NORM+EACH+INT)	Count at each selected clock when the internal comparison signal is true. Event and delay completion occurs with no delay after the 65,536th event.
? TEVT (1)	Alter event count to 1, leaving previously entered delay and mode unaltered.
? TEVT (5,35)	Alter event count to 5 and delay count to 35 leaving previously entered mode unaltered.
? TEVT (>8E4F)	Alter event count to $8E4F_{16}$ (36431) leaving previously entered delay and mode unchanged.

Table 5-2. Event Modes

Event Mode	Meaning
+NORM+EACH+INT	Decrement at each selected clock during which the internal event signal is true.
+NORM+EACH+EXT	Decrement at each selected clock during which the external event signal is true.
+NORM+EDGE+INT	Decrement at selected clock when the internal event signal has changed from false to true since the preceding selected clock.
+NORM+EDGE+EXT	Decrement at selected clock when the external event signal has changed from false to true since the preceding clock.
+INV+EACH+INT	Decrement at each selected clock during which the internal event signal is false.
+INV+EACH+EXT	Decrement at each selected clock during which the external event signal is false.
+INV+EDGE+INT	Decrement at selected clock when the internal event signal has changed from true to false since the preceding selected clock.
+INV+EDGE+EXT	Decrement at selected clock when the external event signal has changed from true to false since the preceding selected clock.

Note: A selected clock is a clock signal selected for storing a value in trace memory. Specified qualifiers in specified states select a clock pulse.

5.11.4 INITIALIZE TRACE COMPARE LOGIC COMMAND. The Initialize Trace Compare Logic command (TCMP) specifies values to be placed in the comparison register and in the compare mask register. The syntax for the TCMP command is as follows:



The first operand specifies the mask and values for the high-order four bits of the compare register, and optionally, the source of data to be compared. Optionally, the keyword ADDR or the keyword DATA may be entered, specifying the type of data being supplied to the trace module in the emulator control and data mode. The remainder of the operand is a series of one to four keywords to specify the mask and values of the high-order bits. Keywords Q0 through Q3 denote the high order bits, with Q0 the most significant bit. When a plus sign precedes a keyword, the corresponding bit in the compare register is set to one. When a minus sign precedes a keyword, the corresponding bit in the compare register is set to zero. The sign may be omitted preceding the first keyword; the plus sign applies. When the keyword corresponding to a bit is omitted, that bit is masked off and does not participate in the comparison. When the keyword OFF is entered, neither data source is specified, all four bits are masked off and the four high-order bits of the compare register do not participate in the comparison.

When the target system uses a TMS 9980 microprocessor and keyword DATA is entered, the appropriate state of keyword Q0 should be specified also. When DATA+Q0 is entered, an equal comparison is possible only when an even memory address is being accessed. The data bus at this time contains the byte at the even address in the most significant eight bits and random data in the least significant eight bits. When DATA-Q0 is entered, an equal comparison is possible only when an odd memory address is being accessed. The data bus at this time contains the two bytes of a memory word, with the even-addressed byte in the most significant eight bits and the odd-addressed byte in the least significant eight bits. Thus when emulating a TMS 9980 and comparing to data, DATA-Q0 should be entered to avoid undesired equal comparisons.

The second operand is the value to be placed in the low-order 16 bits of the compare register. The operand may be any valid expression. When the second operand is omitted, the most recently entered value continues to apply.

The third operand is the value to be placed in the low-order 16 bits of the compare mask register. The operand may be any valid expression.

When the target system uses a TMS 9980 microprocessor the mask value can be used to select the valid bits. When the ADDR keyword is entered, the TMS 9980 address is in the 14 least significant bits with the two most significant bits set to zero. By using a mask operand less than 4000_{16}, the user specifies a comparison that ignores the two most significant bits and compares only valid address bits.

One bits in the mask register correspond to bits that are compared. When the third operand is omitted, the most recently entered value is the low-order 16 bits of the compare mask register continue to apply.

In the emulator control and data connection mode, the four high-order bits of the data word and the four qualifiers are common. When the target system uses a TMS 9900 microprocessor, Q0 and the high-order bit are connected to logic one. When the target system uses a TMS 9980 microprocessor, Q0 and the high-order bit are connected to the inversion of the least significant address bit. Q1, Q2, and Q3 are connected to IAQ, DBIN, and EM.EVENT, respectively. Table 5-3 lists alternate keywords that may be used in the first operand. When the trace module is connected in this manner, the first operand of the TCMP command must contain a subset of the qualifiers specified in the TTRC command. Otherwise, the comparison always fails because the comparison is made to qualified values.

Table 5-3. Alternate Keywords for Qualifiers

Keyword	Connected To	Alternate Keyword
Q0	Logic one (TMS 9900) Inversion of A13 (TMS 9980)	None
Q1	Instruction Acquisition	IAQ
Q2	Data Bus In	DBIN
Q3	Emulator Event	EMU

In the emulator control and data connection mode, the data input is supplied by the emulator. When an ETRC command with either the keyword IAQ or the keyword ADDR is in effect, the emulator supplies memory data to the trace module. When an ETRC command with the keyword OFF is in effect, the emulator may supply either memory data or memory addresses to the trace module, as specified by the TTRC command. Keywords ADDR and DATA in the TCMP command are checked for compatibility with the current ETRC and TTRC commands.

In the stand-alone or emulator control external data connection mode, the four most significant bits of the data word and the four qualifiers are separate probe leads and are connected by the user as the test requires.

The TCMP command specifies an event to provide an event and delay breakpoint as defined in a TEVT command that specifies INT in the event mode. A TBRK command that specifies either EVT or EVT+FULL must be in effect for the breakpoint to occur.

The TCMP command may be entered with no operands to display the compare register and compare mask register contents in effect. When this is done prior to entering a TCMP command with on or more operands, the initial compare register contents and compare mask register contents are displayed, as follows:

```
   ? TCMP;
OFF
>FFFF
+0000
```

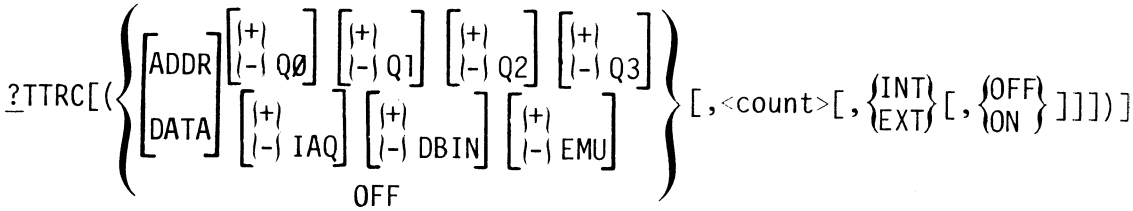
This indicates that the high-order four bits are masked off, the low-order 16 bits of the compare register contain $FFFF_{16}$, and no low-order bits are compared. The first line is either OFF or is some combination of keywords D0 through D3, in which D0 through D3 specify the high-order four bits of the compare and compare mask registers. The signs and the omission of a keyword have the same significance as in the first operand of the command. The second line shows the contents of the 16 low-order bits of the compare register. The third line shows the contents of the 16 low-order bits of the compare mask register.

The following are examples of TCMP commands:

?TCMP (QØ+Q1+Q2+Q3),>4142,>FFFF)	Set the comparison mask to compare all 20 bits, comparing to $F4142_{16}$.
? TCMP (IAQ,>2FEØ)	Set the comparison mask to compare only bit 1 and the bits not masked off by the low 16 mask operand in effect. Compare data to $2FE0_{16}$ and IAQ true. (Emulator control and data mode.)
? TCMP (OFF)	Set the comparison mask to ignore all four high order bits, and compare the low order 16 bits to the most recently entered compare value and mask value.

?TCMP (DATA-Q0+IAQ,>10FF,>FFFF)	Compare instruction data on data bus to $10FF_{16}$ (JMP $ instruction) when the entire word has been accessed and placed on the data bus (TMS 9980). If addresses are being stored in emulator trace memory simultaneously, the stored address is an odd address, one greater than the address of the instruction.

5.11.5 INITIALIZE TRACE MODULE TRACE LOGIC COMMAND. The Initialize Trace Module Trace Logic command (TTRC) specifies the qualifiers, count, and clock for a trace operation. The TTRC command also enables or disables the pulse-latching function. Optionally, the TTRC command initializes the emulator module to provide the specified input data. The syntax for the command is as follows:



The first operand specifies the mask and values for the qualifiers that select the clock for the trace operation, and optionally, the source of data to be traced. When the keyword ADDR is entered as the optional word, the memory address bus of the target system is selected for tracing. When the keyword DATA is entered, the memory data bus is selected. The remainder of the operand is a series of one to four keywords to specify the mask and values of the qualifiers. Keywords Q0 through Q3 denote the qualifiers, with Q0 the most significant qualifier bit. When a plus sign precedes the keyword, the corresponding bit in the qualifier register is set to one. When a minus sign precedes the keyword, the corresponding bit in the qualifier register is set to zero. The sign may be omitted preceding the first keyword; the plus sign applies. When the keyword corresponding to a bit is omitted, that bit is masked off and is not used as a qualifier. When the keyword corresponding to a bit is omitted, that bit is masked off and is not used as a qualifier. When the keyword OFF is entered, neither data source is specified, all four qualifiers are masked off and tracing occurs on every clock.

The second operand specifies the number of values to be stored during the trace operation. The operand may be any valid expression having a value in the range of 1 through 256. The second operand is ignored unless the first operand of the TBRK in effect is either FULL or EVT+FULL and the second operand of that command is SELF, EMU, or EMU + FULL. When the second operand is omitted, the most recently entered value continues to apply.

The third operand selects the clock source for the trace operation. When the keyword INT is entered, the internal clock is the clock for the trace operation. When the keyword EXT is entered, the external clock is the clock for the trace operation. When neither keyword is entered, the most recent clock selection continues to apply.

The fourth operand controls the latch mode of the trace module. When the keyword OFF is entered, the latch mode is disabled. When the ON is entered, the latch mode is enabled. When neither keyword is entered, the most recent latch mode keyword continues to apply.

The choice of DATA or ADDR in the first operand applies only in the emulator control and data mode. When either keyword is entered, AMPL software verifies that the data cable is connected to the emulator. If an ETRC command with the keyword OFF is in effect, the software issues the necessary control signal to the emulator to provide the specified data. Otherwise, the software verifies that the specified data is being supplied by the emulator. When the cable is not connected, or when the other type of data has been selected by an ETRC command, the software prints a warning message.

The qualifiers specified in the first operand must all be in the specified state at clock time in order to select a clock for storing a traced value. When the trace module is connected in the emulator control and data mode, the qualifiers are connected as shown in table 5-3, and the alternate keywords may be used in the first operand.

When the target system uses the TMS 9980 microprocessor, and the keyword DATA is specified, entering -Q0 as a qualifier results in more efficient use of the trace memory. The data stored in trace memory when qualifier -Q0 is in effect consists of words that contain the bytes in the proper order. If qualifier +Q0 is entered, the most significant byte of the addressed word is stored with random data in the least significant byte. If the qualifier Q0 is omitted, two words are stored for each word in memory; the first word contains the most significant byte of the addressed word with random data in the least significant byte, and the second word contains the complete word.

The count operand is used to determine the completion of the trace operation (trace full condition). The count is placed in the trace buffer counter, which is decremented as values are stored. When the counter is at one and the memory stores a value, the trace full signal occurs which causes a breakpoint if a TBRK command that specifies FULL or EVT+FULL is in effect.

The clock operand selects the clock for the trace operation. The internal clock is a 10 MHz clock in the trace module. In the stand-alone and emulator control-external data modes, external clock is the signal to which the probe lead designated CLOCK is connected. In the emulator control and data mode, the external clock is the emulator end-of-memory cycle clock.

The latch operand enables or disables the latch mode. The four most significant lines of data are equipped to latch narrow data pulses when the latch mode is enabled. They function like the other 16 lines when the latch mode is disabled. In the emulator control and data mode, the lines that are equipped with latches are connected to the same signals as the qualifiers.

The TTRC command may be entered with no operands to display the qualifiers, count, clock, and latch in effect. When this is done prior to entering a TTRC command with one or more operands, the initial qualifiers, count, clock, and latch are displayed, as follows:

```
    ? TTRC;
OFF
    256
+INT
+OFF
```

This indicates that the qualifiers are masked off, and that the trace count is 256. Internal clock is to be used, and the latch mode is disabled. The first line is either OFF or some combination of the keywords used in the first operand. The secon dline displays the count in effect. The third line is either INT or EXT, corresponding to internal or external clock, respectively. The fourth line is either ON or OFF, indicating that the latch mode is enabled or disabled.

The following are examples of TTRC commands:

? TTRC (QØ-Q1-Q2+Q3,128,INT,OFF)	Trace 128 values that occur when qualifiers 0 and 3 are true and qualifiers 1 and 2 are false, using the internal clock. Set latch mode OFF.

? TTRC (DATA-DBIN)	Set to trace memory data written during write accesses using previously entered additional operands (emulator control and data mode).
? TTRC (OFF,25,INT,ON)	Trace 25 values on internal clock, ignoring qualifiers with latch mode.
? TTRC (DATA-Q0,256,EXT,OFF)	Trace data words in trace memory, using external clock with latch inhibited. Generate the trace memory full signal when 256 words have been stored. Store when the entire memory word is on the data bus (TMS 9980).

5.11.6 START TRACE COMMAND. The Start Trace command (TRUN) enables the trace operation. The syntax for the TRUN command is as follows:

? TRUN;	Start the trace module.

In the stand-alone connection mode, the trace module starts when the TRUN command is executed. In the emulator control external data mode or emulator control and data mode, the trace module starts when a TRUN command has been executed and the emulator is started. System variable TST supplies the state of the trace module as described in a subsequent paragraph.

The following is an example of a TRUN command used to start tracing when the emulator is connected:

? TRUN; ? ERUN;	Omit in stand-alone mode.

5.11.6.1 System Variable TST. System variable TST is set to the current state of the trace module by the AMPL software. TST may not be assigned a value by the user, but may be displayed at any time. The display contains the status in the least significant three bits, as shown in table 5-4. The following is an example of a display statement to display TST:

? TST; >0001	Display trace module status. Trace is running and neither event and delay nor trace operation has completed.

Table 5-4. Trace Module Status

Contents of Least Significant Digit Of System Variable TST	Status
0	Trace module is halted. Neither event and delay nor trace operation has completed.
1	Trace module is running. Neither event and delay nor trace operation has completed.
2	Trace module is halted and trace operation has completed. Event and delay operation has not completed.
3	Trace module is running and trace operation has completed. Event and delay operation has not completed.
4	Trace module is halted and event and delay operation has completed. Trace operation has not completed.
5	Trace module is running and event and delay operation has completed. Trace operation has not completed.
6	Trace module is halted, event and delay operation has completed, and trace operation has completed.
7	Trace module is running, event and delay operation has completed, and trace operation has completed.

System variable TST may also be used with a mask value to test for a specific status condition as in the following example:

```
? IF TST AND 2 THEN 'TRACE COMPLETED'
```

Display the message if bit 14 of system variable TST is true.

A mask value of 1 tests the status bit that is true when the trace module is running. A mask value of 2 tests the status bit that is true when a trace operation has completed. A mask value of 4 tests the status bit that is true when an event and delay completion has occurred.

5.11.7 STOP TRACE COMMAND. The Stop Trace command (THLT) stops the trace module, and returns the status in system variable TST. The syntax for the THLT command is as follows:

```
?THLT;
```

When the trace module has been started and has not been stopped by the internal circuitry or by the emulator, the THLT command stops the trace module. The command displays the trace module status (system variable TST). Since the status is read after the trace module is halted, only the even numbers in table 5-4 apply. The following is an example of the THLT command:

```
? THLT;
>0002
```

Stop the trace module. Status indicates that the trace is stopped, the trace operation has completed and the event and delay operation has not completed.

5.11.8 READ LOW-ORDER TRACE MODULE MEMORY COMMAND. The Read Low-Order Trace Module Memory command (TTB) is a system function that reads the 16 low-order bits of the specified stored value. The syntax for the command is as follows:

```
?TTB(<index>)
```

The index operand may be any valid expression. When the value of the expression is a valid index to the trace memory, the command displays the value stored in the low-order 16 bits of the location corresponding to the index value. Two system variables contain the limits of valid index values. System variable TTBN contains the index value corresponding to the most recently stored value. System variable TTBO contains the index value corresponding to the oldest value stored.

When the trace module is running, it stores a value in trace memory when the qualifiers have the specified values and a clock occurs. The trace module may be stopped internally when a specified number of values have been stored or when a specified number of events and a specified delay have occurred. In these cases, or when the operation is stopped by the emulator or by command before the event and delay have occurred or the trace operation has completed, the most recently stored value correspond to an index of zero. When the event and delay operation has completed before the trace module halts, the index value of zero corresponds to the value stored by the specified event, and values stored during the delay period (and during the time required by the emulator to complete execution of the current instruction) are accessed by positive index values. Values stored prior to the value corresponding to index 0 are accessed by negative index values. Figure 5-2 shows an example of trace memory contents, assuming that the fourth value traced was the specified event, and that a delay of two was specified. The example also assumes that the completion signal stopped the trace; no additional delay occurred. The index of -3 corresponds to the first value traced and negative values denote successive values traced. The event value corresponds to index value 0. The values corresponding to indexes 1 and 2 were stored at delay counts.

In the trace module, the delay count and a further delay while the currently executing instruction completes, can cause additional values to be stored. When the total delay causes 256 or more additional values to be stored, the value associated with the specified event is overlayed by another value. The index value is also misleading. This problem can be avoided by limiting the magnitude of the value placed in the delay register by a TEVT command, and entering TBRK and EBRK commands that cause the trace module to stop.

When the trace module is connected in the emulator control and data mode, the 16 bits read by the TTB command are either memory data or memory addresses, as selected by a TTRC command prior to the operation. In the other two connection modes, the bits are signals to which the probe leads corresponding to the 16 low-order bits (D4 through D19) are connected.

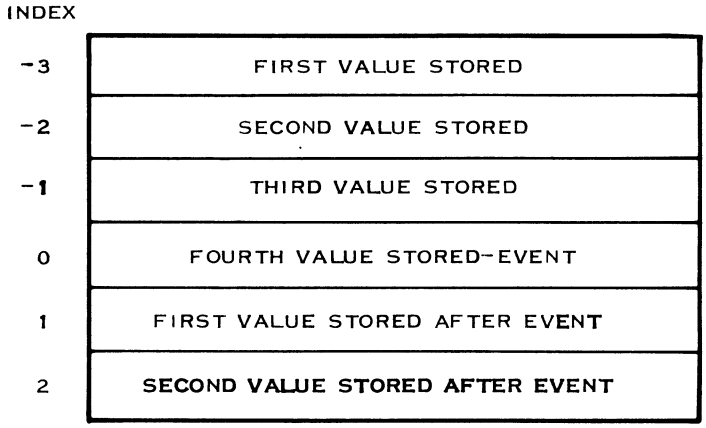


Figure 5-2. Trace Memory Contents After Trace

When the target system uses a TMS 9980 microprocessor, each memory access transfers a single byte. Trace memory stores 16 data bits and four qualifier bits, when the trace module is connected in the emulator control and data mode. Figure 5-3 shows the contents of trace memory when data is traced and qualifier Q0 is masked off (not specified in the TTRC command). The first memory access after starting the emulator is one in which Q0 is true. Subsequent memory accesses show Q0 alternately false and true. At each memory access during which Q0 is true, the most significant byte is the addressed byte and the least significant byte contains random data. At each memory access during which Q0 is false the least significant byte is the addressed byte, and the most significant byte is unaltered. The contents of trace memory when the trace is specified in this manner show the way in which the 8-bit accesses are combined in a 16-bit word, and also show that tracing with qualifier Q0 false selects only alternate memory accesses during which the word on the data bus coincides with the word as stored in target system memory.

Alternatively, a TTB command and its operand may be used as a variable in an AMPL statement. The stored value corresponding to the value of the operand becomes the value of the command when used as a variable.

The following are examples of TTB commands:

```
    ? TTB (Ø)
>78E2
```

Display the low order 16 bits of the value stored as the specified event.

```
    ? TTB (25)
>FFFF
```

Display the low order 16 bits of the value stored at the 25th delay count following the specified event.

```
    ? NUADD = TTB(Ø) + 1Ø
```

Assign a value to user symbol NUADD. The value is the sum of the value in the low order 16 bits of the value stored as the event plus 10.

INDEX	Q0	Q1	Q2	Q3	BITS 4 - 11	BITS 12 - 19
-3	1	X	X	X	BYTE 0	X X X X X X X X
-2	0	X	X	X	BYTE 0	BYTE 1
-1	1	X	X	X	BYTE 2	X X X X X X X X
0	0	X	X	X	BYTE 2	BYTE 3

NOTE: BITS REPRESENTED BY X ARE BITS THAT MAY HAVE EITHER VALUE. BYTES 0 THROUGH 3 ARE FOUR CONSECUTIVE BYTES OF TARGET MEMORY, WITH BYTE 0 CORRESPONDING TO ANY EVEN (WORD) ADDRESS.

(A)137458

Figure 5-3. Trace of Data Stored in Trace Module During Unqualified Trace of TMS 9980

5.11.8.1 System Variable TTBO. System variable TTBO is set to the index value of the oldest value stored in memory by the trace module. TTBO may not be assigned a value by the user, but may be displayed when the trace module is not running. The value of TTBO is the smallest (most negative) value that may be used as an index in a TTB or TTBH command. When this value is greater than the value of system variable TTBN, no values have been stored and a TTB or TTBH command cannot be executed. The following is an example of a display statement to display TTBO:

```
? TTBO                          Display the most negative valid index to
>FF80                           trace memory.
```

5.11.8.2 System Variable TTBN. System variable TTBN is set by AMPL software to the index value of the most recently stored value in trace memory. TTBN may not be assigned a value by the user, but may be displayed whenever the trace module is not running. The value of TTBN is the largest (most positive) value that may be used as an index in a TTB or TTBH command. When this value is less than the value of system variable TTBO, no values have been stored and a TTB or TTBH command cannot be executed. The following is an example of a display statement to display TTBN:

```
? TTBN                          Display the most positive valid index to
>005F                           trace memory.
```

5.11.9 READ HIGH-ORDER TRACE MODULE MEMORY COMMAND. The Read High-Order Trace Module Memory command (TTBH) is a system function that reads the four high-order bits of the specified stored value. The syntax for the command is as follows:

```
? TTBH(<index>)
```

The index operand may be any valid expression. When the value of the expression is a valid index to the trace memory, the command displays the value of the four high-order bits in the location corresponding to the index value. The bits are displayed right-justified in a 16-bit word, zero-filled to the left. The limits of valid index values are the same as described for the TTB command. The four high-order bits accessed are stored by the trace module along with the 16 low-order bits accessed by a TTB command that specifies the same index. The description of index values in paragraph 5.11.8 applies to the TTBH command also.

An alternate form of the command displays a value of zero (false) or one (true), the result of comparing specified bits of the four high-order bits with specified values. The syntax for this form of the TTBH command is as follows:

?TTBH(<index>,{ [(+)/(-)] Q0] [(+)/(-)] Q1] [(+)/(-)] Q2] [(+)/(-)] Q3]
[(+)/(-)] IAQ] [(+)/(-)] DBIN] [(+)/(-)] EMU] })

The index operand has the same form and limits as in the other form of the command. The second operand is a series of one to four keywords each preceded by a sign. The series denotes a value and a mask for comparison with the four high-order bits. When a keyword is preceded by a plus sign the value of the corresponding bit is one. When a keyword is preceded by a minus sign, the value for the corresponding bit is zero. The sign may be omitted preceding the first keyword; the plus sign applies. When a keyword is omitted, the bit is masked off, and is not used in the comparison. When the specified bits have the specified values, the command returns one. Otherwise, the command returns zero. The alternate keywords apply when the trace module is connected in the emulator control and data mode.

Either form of the TTBH command may be used as a variable in an AMPL statement.

The following are examples of TTBH commands:

```
    ? TTBH (Ø)
>ØØØD
```

Display the high order four bits of the value stored as the specified event.

```
    ? TTBH (Ø,Q0+Q1-Q2):B1
<1
```

Display the same value as the result of a comparison.

```
    ? IFØTTBH(Ø,EMU)ØTHEN TTB(Ø)
```

Display the low order 16 bits of the value stored as the event when the event was also an emulator event (emulator control and data mode).

SECTION VI

ERRORS AND RECOVERY

6.1 INTRODUCTION

Error messages are printed on the system console when errors are detected. This section lists the error messages and methods of recovery from errors.

6.2 ERROR MESSAGE FORMATS

AMPL software provides two categories of error messages. The warning messages are displayed when conditions which do not necessarily require termination of the command are detected. The error messages are displayed when error conditions are detected. The following is an example of a warning message:

```
*** WARNING   2Ø4 ØØØØ ØØØ1
    EMULATOR TRACING WITH TRACE/
    CONTROL MEMORY SELECTED
```

The number 204 is the message number. The leftmost hexadecimal number is the contents of workspace register zero, and the second hexadecimal number is the contents of workspace register one. The error message on the next line identifies the warning as the result of an emulator trace operation while the trace operation while the trace memory is available to the target system.

The contents of workspace registers zero and one are printed with all messages; in the case of the example message, only the most significant byte of workspace register zero is relevant. For an emulator DSR error, the DSR error code is in that byte, and it identifies the error as a device error; the emulator is unable to perform the operation required to read the target system registers (WP, PC, and ST). The user must refer to the operating system manual for explanations of DSR error codes other than the three listed in the message.

The following is an example of an error message:

```
*** ERROR      5 >9CC6 >2704
    I/O ERROR;  UNABLE TO OPEN FILE
```

The number 5 is the message number; the leftmost hexadecimal number is the contents of workspace register zero, and the second hexadecimal number is the contents of workspace register one. The error message identifies the error as an I/O error that occurs because the file cannot be opened. The only relevant information is the most significant byte of workspace register one, which contains the I/O error code (DSR error code) as listed in the operating system manual. The code in the example, 27_{16}, indicates that the file name is undefined.

Some types of errors display a symbol also, as follows:

```
*** ERROR      1Ø3 TSTP  =>AAA6 >ØØØ9
    SPECIFIED SYMBOL HAS NOT BEEN DEFINED
```

The number 103 is the message number, which is followed by the characters of a symbol, an equal sign, the contents of workspace register zero, and the contents of workspace register one. The error message on the next line identifies the error. The contents of the workspace registers are not relevant in this case.

Only eight error messages are memory resident; these are error messages for errors that occur more frequently. The messages for errors 1, 5, 6, 7, 103, 113, 117, and 224 are memory resident and are printed whenever these errors occur. Messages for the remainder of the errors are on an error message file; when this file is available messages are printed for all errors. When this file is not available only the first line (containing the error number, optionally a symbol, and the contents of workspace registers zero and one) is printed when errors other than those for which the messages are resident occur. The circumstances under which the error message file is available differ for different operating systems.

Under TX990 the error message file is on the system diskette. When the system diskette is on floppy disk unit 1 (lefthand unit) the error message file is available. If the system diskette is on another floppy disk unit, or if the error file has been deleted to provide more space on the system diskette for other files, the error message file is not available.

Under DX10, the error message file is installed under the pathname .S$SYSLIB.AMPL$ET1. When this is done, error messages are available unless the error message file has been deleted or its pathname changed, or the disk volume has been unloaded.

During initialization of AMPL software, immediately after loading the software, the error message file is opened. If the file is not available, message number 5, UNABLE TO OPEN FILE, is displayed. When errors for which error messages are not memory-resident occur, the user will have to look up the error message from table 6-1.

If the user should remove the system diskette or unload the disk volume that contains the error message file, and an error occurs (other than those for which the messages are memory-resident), the software prints message 7, I/O ERROR; READ ERROR. Any further errors are identified as if the error file had not been opened.

Table 6-1 lists the message numbers and their significance, and the possible ways of recovering from the error. It also indicates which messages display a symbol, and which register contains relevant information. Some message numbers may appear in either a warning message or an error message, depending on the severity of the condition in the context within which it is detected.

Table 6-1 .Error and Warning Messages

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
1	INTERNAL ERROR; UNDEFINED ERROR ID!	No	Not relevant	Not relevant	Invalid error code. Reload and restart AMPL program.
2	FREESPACE LOW, DELETE SOME PROCS/ FUNCS/ARRAYS	No	Not relevant	Not relevant	Working storage is full of procedures, functions, and arrays. Delete any of these that are no longer required.
3	INTERNAL ERROR; UNABLE TO BEGIN ONE-LINER!	No	Not relevant	Not relevant	Possible AMPL system error. Restart AMPL program. Symbol table and working storage are cleared by restart.
4	INTERNAL ERROR; SEGMENT VIOLATION!	No	Not relevant	Not relevant	AMPL system error. Reload and restart AMPL program.
5	I/O ERROR; UNABLE TO OPEN FILE	No	Not relevant	I/O error code in most significant byte	Typically, undefined file name. Check error code and correct input.
6	I/O ERROR; WRITE ERROR ESCAPE KEY - R1 = Ø6XX	No	Not relevant	I/O error code in most significant byte	If I/O error code is 06_{16}, the ESC key was pressed during a display. Otherwise, make another attempt to execute the command or statement.
7	I/O ERROR; READ ERROR	No	I/O error code in most significant byte	Not relevant	Make another attempt to execute the command or statement.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
8	MODULE SYMBOL TABLE TOO LARGE	No	Not relevant	Not relevant	Working storage does not have room for the load module symbols being loaded. Try the load operation again. If error occurs again, delete any procedures, functions, or arrays that are no longer required, or use OFF keyword in LOAD command. A DELE or MDEL command within a procedure or function is effective when the procedure or function completes.
9	POSSIBLE INVALID MODULE SYMBOL REFERENCES MORE THAN 15 LOADS SINCE LAST CLR	No	Not relevant	Not relevant	More than fifteen load operations with module symbols requested have been performed since a CLR command. Execution of a procedure or function that references load module symbols of a module other than the most recently loaded module may not execute correctly.
10	REQUESTED USER SYMBOL TABLE TOO LARGE	No	Not relevant	Not relevant	Insufficient memory to provide user symbol table of requested size. When this message is issued following a CLR command, reenter the command with a smaller value. When this occurs during initialization of AMPL software, the system will not execute in memory of that size.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
101	SPECIFIED SYMBOL CANNOT BE READ	Yes	Not relevant	Not relevant	User is not permitted to read the specified system symbol.
102	SPECIFIED SYMBOL CANNOT BE WRITTEN	Yes	Not relevant	Not relevant	User is not permitted to alter the value of the specified system symbol.
103	SPECIFIED SYMBOL HAS NOT BEEN DEFINED	Yes	Not relevant	Not relevant	Define the symbol, or enter a symbol that has been defined.
104	INTERNAL ERROR; CODEGEN BRANCH TABLE INDEX OUT OF RANGE	No	Not relevant	Parser index	The value assigned as the index by the parser is invalid. Reenter the command or statement. If error persists, reload AMPL program or perform maintenance on hardware.
105	NOT ENOUGH MEMORY FOR INTERNAL PROGRAM STATEMENT	No	Not relevant	Not relevant	Delete any procedures, functions, or arrays that are no longer required, and reenter command or statement.
106	SPECIFIED SYMBOL CANNOT BE REDEFINED	Yes	Not relevant	Not relevant	This message applies to the name of a procedure, function, or array. To redefine the symbol, delete it and reenter the symbol. When a symbol has been entered incorrectly, enter the correct symbol.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
107	NOT ENOUGH MEMORY TO CREATE A PROCEDURE/FUNCTION	No	Not relevant	Not relevant	Delete any procedures, functions, or arrays that are no longer required.
108	INPUT STATEMENT IS LEXICALLY INVALID	No	Not relevant	Not relevant	Check symbols, constants, format characters, operators, strings, and comments. Correct any misspelling, and supply missing blanks or other required characters.
109	INPUT TOKEN IS TOO LONG (64 CHARACTERS MAX)	No	Not relevant	Not relevant	A character string of more than 64 characters overflows the buffer. Enter a shorter character string.
110	INTERNAL ERROR: SCANNER BRANCH TABLE INDEX OUT OF RANGE	No	Not relevant	Not relevant	The value assigned as the index by the scanner is invalid. Reenter the command or statement. If error persists, reload AMPL program or perform maintenance on hardware.
111	INVALID SCAN TOKEN	No	Not relevant	Not relevant	Check that constants are entered correctly.
112	INTERNAL ERROR; UNDEFINED OPERATOR	No	Not relevant	Not relevant	Check that operators are correct. If so, error is an AMPL system error. Reload AMPL program, or perform maintenance on hardware.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
113	SYMBOL TRUNCATED TO 6 CHARACTERS	No	Not relevant	Not relevant	The symbol has been truncated to six characters, and used in processing.
114	SPECIFIED SYMBOL NOT FOUND AS A MODULE SYMBOL	No	Not relevant	Not relevant	Verify spelling of symbol and module that was loaded. Verify that module contains the symbol.
115	USER SYMBOL TABLE FULL CLEAR THE SYMBOL TABLE WITH THE CLR PROCEDURE	No	Not relevant	Not relevant	User must enter a CLR command to clear the symbol table. Required user symbols, procedures, functions, and arrays must be redefined.
116	INVALID CONSTANT, GREATER THAN 16 BITS	No	Not relevant	Not relevant	The number does not correctly convert to a 16-bit binary integer. Check that constants contain only the valid numbers for their type and are within the range for their type.
117	SYNTAX ERROR	No	Not relevant	Not relevant	Check the syntax of the command or statement, and reenter.
118	INTERNAL ERROR; BAD KEYWORD STRING LENGTH	No	Not relevant	Not relevant	This is an AMPL system error. Reload AMPL program, or perform maintenance on hardware.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
119	SYNTAX ERROR IN ONE-LINE-ASSEMBLY STATEMENT CHECK SYNTAX BETWEEN # # DELIMITERS	No	Not relevant	Not relevant	The source code statement between the pound signs contains a syntax error. Check the instruction syntax and reenter the instruction constant.
120	TOO MANY OR TOO FEW ARRAY SUBSCRIPTS	No	Not relevant	Not relevant	A reference to an array element does not have the correct number of subscripts within parentheses.
121	ESCAPE DID NOT OCCUR IN A LOOP CONSTRUCT	No	Not relevant	Not relevant	An ESCAPE statement is only valid within a WHILE, REPEAT, or FOR statement.
122	ARRAY REDEFINITION MUST USE SAME NUMBER OF INDICES AS USED IN ORIGINAL DEFINITION	No	Not relevant	Not relevant	When an array has been deleted, the array name may only be redefined with the same number of indices.
201	SYMBOL TO BE DELETED NOT FOUND IN SYMBOL TABLE	No	Number of operand in error	Not relevant	An operand in a DELE command does not appear in the user symbol table. Other symbols, if any, are deleted.
202	SPECIFIED SYMBOL CANNOT BE DELETED; ONLY PROCEDURES/ FUNCTIONS/ARRAYS MAY BE DELETED	No	Number of operand in error	Not relevant	An operand in a DELE command is not a name of a procedure, function, or array. User symbols cannot be deleted. Other symbols in the command, if any, are deleted.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
203	INVALID DISPLAY MODIFIER FOLLOWING:	No	ASCII code for format character in error in most significant byte	Not relevant	Correct the format specification and reenter statement.
204	EMULATOR TRACING WITH TRACE/ CONTROL MEMORY SELECTED	No	Not relevant	Not relevant	Possible contention between emulator and target system for trace memory. Target system must not access memory addresses from $FE00_{16}$ through $FFFF_{16}$ while tracing with ETM set to one.
205	EMULATOR DSR ERROR; 01XX = EMULATOR NOT INITIALIZED, 02XX = ILLEGAL OPERATION, 05XX = MEMORY WRITE ERROR, 06XX = OPERATION TIMED OUT, 07XX = DEVICE ERROR.	No	I/O error code in most significant byte.	Not relevant	Enter EINT command and reenter command or statement. I/O error code 02_{16} (Illegal Operation) occurs when a read or write of target memory is attempted while emulator is running. I/O error code 05_{16} (Memory Write Error) occurs when access to an invalid address is attempted. I/O error code 06_{16} (Operation Timed Out) occurs when emulator cannot be placed in Hold. Check target system clock. I/O error code 07_{16} (Device Error) occurs when the system cannot read the target system registers. This can result if target system enters the load routine when in Hold.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
206	TARGET SYSTEM MUST BE IN HOLD TO PERFORM THE SPECIFIED OPERATION	No	Not relevant	Not relevant	The command or statement may only be entered when the target system is in Hold. Enter an EHLT command and reenter the command or statement.
209	INVALID INDEX INTO EMULATOR TRACE BUFFER	No	Not relevant	Not relevant	Check index value, and reenter command with correct index.
210	INCONSISTENT USE OF TRACE DATA CABLE REQUESTED	No	TTRC command value	Not relevant	Either enter TTRC command with other value or enter ETRC command with new value and reenter TTRC command.
211	INCONSISTENT USE OF TRACE DATA CABLE REQUESTED	No	TTRC command value	Not relevant	Enter ETRC command that specifies OFF and reenter TTRC command.
212	TRACE DATA CABLE NOT CONNECTED TO EMULATOR	No	Not relevant	Not relevant	Either connect data cable to emulator and reenter TTRC command, or enter TTRC command that does not specify DATA or ADDR.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
213	CONFLICTING DATA SOURCES; CHANGE TTRC OR TCMP	No	Not relevant	Not relevant	A TCMP command specifies either ADDR or DATA but the TTRC command in effect specifies the other. Either enter a TTRC command that specifies the desired source and reenter the TCMP command or enter a TCMP command that specifies the source in effect.
214	INCONSISTENT QUALIFICATION IN TRACING AND COMPARISON	No	TTRC command qualifiers in effect	TCMP command high four bits	No valid comparison will occur unless either a TTRC command with other qualifiers or a TCMP command with other high four bits is entered.
215	BAD BREAKPOINT ACTION SPECIFIED	No	Not relevant	Not relevant	Check the second operand of the TBRK command and reenter command with correct operand.
216	BAD TRACE MODE OR COMPARISON MODE SELECTED	No	Not relevant	Not relevant	First operand of TTRC, TCMP, ETRC or ECMP is invalid. Check command and reenter.
217	BAD EMULATOR TRACE OR COMPARISON MODE SELECTED	No	Not relevant	Not relevant	ETRC or ECMP command specifies mode that emulator cannot perform. Enter command with correct first operand.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
218	TRACE MODULE NOT INITIALIZED	No	Not relevant	Not relevant	Enter TINT command specifying the device name of a trace module entered when the operating system was generated. Reenter command of statement.
219	TRACE MODULE DSR ERROR; >02XX = ILLEGAL WHILE TRACING, >06XX = OPERATION ABORTED, >07XX = TRACE MODULE DEVICE ERROR	No	Operation code in most significant byte	Error code in most significant byte	If error occurs when a TRUN command is entered, enter a THLT command and reenter TRUN command. Error codes are listed in message. Operation codes are: 05 Read status 06 Start tracing 07 Stop tracing 08 Read trace memory The device error indicates that the trace module will not halt. User must reload and restart system or perform maintenance on the trace module.
220	TRACE DATA CABLE MISSING	No	Not relevant	Not relevant	Connect trace data cable and reenter command.
221	ILLEGAL WHILE TRACING	No	Not relevant	Not relevant	Enter a THLT command and reenter command.
222	BAD INDEX INTO TRACE BUFFER	No	Index value	Not relevant	Enter TTB or TTBH command with an index operand not less than the value of TTBO or greater than the value of TTBN.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
223	TOO FEW ARGUMENTS IN PROCEDURE/ FUNCTION CALL	Yes	Not relevant	Number of arguments required	Enter statement again with proper number of arguments in the procedure or function call.
224	STACK OVERFLOW!	No	Not relevant	Not relevant	Working storage is probably full. Delete any procedures, functions, or arrays that are no longer required, and re-enter the command or function.
225	EVALUATION OF UNDEFINED SYMBOL ATTEMPTED; PROCEDURE/FUNCTION SHOULD BE DELETED.	Yes	Not relevant	Not relevant	Either redefine the symbol or delete the procedure or function and redefine it using valid symbols. Reenter statement.
227	NO DEVICE/FILE OPEN FOR LIST/CNSL	No	Not relevant	Not relevant	Enter the LIST or CNSL command with the device or file name operand. The ON operand for these commands is only valid to restore functions terminated with an OFF operand.
228	BAD EXPRESSION OR COMMAND TO DISPLAY/MODIFY	No	Not relevant	Not relevant	Check statement for syntax, undefined symbol, or symbol whose value cannot be determined. Reenter statement with correct expression.
229	INVALID CHARACTER IN OBJECT FILE BEING LOADED	No	Not relevant	Not relevant	Reenter command. If error persists reassemble or relink to obtain good object code.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
230	CHECKSUM ERROR OBJECT FILE BEING LOADED	No	Difference between expected and computed checksums	Not relevant	If object code has been altered without correcting checksum, ignore error. If object file is bad, reassemble or relink object file.
231	ARITHMETIC OVERFLOW	No	Not relevant	Not relevant	The result of an arithmetic operation could not be correctly respresented in 16 bits. Result may or may not be useful. Sometimes operations can be reordered to avoid the error. To allow address arithmetic (unsigned) this message is omitted for addition and subtraction.
232	INVALID MODE SELECTION FOR COMPARE OR TRACE EMULATOR FUNCTION	No	Mode operand	Not relevant	Correct mode of ECMP or ETRC command and reenter.
233	PROCEDURE/FUNCTION CALL ARGUMENT OUT OF RANGE R0 = POSITION OF ARGUMENT IN CALL R1 = MAXIMUM VALUE ALLOWABLE FOR ARGUMENT	No	Position of argument	Maximum valid value of argument	Correct argument and reenter command.
234	BAD INDEX TO ARG OR LOC FOR WRITING	No	Not relevant	Not relevant	Index to LOC or ARG must be greater than zero and not greater than value of LOC 0 or ARG 0, respectively. Reenter procedure or function with correct values. The error may result from evaluating an expression used as the index.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
235	BAD INDEX TO ARG OR LOC FOR READING	No	Not relevant	Not relevant	Index to LOC or ARG must be greater than zero and not greater than the value of LOC 0 or ARG 0, respectively. Reenter procedure or function with correct values. The error may result from evaluating an expression used as the index.
236	INVALID INPUT ENTERED FOR THE READ FUNCTION INPUT MUST BE A VALID AMPL CONSTANT	No	Not relevant	Not relevant	An invalid input character was read in a READ operation. When this message is a warning, enter a valid constant. When this message is an error message, correct the file and repeat the READ operation.
237	ARRAY ALREADY DEFINED	Yes	Not relevant	Not relevant	Enter a DELE command to delete the array and reenter the ARRAY statement.
238	BAD ARRAY DIMENSIONS–ZERO OR TOO LARGE	Yes	Not relevant	Not relevant	When an array dimension is zero, a negative value, or a value greater than 32767, or when the product of two dimensions is greater than 32767, correct the dimension and reenter the ARRAY statement.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
239	NOT ENOUGH FREESPACE FOR ARRAY	Yes	Not relevant	Not relevant	Delete any procedures, functions, or arrays that are no longer required and reenter ARRAY statement.
240	UNDECLARED ARRAY ACCESSED	No	Not relevant	Not relevant	An array accessed in the command or statement has not been declared. Enter an ARRAY statement and reenter the command or statement.
241	BAD SUBSCRIPT IN ARRAY ACCESS	No	Subscript	Declared Dimension	A value for a subscript is greater than the dimension declared in the ARRAY statement. Reenter statement with corrected subscript.
242	INTERNAL ERROR; BAD ARRAY SEGMENT	No	Not relevant	Not relevant	Delete the array; reenter the ARRAY statement; and reenter the statement. If error persists, reload the AMPL program or perform maintenance on hardware.
243	DELETED MODULE SYMBOL ACCESSED	No	Not relevant	Not relevant	A procedure or function referenced a load module symbol that was replaced when another module was loaded. Delete and redefine the procedure or function.

Table 6-1. Error and Warning Messages (Continued)

Message Number	Message	Symbol Printed?	R0 Contains	R1 Contains	Recovery
244	ALL MODULE SYMBOLS DELETED	No	Not relevant	Not relevant	A load module symbol was referenced after the load module symbol table was deleted. Reload the module or redefine the procedure or function.
245	INVALID DEVICE SPECIFIED	No	Not relevant	Not relevant	The device name operand of an EINT or TINT command is not that of an emulator or trace module, respectively. Reenter command with correct device name.
246	INVALID FILENAME IN EDIT CALL	No	Not relevant	Not relevant	The filename is not a valid TX990 filename. Reenter with correct filename. (TX990 systems only).
247	INADEQUATE MEMORY FOR RESTORE OPERATION	No	Not relevant	Not relevant	AMPL task requires more memory. For DX10, reassign synonym $AMPL$E to pathname of the SAVE file, and restart the AMPL task with more memory.
248	INVALID ADDRESS FOR HOST MEMORY RESTORE	No	Not relevant	Not relevant	The host memory cannot be restored from the specified file. The address is not valid. A file can properly restore only in the same system.

SECTION VII

EXAMPLES

7.1 INTRODUCTION

This section contains examples of the use of AMPL statements and commands to debug a program. The examples all relate to an example program, showing how commands may be used to load the program, to determine that the instructions in the program are executed in the proper sequence, and that the results are correct. The examples and applications of an AMPL system to the debugging of a program are not intended to be exhaustive, but to be typical of what the user can do.

7.2 EXAMPLE PROGRAM

The example program consists of a driver and a sine-cosine subroutine. The driver includes a set of values of angles, buffers for results, and a loop of instructions to call the subroutine to compute the sine and cosine for each angle, and store the results. The subroutine consists of a set of instructions to initialize the registers for the computation, and a loop of instructions to perform the computation. After executing the loop 12 times, the subroutine returns control to the driver program. The subroutine uses a table of 12 arctangents in the computations.

Assemble the source code either with TXMIRA under TXDS or with SDSMAC under DX10, following the operating instructions in the appropriate system documentation. Figure 7-1 shows the source listing printed when the program was assembled by TXMIRA.

1. Perform steps 1 through 5 of the procedure in paragraph 5.2.2 to load TXDS and to start execution of the operating system.

2. Enter an exclamation point (!). TXDS should request a program name. Request TXMIRA as follows:

```
PROGRAM: :TXMIRA/SYS
```

3. Define the input (source file) as follows, using the file name assigned when the file was written:

```
INPUT: :SINTST/TEM
```

4. Enter the device name of the device on which the listing is to be printed, as follows:

```
OUTPUT: ,LOG
```

5. Enter the desired assembler options, as follows:

```
OPTIONS: LS
```

6. TXDS prints the following messages as it loads and executes TXMIRA:

```
TXMIRA 936227 **
```

When TXMIRA completes the assembly, control returns to TXDS, which requests another program.

```
SINTST      TXMIRA 936227 **                                PAGE 0001

0001                *
0002                *  DRIVER FOR SINE - COSINE SUBROUTINE
0003                *
0004                       IDT  'SINTST'
0005        0000    R0     EQU  0
0006        0001    R1     EQU  1
0007        0002    R2     EQU  2
0008        0003    R3     EQU  3
0009        0004    R4     EQU  4
0010        0005    R5     EQU  5
0011        0006    R6     EQU  6
0012        0009    R9     EQU  9
0013        000A    R10    EQU  10
0014        000B    R11    EQU  11
0015 0000           WSP1   BSS  18                  REGISTERS 0 - 8
0016 0012 000E             DATA 14                  REGISTER 9 - LOOP COUNT
0017 0014 0000             DATA 0                   REGISTER 10 - LOOP COUNTER
0018 0016                  BSS  10                  REGISTERS 11 - 15
0019 0020 0100  ANGLE      DATA >100                1 DEGREE
0020 0022 0200             DATA >200                2 DEGREES
0021 0024 0400             DATA >400                4 DEGREES
0022 0026 0800             DATA >800                8 DEGREES
0023 0028 1000             DATA >1000               16 DEGREES
0024 002A 1E00             DATA >1E00               30 DEGREES
0025 002C 3C00             DATA >3C00               60 DEGREES
0026 002E           SINE   BSS  14                  SINE BUFFER
0027 003C           COS    BSS  14                  COSINE BUFFER
0028 004A 02E0  BEGIN      LWPI WSP1                LOAD WORKSPACE POINTER
     004C 0000'
0029 004E 04CA             CLR  R10                 RESET COUNTER
0030 0050 C82A  COMP       MOV  @ANGLE(R10),@INP    MOVE ANGLE TO CALLING SEQUENCE
     0052 0020'
     0054 005A'
0031 0056 06A0             BL   @SINCOS             BRANCH TO SUBROUTINE
     0058 0074'
0032 005A 0000  INP        DATA 0                   ANGLE
0033 005C 0000  SIN        DATA 0                   SINE
0034 005E 0000  COSN       DATA 0                   COSINE
0035 0060 CAA0             MOV  @SIN,@SINE(R10)     MOVE SINE TO BUFFER
     0062 005C'
     0064 002E'
0036 0066 CAA0             MOV  @COSN,@COS(R10)     MOVE COSINE TO BUFFER
     0068 005E'
     006A 003C'
0037 006C 05CA             INCT R10                 INCREMENT INDEX
0038 006E 824A             C    R10,R9              FINISHED?
0039 0070 11EF             JLT  COMP                CONTINUE
0040 0072 0340             IDLE                     THAT'S ALL!

(A)136544 (1/3)
```

Figure 7-1. Example Program Listing (Sheet 1 of 3)

```
SINTST        TXMIRA 936227 **                                PAGE 0002

0042                    *
0043                    *  SINCOS SUBROUTINE
0044                    *
0045                    *  SINCOS COMPUTES THE SINE AND COSINE OF AN ANGLE
0046                    *
0047                    *  CALLING SEQUENCE:
0048                    *       BL    @SINCOS
0049                    *       DATA ANGLE                IN DEGREES * 256
0050                    *       DATA 0                    SINE RETURNED HERE
0051                    *       DATA 0                    COSINE RETURNED HERE
0052                    *
0053                    *  THE SUBROUTINE ALTERS THE CONTENTS OF R0 - R6 AND R11
0054                    *
0055                    *  THE OUTPUTS ARE SIGNED BINARY FRACTIONS
0056                    *
0057 0074 04C0  SINCOS  CLR   R0                  INITIALIZE SHIFT COUNT
0058 0076 C07B          MOV   *R11+,R1            GET ANGLE
0059 0078 0501          NEG   R1                  NEGATE ANGLE
0060 007A 04C2          CLR   R2                  INITIALIZE SINE
0061 007C 0203          LI    R3,>4DBA            INITIALIZE COSINE
     007E 4DBA
0062 0080 04C4          CLR   R4                  INITIALIZE SINE ONE
0063 0082 C143          MOV   R3,R5               INITIALIZE COSINE ONE
0064 0084 04C6          CLR   R6                  INITIALIZE INDEX REGISTER
0065 0086 C041  SER     MOV   R1,R1               START SERIES, TEST SIGN
0066 0088 1105          JLT   NEG                 JUMP IF NEGATIVE
0067 008A 6085          S     R5,R2               SUBTRACT COSINE ONE FROM SINE
0068 008C A0C4          A     R4,R3               ADD SINE ONE TO COSINE
0069 008E 6066          S     @ATAN(R6),R1        SUBTRACT ARCTANGENT FROM ANGLE
     0090 00B4
0070 0092 1004          JMP   COM                 BRANCH TO COMMON CODE
0071 0094 A085  NEG     A     R5,R2               ADD COSINE ONE TO SINE
0072 0096 60C4          S     R4,R3               SUBTRACT SINE ONE FROM COSINE
0073 0098 A066          A     @ATAN(R6),R1        ADD ARCTANGENT TO ANGLE
     009A 00B4
0074 009C 0580  COM     INC   R0                  INCREMENT SHIFT COUNT
0075 009E 05C6          INCT  R6                  INCREMENT INDEX REGISTER
0076 00A0 C102          MOV   R2,R4               MOVE SINE TO SINE ONE
0077 00A2 0804          SRA   R4,R0               DIVIDE SINE ONE BY 2**R0
0078 00A4 C143          MOV   R3,R5               MOVE COSINE TO COSINE ONE
0079 00A6 0805          SRA   R5,R0               DIVIDE COSINE ONE BY 2**R0
0080 00A8 0280          CI    R0,12               SHIFT COUNT LESS THAN 12?
     00AA 000C
0081 00AC 11EC          JLT   SER                 YES - CONTINUE
0082 00AE CEC2          MOV   R2,*R11+            MOVE SINE TO CALL SEQUENCE
0083 00B0 CEC3          MOV   R3,*R11+            MOVE COSINE TO CALL SEQUENCE
0084 00B2 045B          RT    RETURN
```

Figure 7-1. Example Program Listing (Sheet 2 of 3)

```
SINTST        TXMIRA 936227 **                          PAGE 0003

0084·                 *  ARCTANGENTS
0085 00B4 2D00  ATAN    DATA >2D00
0086 00B6 1A90          DATA >1A90
0087 00B8 0E09          DATA >E09
0088 00BA 0720          DATA >720
0089 00BC 0394          DATA >394
0090 00BE 01CA          DATA >1CA
0091 00C0 00E5          DATA >E5
0092 00C2 0073          DATA >73
0093 00C4 0039          DATA >39
0094 00C6 001D          DATA >1D
0095 00C8 000E          DATA >E
0096 00CA 0007          DATA >7
0097                    END  BEGIN


R ANGLE   0020    R ATAN    00B4    R BEGIN   004A    R COM     009C
R COMP    0050    R COS     003C    R COSN    005E    R INP     005A
R NEG     0094      R0      0000      R1      0001      R10     000A
  R11     000B      R2      0002      R3      0003      R4      0004
  R5      0005      R6      0006      R9      0009    R SER     0086
R SIN     005C    R SINCOS  0074    R SINE    002E    R WSP1    0000

0000 ERRORS

(A)136544 (3/3)
```

Figure 7-1. Example Program Listing (Sheet 3 of 3)

7.3 LOADING THE EXAMPLE PROGRAM

Prior to loading the example program, the AMPL program must be loaded and started. Perform the steps in the appropriate paragraph of Section V (depending on the operating system).

Before loading the program, accessing system variables ETM or EUM, or accessing target memory, the emulator must be initialized. The first command to be entered is an EINT command, as in the following example:

```
? EINT ('EMU')                          Initialize Emulator EMU.
```

The example uses the device name EMU which applies to a TX990 system supplied by Texas Instruments. Substitute the correct device name for other operating systems; e.g., EM01 for DX10.

The AMPL program loads programs into target system memory by executing a LOAD command. Target system memory addresses 0 through $1FFF_{16}$ may be mapped either into emulator memory or target system memory. Prior to entering a LOAD command the user should display the system variables that control mapping, as follows:

```
?EUM:B1
<1
```

The display of system variable EUM shows that it has been set to one, mapping address 0 through $1FFF_{16}$ into the user memory of the emulator. It is assumed that the example program will be stored in PROM, and should be tested in emulator memory. The value of one, then, is the desired value. Display the other variable as follows:

```
      ?ETM:B1
<1
```

The display of system variable ETM shows that it has been set to one, mapping addresses $FE00_{16}$ through $FFFF_{16}$ ($3E00_{16}$ through $3FFF_{16}$ in a target system using the TMS 9980) into the trace memory of the emulator. Since the example program does not occupy this area of memory, either value of ETM would be satisfactory. If it were necessary to map memory differently, the system variables EUM and ETM would be assigned the correct values by entering assign statements. The user should verify the mapping of memory before beginning the debug session.

Next, enter the LOAD command to load the example program, as follows:

```
?LOAD('DSC:SINTST/OBJ',>1ØØ)
```

The first operand consists of the device name of the device on which the object file resides, and the file name. The example shown applies to TX990; a DX10 pathname would be used in a DX10 system. The second operand causes the AMPL program to load the example program at address 100_{16}. If the operand is omitted, the program is loaded at address $A0_{16}$. The value of 100_{16} is chosen to simplify the computation of target memory addresses from the relative addresses shown in the assembly listing. In many cases the load address may not be chosen arbitrarily, but is determined when available memory is allocated for target system requirements.

7.4 INITIAL DEBUGGING

To check that the program is loaded as intended, and to verify a constant in the subroutine enter the following command:

```
      ? @(>7E+>1ØØ):H
>4DBA
```

Referring to the assembly listing on sheet 2 of figure 7-1, notice that the fifth instruction in subroutine SINCOS is an LI instruction that places an initial value in workspace register 3. The preceding command displays the constant (at location $7E_{16}$ relative to the first location of the program) as a hexadecimal value. The result is the correct value, verifying both the loading of the program and the constant on which the computations depend.

The subroutine also uses a table of arctangent values in the computation. The following example uses a command statement to verify these values:

```
 ? BEGIN
1?   ATAN = >B4+>1ØØ;
1?   CNT = Ø;
1?   WHILEØCNTØLTØ24ØDO
1?         BEGIN
2?         @(ATAN + CNT):HXXXXX;
2?         @(ATAN + CNT + 2):HXXXXX;
2?         @(ATAN + CNT + 4):HN:
2?         CNT = CNT + 6
2?         END
1? END

>2DØØ        >1A9Ø        >ØE09
>Ø72Ø        >Ø394        >Ø1CA
>ØØE5        >ØØ73        >ØØ39
>ØØ1D        >ØØØE        >ØØØ7
```

The example uses an assign statement to assign the user symbol ATAN to the address of the table, and another assign statement to clear the user variable CNT. Then, using a WHILE statement that contains a compound statement, the examples prints the 12 constants as decimal values, three on a line. The values are correct, and verify that the data to be used in the computations is correct.

7.5 USING THE EMULATOR

In order to execute the program, an ERUN command must be entered. However, the user should set up a condition or conditions for stopping the emulation before executing the ERUN command. The EBRK command defines a breakpoint and specifies whether the breakpoint occurs when an event occurs, when a trace operation completes, or at either an event or a completion of a trace operation. The EEVT command defines an event as either the result of a comparison defined by an ECMP command, or an external event. The ETRC command defines a trace operation. These commands can be used in various combinations to stop emulation at a point that allows the user to verify whether the portion of the program being tested has executed properly.

Referring to sheet 3 of figure 7-1, the END directive of the source program defines BEGIN as the entry point. The LOAD command should have placed the address corresponding to label BEGIN in the target system program counter. If the user sets up a trace of five instructions and a program count comparison of $14E_{16}$, the emulator should execute two instructions. However, should the program contain an error that causes a transfer of control that prevents the execution of the specified instruction, the trace operation would stop emulation after execution of five instructions (TMS 9900), and examination of trace memory contents would show which instructions had been executed. The following example shows the necessary commands:

?ETM = Ø	Target memory addresses should not be mapped into trace memory while tracing is being done.
?EEVT(INT)	Event is internal.
?ECMP(IAQ,>14E)	Compare program counter to $14E_{16}$.

?ETRC(IAQ,5,INT)	Trace five program counter values (TMS 9900).
?ERUN;	Start emulation.

In a TMS 9980 system, the same commands could be used, but if the comparison did not occur before the trace completion signal stopped the emulator, six addresses would have been stored. These addresses would be the addreses of the bytes that contain three instructions. To trace five instructions as in the example, the ETRC command for a TMS 9980 system would be:

```
?ETRC(IAQ,10,INT)
```

The emulator should begin emulation at the address in the target system program counter and halt emulation and request an entry by printing a question mark. The user enters a command to determine where emulation was halted, as in the following example:

```
? DR;
R0  = >0000 R8  = >0000 PC  = >0150 / >C82A MOV @>0120(R10),@>015A
R1  = >0000 R9  = >000E WP  = >0100
R2  = >0000 R10 = >0000 ST  = >2200
R3  = >0000 R11 = >0000
R4  = >0000 R12 = >0000
R5  = >0000 R13 = >0000
R6  = >0000 R14 = >0000
R7  = >0000 R15 = >0000
```

The contents of PC is 150_{16}, indicating that the emulator was halted by the breakpoint at $14E_{16}$, and that the program has executed correctly to this point. The contents of WP (100_{16}) show that the LWPI instruction executed correctly, and the contents of R10 show that the CLR instruction executed. Contents of other registers and workspace registers may differ from those shown in the example.

7.6 TRACING WITH THE EMULATOR

The user could set up another breakpoint comparison, or another trace, at this point. However, the user could also execute another ERUN command, using the same breakpoint comparison already set up, and the same trace parameters. The comparison should not occur again, because control should not return to this point in the program unless it is restarted. Tracing five instructions should cause the emulation to halt following the third instruction of subroutine SINCOS. The user enters an ERUN command, as follows:

```
?ERUN;
```

The command starts emulation, which halts either as a result of the specified comparison or after tracing five instruction addresses. The user displays the PC to determine which breakpoint was effective, as follows:

```
?PC

PC = >017A
```

This value in the PC indicates that the emulator was halted when five instructions had been traced. The user may enter the following to determine which instructions were traced:

```
? N = ETBO                          Set N to index of oldest value traced.
? WHILEØNØLEØETBNØDOØBEGIN          Perform compound statement until N is
1?    ETB(N):HN                     equal to index of newest value traced.
1?    N = N + 1                     The compound statement displays the
1? END                              indexed value and increments the index.

>Ø15Ø
>Ø156
>Ø174
>Ø176
>Ø178
```

The results of the preceding example are the addresses of the five instructions emulated following entry of the ERUN command.

In a TMS 9980 system, using the proper ETRC command, ten addresses would be stored as follows:

```
>0150
>0151
>0156
>0157
>0174
>0175
>0176
>0177
>0178
>0179
```

The correct instructions were emulated. The user could display register contents to verify the accuracy of the computation, as follows:

```
? RØ
RØ = >ØØØØ
? R1
R1 = >FFØØ
```

The zero in R0 verifies operation of the CLR instruction at location 0174_{16}. The result in R1 verifies that the MOV instruction at location $15Ø_{16}$, the MOV instruction at location 176_{16}, and the NEG instruction at location 178_{16} operated correctly.

7.7 MONITORING PROGRAM EXECUTION

The user may then enter a series of commands to emulate the loop of instructions in the SINCOS subroutine and display register contents showing the results. The commands repeat the operation for each iteration of the instructions, allowing the user to verify the steps of the computation. First, the EBRK command should be entered as follows:

```
? ERBK (EVT)                        Define breakpoint to occur on compare
                                    only.
```

Change the comparison by entering an ECMP command as follows:

```
? ECMP (IAQ,>1A8)
```

Breakpoint at CI instruction at end of loop.

The breakpoint has been altered so that completion of the defined trace operation does not halt the emulator. Tracing may still be done, however, and tracing of addresses yields helpful information. Change the trace definition as follows:

```
? ETRC (IAQ,256)
```

Change the trace count to 256.

Enter a REPEAT statement to emulate the instructions repeatedly, as follows:

```
 ? REPEATØBEGIN
1?   ERUN;
1?   RØ
1?   R1
1?   R2
1?   R3:HN
1?   R4
1?   R5
1?   R6:HN
1? ENDØUNTILØROØEQØ12
```

Execute a compound statement repeatedly until target system workspace register Ø contains 12. The compound statement starts emulation and displays target system workspace contents when a breakpoint stops emulation.

The statement should produce 12 sets of contents of the seven registers used in the computation. The following shows the portion of the results displayed during the first iteration of the loop:

```
RØ  = >ØØØ1 R1  = >2CØØ R2  = >4DBA R3  = >4DBA
R4  = >26DD R5  = >26DD R6  = >ØØØ2
```

7.8 DISPLAYING TRACED ADDRESSES

Before analyzing the data in the registers, the user may wish to display the traced addresses. If the target system has correctly executed the program in a TMS 9900 system, the trace should consist of 14 addresses stored during the last iteration, following execution of the ERUN command. If the program has not been correct, there could be as many as 256 addresses stored. These would be the addresses of the last 256 instructions emulated. The following is an example of commands to display trace memory contents:

```
?N = ETBO
```

Set user variable N to the index of the oldest value traced.

```
 ? REPEATØBEGIN
1?   ETB(N):HN;
1?   N = N + 1
1? END UNTIL N GE Ø
```

Display stored values until the index equals zero.

>Ø1AC
>Ø186
>Ø188
>Ø18A
>Ø18C
>Ø18E
>Ø192
>Ø19C
>Ø19E
>Ø1AØ
>Ø1A2
>Ø1A4
>Ø1A6
>Ø1A8

In a TMS 9980 system, the trace consists of 28 addresses if the target system has correctly executed the program; there could otherwise be as many as 256 addresses stored. These addresses would be the addresses of both bytes of the last 128 instructions emulated. The stored addresses for a correctly executing system are:

>01AC
>01AD
>0186
>0187
>0188
>0189
>018A
>018B
>018C
>018D
>018E
>018F
>0192
>0193
>019C
>019D
>019E
>019F
>01A0
>01A1
>01A2
>01A3
>01A4
>01A5
>01A6
>01A7
>01A8
>01A9

The display of addresses in the preceding example shows that the loop of instructions was executed in the proper order. The user may analyze the workspace register contents previously displayed and determine that the computation is correct. The first instruction to be executed when an ERUN command is entered is a JLT instruction. The user may display the status register contents as follows:

```
? ST
ST = >32ØØ
```

Status register bits 2, 3, and 6 are set to one, indicating that the Equal, Carry, and XOP bits have been set. Bits 0 and 1 equal to zero and bit 2 equal to one indicate that the result of the comparison in the last instruction emulated is correct, and the MOV instruction at address $1AE_{16}$ should be executed following the JLT instruction when an ERUN command is entered.

7.9 CHECKING PROGRAM RESULTS

The next breakpoint should allow the user to verify proper exit from the subroutine and storing of the sine and cosine values computed. The user may enter another ECMP command and an ERUN command as follows:

```
? ECMP (-DBIN,>13C)
```
Compare the addresses of write operations to $13C_{16}$.

```
? ERUN;
```
Resume emulation.

The breakpoint should occur following execution of the MOV instruction at location 166_{16}. When the AMPL program requests another command, the user may verify the point at which emulation halted by displaying the PC, as follows:

```
? PC
PC = >Ø16C
```

The PC contents indicate that the instructions were executed in the proper sequence. If the emulation had halted at some other point, the traced addresses could be examined to determine what happened.

At this point the sine and cosine of 1° should have been stored in locations SINE and COS of the driver program. The values are signed binary fractions, and are not directly convertible to decimal fraction values in trigonometric tables. The values may be displayed as decimal integers, as follows:

```
? @>12E:DN
  57Ø
? @>13C:DN
 32761
```
Display contents of address $12E_{16}$ (SINE) in decimal format.

Display contents of address $13C_{16}$ (COS) in decimal format.

Divide these values by 32.768 to obtain decimal fraction equivalents. The computations of subroutine SINCOS for a single value are valid. The user may set the breakpoint to another value, emulate the subroutine for each angular value, and display the answers, as follows:

```
?ECMP (ADDR,>172)
```

Set breakpoint to compare memory addresses to 172_{16}.

```
?ERUN;
```

Start emulator.

```
?PC
PC = >Ø174
```

Verify breakpoint.

```
 ? PROCØSCDUMP(Ø,1)ØBEGIN
1?    'ANGLE    SINE       COSINE';
1?    NL;
1?    LOCØ1 = Ø;
1?    WHILEØLOCØ1ØLEØ12ØDO
1?       BEGIN
2?         @(>12Ø + LOCØ1)/256:DX;
2?         @(>12E + LOCØ1):DXXX;
2?         @(>13C + LOCØ1):DN;
2?         LOCØ1 = LOCØ1 + 2
2?       END
1?   END
```

Define procedure to display results in tabular format.

```
?SCDUMP;
```

Execute procedure.

```
ANGLE     SINE       COSINE
    1       57Ø      32761
    2     1144       32747
    4     2294       32689
    8     4450       32448
   16     9Ø11       31504
   3Ø    16388       28367
   6Ø    28376       16388
```

When the values in the SINE and COSINE columns are divided by 32,768, the results are the more familiar decimal fraction equivalents. The user may enter new values for angles and repeat the test, as follows:

```
?>12Ø:H?:H
```

Display and modify first angle.

```
>Ø12Ø / >Ø1ØØ =? = >2CØØ
>Ø12Ø / >2CØØ =?
>Ø122 / >Ø2ØØ =? = >16ØØ
>Ø122 / >16ØØ =?
>Ø124 / >Ø4ØØ =? = >BØØ
>Ø124 / >ØBØØ =?
>Ø126 / >Ø8ØØ =? = >59ØØ
>Ø126 / >59ØØ =?
>Ø128 / >1ØØØ =? = ØFØØ
>Ø128 / >ØFØØ =?
>Ø12A / >1EØØ =? = Ø7ØØ
>Ø12A / >Ø7ØØ =?
>Ø12C / >3CØØ =? = Ø3ØØ
>Ø12C / >Ø3ØØ =? ;
```

Display and modify second angle.

Display and modify third angle.

Display and modify fourth angle.

Display and modify fifth angle.

Display and modify sixth angle.

Display and modify seventh angle.

```
?PC =>14A
?ERUN;
?SCDUMP;
```

Set PC to start of program.
Emulate program with new data.
Display results.

At this point, the subroutine computations have been verified using 14 different angle values. The user may verify the subroutine more thoroughly by computing additional sines and cosines, or may accept the program as having been tested adequately.

7.10 USING THE TRACE MODULE

Had the sines and cosines not been correctly calculated, the user may want to execute subroutine SINCOS tracing all memory addresses and data. Analysis of traced addresses and the data in those addresses should identify the error in the calculations. Assuming that the trace module is connected in the emulator control and data mode, the following commands would trace the SINCOS subroutine:

?TINT('TRA')	Initialize trace module (TX990; TR01 for DX10).
?TBRK(OFF,OFF)	Turn off trace module breakpoint conditions and action. Trace module is to operate under emulator control.
?TTRC(DATA+OFF,256,EXT,OFF)	Trace 256 words of memory data, following the clock from the emulator, with the latch mode off.
?TRUN;	Trace module starts when ERUN command is executed.
?EBRK(EVT,SELF)	Define breakpoint to stop the microprocessor when an event occurs.

?EEVT(INT)	Define an event as a signal from the internal comparison circuitry.
?ECMP(IAQ,>1A8)	Compare instruction addresses to $1A8_{16}$. A breakpoint occurs when the instruction that closes the loop in subroutine SINCOS is executed.
?ETRC(ADDR,256,EXT)	Trace 256 addresses, clocked by the target system clock enabled by a signal from the trace module.
?PC = >14A	Start execution at location BEGIN of driver routine.
?ERUN;	Start the microprocessor.

The commands in the example trace all memory accesses in a TMS 9900 system, storing the memory address in the emulator trace memory, and the memory data in the trace module memory. The trace module memory contains signals IAQ, DBIN, and EMU in the high-order bits. The addresses were stored in the emulator simultaneously with the storage of data in the trace module; however, the indexes are not necessarily equal. The address stored at the index in system variable ETBO corresponds to the data stored at the index in system variable TTBO, and both addresses and data are stored sequentially following these indexes.

In a TMS 9980 system, the commands listed for the TMS 9900 system could be used, but would trace addresses and data that serve no purpose. By changing the TTRC command as follows, the trace of a TMS 9980 system is similar to that of a TMS 9900 system:

?TTRC(DATA-Q0,256,EXT,OFF)	Trace 256 words of memory data, storing the contents of the data bus only when the complete memory word is on the bus.

The ETRC command specifies EXT, which causes the emulator trace clock to be enabled by the trace module trace clock, synchronizing the trace in the emulator to that in the trace module. The addresses traced in the emulator are odd addresses, each of which is one greater than the address of the word stored in the trace module memory.

To determine the cause of erroneous results the user is mainly concerned with memory write accesses. The data stored in the trace module memory is the data written, and represents the results of the computations. The following example shows commands to print the emulator trace index, the address, and the data for each write access stored in the trace memories:

?N = ETBO	Set variable N to oldest index for emulator trace memory.
?M = TTBO	Set variable M to oldest index for trace module memory.

```
 ?WHILE N¤LE¤0¤DO¤BEGIN
1?   IF TTBH (M,-DBIN)¤THEN¤BEGIN
2?         N:DXX
2?         ETB(N):HXX
2?         TTB (M):HN
2?         END
1?   M = M+1
1?   N = N+1
1?END
```

Examining the results of executing these commands should identify the error in the calculations.

The commands in the preceding example may be used in a TMS 9980 system, and will display the addresses of the least significant bytes of the words displayed. By changing the statement on the fourth line (the one that displays the address) the user may display the same information as on a TMS 9900 system. The statement may be changed as follows:

```
2?                                   (ETB(N) - 1):HXX
```

APPENDIX A

SYSTEM GENERATION

APPENDIX A

TX990 SYSTEM GENERATION

Texas Instruments supplies three versions of the TX990 operating system with the AMPL system. One of these versions uses the 913 Video Display Terminal (VDT) as the system console, another uses the 733 ASR Electronic Data Terminal as the system console, and the third uses the 911 VDT as the system console. The user may wish to generate a TX990 operating system to support the AMPL system for various reasons. Some reasons for generating a system are as follows:

- To provide more working storage in memory by eliminating support of devices that are not required.
- To include the Operator Command Package, which provides operator interface with TX990.
- To include support of a device (or devices) not supported by a standard TX990 system.

Generating a system requires the following:

- The TX990 parts diskette (part number 937803-2601)
- AMPL system diskette (part number 937745-0007)
- One of the following TXDS system diskettes:
  - For 911 VDT System Console (part number 937803-2605)
  - For 913 VDT System Console (part number 937803-2602)
  - For 733 ASR System Console (part number 937803-2603)
- A scratch diskette on which the newly generated system is stored.

In the following system generation procedures, the characters output by the system are underlined when both output and user input are shown. Commands are preceded by asterisks (*).

The procedures include those for executing the following utility programs:

- SYSUTL — Initialize time and date.
- TXCCAT — Copy object code of TXDS control program to the TX990 parts diskette.
- INITDSC — Initialize new system diskette.
- GENTX — To generate source code for TASKDF and TXDATA modules of system.

- OBJMGR – To select and copy modules required for new system to file TXPART/OBJ for input to TXLINK.
- TXMIRA – To assemble modules TASKDF and TXDATA.
- TXLINK – To link modules TASKDF, TXDATA, and those copied to TXPART/OBJ to form new system on file NEWSYS/SYS.
- SYSUTL – To copy system loader to the new system diskette, and to designate the newly linked system as the current system.

GENTX is an interactive program that accepts system parameters and device designations and generates source code for the system data modules TASKDF and TXDATA. In the portion of the procedure that applies to GENTX, default values apply when no value (carriage return only) is entered. Refer to the *Model 990 Computer TX990 Operating System Programmer's Guide* for the default values and for complete system generation information. In the portion of the procedure that applies to defining devices, omit the definition of any devices that are not to be supported by the new system. All device names are completely arbitrary and may be changed as desired.

To generate a new TX990 system, perform the following steps:

1. Place the TX990 parts diskette in the right hand floppy disc unit, and the TXDS system diskette in the left hand unit. Place both units in ready.
2. Press the HALT/SIE switch on the programmer panel of the computer.
3. Press the RESET switch on the same panel.
4. Press the LOAD switch on the same panel.
5. The computer loads the system and prints a message similar to the following:

```
TX99Ø  SYSTEM      RELEASE 2.2
MEMORY SIZE(WORDS): 24576      AVAILABLE:  16749
```

6. Enter an exclamation point (!) at the keyboard of the system console. The computer prints the following:

```
TXDS  936215 **        1/Ø       Ø: Ø

PROGRAM:
```

7. Enter the following information to execute utility SYSUTL to initialize the time and date. Terminate each line with a carriage return. The example shows entering data corresponding to 2:15 PM April 1, 1977.

```
PROGRAM:  :SYSUTL/SYS
INPUT:
OUTPUT:
OPTIONS:  ID,1977,4,1,14,15.
```

8. SYSUTL executes the command, printing the following message, and returning control to TXDS:

```
TX99Ø SYSTEM UTILITY  937544**

14:15:ØØ APR  1, 1977
```

9. Enter the following information to execute utility TXCCAT to copy object code of the TXDS control program to the TX990 parts diskette.

```
PROGRAM:  :TXCCAT/SYS
  INPUT:  :CNTROL/OBJ
 OUTPUT:  DSC2:CNTROL/OBJ
OPTIONS:                          *NO OPTIONS
```

10. Remove the TXDS system diskette from the left hand floppy disc unit, and place the scratch diskette on which the new system is to be written in the left hand unit.

11. Enter the following information to execute utility INITDSC to initialize the new system diskette:

```
PROGRAM:  :INITDSC/SYS*

TX99Ø DISC INITIALIZATION 937545 **

DISC NAME?  DSC
DISC I.D.?  TXDS SYSTEM DISC
OK TO ERASE DISC ?? Y OR N  Y
CHECKING DSC
INITIALIZATION COMPLETE
```

12. Enter the following information to execute utility GENTX to generate source code for system modules TASKDF and TXDATA:

```
PROGRAM:  :GENTX/SYS*
```

13. GENTX requests parameters as follows. Enter the values shown. Where no value is shown, the default applies. A carriage return terminates each line. A value other than the default may be entered if required.

```
TX99Ø SYSTEM GENERATION 945673 *C
MEMORY AVAILABLE -  2ØØØ
LINE FREQ. -
TIME SLICE -
PL Ø WT. FACTOR -
PL 1 WT. FACTOR -
PL 2 WT. FACTOR -
PL 3 WT. FACTOR -
COMMON SIZE -  17Ø
# OF EXP CHASSIS -
CHASSIS - Ø
   DEV NAME -   LOG
```

14. Define the system console. When the system console is a 913 VDT, perform this step, and skip to step 17.

```
DEV TYPE - V913
STATION # - 1
CRU BASE ADDR -
ACCESS MODE -
INT LEVEL -
TIME-OUT COUNT -
```

15. When the system console is a 911 Video Display Terminal, perform this step and skip to step 17.

```
DEV TYPE - V911
STATION # - 1
CRU BASE ADDR -
ACCESS MODE -
INT LEVEL -
TIME-OUT COUNT -
```

16. When the system console is a 733 ASR Data Terminal, perform this step.

```
DEV TYPE - ASR
LEFT CASS/PTP NAME - CS1
RIGHT CASS/PTR NAME - CS2
CRU BASE ADDR -
ACCESS MODE -
INT LEVEL -
TIME-OUT COUNT -
```

17. When one or more 913 VDTs other than the system console is required, enter an appropriate device name and the information requested in step 14 for each. When one or more 911 VDTs other than the system console is required, enter an appropriate device name and the information requested in step 15 for each. When one or more 733 ASR Data Terminals other than the system console is required, enter an appropriate device name and the information requested in step 16 for each.

18. Enter the following information to define other devices. Omit any that do not apply.

```
DEV NAME - DSC          * FOR FLOPPY DISC UNIT
DEV TYPE - FD
CRU BASE ADDR -
INT LEVEL -
# OF DRIVES - 2         * ENTER NUMBER OF DRIVES (MAX 4)

DEV NAME - LP           * FOR LINE PRINTER
DEV TYPE - LP
CRU BASE ADDR - >6Ø
ACCESS MODE -
INT LEVEL -             * FOR 913 VDT SYSTEM, ENTER 4.
TIME-OUT COUNT -          OTHERWISE, USE THE DEFAULT VALUE.
```

```
DEV NAME - TTY              * FOR TELETYPEWRITER
DEV TYPE - TTY
LEFT CASS/PTP NAME - PTP
RIGHT CASS/PRT NAME - PTR
CRU BASE ADDR -
ACCESS MODE -
INT LEVEL -
TIME-OUT COUNT -
```

19. Enter the following information to define emulator and trace modules.

```
DEV NAME - EMU              * FOR EMULATOR MODULE
DEV TYPE - EMU
CRU BASE ADDR -
ACCESS MODE -
INT LEVEL -                 * FOR 913 VDT SYSTEM, ENTER 4.
TIME-OUT COUNT -              OTHERWISE, USE THE DEFAULT VALUE.

DEV NAME - TRA              * FOR TRACE MODULE
DEV TYPE - TRA
CRU BASE ADDR -
ACCESS MODE -
INT LEVEL -                 * FOR 913 VDT SYSTEM, ENTER 4.
TIME-OUT COUNT -              OTHERWISE, USE THE DEFAULT VALUE.
```

20. Enter the following carriage returns to terminate the device definition portion of GENTX:

```
    DEV NAME -              * NO MORE DEVICES
CHASSIS -                   * NO MORE CHASSIS
```

21. Enter the following information to define the standard TXDS tasks:

```
SVC # -                     * NO USER DEFINED SUPERVISOR CALLS
XOP # -                     * NO USER DEFINED EXTENDED OPERATIONS

TASK ID# - >FØ              * FOR FLOPPY DISC DRIVE 1
    PRIORITY LEVEL - Ø
    INITIAL DATA LABEL - FMP1

TASK ID# - >F1              * FOR FLOPPY DISC DRIVE 2
    PRIORITY LEVEL - Ø
    INITIAL DATA LABEL - FMP2
```

22. When more than two floppy disk drives are required, repeat the floppy disk drive sequence for each additional drive. The task ID for the third disk drive is $F2_{16}$ and the initial data label is FMP3. The task ID for the fourth disk drive is $F3_{16}$ and the initial data label is FMP4.

23. Enter the following information to define additional standard TXDS tasks:

```
TASK ID# -   >B                  *FILE MANAGEMENT
    PRIORITY LEVEL -  1
    INITIAL DATA LABEL -  FUR

TASK ID# -   >D                  * DIAGNOSTIC TASK
    PRIORITY LEVEL -  1
    INITIAL DATA LABEL -  DIAGTS

TASK ID# -   >F                  * OCP - OMIT THIS SEQUENCE WHEN OCP IS
    PRIORITY LEVEL -  1          * NOT REQUIRED
    INITIAL DATA LABEL -  OCP

TASK ID# -   >16                 * TXDS CONTROL PROGRAM
    PRIORITY LEVEL -  1
    INITIAL DATA LABEL -  CNTROL

TASK ID# -                       * NO MORE TASKS
```

24. Enter the following information to complete GENTX:

```
MULTIPLE DYNAMIC TASKS (Y OR N) - N
CONSOLE DEV NAME - LOG
DEFAULT DISC NAME - DSC
DEFAULT PRINT DEV - LOG          * NO ENTRY IF PRINT DEVICE NOT SUPPORTED
ASSIGN LUNO -                    * NO PREASSIGNED LUNOS
# OF SPARE DEV LUNO BLOCKS -
# OF SPARE FILE LUNO BLOCKS -
# OF FILE CONTROL BLOCKS -
# OF DEFAULT BUFFERS - 0
# OF GENERAL BUFFERS - 0
TASKDF OUTPUT FILE NAME - :TASKDF/SRC
                                 * GENERATE SOURCE MODULE ON FILE :TASKDF/SRC
TXDATA OUTPUT FILE NAME - :TXDATA/SRC
                                 * GENERATE SOURCE MODULE ON FILE :TXDATA/SRC
END TX99Ø SYSGEN
```

25. Enter the following information to execute OBJMGR to select and copy modules to an input file for TXLINK.

```
PROGRAM:  :OBJMGR/SYS*

99Ø OBJECT MANAGER 945672 *B

                  OUTPUT FILE:  :TXPART/OBJ
```

26. Select object modules from file :DSRLIB/OBJ. Enter C for supported devices, and S for devices that are not supported. Selections must be consistent with devices defines in steps 14 through 19. Selections shown are for a system that has a 911 VDT, a line printer, two floppy disk units, an emulator, and a trace module.

```
      INPUT FILE:   DSC2:DSRLIB/OBJ
      OPEN INPUT FILE, WITH REWIND  ?  Y
FPYDSR      ?  C                        * FLOPPY DISC DSR
DSR733      ?  C                        * 733 ASR DSR
KSRDSR      ?  S                        * 743 KSR DSR
LPDSR       ?  C                        * LINE PRINTER DSR
FLP         ?  S                        * FAST LINE PRINTER PSR
DSR93       ?  S                        * 913 VDT DSR
CRDSR       ?  S                        * CARD READER DSR
DSRTTY      ?  S                        * TELETYPEWRITER DSR
DSR911      ?  C                        * 911 VDT DSR
END-OF-FILE
```

27. Remove the TXDS parts diskette from the right-hand disk unit, and place the AMPL diskette in the right-hand unit. Select modules from file :AMPDSR/OBJ for emulator and trace module DSRs. Enter C to include a module.

```
     INPUT FILE:     DSC2:AMPDSR/OBJ
     OPEN INPUT FILE, WITH REWIND ?  Y
EMDSR     ?   C                       * INCLUDE FOR EMULATOR SUPPORT
TMDSR     ?   C                       * INCLUDE FOR TRACE MODULE SUPPORT
```

28. Remove the AMPL diskette from the right-hand disk unit, and place the TXDS parts diskette in the right-hand unit.

29. Select all modules from file :OCPLIB/OBJ when OCP is required. Otherwise, omit this step.

```
INPUT FILE:   DSC2:OCPLIB/OBJ
OPEN INPUT FILE, WITH REWIND ?  Y

OCPTSK     ?  A                     * INCLUDE ENTIRE FILE
END-OF-FILE
```

30. Select modules from file FMPLIB/OBJ for floppy disk drives. Enter C to include module. module.

```
      INPUT FILE:    DSC2:FMPLIB/OBJ
      OPEN INPUT FILE, WITH REWIND  ?  Y
TXFMP1    ?  C
TXFMP2    ?  C
TXFMP3    ?  S                  * INCLUDE FOR THIRD FLOPPY DISC DRIVE
TXFMP4    ?  S                  * INCLUDE FOR FOURTH FLOPPY DISC DRIVE
TXFMP     ?  A                  * INCLUDE REMAINDER OF FILE
END-OF-FILE
```

31. Select the module on file :CNTROL/OBJ and appropriate modules from file :TXLIB/OBJ. Enter C to include module.

```
        INPUT FILE:    DSC2:CNTROL/OBJ
        OPEN INPUT FILE, WITH REWIND ?  Y
CNTROL     ?  C
END-OF-FILE

        INPUT FILE:      DSC2:TXLIB/OBJ
        OPEN INPUT FILE, WITH REWIND ? Y

TXROOT     ?  C
EVENTK     ?  C              *REQUIRED FOR AMPL
TITTCM     ?  S
CRTPRO     ?  C              *INCLUDE TO SUPPORT EITHER VDT
STA913     ?  S              *FOR 913 VDT SUPPORT
SVC913     ?  S              *FOR 913 VDT SUPPORT
STA911     ?  C              *INCLUDE FOR 911 SUPPORT
SVC911     ?  C              *INCLUDE FOR 911 SUPPORT
IOSUPR     ?  A              *INCLUDE REMAINDER OF FILE
END-OF-FILE
```

32. Respond to next request for an input file with a carriage return, terminating OBJMGR, as follows:

```
     INPUT FILE:                        * NO MORE FILES
END OBJECT MANAGER
```

33. Remove the TX990 parts diskette from the right-hand disc unit, and place the TXDS system diskette in the right-hand unit. (The TXDS system diskette was removed from the left-hand unit in step 10).

34. Enter the following information to execute TXMIRA to assemble modules TXDATA and TASKDF.

```
PROGRAM:  :TXMIRA/SYS                 * ASSEMBLE TXDATA
  INPUT:  :TXDATA/SRC
 OUTPUT:  :TXDATA/OBJ,:TXDATA/LST

OPTIONS:  CLS

TXMIRA 936227 **

PROGRAM:  :TXMIRA/SYS                 * ASSEMBLE TASKDF
  INPUT:  :TASKDF/SRC
 OUTPUT:  :TASKDF/OBJ,:TASKDF/LST
OPTIONS:  CLS
TXMIRA 936227 **
```

35. Enter the following information to execute TXLINK to link the new system.

```
PROGRAM:   :TXLINK/SYS
  INPUT:   :TXDATA/OBJ,:TASKDF/OBJ,:TXPART/OBJ
 OUTPUT:   :NEWSYS/SYS,:NEWSYS/LST
OPTIONS:   CLITX990
TXLINK  937357 **
```

36. Remove the TXDS system diskette from the right-hand floppy disc unit, and place the TX990 parts diskette in the right-hand unit. (The TX990 parts diskette was removed from the right-hand unit in step 30.)

37. Enter the following information to execute SYSUTL to copy the system loader to the new system diskette and to designate the newly linked system as the current system.

```
PROGRAM:   :SYSUTL/SYS*

TX99Ø SYSTEMS UTILITY   937544 **

OP:  BC,DSC.SF,:NEWSYS/SYS.TE.
```

38. The newly linked system may be loaded by repeating steps 2, 3, and 4. The system should print the initial message. Actual memory size numbers vary:

```
TX99Ø  SYSTEM     RELEASE 2.2
MEMORY SIZE(WORDS): 24576     AVAILABLE:  14631
```

39. Enter an exclamation point (!). If OCP was included in the system, OCP begins execution and prints a period (.) to request input. If OCP was *not* included, TXDS begins execution and prints the message shown in step 40.

40. Enter the following to execute TXDS if OCP *was* included:

```
. EX,16.TE.
```

41. TXDS prints the following:

```
TXDS  936215 **   1/ Ø   Ø: Ø
```

Should the system fail to begin execution or to execute OCP or TXDS, carefully check the parameters supplied to GENTX and the other input to the system generation process. If an error is identified, repeat the procedure correctly.

APPENDIX B

DX10 SYSTEM GENERATION

APPENDIX B

DX10 SYSTEM GENERATION

B.1 INTRODUCTION

The DX10 operating system supplied by Texas Instruments does not support the emulator and trace modules for an AMPL system. The user must generate a version of DX10 that supports these modules as Special Devices and must install the AMPL program in the newly generated system in order to execute the AMPL program under DX10. System generation requires execution of the following system utilities:

- SDSTIE — text editor
- GEN990 — auto sysgen utility
- SDSMAC — macro assembler
- SDSLNK — link editor

The user should have a basic knowledge of the operation of these utilities. Generating a system requires the following:

- A system disk volume that contains the DX10 object modules (accessed through directory .S$SYSGEN).
- The AMPL disk volume or the AMPL parts already installed on the system disk.
- The AMPL installation package.

The AMPL installation package may reside on the system disk volume, on a separate disk volume, or on magnetic tape. When the package is on magnetic tape, perform the procedures in paragraph B.8 to copy the package to either the system disk or to a separate disk volume, then generate the system. On either medium, the installation package is defined in directory DXAMPL30.

This appendix describes the initial steps prior to system generation, execution of the text editor to prepare the PDT modules, and execution of the auto sysgen utility. The appendix also describes preparing the generated system for execution and installation of the system as the primary system. Finally, this appendix describes installation of the AMPL program in the newly created system.

B.2 INITIAL STEPS

When the installation package is on a separate disk volume, that volume must be installed and the modules required for system generation must be moved to the system disk volume. In all cases, synonyms must be assigned and the modules required must be cataloged in directory .S$SYSGEN. The following procedure assumes that DX10 has been loaded and initialized, and is executing. If not, perform steps 1 through 19 of paragraph 5.2.2 prior to the following procedure:

1. Mount the volume containing directory DXAMPL30 in a disk unit and place the unit in ready. Omit this step when the directory is already on the system disk.

2. Press the blank orange key (upper right of keyboard) on the Model 911 VDT to be used to input commands. If the terminal that was used to initialize DX10 is to be used, omit this step and skip to step 6.

3. Verify that the UPPER CASE LOCK key is in the upper case position.

4. Enter an exclamation point (!). When the system displays the initial menu of the System Command Interpreter (SCI), no log in is required and you may skip to step 6. Otherwise, the system diplays:

```
SYSTEM COMMAND INTERPRETER—PLEASE LOG IN
    USER ID:
    PASSCODE:
```

5. Enter the identifier that you have been assigned, and press the RETURN key.

6. Enter your passcode and press the RETURN key. The system displays the initial SCI menu if the identifier and passcode are valid. If not, you may have entered the identifier or passcode incorrectly, or your privilege level may be too low for the terminal. Either reenter the identifier and passcode correctly, or use a terminal with a lower privilege level, respectively. If directory .DXAMPL30 already resides on the system disk skip to step 10.

7. Enter the following command to install the disk volume:

   IV

   The system displays:

   INSTALL VOLUME

   UNIT NAME:

   VOLUME NAME:

8. Enter the device name of the disk unit in which the AMPL disk volume was mounted; e.g., DS02, DS03, etc.

9. Enter the volume name as follows:

   VOLUME NAME: DXAMPL30

10. Enter the following command to assign a synonym for the system disk volume:

    AS

    The system displays:

    ASSIGN SYNONYM VALUE

    SYNONYM:

    VALUE:

11. Enter the synonym as follows:

    SYNONYM:DSC

12. Enter the volume name of the system disk volume as the value.

13. Enter AS to assign a synonym to DXAMPL30. The display is shown in step 10.

14. Enter the synonym as follows:

    SYNONYM: AMPL

15. Enter the value as follows:

    VALUE: DXAMPL30 (If AMPL parts are on a separate disk)

    VALUE: .DXAMPL30 (If AMPL parts are on the system disk)

16. Enter the following command to execute a batch procedure:

    XB

    The system displays:

    EXECUTE BATCH

    INPUT ACCESS NAME:

    LISTING ACCESS NAME:

17. Enter the pathname of the file that contains the batch procedure, as follows:

    INPUT ACCESS NAME:AMPL.AMPL$IGP

18. Enter the Line Printer as the listing device, as follows:

    LISTING ACCESS NAME: LP01

The batch procedure on file AMPL.AMPL$IGL copies the AMPL software modules onto the system disk, cataloged in directory .S$SYSGEN. Messages output during execution of the procedure are printed on the line printer.

B.3 EXECUTING TEXT EDIT

A DX10 system requires a Peripheral Device Table (PDT) for each device supported by the system. The AMPL disk volume contains source code modules for two generalized PDTs, one for the emulator and one for the trace module. These PDTs must be made specific by placing the correct device name in certain source code statements and the interrupt-level-minus-one in another source code statement. The user executes the Text Editor to insert the correct data into the source code statements, and to copy the resulting source code into a source code module for the PDT for a specific device. The generalized emulator PDT may be edited and copied as many times as required to obtain a PDT for each emulator. Similarly, the generalized trace module PDT may be edited and copied for each trace module.

The device name of the emulator in a single-emulator laboratory is EM01, and additional emulators are EM02 . . . EM0n, where n is the total number of emulators. Similarly, when only one trace module is included, the device name is TM01. Additional trace modules are TM02 . . . TM0n, where n is the total number of trace modules. Device names EMnn and TMnn are also valid when more than nine emulators or trace modules are included.

The interrupt level for an emulator or trace module that is installed in an expansion chassis is determined by the slot in the main chassis into which the CRU expander board is connected. The same interrupt level applies to all emulators, trace modules, or other devices in all expansion chassis connected to the expander. The chassis number and position number are supplied to the auto sysgen utility to provide for identifying the specific device on the interrupt level. The interrupt level for an emulator or trace module that is installed in the main chassis is determined by the slot into which it is installed. The interrupt-level-minus-one value is the difference obtained by subtracting one from the interrupt level; e.g., 12, for interrupt level 13.

The following procedure calls the Text Editor to edit a PDT for the first or only emulator and a PDT for the first or only trace module. Perform the following steps:

1. Enter the following command:

   XE

   The system displays the following:

   INITIATE TEXT EDITOR

   FILE ACCESS NAME:

2. Enter the name of the file that contains the generalized emulator PDT, as follows:

   FILE ACCESS NAME: .S$SYSGEN.PDT$EM

3. Enter the following command to replace the string ?I with the interrupt-level-minus-one:

   RS

   The Text Editor displays the following:

   REPLACE STRING

   NUMBER OF OCCURRENCES: 1

   START COLUMN: 1

   END COLUMN: 80

   STRING:

   CHANGE:

4. Press the TAB key to enter the displayed number of occurrences.

5. Enter the number of the first column of the field in which ?I occurs:

   START COLUMN: 13

6. Enter the number of the last column of the field in which ?I occurs:

   END COLUMN: 20

7. Enter the characters of the string to be replaced:

   STRING: ?I

8. Enter the value of interrupt-level-minus-one; the following value applies to interrupt level 13:

   CHANGE: 12

9. The Text Editor executes the command and displays the line in which ?I is replaced, as follows:

   9 DATA 12 R2 = STATUS FLG/INTERRUPT-1

10. Enter the following command to terminate the edit operation and create the file specifically for EM01:

    QE

    The Text Editor displays the following:

    QUIT EDIT

    ABORT?: NO

11. Press the TAB key to enter the displayed response:

    The Text Editor displays the following:

    QUIT EDIT

    OUTPUT FILE ACCESS NAME: S$SYSGEN.PDT$EM

    REPLACE?: NO

    MOD LIST ACCESS NAME:

12. Enter the name of the PDT file for EM01 as follows:

    OUTPUT FILE ACCESS NAME: .S$SYSGEN.PDT$EM01

13. Press the TAB key to enter the displayed NO response.

14. Press the RETURN key to execute the command.

15. Enter the following command:

    XE

    The system display is shown in step 1.

16. Enter the name of the file that contains the generalized trace module PDT, as follows:

    FILE ACCESS NAME: .S$SYSGEN.PDT$TM

17. Enter the following command to replace the string ?I with the interrupt-level-minus-one:

    RS

    The Text Editor display is shown in step 3.

18. Enter the number of occurrences of ?I as follows:

    NUMBER OF OCCURRENCES: 1

19. Enter the number of the first column of the field in which ?I occurs:

    START COLUMN: 13

20. Enter the number of the last column of the field in which ?I occurs:

    END COLUMN: 20

21. Enter the characters of the string to be replaced:

    STRING: ?I

22. Enter the value of interrupt-level-minus-one; the following value applies to interrupt level 13:

    CHANGE: 12

23. The Text Editor executes the command and displays the line in which ?I is replaced, as follows:

    9 DATA 12 R2 = STATUS FLG/INTERRUPT-1

24. Enter the following command to terminate the edit operation and create the file specifically for TM01:

    QE

    The Text Editor display is shown in step 11.

25. Press the TAB key to enter the displayed output file access name.

26. Press the TAB key to enter the displayed NO response.

27. Press the RETURN key to execute the command.

The preceding procedure does not alter the device name; the device name in .S&SYSGEN.PDT$EM is EM01 and the device name in .S&SYSGEN.PDT$TM is TM01. For additional emulators and trace modules, perform the following steps:

1. Perform steps 1 and 2 of the preceding procedure.

2. Enter the following command to replace the string EM01 with the device name of an additional emulator:

   RS

   The system display is shown in step 3 of the preceding procedure.

3. Enter the number of occurrences of EM01 as follows:

   NUMBER OF OCCURRENCES: 7

4. Enter the number of the first column in which EM01 appears in any source statement:

   START COLUMN: 1

5. Enter the number of the last column in which EM01 appears in any source statement:

   END COLUMN: 25

6. Enter the characters of the string to be replaced:

   STRING: EM01

7. Enter the device name of an additional emulator. The example applies to the second emulator:

   CHANGE: EM02

8. The Text Editor executes the command and displays the last line in which EM01 is replaced, as follows:

   41 PLEM02 EQU $-PSEM02 LENGTH OF PDT

9. Enter the following command to return to the beginning of the file:

   SL

   The Text Editor displays the following:

   SHOW LINE

   LINE: 1

10. Press the TAB key to enter the displayed line number in preparation for another Replace String command.

11. Perform steps 3 through 16 of the preceding procedure.

12. Enter the following command to replace string TM01 with the device name of an additional trace module:

    RS

13. Enter the number of occurrences of TM01 as follows:

    NUMBER OF OCCURRENCES: 7

14. Enter the number of the first column in which TM01 appears in any source statement:

    START COLUMN: 1

15. Enter the number of the last column in which TM01 appears in any source statement:

    END COLUMN: 25

16. Enter the characters of the string to be replaced:

    STRING: TM01

17. Enter the device name of an additional trace module. The example applies to the second emulator:

    CHANGE: TM02

18. The Text Editor executes the command and displays the last line in which TM01 is replaced, as follows:

    41 PLTM02 EQU $-PSTM02 LENGTH OF PDT

19. Enter the following command to return to the beginning of the file:

    SL

    The Text Editor display is shown in step 9.

20. Press the TAB key to enter the displayed line number.

21. Perform steps 17 through 27 of the preceding procedure.

22. Repeat this procedure for each additional emulator and trace module, using the proper device names.

B.4 EXECUTING GEN990

GEN990, the auto sysgen utility, builds a source code file for the operating system data base and a control file for the linking operation from responses supplied by the user. GEN990 displays requests for system parameters, and accepts commands that support the input of the required data. An important capability of GEN990 is that of displaying two levels of description for most of the requested parameters.

GEN990 processes the emulator and trace module as special devices. The responses shown in the example for these devices are required for support of these devices. The other responses are typical; the devices specified are those of a recommended AMPL system. An actual DX10 system may support other devices and/or additional devices. Similarly, other parameters may require different values in a system, to meet the requirements of specific applications.

The following procedure calls and executes GEN990 to generate a DX10 system that supports an AMPL laboratory:

1. Enter the following command to activate GEN990:

   XGEN

   The system displays the following:

```
EXECUTE AUTO SYSGEN


= = = = = = = = = = = = = = =  GEN990-AUTO SYSGEN        RELEASE 3.1 * * = = = = = = = = =
```

2. Enter the indicated responses to the requests displayed. For further information about the parameters, refer to *Model 990 Computer DX10 Operating System Release 3 Reference Manual Vol. 5*. Notice that the default values are shown in parenthesis in each prompt. Specify the initial system parameters as follows:

```
DATA DISK:  (DS01)
TARGET DISK:  (DS01)
INPUT:
OUTPUT:  SYS1
LINE:  (60)
SLICE:  (50)
QUEUE  0:  (10)  255
QUEUE  1:  (10)  255
QUEUE  2:  (10)  255
QUEUE  3:  (10)  255
TABLE:  6000                                  See note
COMMON:  (NONE)
INTERRUPT DECODER:  (NONE)
FILE MANAGEMENT TASKS:  (2)
CLOCK:  (5)
ID:  (NONE)
OVERLAYS:  (2)
SYSLOG:  (6)
BUFFER MANAGEMENT:  (1K)
I/O BUFFERS:  (0)  1152                       See note
INTERTASK:  (100)
KIF?  (YES)  NO
SCI BACKGROUND:  (2)
SCI FOREGROUND:  (8)
BREAKPOINT:  (16)
CARD 1:  10
CARD 2:
```

NOTE

The values shown for the TABLE response and for the I/O BUFFERS are typical, but may not be adequate for systems with many peripherals or systems with multiple emulator and trace modules. The value for TABLE may be estimated using the information displayed when the WHAT? command is entered the second time. The value for I/O BUFFERS allows 512 bytes for an emulator module and 640 words for a trace module. This may be adequate for multiple emulators and trace modules when contention for buffer space is rare (e.g., concurrent use of more than one emulator and trace module is infrequent).

3. Enter the following responses to define the supported devices:

```
DEVICE: DS                                  Define the disk controller
TILINE: (F800)
DRIVES: (1) 2
DEFAULT RECORD SIZE: (864)
INTERRUPT: (13)
DEVICE: LP                                  Define the line printer
CRU: ()60)
ACCESS TYPE: (FILE)
TIME OUT: (30) 0
WIDTH: (80) 132
PRINT MODE: (SERIAL)
EXTENDED? (NO) YES
3270 CRU ADDRESS: (NONE)
INTERRUPT: (14)
DEVICE: ASR                                 Define the 733 ASR data terminal
CRU: ()00)
ACCESS TYPE: (RECORD)
TIME OUT: (0)
CASSETTE ACCESS TYPE: (FILE)
CASSETTE TIME OUT: (3)
CHARACTER QUEUE: (6)
INTERRUPT: (6)
DEVICE: CRT                                 Define 911 VDT
CRU: ()C0) >0500
ACCESS TYPE: (RECORD)
TIME OUT: (0)
CRT TYPE: (913) 911
3270 CRU ADDRESS: (NONE)
CHARACTER QUEUE: (6)
INTERRUPT: (11) 10
EXPANSION CHASSIS: (1)
EXPANSION POSITION: 3
```

4. Enter the following responses to define the emulator and trace modules:

```
DEVICE: SD                                        Define emulator
CRU: ()20) >0440
INTERRUPT CRU BIT: (NA)
NAME: EM01
KSB: (NONE)
DSR WORKSPACE: PDEM01
INTERRUPT ENTRY: EMINT
PDT FILE: .S$SYSGEN.PDT$EM01
DSR FILE: .S$SYSGEN.DSR$EM
OVERRIDE? (YES)
INTERRUPT: (15) 10
EXPANSION CHASSIS: (1)
EXPANSION POSITION: 1
DEVICE: SD
CRU: ()20) >0400                                  Define trace module
INTERRUPT CRU BIT: (NA)
NAME: TM01
KSB: (NONE)
DSR WORKSPACE: PDTM01
INTERRUPT ENTRY: TMINT
PDT FILE: .S$SYSGEN.PDT$TM01
DSR FILE: .S$SYSGEN.DSR$TM
OVERRIDE? (YES)
INTERRUPT: (15) 10
EXPANSION CHASSIS: (1)
EXPANSION POSITION: 2
```

5. Repeat the entries of step 4 (with appropriate CRU addresses, names, DSR workspaces, and PDT files) for additional emulators and trace modules as required.

6. Enter the following response to complete the input to GEN990:

```
DEVICE:          No more devices
SVC:             No user-supplied supervisor calls
XOP:             No user-supplied XOPs
```

7. Enter the following responses to terminate GEN990:

```
CONFIGURATION FILE IS COMPLETE.  DO YOU WANT TO SAVE IT?  (YES)
********** CONFIGURATION FILE SAVED **********
******** D$DATA SOURCE FILE IS NOW BEING BUILT **********
******** THE SDSLED COMMAND STREAM SOURCE FILE IS BEING BUILT *******
******** BATCH FILE FOR SYSGEN COMPLETION IS NOW BEING BUILT *******
```

8. GEN990 displays the following question. If the user enters Y instead of N as shown in the following example, GEN990 displays detailed instructions for assembling D$DATA and linking the system. If the terminal which called GEN990 is a 733 ASR, GEN990 prints a convenient set of instructions.

```
DO YOU NEED INSTRUCTIONS TO COMPLETE THE SYSGEN?  NO
********* GEN990 TERMINATED *********
```

B.5 COMPLETING SYSTEM GENERATION

The data base for the system must be assembled and linked with the remainder of the system. GEN990 has created a batch stream to assemble and link the system. The ALGS procedure may be called to bid this batch stream.

Prompt	Response
TARGET DISK:	(target)
SYSTEM NAME:	(output)
D$DATA LISTING:	(user defined)
BATCH LISTING:	(user defined)

where

(target) = target disk specified in sysgen
(output) = output parameter definition

When this batch stream completes without any errors, the message ***ALGS - NORMAL COMPLETION *** appears on the screen. If any other message appears, an error has occurred and the batch listing should be examined to determine the cause of the error.

After the system is linked, the patch stream must be run against the linked image. To patch the system, bid the PGS procedure.

Prompt	Response
TARGET DISK:	(target)
SYSTEM NAME:	(output)
BATCH LISTING:	(user defined)

When the batch stream is completed without errors, the message *** PGS - NORMAL TERMINATION *** appears on the screen. If any other message appears, an error has occurred and the batch listing should be examined to determine the cause of the error.

B.6 INSTALLING AND TESTING THE SYSTEM

When the batch commands have been executed, generation of the system is complete, and it is ready to be installed as the secondary system for test purposes. To set up this system to be tested at IPL time, bid the TGS procedure.

Prompt	Response
TARGET DISK:	(target)
SYSTEM NAME:	(output)

After the procedure is entered, it displays the system IPL status. The system name should be the secondary system and the IPL status should say TEST SECONDARY SYSTEM.

The system can now be booted so that the new system can be tested. Verify that the new system is acceptable. If it is not, the system can be booted and control returns to the original system.

Perform the procedure in paragraph 5.2.2 to load the newly generated system. Execute the Hardware Demonstration Test to verify system operation. If it is not possible to load the system or to perform the test properly, verify the system generation. When the system is acceptable, it may be installed as the primary system by bidding the IGS procedure.

Prompt	Response
TARGET DISK:	(target)
SYSTEM NAME:	(output)

After the procedure is entered, it displays the current IPL status. The system specified should be the primary system and the IPL status should be set to IPL the primary system. The sysgen is then complete.

B.7 INSTALLING THE AMPL PROGRAM

In order to conveniently execute the AMPL program under a DX10 system, the SCI command AMPL must be defined, the software must be installed, the error text file must be installed, and the AMPL menu must be installed. The AMPL software disk volume contains these items and a set of SCI batch commands to install them in the system. If this procedure is not being done following generation of the DX10 system that supports an AMPL laboratory, perform steps 1 through 15 of the procedure in paragraph B.2 prior to the following procedure. Then perform the following steps:

1. Check that program file .SSDS is not protected. This is the program file on which the AMPL task and procedure are to be installed.

2. Enter the following command to assign a synonym to the system program file:

   AS

   The system displays:

   ```
   ASSIGN SYNONYM VALUE
   SYNONYM:
   VALUE:
   ```

3. Enter the synonym as follows:

   SYNONYM: PROGF

4. Enter the pathname of the program file as the value:

   VALUE: . SSDS

5. Enter the following command to execute the batch commands:

   XB

   The system displays:

   ```
   EXECUTE BATCH
   INPUT ACCESS NAME:
   LISTING ACCESS NAME:
   ```

6. Enter the pathname of the file that contains the batch commands, as follows:

   INPUT ACCESS NAME: AMPL.AMPL$IAT

7. Enter the Line Printer as the listing device, as follows:

   LISTING ACCESS NAME: LP01

When the execution of the batch commands completes, the AMPL laboratory is installed in the DX10 system.

The AMPL software installed by the preceding procedure includes an SCI command, AMPL, that is executed to call the AMPL program. The command requests the user to specify the amount of memory in 1K word (2K byte) blocks, and provides a default value of 8 for 8K words (16K bytes). The user may change the default value to any number from 1 through 32 (1K through 32K words, or 2K through 64K bytes) by performing the following steps:

1. Enter the following command to activate the Text Editor:

   XE

   The system displays the following:

   INITIATE TEXT EDITOR
   FILE ACCESS NAME:

2. Enter the pathname of the AMPL SCI command file, as follows:

   FILE ACCESS NAME: .S$PROC.AMPL

3. When the Text Editor displays the command, position the cursor on the following line:

   USER MEMORY (K) = INT (8)

4. Move the cursor to the number in parentheses (8 originally) following the keyword INT, and enter the desired number. The following example shows the contents of the line for a default value of 10:

   USER MEMORY (K) = INT (10)

5. Press the CMD key and enter the following command:

   QE

   The Text Editor displays the following:

   QUIT EDIT
   ABORT?: NO

6. Press the TAB key to enter the NO response.

   The Text Editor displays the following:

```
QUIT EDIT
    OUTPUT FILE ACCESS NAME: .S$PROC.AMPL
    REPLACE?: NO
    MOD LIST ACCESS NAME:
```

7. Press the TAB key to specify the input file for output.

8. Enter YES to specify replacing the old file with the edited file:

```
REPLACE?: YES
```

9. Press the RETURN key to execute the command.

Subsequent executions of the AMPL SCI command display the newly entered default value for memory size.

After the AMPL Laboratory has been installed, and the memory size default value changed if necessary, synonyms AMPL, DSC, and PROGF may be deleted. To delete synonym AMPL, perform the following steps:

1. Enter the following command:

```
AS
```

   The system displays:

```
ASSIGN SYNONYM VALUE
SYNONYM:
VALUE:
```

2. Enter the synonym as follows:

```
SYNONYM: AMPL
```

3. Press the TAB key to enter blanks for the value, deleting the synonym.

4. Repeat the preceding steps to delete synonyms DSC and PROGF.

B.8 DX10 SYSTEM GENERATION AND AMPL PROGRAM INSTALLATION FROM MAGNETIC TAPE

When the AMPL software is supplied on magnetic tape, the software may be copied to a disk volume. The disk volume may then be used in system generation and in AMPL program installation as previously described for software supplied on a disk volume. To copy the software, which is listed in directory DXAMPL30, perform the following steps:

1. Mount the magnetic tape that contains the installation package on an available magnetic tape unit and place the unit in ready.

2. If the system disk is not to be used for the disk files to be written, install the scratch secondary disk in a disk unit.

3. Enter the following command to assign a synonym to the file to be written:

   AS

   The system displays:

   ```
   ASSIGN SYNONYM VALUE
   SYNONYM:
   VALUE:
   ```

4. Enter the synonym as follows:

   SYNONYM: AMPL

5. Enter the value, consisting of the disk volume name concatenated with the directory name DXAMPL30. The volume name used in the following example is VSYS:

   VALUE: VSYS.DXAMPL30

6. Enter the following command to copy the file from tape to disk:

   RD

   The system displays the following:

   ```
      RESTORE DIRECTORY
   SEQUENTIAL ACCESS NAME:
     DIRECTORY PATHNAME:
    LISTING ACCESS NAME:
               OPTIONS:
   ```

7. Enter the device name of the magnetic tape unit on which the tape was mounted. The example uses device name MT01, as follows:

   SEQUENTIAL ACCESS NAME: MT01

8. Enter the directory pathname as follows:

   DIRECTORY PATHNAME: AMPL

9. Press the TAB key to enter the default value for the listing access name.

10. Enter the option as follows:

    OPTIONS: ADD

11. Perform the steps of system generation and program installation using the files copied to the disk. Do not assign a new value to synonym AMPL.

The user may either retain the files on the disk or delete it. To delete the files perform the following steps:

1. Enter the following command to delete the file directory:

   DD

   The system displays the following:

   ```
   DELETE DIRECTORY:
             PATHNAME:
   LISTING ACCESS NAME:
          ARE YOU SURE?:
   ```

2. Enter AMPL as the pathname, as follows:

   PATHNAME: AMPL

3. Press the TAB key to enter the default value for the listing access name.

4. Enter YES to cause the directory and files to be deleted, as follows:

   ARE YOU SURE?: YES

If the files are to remain on the disk, perform the third procedure in paragraph B.7 to delete the synonyms.

APPENDIX C

AMPL GRAMMAR

APPENDIX C

AMPL GRAMMAR

The following is a formal definition of the AMPL grammar. An arrow in a statement means "consists of". A vertical line on a subsequent line of a statement means "or". Words in upper case letters are entered as shown. Words in lower case letters enclosed in angle brackets are defined in grammar statements. Words in lower case letters that are not enclosed in angle brackets are defined in the note that follows the grammar definition.

```
<amplprog> → <stmt>
             | PROC <procname> <parmlist> <stmt>
             | PROC <procname> <stmt>
             | FUNC <funcname> <parmlist> <stmt>
             | FUNC <funcname> <stmt>

<procname> → <procname>
             | name

<funcname> → <funcname>
             | name

<parmlist> → (constant,constant)
             | (constant)

<stmt> → BEGIN <stmtlist> END
       | IF <expr> THEN <stmt>
       | IF <expr> THEN <stmt> ELSE <stmt>
       | WHILE <expr> DO <stmt>
       | REPEAT <stmt> UNTIL <expr>
       | CASE <expr> OF <caselist> ELSE <stmt>
       | CASE <expr> OF <caselist>
       | FOR <leftpart> = <expr> TO <expr> BY <expr> DO < stmt >
       | FOR <leftpart> = <expr> TO <expr> DO < stmt >
       | <proccall>
       | <proccall> (<arglist>)
       | RETURN
       | RETURN <expr>
       | <assign>
       | <display>
       | NULL
       | <stmt>;
       | ARRAY <arraylist>
       | ESCAPE

<stmtlist> → <stmtlist >; < stmt>
             | <stmt>

<proccall> → <procname>
```

```
<arraylist> → <arraylist>, <arrayname> <exprlist>
            |<arrayname> <exprlist>

<arrayname> → <arrayname>
            | name

<caselist> → <expr>::<stmt>
           |<caselist> <expr :: < stmt >

<assign> → <left part> = <expr>

<left part> → symbol
            |@ < primary >
            | undefined
            | <arrayname>
            | <arrayname> <exprlist>

<display> → string
          | <expr>
          | <expr>: format
          | <expr>?
          | <expr>: format ?
          | <expr>: format ? :<format>
          | <range>?
          | <range>: format ?
          | <range>: format ? : format

<range> → <expr> TO <expr>

<exprlist> → (<expr>)
           |(<expr>,<expr>)

<expr> → <expr> OR <expr>
       | <expr> AND <expr>
       | NOT <expr>
       | <expr> relop <expr>
       | <expr> addop <expr>
       | <expr> multop <expr>
       | <primary>

<primary> → (<expr>)
          | symbol
          | constant
          | @ <primary>
          | addop <primary>
          | <funccall>
          | <funccall> (<arglist>)
          | <arrayname> <exprlist>

<funccall> → <funcname>

<arglist> → <arglist>,<expr>
          | <arglist>,<string>
          | <expr>
          | string
```

Note: The items in the grammar definition that are not defined in the statements of the definition are as follows:

addop – The addition or subtraction operator; i.e., + or –.

constant – A constant.

format – A format specification consisting of one or more of the format characters in table 2-2.

multop – The multiplication or division operator; i.e., *, /, or MOD.

name – A user symbol that is a procedure, function, or array name.

relop – A relational operator; i.e., EQ, GE, GT, HI, HIE, LE, LO, LOE, LT, or NE.

string – A character string.

symbol – A user symbol or variable.

undefined – A user symbol or variable not previously defined.

APPENDIX D

AMPL STATEMENT SUMMARY

APPENDIX D

AMPL STATEMENT SUMMARY

The statements of AMPL language with the syntax for each statement are as follows:

ARRAY statement:

```
ARRAY␣<array name>(<expr>[,<expr>])[,<array name>(<expr>[,<expr>])]...
```

Assign statement:

```
<symbol> = <expression>
```

CASE statement:

```
CASE␣<expr>␣OF␣<expr>::<stmt>;[<expr>::<stmt>;]...[␣ELSE␣<stmt>]␣END
```

Compound statement:

```
BEGIN␣<statement>[[;]<statement>]...␣END
```

Display statement:

```
<addr>[␣TO␣<addr>][:<format>...][?[:<format>...]]
```

ESCAPE statement:

```
ESCAPE
```

FOR statement:

```
FOR␣<symbol>=<expr>␣TO␣<expr>[␣BY␣<expr>]␣DO␣<stmt>
```

FUNC statement:

```
FUNC␣<function name>[(<parm1>[,<parm2>])]<statement>
```

IF statement:

```
IF<expression>␣THEN␣<statement>[␣ELSE␣<statement>]
```

NULL statement:

```
NULL
```

PROC statement:

```
PROC␣<procedure name>[(<parm1>[,<parm2>])]<statement>
```

REPEAT statement:

```
REPEAT␣<statement>␣UNTIL␣<expression>
```

RETURN statement:

```
RETURN[␣<expression>]
```

WHILE statement

```
WHILE␣<expression>␣DO␣<statement>
```

APPENDIX E

AMPL RESERVED WORDS

APPENDIX E

AMPL RESERVED WORDS

The following are the reserved words of the AMPL language:

AND
ARG
ARRAY
BEGIN
BY
CASE
DO
ELSE
END
EQ
ESCAPE
FOR
FUNC
GE
GT
GUBED
HI
HIE
IF
LE
LO
LOC
LOE
LT
MOD
NE
NOT
NULL
OF
OR
PROC
REPEAT
RETURN
THEN
TO
UNTIL
WHILE

APPENDIX F

SYSTEM SYMBOLS

APPENDIX F

SYSTEM SYMBOLS

The following is a list of system symbols and their definitions:

ADDR – An operand keyword used in the ECMP, ETRC, TCMP, and TTRC commands.

CC – A system variable that contains a running total of clock cycles.

CRUB – A system variable that contains the CRU base address for the CRUR and CRUW commands.

D0 - D3 – Operand keywords that name high order data bits in TCMP command.

DATA – An operand keyword used in the TCMP and TTRC commands.

DAY – A system variable that contains the day of the year. The user may only read this variable.

DBIN – An operand keyword used in the ECMP, TTRC, TCMP, and TTBH commands.

DEF – An operand keyword used in the LOAD command.

DST – A system variable that contains the destination address of an instruction being displayed in the instruction mode.

EACH – An operand keyword used in the TEVT command.

EDGE – An operand keyword used in the TEVT command.

EMT – A system variable that identifies the type of microprocessor being emulated. The user may only read this variable. Values are:

0 = TMS 9980 microprocessor
1 = TMS 9900 microprocessor

EMU – An operand keyword used in the TBRK, TTRC, TCMP, and TTBH commands.

ENI – A system variable that contains the number of emulator interrupts that have occurred since the emulator was started, when the EBRK command specifies turning breakpoints off. The user may only read this variable.

EST – A system variable that contains the current status of the emulator module. The user may only read this variable.

ETBN – A system variable that contains the index of the newest address stored in trace memory of the emulator. The user may only read this variable.

ETM – A system variable that maps trace memory into target system memory addresses $FE00_{16}$ through $FFFF_{16}$. Values are:

0 = Target system memory
1 = Trace memory

EUM – A system variable that maps user memory into target system memory addresses 0 through $1FFF_{16}$. Values are:

0 = Target system memory
1 = User memory

EVT – An operand keyword used in the EBRK and TBRK commands.

EXT – An operand keyword used in the EEVT, ETRC, TEVT, and TTRC commands.

FULL – An operand keyword used in the EBRK and TBRK commands.

HCRB – A system variable that contains the CRU base address for the host CRU commands HCRR and HCRW.

HR – A system variable that contains the hour of the day. The user may only read this variable.

IAQ – An operand keyword used in the ECMP, ETRC, TCMP, TTRC, and TTBH commands.

IDT – An operand keyword used in the LOAD command.

INT – An operand keyword used in the EEVT, ETRC, TEVT, and TTRC commands.

INV – An operand keyword used in the TEVT command.

MC – A system variable that contains a running total of memory accesses.

MDR – A system variable used with the MPY and DIV commands, and with addition and subtraction operations in expressions. Contains the most significant 16 bits of the product or the remainder. Is set to one when a carry occurs in an addition or subtraction operation, and to zero when a carry does not occur in an addition or subtraction. The user places the most significent 16 bits of the dividend in MDR for a division operation.

MIN – A system variable that contains the minute of the hour. The user may only read this variable.

NORM – An operand keyword used in the TEVT command.

OFF – An operand keyword used in the EBRK, ECMP, ETRC, TBRK, TCMP, and TTRC commands. May also be used in an assign statement to assign the value of zero to a variable.

ON – An operand keyword in the TTRC command. May also be used in an assign statement to assign the value of one to a variable.

PC – A system variable that contains the value in the target system program counter.

Q0 – Operand keyword used in emulation control and data mode to specify odd address selection for the TMS 9980 microprocessor.

Q0-Q3 – Operand keywords that name qualifier bits in the TTRC command.

R0-R15 – System variables that contain the values in the target system workspace registers.

REF – An operand keyword used in the LOAD command.

SEC – A system variable that contains the second of the hour. The user may only read this variable. variable.

SELF – An operand keyword used in the EBRK and TBRK commands.

SRC – A system variable that contains the source address of an instruction being displayed in the instruction mode.

ST – A system variable that contains the value in the target system Status Register.

TIME – A system variable used as a switch to control printing of instruction time counts when data is displayed in the instruction mode.

TNE – A system variable that contains the number of trace module events that have occurred since the module was started. The user may only read this variable.

TNI – A system variable that contains the number of trace module interrupts that have occurred since the module was started. The user may only read this variable.

TST – A system variable that contains the current status of the trace module. The user may only read this variable.

TTBN – A system variable that contains the index of the newest value stored in memory of the trace module. The user may only read this variable.

TTBO – A system variable that contains the index of the oldest value stored in memory of the trace module. The user may only read this variable.

WP – A system variable that contains the value in the target system workspace pointer.

YR – A system variable that contains the year. The user may only read this variable.

APPENDIX G

USER COMMANDS

APPENDIX G

USER COMMANDS

The commands supported by AMPL software with the syntax for each command are as follows:

```
CLR(<symbols>)

CLSE;

CNSL( {'<device name>'} )
     {     OFF      }
     {     ON       }

COPY[(<access name>)]

CRUR(<displacement>,<length>)

CRUW(<displacement>,<length>,<value>)

DELE('<name>' [,'<name>']...)

DIV(<divisor>,<dividend>)

DR;

DUMP ('<access name>',<start>,<end>[,<entry>])

EBRK[( {  EVT    } [, {SELF} ])]
      {  FULL    }     {OFF }
      {EVT+FULL  }
      {  OFF     }

ECMP[( {    ADDR     } [,<address>])]
      {[ADDR]-DBIN   }
      {[ADDR+] IAQ   }
      {    OFF       }

EDIT [ (' access name ')]

EEVT[( {INT} )]
      {EXT}

EOF;

EHLT;

EINT('<device name>')

ERUN;

ETB(<index>)
```

```
ETRC[({ ADDR          }[,<count>[,{INT}]])]
      {[ADDR+]IAQ     }          {EXT}
      { OFF           }

EXIT  [('< access name > ',  <start> , <end>[ , <entry >])]

HCRR(<displacement>,<length>)

HCRW(<displacement>,<length>,<value>)

LIST({'<access name>'})
     {     OFF      }
     {     ON       }

LOAD('<access name>'[,<bias>[,{[IDT][+REF][+DEF]}]])
                              {       OFF       }

MDEL;

MPY(<multiplicand>,<multiplier>)

MSYM;

NL;

OPEN('<access name>')

READ;

RSTR('<access name>')

SAVE('<access name>'[,<start>,<end>[<entry>]]

SSYM;

TBRK[({ EVT      }[,{ SELF     }])]
      { FULL     }  { EMU      }
      {EVT+FULL  }  {SELF+EMU  }
      { OFF      }  { OFF      }

TCMP[({[ADDR][{+}][{+}][{+}][{+}]                                    }[,<low 16 value>[,<low 16 mask>]])]
      {      {-}Q0 {-}Q1 {-}Q2 {-}Q3                                 }
      {[DATA][{+}][{+}][{+}]                                         }
      {      {-}IAQ {-}DBIN {-}EMU                                   }
      {                OFF                                           }

TEVT[(<events>[,<delays>[,[NORM]+[EACH]+[INT]]])]
                         [INV]  [EDGE] [EXT]

THLT;

TINT('<device name>')
```

TRUN;

TTB(<index>)

TTBH(<index>,[[{+/-}Q0][{+/-}Q1][{+/-}Q2][{+/-}Q3] [{+/-}IAQ][{+/-}DBIN][{+/-}EMU]])

TTRC ({[ADDR/DATA] [{+/-}Q0][{+/-}Q1][{+/-}Q2][{+/-}Q3] [{+/-}IAQ][{+/-}DBIN][{+/-}EMU] / OFF}[,<count>[,{INT/EXT}[,{OFF/ON}]]])

USYM;

VRFY('<access name>'[,<bias>])

WAIT(<number>)

APPENDIX H

AMPL PROCEDURE AND FUNCTION LIBRARY

APPENDIX H

AMPL PROCEDURE AND FUNCTION LIBRARY

AMPL software includes a library of procedures and functions described in this appendix. These procedures and functions are stored on the AMPL system diskette (part number 937745-0001) in a set of 17 files. Table H-1 lists the procedures and functions in the library, and the filename by which each is accessed. The filenames shown apply when the diskette is loaded in floppy disk unit 1.

To use the procedure and function library, the user loads the diskette into either floppy disk unit and executes COPY command for each file. Floppy disk unit 1 (left-hand unit) is normally used; the filenames for floppy disk unit 2 begin with DSC2 instead of DSC.

AMPL software executes the COPY command by reading the file, printing a message that identifies the procedure or function being defined, and storing the definition for subsequent use. When the name of the procedure has previously been defined as a procedure, the user may enter a DELE command to delete the procedure, and reenter the COPY command. Similarly, when the name of the function has previously been defined as a function, the user may enter a DELE command to delete the function, and reenter the COPY command. Any other type of name duplication (name previously defined as a variable or array, function name previously defined as a procedure, or procedure name previously defined as a function) requires that the user clear the symbol table with a CLR command, reenter the COPY commands previously entered, and redefine required variables, procedures, functions, and arrays. By entering the desired COPY commands early in the debug session, the user minimizes the problems of duplicate names.

Some files contain a single procedure or function; others contain several procedure or function definitions. When a file contains more than one definition, a separate message identifies each procedure or function. Figures H-1 through H-17 show the listings of the procedure and function library files. Each listing begins with a set of comments that describe the procedure or function and the syntax used to call it.

The comments are followed by the message that identifies the procedures or function, and the AMPL statements that define it.

After a file has been read and processed, the user may call the procedure or function in accordance with the calling information contained in the listing of the file.

The following is an example of a COPY statement (TX990) to install procedures ESTAT and TSTAT:

?COPY('DSC:STAT/PRC')	Copy file DSC:STAT/PRC and install procedures ESTAT and TSTAT.

The COPY statement to install the same procedures in a DX10 system is:

?COPY ('.S$SYSLIB.AMPL$LIB.STAT')	Copy file .S$SYSLIB.AMPL$LIB.STAT and install procedures ESTAT AND TSTAT.

The following is an example of a call to procedure ESTAT:

```
    ? ESTAT
EMULATOR IS EXECUTING AN IDLE INSTRUCTION.
EMULATOR TRACE BUFFER IS NOT FULL.
EVENT CONDITIONS ARE NOT SATISFIED.
  ØØØF BREAKPOINTS COUNTED.
```

The largest file is that of the 14 debug procedures. Because of the size of this file (and the time required for copying it into the program) the message that describes the procedures and their use is on a separate file, shown in figure H-4. In a TX990 system, execute LIST80 to obtain additional copies. In a DX10 system, execute a Print File command to obtain additional copies.

Table H-1. Procedures and Functions

Procedure or Function	Description	TX990 Filename	DX10 Pathname	Figure
AMPHDT	System test of AMPL hardware.	DSC:AMPHDT/PRC	.S$SYSLIB.AMPL$LIB.AMPHDT	H-1
AT	Debug procedure; provides a value for an emulator comparison. See Figure H-4.	DSC:DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3
BL	Simulates execution of a BL instruction in the target system.	DSC:INSTR/PRC	.S$SYSLIB.AMPL$LIB.INSTR	H-9
BLWP	Simulates execution of a BLWP instruction in the target system.	DSC:INSTR/PRC	.S$SSYLIB.AMPL$LIB.INSTR	H-9
CB	Clears a software breakpoint in the target program.	DSC:SB/PRC	.S$SYSLIB.AMPL$LIB.SB	H-10
DELAY	Debug procedure; provides a delay count for trace	DSC:DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3
DT	Debug procedure; displays traced addresses and data. See Figure H-4.	DSC:DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3
EB	Starts execution of target system with software breakpoints.	DSC:SB/PRC	.S$SYSLIB.AMPL$LIB.SB	H-10
EDUMP	Displays addresses from emulator trace memory, and contents of the addresses.	DSC:DUMPS/PRC	.S$SYSLIB.AMPL$LIB.DUMPS	H-5
ESTAT	Displays status of emulator.	DSC:STAT/PRC	.S$SYSLIB.AMPL$LIB.STAT	H-13
FNDBYT	Searches target system memory for a byte that contains	DSC:FNDBYT/PRC	.S$SYSLIB.AMPL$LIB.FNDBYT	H-6
FNDWRD	Searches target system memory for a word that contains	DSC:FNDWRD/PRC	.S$SYSLIB.AMPL$LIB.FNDWRD	H-7
GETCHR	Reads a character from a 743 KSR or 733 ASR connected to the target system.	DSC:ASRKSR/PRC	.S$SYSLIB.AMPL$LIB.ASRKSR	H-2
HALT	Debug procedure; halts the emulator and trace modules. See Figure H-4.	DSC:DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3

Table H-1. Procedures and Functions (Continued)

Procedure or Function	Description	TX990 Filename	DX10 Pathname	Figure
HELP	Debug procedure; displays the syntax of the debug commands. See Figure H-4.	DSC:DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3
HIST	Displays a histogram.	DSC:HIST/PRC	.S$SYSLIB.AMPL$LIB.HIST	H-8
INITRM	Initializes a 743 KSR or 733 ASR connected to the	DSC:ASRKSR/PRC	.S$SYSLIB.AMPL$LIB.ASRKSR	H-2
INSTR	Debug procedure; sets an instruction access comparison event in the trace module. See Figure H-4.	DSC:DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3
INTR	Simulates an interrupt in the target system.	DSC:INSTR/PRC	.S$SYSLIB.AMPL$LIB.INSTR	H-9
LOADUP	Debug procedure; selects and loads target system memory; initializes emulator and trace modules. See Figure H-4.	DSC:DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3
MREAD	Debug procedure; sets a read access comparison event in the trace module. See Figure H-4.	DSC:DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3
MWRITE	Debug procedure; sets a write access comparison event in the trace module. See Figure H-4.	DSC:DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3
PUTCHR	Writes a character to a 743 KSR or 733 ASR connected to the target system.	DSC:ASRKSR/PRC	.S$SYSLIB.AMPL$LIB.ASRKSR	H-2
RT	Simulates execution of an RT pseudo-instruction in the target system.	DSC:INSTR/PRC	.S$SYSLIB.AMPL$LIB.INSTR	H-9
RTWP	Simulates execution of an RTWP instruction in the target system.	DSC:INSTR/PRC	.S$SYSLIB.AMPL.LIB.INSTR	H-9
RUN	Debug procedure; initializes PC and WP; starts trace and emulator modules. See Figure H-4.	DSC:DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3
SB	Sets a software breakpoint in the target system.	DSC:SB/PRC	.S$SYSLIB.AMPL$LIB.SB	H-10
SETMEM	Writes a constant value into a range of target memory addresses.	DSC:SETMEM/PRC	.S$SYSLIB.AMPL$LIB.SETMEM	H-11

Table H-1. Procedures and Functions (Continued)

Procedure or Function	Description	TX990 Filename	DX10 Pathname	Figure
SIE	Executes a single instruction in the target system.	DSC:SIE/PRC	.S$SYSLIB.AMPL$LIB.SIE	H-12
STAT	Debug procedure; displays emulator and trace module status, and trace parameters in effect. See Figure H-4.	DSC:DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3
TDATA	Displays traced values from trace module memory in hexadecimal and ASCII formats.	DSC:TDUMPS/PRC	.S$SYSLIB.AMPL$LIB.TDUMPS	H-14
TDUMP	Displays interpreted traced values from trace module memory.	DSC:DUMPS/PRC	.S$SYSLIB.AMPL$LIB.DUMPS	H-5
TEDUMP	Displays traced values from emulator and trace module memory.	DSC.DUMPS/PRC	.S$SYSLIB.AMPL$LIB.DUMPS	H-5
TFOUR	Displays traced values from trace module memory in 4-bit groups.	DSC:TDUMPS/PRC	.S$SYSLIB.AMPL$LIB.TDUMPS	H-14
TILL	Debug procedure; provides the count of events and sets an event breakpoint in the trace module. See Figure H-4.	DSC.DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3
TIMER	Times the emulator and displays the elapsed time.	DSC:TIMER/PRC	.S$SYSLIB.AMPL$LIB.TIMER	H-15
TRACE	Debug procedure: initialize a track operation in the emulator and trace modules. See Figure H-4.	DSC:DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3
TRACE	Displays traced bits stored in trace module memory plotted on a time base.	DSC:TRACE/PRC	.S$SYSLIB.AMPL$LIB.TRACE	H-16
TSAVE	Store traced values from trace module memory on a file.	DSC:TRSAVR/PRC	.S$SYSLIB.AMPL$LIB.TRSAVR	H-17
TSTAT	Displays status of trace module.	DSC:STAT/PRC	.S$SYSLIB.AMPL$LIB.STAT	H-13
TVRFY	Verify traced values from trace module memory by comparing with other values in file stored by TSAVE.	DSC:TRSAVR/PRC	.S$SYSLIB.AMPL$LIB.TRSAVR	H-17
VALUE	Debug procedure; provides value and mask for comparison for trace module event.	DSC:DEBUG/PRC	.S$SYSLIB.AMPL$LIB.DEBUG	H-3
XOP	Simulates execution of an XOP instruction in the target system.	DSC:INSTR/PRC	.S$SYSLIB.AMPL$LIB.INSTR	H-9

```
..TITLE:    HARDWARE DEMONSTRATION TEST FOR AMPL HARDWARE
..REVISION: 09/16/77
..
..ABSTRACT:
..      THIS COPY FILE IS READ AS PART OF THE SYSTEM TEST OF
..AN AMPL HARDWARE FOR QUICK VERIFICATION THAT THE EMULATOR
..AND TRACE MODULES WORK CORRECTLY UNDER THE AMPL SOFTWARE.
..
..USAGE:
..      THE OPERATOR PLACES THE DISKETTE OR DISC WITH THIS FILE ON
..IT INTO A DRIVE "N" AND INVOKES AMPL.  THEN HE GIVES THE
..FOLLOWING COMMAND:
..
..          COPY('DSCN:AMPHDT/PRC')<RETURN>
..          ..WHICH COPIES THIS FILE FROM A FLOPPY DISKETTE.
..OR:
..          COPY('DS0N.S$SYSLIB.AMPL$LIB.AMPHDT')<RETURN>
..          ..WHICH COPIES THIS FILE FROM A HARD DISC.
..
..      THE FILE DEFINES THE PROCEDURE "AMPHDT" AND INVOKES IT.
..REFER TO THE TEXT MESSAGES BELOW FOR OPERATIONAL DETAILS.
..
..NOTE:  THIS PROCEDURE IS NOT A HARDWARE DIAGNOSTIC!
..
..-------------------------------------------------------------------------
..
NL;'DEFINING PROCEDURE AMPHDT - PLEASE WAIT';NL
..
..  'LOCAL' PROC DEFINITION
..
CLR(10)
C = 0;
..
FUNC MEMTST (3,1) BEGIN     ..MEMORY TEST FUNCTION
       FOR LOC 1 = ARG 1 TO ARG 2 BY ARG 3 DO @LOC 1 = LOC 1
       FOR LOC 1 = ARG 2 TO ARG 1 BY - ARG 3 DO
            IF @LOC 1 NE LOC 1 THEN RETURN -1 ..MEMORY ERROR
       RETURN 0 ..MEMORY OK !
       END
..
PROC RESET BEGIN    ..INITIALIZE CPU REGISTERS PROCEDURE
       PC = 0100
       WP = 0A0
       R0 = 0
       R1 = 1
       R2 = 2
       R3 = 3
       END
..
FUNC ETST (1,1) BEGIN
       IF PC NE 0106 OR EST NE ARG 1 THEN RETURN -1
       IF EMT EQ 0
         THEN BEGIN          ..9980 ADDRESSES TRACED
            LOC 1 = 0105
            END;
         ELSE BEGIN          ..9900 ADDRESSES TRACED
            LOC 1 = 0104
            END;          ,
       IF ETBO NE -2+(EMT-1)*3
         OR ETB(ETBO) NE 0100 OR ETB(ETBN) NE LOC 1
            THEN RETURN -1
       RETURN 0
```

Figure H-1. Listing of AMPHDT Procedure File (Sheet 1 of 6)

```
        END;
..
FUNC TTST(3) BEGIN
        IF EST NE 0 OR TST NE ARG 1 OR TTBN-TTBO+1 NE ARG 2 OR
           TTB(TTBO) NE #MOV R0,R1# OR TTB(TTBN) NE ARG 3
          THEN RETURN -1
        RETURN 0
        END;
..
.....................................................................
..
PROC AMPHDT (0,1) BEGIN
..
 IF C EQ 0 THEN BEGIN
        C:NX0'-EMULATOR INITIALIZATION TEST-';NL
        NL;'PLEASE INITIALIZE THE EMULATOR WITH THE COMMAND:'
        NL;'      EINT(''<EMULATOR NAME>'')'
        NL;'THE EMULATOR NAME WILL PROBABLY BE ''EMU'' FOR THE'
        NL;'TXDS OPERATING SYSTEM OR ''EM01'' FOR THE DX/10 3.0 OS.'
        NL;'IF THE INITIALIZATION SUCCEEDS CONTINUE THE TEST BY ENTERING:'
        NL;'      AMPHDT<RETURN>'
        C = 1
        RETURN;
        END;
..
 IF C EQ 1 THEN BEGIN
        C:NX0;'-EMULATOR MEMORY SELECTION TEST-';NL
        EUM=OFF
        ETM=OFF
        IF EUM NE 0 OR ETM NE 0 THEN BEGIN
             NL;'*** ERROR: MEMORY SELECTION TEST FAILS';
             RETURN
             END
        C = 2
        END;
..
IF C EQ 2 THEN BEGIN
        C:NX0;'-EMULATOR USER MEMORY TEST (4K WORDS)-';NL
        EUM = ON;
        IF MEMTST(0,>2000-32,32) THEN BEGIN
                  C:NX0;'*** ERROR: EMULATOR USER MEMORY ERROR'
                  RETURN
                  END;
    ..PLACE A PROGRAM IN MEMORY
        @0100 = #MOV R0,R1#;
        @0102 = #MOV R1,R2#;
        @0104 = #MOV R2,R3#;
        @0106 = #IDLE#;
        C = 3;
        END;
..
  IF C EQ 3 THEN BEGIN
        C:NX0;'-EMULATOR TRACE BUFFER MEMORY TEST (256 WORDS)-';NL
        ETM = ON;
        IF MEMTST(0FE00,0FFFE,2) THEN BEGIN
             C:NX0;'*** ERROR: EMULATOR TRACE MEMORY FAILURE'
             RETURN;
             END;
        C = 4;
        END;
..
  IF C EQ 4 THEN BEGIN
```

Figure H-1. Listing of AMPHDT Procedure File (Sheet 2 of 6)

```
      C:NX0;'-EMULATOR RUN TEST-';NL
      ETM = OFF
      ETRC(OFF)
      ECMP(OFF)
      EEVT(INT)
      EBRK(OFF,OFF)
      RESET
      ERUN
      IF EST NE 9 THEN BEGIN
           C:NX0;'*** ERROR: EMULATOR EXECUTION FAILURE'
           RETURN
           END;
      C = 5
      END;

IF C EQ 5 THEN BEGIN
      C:NX0;'-EMULATOR HALT TEST-';NL
      LOC 1 = EHLT
      IF EST NE 0 THEN BEGIN
           C:NX0;'*** ERROR: EMULATOR FAILED TO HALT PROPERLY'
           RETURN
           END;
      C = 6;
      END;

IF C EQ 6 THEN BEGIN
      C:NX0;'-EMULATOR TRACE TEST';NL
      ETRC(ADDR+IAQ,3*((NOT EMT)+3),INT);
      EBRK(FULL,SELF);
      RESET
      ERUN;
      IF ETST(2) THEN BEGIN
           LOC 1 = EHLT;
           C:NX0;'*** ERROR: EMULATOR FAILED TRACE TEST'
           RETURN;
           END;
      C = 7;
      END;

IF C EQ 7 THEN BEGIN
      C:NX0;'-EMULATOR ADDRESS COMPARISON TEST-';NL
      RESET;
      ECMP(IAQ,0104);
      EEVT(INT);
      EBRK(EVT,SELF);
      ERUN;
      IF ETST(4) THEN BEGIN
           EHLT;
           C:NX0;'*** ERROR: EMULATOR FAILED PC BREAKPOINT'
           RETURN;
           END
      NL;'*** EMULATOR PASSES HARDWARE DEMONSTRATION TEST ***';NL
      Y = 0; N = -1;
     C:NX5;'DO YOU WISH TO TEST A LOGIC STATE TRACE MODULE? (Y=YES,N=NO)'
      IF READ EQ 0
           THEN C = 8
           ELSE C = 14
      END;

IF C EQ 8 THEN BEGIN
      C:NX0;'-LOGIC STATE TRACE MODULE INITIALIZATION TEST-';NL
   NL;'TO TEST THE AMPL LOGIC STATE TRACE MODULE IT MUST BE STRAPPED TO'
```

Figure H-1. Listing of AMPHDT Procedure File (Sheet 3 of 6)

```
      NL;'THE EMULATOR JUST TESTED WITH BOTH THE CONTROL AND DATA CABLES.'
      NL;'PLEASE VERIFY THAT THESE CABLES ARE CONNECTED.'
      NL;'INITIALIZE THE LOGIC STATE TRACE MODULE WITH THE COMMAND:'
      NL;'      TINT(''<TRACE MODULE NAME>'')'
      NL;'THE TRACE MODULE NAME WILL PROBABLY BE ''TRA'' FOR THE'
      NL;'TXDS OPERATING SYSTEM OR ''TM01'' FOR THE DX/10 3.0 OS.'
      NL;'IF THE INITIALIZATION SUCCEEDS CONTINUE THE TEST BY ENTERING:'
      NL;'      AMPHDT<RETURN>'
         C = 9
         RETURN;
         END;

IF C EQ 9 THEN BEGIN
         C:NX0;'-TRACE MODULE INTERNAL CLOCK TEST-';NL
         ECMP(OFF)          . . PUT THE EMULATOR IN A NEUTRAL STATE
         EEVT(INT)
         ETRC(OFF)
         EBRK(OFF,OFF)
         TTRC(OFF,10,INT,OFF);
         TEVT(1,0,0);
         TBRK(FULL,SELF);
         TRUN;
         RESET;
         ERUN; LOC 1 = EHLT;
         IF TST NE 6 THEN BEGIN
              LOC 1 = EHLT;
              C:NX0;'*** ERROR: TRACE MODULE FAILED INTERNAL CLOCK TEST'
              RETURN
              END
         C = 10;
         END;

IF C EQ 10 THEN BEGIN
         C:NX0;'-TRACE MODULE DATA TRACE TEST-';NL
         TTRC(DATA+IAQ+(EMT-1)*Q0,3,EXT);
         TBRK(FULL,EMU)
         TRUN;
         RESET;
         EBRK(OFF,SELF)
         ERUN;
         IF TTST(6,3,#MOV R2,R3#)
           THEN BEGIN
              LOC 1 = EHLT;
              C:NX0;'*** ERROR: TRACE MODULE FAILED DATA TRACE'
              RETURN;
              END
         C = 11
         END;

IF C EQ 11 THEN BEGIN
         C:NX0;'-TRACE MODULE EVENT TEST-';NL
         TCMP(DATA+IAQ+(EMT-1)*Q0,#MOV R0,R1#,>FFFF)
         TEVT(1,0)
         TBRK(EVT,EMU)
         TRUN
         RESET
         ERUN
         IF TTST(4,1,#MOV R0,R1#)
           THEN BEGIN
              LOC 1 = EHLT
              C:NX0;'*** ERROR: TRACE MODULE EVENT FAILURE'
              RETURN
```

Figure H-1. Listing of AMPHDT Procedure File (Sheet 4 of 6)

```
            END;
        C = 12
        END

IF C EQ 12 THEN BEGIN
        C:NX0'-TRACE MODULE: TRACE DELAY TEST-';NL
        TEVT(1,2)
        RESET
        TRUN
        ERUN
        IF TTST(4,3,#MOV R2,R3#)
          THEN BEGIN
            LOC 1 = EHLT
            C:NX0;'*** ERROR: TRACE MODULE DELAY FAILURE'
            RETURN
            END
        C = 13
        END

IF C EQ 13 THEN BEGIN
        NL;'*** TRACE MODULE PASSES HARDWARE DEMONSTRATION TEST ***';NL
        C = 14
        END;

IF C EQ 14 THEN BEGIN
        NL;'*** CONCLUSION OF AMPL HARDWARE DEMONSTRATION TEST ***';NL
        C = 0
        END
 END

------------------------------------------------------------------------

 AMPHDT INTRODUCTION

BEGIN
        C = 0
        C:NX0;'*** INTRODUCTION: AMPL HARDWARE DEMONSTRATION TEST ***';NL
    NL;'      AMPHDT IS AN AMPL PROCEDURE TO VERIFY THE CORRECT OPERATION'
    NL;'      OF THE EMULATOR AND LOGIC STATE TRACE MODULES. '
        C:NX5'ONLY A GO/NO GO TEST IS PERFORMED. ';NL
        C:NX0;'*** THIS IS NOT A HARDWARE DIAGNOSTIC TEST ***';NL
        C:NX5;'TYPE C<RETURN> FOLLOWING THE "=?" TO CONTINUE'
        C = READ
    NL;'      TO TEST THE TRACE MODULE, AN EMULATOR MUST BE PRESENT AND'
    NL;'CONNECTED WITH THE DATA AND CONTROL CABLES TO THE TRACE MODULE. '
    NL;'      TO TEST THE EMULATOR, A BUFFER MODULE (EITHER THE TMS9900'
    NL;'OR TMS9980) SHOULD BE CONNECTED TO THE EMULATOR MODULE.  THE CLOCK'
    NL;'SELECT SWITCH ON THE BUFFER MODULE MUST BE SET TO "INTERNAL". '
    NL;'THE TARGET MICROPROCESSOR CONNECTOR CABLES FROM THE'
    NL;'BUFFER MODULE SHOULD NOT BE PLUGGED INTO A TARGET SYSTEM. '
    NL;'REFER TO SECTION 2.2.3 OF THE AMPL SYSTEM OPERATION GUIDE'
    NL;'(#946244-9701) FOR DETAILS OF THESE CONNECTIONS. '
        C:NX5;'TYPE C<RETURN> FOLLOWING THE "=?" TO CONTINUE'
        C = READ
    NL;'      THE TITLE OF EACH TEST IS PRINTED AS IT IS PERFORMED. '
    NL;'FAILURE OF THE TEST WILL RESULT IF AMPL DETECTS AN ERROR'
    NL;'OR INCORRECT RESULTS ARE DETECTED BY THE TEST PROCEDURE. '
    NL;'IN EITHER CASE A MESSAGE WILL BE PRINTED AND THE TEST ABORTED. '
    NL;'YOU MAY RETRY THE TEST THAT JUST FAILED BY ENTERING:'
    NL;'           AMPHDT<RETURN>      FOLLOWING "?"'
    NL;'THE COMPLETE TEST PASSES ONLY IF NO ERRORS ARE DETECTED. '
        C:NX5;'TYPE C<RETURN> FOLLOWING THE "=?" TO CONTINUE'
```

Figure H-1. Listing of AMPHDT Procedure File (Sheet 5 of 6)

```
        C = READ
END
..
AMPHDT
```

Figure H-1. Listing of AMPHDT Procedure File (Sheet 6 of 6)

```
...TITLE:          ASRKSR:  ROUTINES TO USE AN ASR OR KSR TERMINAL
...REVISION:       03/24/77
...
...ABSTRACT:
...     THREE ROUTINES ARE CONTAINED IN THIS FILE.  THEY INTERACT WITH
...EITHER AN ASR733 OR A KSR743 TERMINAL CONNECTED TO THE TARGET
...SYSTEM.  THE INITRM PROCEDURE INITIALIZES THE INTERFACE LOGIC AT
...THE SPECIFIED CRU BASE ADDRESS.  GETCHR AND PUTCHR READ AND WRITE
...A SINGLE ASCII CHARACTER, RESPECTIVELY, ONCE INITRM HAS BEEN CALLED.
...
...USAGE:
...     INITRM HAS ONE ARGUMENT, THE CRU BASE ADDRESS.  IF THIS
...IS OMITTED, CRU BASE ZERO IS USED.  INITRM LEAVES SYSTEM VARIABLE
...CRUB SET TO THE CRU BASE OF THE DEVICE, SO PUTCHR AND GETCHR CAN
...USE IT.  INITRM MAY BE CALLED ONLY WHEN THE EMULATOR IS HALTED.
...GETCHR READS ONE CHARACTER FROM THE DEVICE INTERFACE AT THE BASE
...ADDRESS IN SYSTEM VARIABLE CRUB.  GETCHR HAS NO ARGUMENTS AND
...RETURNS THE ASCII CODE FOR THE CHARACTER TYPED.  GETCHR DOES NOT
...RETURN UNTIL A CHARACTER IS TYPED ON THE TERMINAL.  GETCHR MAY BE
...CALLED ONLY WHEN THE EMULATOR IS HALTED.
...PUTCHR WRITES ITS ONE ARGUMENT ON THE DEVICE INTERFACE AT THE BASE
...ADDRESS IN SYSTEM VARIABLE CRUB.  PUTCHR IS A PROCEDURE AND DOES
...NOT RETURN A VALUE.  PUTCHR MAY BE CALLED ONLY WHEN THE EMULATOR
...IS HALTED.
...
'DEFINING INITRM'
...
PROC INITRM(0) BEGIN
  IF ARG 0
       THEN CRUB = ARG 1              ...BASE ADDRESS WAS SPECIFIED
       ELSE CRUB = 0;                 ...NOT SPECIFIED
  CRUW(09,1,1);                       ...SET DATA TERMINAL READY
  CRUW(0A,1,1);                       ...SET REQUEST TO SEND
  CRUW(0B,1,1);                       ...CLEAR WRITE REQUEST
  CRUW(0C,1,1);                       ...CLEAR READ REQUEST
  CRUW(0E,1,0);                       ...DISABLE INTERRUPTS
  CRUW(0F,1,0);                       ...DISABLE DIAGNOSTIC MODE
END
...
NL
'DEFINING FUNCTION GETCHR'
...
FUNC GETCHR (0,1) BEGIN
  REPEAT NULL UNTIL CRUR(0C,1);       ...WAIT UNTIL READ REQUEST
  LOC 1 = CRUR(0,8);                  ...READ THE CHARACTER
  CRUW(0C,1,1);                       ...CLEAR READ REQUEST
  RETURN LOC 1;                       ...RETURN CHARACTER
END
...
NL
'DEFINING PUTCHR'
...
PROC PUTCHR(1) BEGIN
  REPEAT NULL UNTIL CRUR(0E,1);       ...WAIT UNTIL DATA SET READY
  CRUW(0,8,ARG 1);                    ...SEND THE CHARACTER
  REPEAT NULL UNTIL CRUR(0B,1);       ...WAIT ON WRITE REQUEST FINISHED
  CRUW(0B,1,1);                       ...CLEAR WRITE REQUEST
END
```

Figure H-2. Listing of ASRKSR Procedure File

```
..TITLE:      SIMPLE DEBUG COMMANDS
..REVISION:   09/27/77
..
..ABSTRACT:
..        THIS SET OF AMPL FUNCTIONS AND PROCEDURES IMPLEMENTS
.. A SMALL AND SIMPLE SET OF COMMANDS FOR THE USER TO DEBUG A
.. PROTOTYPE SYSTEM.  THE COMMANDS ARE GROUPED INTO THREE BASIC
.. COMMAND CLASSES:
..
..     1) EMULATION CONTROL:                    (LOADUP, RUN, HALT)
..              LOAD MEMORY, RUN AND HALT THE EMULATOR
..
..     2) TARGET SYSTEM STATUS:                 (STAT, DR, DT)
..              CURRENT EMULATION STATUS, REGISTERS AND TRACE BUFFER
..
..     3) TRACE/BREAKPOINT SPECIFICATION:       (TRACE,TILL,VALUE,AT,DELAY)
..              USER SELECTION OF WHAT TO TRACE AND WHEN
..              TO HALT THE EMULATION
..
..   THE USER MAY OBTAIN A LIST OF THESE COMMANDS AND THE SYNTAX FOR
..   THEIR ARGUMENTS AT ANY TIME BY USE OF THE 'HELP' COMMAND.
..
..   ***** IMPORTANT NOTE:
..          THIS SET OF DEBUG COMMANDS PRESUMES THE FOLLOWING
..          CONFIGURATION OF THE AMPL HARDWARE:
..           - THERE IS ONE EMULATOR MODULE AND ONE TRACE MODULE
..               (NAMED EM01 AND TM01).
..           - THE EMULATOR AND TRACE MODULE ARE 'STRAPPED' TOGETHER
..               WITH THE DATA AND CONTROL CABLES.
..           - THE USER HAS A 911 VDT CONSOLE FOR INTERACTIVE USE.
..
..   COMPLETE USER DOCUMENTATION OF THE DEBUG COMMANDS IS AVAILABLE ON
..   FILE: DEBUG/DOC (FS990) OR . S$SYSLIB. AMPL$LIB. DEBUGDOC  (DX/10 3.0)
..
.. ***************************************************************************
..
NL
'                        "DEBUG COMMANDS"'
..
ARRAY ZDBG(9)
..
.. ---------------------------------------------------------------------------
..
NL
'DEFINING: HELP'
..
PROC HELP(0,1) BEGIN
     NL
     'EMULATOR CONTROL COMMANDS:'
     LOC 1:NX5;'LOADUP [('''FILENAME'''[,BIAS[,EUM]])]'
       ' - INITIALIZE AND LOAD TARGET'
     LOC 1:NX5;'RUN [(PC[,WP])] - RUN THE TARGET SYSTEM'
     LOC 1:NX5;'HALT            - HALT THE TARGET SYSTEM'
     ..
     NL;NL
     'TARGET SYSTEM STATUS:'
     LOC 1:NX5;'STAT                - TARGET SYSTEM STATUS'
     LOC 1:NX5;'DR                  - DISPLAY REGISTERS'
     LOC 1:NX5;'DT [(FIRST[,LAST])] - DISPLAY TRACE'
     ..
     NL;NL
     'TRACE SPECIFICATION:'
```

Figure H-3. Listing of DEBUG Procedure File (Sheet 1 of 6)

```
     LOC 1:NX5;' TRACE [(COUNT[,QUAL])]'
     LOC 1:NX0;'COUNT: 0..256 (0=INFINITY), QUAL: IAQ,DBIN,-DBIN'
     ..
     NL;NL
     'BREAKPOINT SPECIFICATION:'
     LOC 1:NX5;' TILL [(COUNT)]              COUNT: 1..>FFFF'
     LOC 1:NX5;' MREAD, MWRITE, OR INSTR     CHOOSE ONE OR NONE'
     LOC 1:NX5;' VALUE (DATA[,MASK])         DEFAULT MASK IS >FFFF'
     LOC 1:NX5;' AT (ADDR)'
     LOC 1:NX5;' DELAY (COUNT)               COUNT: 1..256'
     ..
     END
..
.. ------------------------------------------------------------------------
..
PROC TRACE NULL          ..DUMMY TRACE PROCEDURE
PROC STAT NULL           ..DUMMY STAT PROCEDURE
NL
'DEFINING: LOADUP [(''FILENAME''[,BIAS[,EUM]])]'
..
PROC LOADUP (0,1) BEGIN
     NL;'TX OR DX ? (0=TX,1=DX)'
     IF READ THEN BEGIN
          EINT('EM01')
          TINT('TM01')
          END
       ELSE BEGIN
          EINT('EMU')
          TINT('TRA')
          END
       ETM=OFF
     ..INITIALIZE EMULATOR USER MEMORY
       IF ARG 0 GE 3
          THEN EUM = ARG 3
          ELSE EUM = ON  ..DEFAULT TO EMULATOR MEMORY ENABLED
     ..LOAD THE USERS PROGRAM
       IF ARG 0 GE 1 THEN BEGIN
          LOC 1 = >A0     ..DEFAULT LOAD BIAS
          IF ARG 0 GE 2 THEN LOC 1 = ARG 2
          LOAD(ARG 1,LOC 1)
          WP = @LOC 1
          PC = @(LOC 1 +2)
          END
     ..INITIALIZE TRACING
       TRACE
     ..
       STAT
     ..
       END
..
.. ------------------------------------------------------------------------
..
NL
'DEFINING: RUN [ (PC [,WP] ) ]'
..
PROC RUN BEGIN
     TRUN
     IF ARG 0 GE 1 THEN PC=ARG 1
     IF ARG 0 GE 2 THEN WP=ARG 2
     ERUN
     STAT
     END
```

Figure H-3. Listing of DEBUG Procedure File (Sheet 2 of 6)

```
..
.. ------------------------------------------------------------------------
..
NL
'DEFINING: HALT'
..
PROC HALT BEGIN
     EHLT:X
     THLT:X
     STAT
     END
..
.. ------------------------------------------------------------------------
..
... TITLE:         STAT
... REVISION:      08/30/77
...
... ABSTRACT:
...      STAT USES SYSTEM VARIABLES EST, TNE, ETBO, AND EUM TO PRINT
... THE STATUS OF THE EMULATOR AND THE DEBUG SETUP.
...
... USAGE:
...      STAT HAS NO ARGUMENTS AND MAY BE CALLED AT ANY TIME.
...
DELE('STAT')
...
PROC TQUAL CASE ZDBG(2)-DATA OF
       0 :: ' MEMORY CYCLES'
       IAQ :: ' INSTRUCTION ACQUISITIONS'
       DBIN :: ' READ MEMORY CYCLES'
       -DBIN :: ' WRITE MEMORY CYCLES'
      END
...                 '
NL
'DEFINING: STAT'
...
PROC STAT (0,1) BEGIN
        LOC 1 = (EST  AND 1) EQ 0
     ..
     .. EMULATION CONTROL
        NL;'EMULATION CONTROL:';LOC 1:NX5;
        IF LOC 1 THEN BEGIN
                  'HALTED'
                  IF EUM THEN '      (EMULATOR USER MEMORY ENABLED)'
                  1-ETBO:N2X5D;'SAMPLES TRACED:'
                  IF TTBO LE TTBN THEN BEGIN
                       TTBO:D;'...';TTBN:D;
                       END;
                  TNE:NX5U;'EVENTS COUNTED.'
                  NL;DR
                  END
             ELSE BEGIN
                  'RUNNING'
                  IF EST AND 8 THEN ' AT IDLE'
                  END
     ..
     .. DEBUG CONFIGURATION
        NL;NL;'DEBUG CONFIGURATION:';LOC 1:NX5;
        'TRACE: '
        IF ZDBG(1) NE 0 THEN ZDBG(1):D
            ELSE ' ALL';
        TQUAL
```

Figure H-3. Listing of DEBUG Procedure File (Sheet 3 of 6)

```
        IF ZDBG(9) AND EVT THEN BEGIN
            NL;'     TILL: ' ZDBG(3):DNX5
            CASE (ZDBG(4) - DATA) OF
                 0 :: 'MEMORY CYCLE'
                 IAQ :: 'INSTRUCTION ACQUISITION'
                 DBIN-IAQ :: 'MEMORY READ'
                 -DBIN :: 'MEMORY WRITE'
                 END
            IF ZDBG(6) NE 0 THEN BEGIN
                 '  VALUE: ';ZDBG(5)
                 IF ZDBG(6) NE -1 THEN BEGIN
                      ' MASK: ' ZDBG(6):H
                    END
                 END
            IF ZDBG(7) NE 0FFFF THEN BEGIN
                 ' AT ADDRESS: ';ZDBG(7);
                 END
            IF ZDBG(8) NE 0 THEN BEGIN
                 NL;'     DELAY: ';ZDBG(8):D;TQUAL;
                 END
            END

      END
..
.. ------------------------------------------------------
..
NL
'DEFINING: DT [(FIRST[,LAST])]'
...
PROC DT(0,4) BEGIN
 NL;
 '  INDEX    ADDR       DATA(HEX,ASCII)  INTERPRETATION';
 NL;
  LOC 2 = TTBN-19;                          ..LOC 2 IS TRA LOWER INDEX
  IF LOC 2 LT TTBO THEN LOC 2 = TTBO;
  LOC 3 = TTBN                              ..LOC 3 IS TRA UPPER INDEX
..
  IF ARG 0 GE 1                                        ..SET TRA LOWER INDEX
       THEN IF ARG 1 GE TTBO AND ARG 1 LE TTBN
                 THEN LOC 2 = ARG 1
  IF ARG 0 GE 2                                        ..SET TRA UPPER INDEX
       THEN IF ARG 2 GE LOC 2 AND ARG 2 LE TTBN
                 THEN LOC 3 = ARG 2;
  LOC 1 = ETBO + LOC 2 - TTBO                          ..ADJUST EMU INDEX
..
  WHILE LOC 2 LE LOC 3
       DO BEGIN
          LOC 2:NDX3;                       ..PRINT INDEX IN TRACE BUFFER
          IF LOC 1 GE ETBO AND LOC 1 LE ETBN
               THEN ETB(LOC 1):H    ..PRINT TRACED ADDRESS IN HEX
               ELSE '        ';
          TTB(LOC 2):X4HA;          ..PRINT TRACED VALUE IN HEX,ASCII
          IF TTBH(LOC 2,IAQ)
              THEN TTB(LOC 2):X6I     ..DISPLAY THE INSTRUCTION
              ELSE BEGIN                ...ITS A DATA ACCESS
                   IF TTBH(LOC 2,DBIN)
                        THEN '          READ  '  ...IT'S A READ
                        ELSE '          WRITE ';  ...IT'S A WRITE
                   TTB(LOC 2):H;       ...DISPLAY THE DATA
                   END;
          LOC 2 = LOC 2 + 1;            ...STEP TTB INDEX
          LOC 1 = LOC 1 + 1;            ...STEP ETB INDEX
```

Figure H-3. Listing of DEBUG Procedure File (Sheet 4 of 6)

```
          END
END
..
.. ------------------------------------------------------------------
..
DELE('TRACE')
NL
'DEFINING: TRACE [(COUNT[,QUAL])]'
..
..
PROC TRACE (0) BEGIN
     .. INITIALIZE THE DEFAULT STATE
        ZDBG(1) = 0              .. TRACE COUNT
        ZDBG(2) = DATA           .. TRACE QUALIFIER
        ZDBG(3) = 1              .. EVENT COUNT
        ZDBG(4) = DATA           .. BUS CONTROL QUALIFIER
        ZDBG(5) = 0              .. DATA BUS COMPARISON VALUE
        ZDBG(6) = 0              .. DATA BUS COMPARISON MASK
        ZDBG(7) = 0FFFF          .. ADDRESS BUS COMPARISON VALUE
        ZDBG(8) = 0              .. TRACE DELAY COUNT
        ZDBG(9) = OFF            .. HALTING CONDITIONS
     .. INITIALIZE AMPL HARDWARE PROCEDURES
        ETRC(ADDR,256,EXT)          .. TRACE ADDRESS IN THE EMULATOR
        ECMP(IAQ,0FFFF)             .. NO EMULATOR ADDRESS COMPARISON
        EEVT(INT)                   .. BUT WILL USE COMPARISON IF SELECTED
        EBRK(OFF,SELF)              .. TRACE MODULE MAY HALT THE EMULATOR
        TTRC(DATA,0,EXT)
        TCMP(DATA+IAQ-DBIN,0,0,OFF) .. TM DATA COMPARISON, NO LATCHES
        TEVT(1,0,INT+NORM+EACH)     .. TM EVENT COUNTER
        TBRK(OFF,EMU)               .. STOP EMULATOR IF BREAKPOINT DEFINED
     .. CHECK FOR TRACE PROCEDURE ARGUMENTS
        IF ARG 0 GE 2 THEN BEGIN
                          ZDBG(2) = ZDBG(2)+ARG 2  .. TRACE QUALIFIER
                          ZDBG(4) = DATA+ARG 2     .. CMPR QUALIFIER
                          END
        IF ARG 0 GE 1 THEN IF ARG 1 NE 0 THEN BEGIN
                           ZDBG(1) = ARG 1   .. TRACE COUNT
                           ZDBG(9) = FULL    .. BREAK ON TRACE BUFFER FULL
                           END
     .. INITIALIZE 'TTRC'
        TTRC(ZDBG(2)+(EMT-1)*Q0,ZDBG(1),EXT)
        TBRK(ZDBG(9))
     ..
        END
..
.. ------------------------------------------------------------------
..
NL
'DEFINING: TILL [(COUNT)]'
..
PROC TILL (0) BEGIN
     .. CHECK FOR OPTIONAL ARGUMENT
        IF ARG 0 GE 1 THEN BEGIN
                  ZDBG(3) = ARG 1        .. EVENT COUNTER
                  TEVT(ZDBG(3))
                  END
     .. HALT ON AN EVENT (DEFAULT TO SAME AS TRACING)
        TCMP(ZDBG(4))
        ZDBG(9) = ZDBG(9) + EVT
        TBRK(ZDBG(9))
     ..
        END
```

Figure H-3. Listing of DEBUG Procedure File (Sheet 5 of 6)

```
..
.. ------------------------------------------------------------------
..
NL
'DEFINING: MREAD, MWRITE, INSTR'
PROC MREAD BEGIN
       ZDBG(4) = DATA+DBIN-IAQ  ..LOOK FOR MEMORY READ
       TCMP(ZDBG(4))
       END
..
PROC MWRITE BEGIN
       ZDBG(4) = DATA-DBIN      ..LOOK FOR MEMORY WRITE
       TCMP(ZDBG(4))
       END
..
PROC INSTR BEGIN
       ZDBG(4) = DATA+IAQ       ..LOOK FOR INSTRUCTION ACQUISITON
       TCMP(ZDBG(4))
       END
.. ------------------------------------------------------------------....
..
NL
'DEFINING: VALUE (DATA[,MASK])'
..
PROC VALUE (1) BEGIN
     ..CHECK FOR OPTIONAL ARGUMENT
       IF ARG 0 GE 2 THEN ZDBG(6) = ARG 2  ..TM COMPARISON MASK
                     ELSE ZDBG(6) = -1   ..DEFAULT MASK
     ..INITIALIZE TRACE MODULE COMPARATOR
       ZDBG(5) = ARG 1   ..TM COMPARISON DATA VALUE
       TCMP(ZDBG(4),ZDBG(5),ZDBG(6))
     ..
       END
.. ------------------------------------------------------------------.
..
NL
'DEFINING: AT (ADDR)'
..
PROC AT (1) BEGIN
     .. INITIALIZE EMULATOR COMPARATOR
       ZDBG(7) = ARG 1
        ECMP(ADDR,ARG 1 OR (1-EMT))
       TCMP(ZDBG(4)+EMU)
     ..
       END
.. ------------------------------------------------------------------..
..
NL
'DEFINING: DELAY (COUNT)'
..
PROC DELAY (1) BEGIN
     ..INITIALIZE EVENT DELAY
       ZDBG(8) = ARG 1   ..EVENT DELAY
       TEVT(ZDBG(3),ARG 1)
     ..
       END
.. ------------------------------------------------------------------
```

Figure H-3. Listing of DEBUG Procedure File (Sheet 6 of 6)

```
TITLE:      SIMPLE DEBUG COMMANDS (DOCUMENTATION)
REVISION:   09/27/77

ABSTRACT:
        THIS SET OF AMPL FUNCTIONS AND PROCEDURES IMPLEMENTS
 A SMALL, SIMPLE, UNIFORM SET OF COMMANDS FOR THE USER TO DEBUG A
 PROTOTYPE SYSTEM.  THE COMMANDS ARE GROUPED INTO THREE BASIC
 COMMAND CLASSES:

     1) EMULATION CONTROL:                          (LOADUP, RUN, HALT)
              LOAD MEMORY, RUN AND HALT THE EMULATOR

     2) TARGET SYSTEM STATUS:                       (STAT, DR, DT)
              CURRENT EMULATION STATUS, REGISTERS AND TRACE BUFFER

     3) TRACE/BREAKPOINT SPECIFICATION:             (TRACE, TILL, VALUE, AT, DELAY)
              USER SELECTION OF WHAT TO TRACE AND WHEN
              TO HALT THE EMULATION

   THE USER MAY OBTAIN A LIST OF THESE COMMANDS AND THE SYNTAX FOR
   THEIR ARGUMENTS AT ANY TIME BY USE OF THE 'HELP' COMMAND.

   ***** IMPORTANT NOTE:
          THIS SET OF DEBUG COMMANDS PRESUMES THE FOLLOWING
          CONFIGURATION OF THE AMPL HARDWARE:
           - THERE IS ONE EMULATOR MODULE AND ONE TRACE MODULE
               (NAMED EM01 AND TM01).
           - THE EMULATOR AND TRACE MODULE ARE 'STRAPPED' TOGETHER
               WITH THE DATA AND CONTROL CABLES.
           - THE USER HAS A 911 VDT CONSOLE FOR INTERACTIVE USE.

   THIS IS THE USER DOCUMENTATION FOR THE ABOVE COMMANDS.  THE COMMANDS
   ARE DEFINED ON:  DEBUG/PRC (FS990) OR
                    . S$SYSLIB. AMPL$LIB. DEBUG (DX 3. 0)

*******************************************************************************


HELP COMMAND:

     DISPLAY THE SYNTAX OF THE DEBUG COMMANDS.

          EXAMPLE DISPLAY:

     EMULATOR CONTROL COMMANDS:
          LOADUP [(<'FILENAME'[, BIAS[, EUM]])] - INITIALIZE AND LOAD TARGET RY
          RUN [(PC[, WP])] - RUN THE TARGET SYSTEM
          HALT             - HALT THE TARGET SYSTEM

     TARGET SYSTEM STATUS:
          STAT                  - TARGET SYSTEM STATUS
          DR                    - DISPLAY REGISTERS
          DT [(FIRST[, LAST])] - DISPLAY TRACE

     TRACE SPECIFICATION:
          TRACE [(COUNT[, QUAL])]
               COUNT: 0. . 256 (0=INFINITY), QUAL: IAQ, DBIN, -DBIN

     BREAKPOINT SPECIFICATION:
          TILL [(COUNT)]              COUNT: 1. . >FFFF
          MREAD, MWRITE, OR INSTR     CHOOSE ONE OR NONE
          VALUE (DATA[, MASK])        DEFAULT MASK IS >FFFF
```

Figure H-4. Listing of DEBUG Documentation File (Sheet 1 of 7)

```
        AT (ADDR)
        DELAY (COUNT)                COUNT: 1..256

*******************************************************************************

EMULATOR CONTROL:

    LOAD AND INITIALIZE EMULATOR MEMORY

        INITIALIZE THE EMULATOR AND LOGIC STATE TRACE MODULES.
        TRANSFER AN OBJECT FILE INTO MEMORY OF THE TARGET
        SYSTEM AND INITIALIZE THE WP AND PC REGISTERS.

              LOADUP [('FILENAME'[,BIAS[,EUM]])]'

        'FILENAME' - MUST BE AN OBJECT MODULE.  THE STANDARD TI
             OPERATING SYSTEM INTERFACE IS EXPECTED IN THE OBJECT
             MODULE - THE FIRST TWO WORDS ARE THE WORKSPACE POINTER
             AND INITIAL PROGRAM COUNTER RESPECTIVELY.  THE WORKSPACE
             POINTER (WP) AND PROGRAM COUNTER (PC) ARE INITIALIZED
             TO THESE VALUES.
        BIAS - IS AN ADDRESS BY WHICH TO RELOCATE THE OBJECT MODULE.
             THE DEFAULT BIAS IS >A0.
        EUM - SELECTS 'EMULATOR USER MEMORY', A 4K WORD (ADDRESSES
             0 THROUGH >1FFF) BLOCK OF MEMORY ON THE EMULATOR MODULE.
             'ON' WILL CAUSE THE TARGET SYSTEM TO USE THE EMULATOR
             MODULE MEMORY IN THIS ADDRESS RANGE.  'OFF' WILL ALLOW
             THE TARGET SYSTEM TO ADDRESS ITS OWN MEMORY IN THE ADDRESS
             RANGE >0000 THROUGH >1FFF.  THE DEFAULT VALUE OF EUM IS
             'ON' (USE EMULATOR USER MEMORY).

    -------------------------------------------------------------------------

    RUN EMULATION

        BEGIN EXECUTION IN THE TARGET SYSTEM WITH THE CURRENT
        WORKSPACE AND PROGRAM COUNTER IF NOT SPECIFIED IN THE
        COMMAND.

             RUN [(PC [,WP])]

    -------------------------------------------------------------------------

    HALT EMULATION

        STOP EMULATION IN THE TARGET SYSTEM.  A 'STAT' COMMAND
        IS ALSO PERFORMED.

             HALT

*******************************************************************************

TARGET SYSTEM STATUS:
```

Figure H-4. Listing of DEBUG Documentation File (Sheet 2 of 7)

```
EMULATION STATUS

     DISPLAY INFORMATION CONCERNING THE CURRENT STATUS OF
     THE TARGET SYSTEM (EMULATION CONTROL), AND THE
     TRACE/BREAKPOINT SELECTION (DEBUG CONFIGURATION).

          STAT

     EXAMPLE DISPLAY:


EMULATION CONTROL:
     HALTED      (EMULATOR USER MEMORY ENABLED)

        256  SAMPLES TRACED:  -255  ...      0
          2  EVENTS COUNTED.

R0  = >FFFF  R8  = >1C45  PC  = >0132  / >0601  DEC  R1
R1  = >0FE6  R9  = >1C44  WP  = >0138
R2  = >0000  R10 = >1444  ST  = >D200
R3  = >0000  R11 = >010E
R4  = >1C45  R12 = >1FE0
R5  = >1C44  R13 = >1844
R6  = >1C45  R14 = >1845
R7  = >1444  R15 = >1800



DEBUG CONFIGURATION:
     TRACE:  ALL MEMORY CYCLES
     TILL:       2
     INSTRUCTION ACQUISITION AT ADDRESS: >0106
     DELAY:    128   MEMORY CYCLES

-------------------------------------------------------------------------

DISPLAY THE WORKSPACE REGISTERS

     DISPLAY THE CURRENT VALUE OF THE PROGRAM COUNTER,
     THE WORKSPACE, AND THE VALUES OF THE WORKSPACE
     REGISTERS THIS INFORMATION IS ALSO INCLUDED IN THE STAT
     COMMAND.  THIS COMMAND MAY ONLY BE USED WHEN THE
     EMULATOR IS NOT EXECUTING.

          DR

       EXAMPLE DISPLAY:

R0  = >FFFF  R8  = >1C45  PC  = >0132  / >0601  DEC  R1
R1  = >0FE6  R9  = >1C44  WP  = >0138
R2  = >0000  R10 = >1444  ST  = >D200
R3  = >0000  R11 = >010E
R4  = >1C45  R12 = >1FE0
R5  = >1C44  R13 = >1844
R6  = >1C45  R14 = >1845
R7  = >1444  R15 = >1800

-------------------------------------------------------------------------

DISPLAY THE TRACE BUFFER
```

Figure H-4. Listing of DEBUG Documentation File (Sheet 3 of 7)

```
DISPLAY AND INTERPRET THE TRACE SINCE THE LAST RUN.

EACH SAMPLE OF THE TRACE BUFFER CONSISTS OF THE ADDRESS BUS,
DATA BUS, AND BUS CONTROL LINES.  THIS INFORMATION IS
DISPLAYED ALONG WITH THE INDEX INTO THE BUFFER ALONG THE
LEFT HAND SIDE.  THE NUMBER OF SAMPLES TRACE IS GIVEN
BY THE 'STAT' COMMAND.  THIS COMMAND MAY ONLY BE USED WHEN
THE EMULATOR IS NOT EXECUTING.

     DT [<FIRST[,LAST]>]

NO ARGUMENTS WILL DISPLAY THE LAST 20 SAMPLES TAKEN.
ONLY ONE ARGUMENT WILL DISPLAY 20 SAMPLES BEGINNING
 WITH THE THE ARGUMENT SPECIFIED.
TWO ARGUMENTS DISPLAYS THE TRACE BETWEEN THE TWO INDICES.

EXAMPLE DISPLAY:

INDEX    ADDR       DATA<HEX,ASCII>  INTERPRETATION

 -12     >0132      >0601  ..        DEC  R1
 -11     >013A      >0002  ..            READ  >0002
 -10     >013A      >0001  ..            WRITE >0001
  -9     >0134      >16FE  .^        JNE  $->0002
  -8     >0132      >0601  ..        DEC  R1
  -7     >013A      >0001  ..            READ  >0001
  -6     >013A      >0000  ..            WRITE >0000
  -5     >0134      >16FE  .^        JNE  $->0002
  -4     >0136      >045B  .[        RT
  -3     >014E      >0120  .             READ  >0120
  -2     >0120      >10F2  .R            READ  >10F2
  -1     >0120      >10F2  .R        JMP  $->001A
   0     >0106      >0200  ..        LI   R0,>0000
   1     >0108      >FFFF  ..            READ  >FFFF
   2     >0138      >FFFF  ..            WRITE >FFFF
   3     >010A      >06A0  .         BL   @>0000
   4     >010C      >0122  ."            READ  >0122
   5     >0122      >020C  ..            READ  >020C
   6     >014E      >010E  ..            WRITE >010E
   7     >0122      >020C  ..        LI   R12,>0000

*******************************************************************************

TRACE/BREAKPOINT SELECTION:

     THIS SET OF DEBUG COMMANDS ALLOWS THE USER TO QUICKLY SPECIFY
      - WHAT TO TRACE.
      - A MAXIMUM NUMBER OF TRACE SAMPLES DESIRED.
      - THE NUMBER OF 'EVENTS' TO COUNT BEFORE GENERATING A 'BREAKPOINT'
          CONDITION.
      - WHAT CONSTITUTES AN 'EVENT'.
      - THE NUMBER OF SAMPLES TO TAKE AFTER 'BREAKPOINT'
          BEFORE ACTUALLY HALTING THE EMULATOR.

     WHERE AN 'EVENT' IS DEFINED AS THE SUCCESSFUL MATCHING OF A
     USER SPECIFIED DATA BUS VALUE AND/OR ADDRESS BUS VALUE AND/OR
     BUS CONTROL QUALIFIER.  FOR EXAMPLE A TYPICAL 'EVENT' IS AN
     INSTRUCTION ACQUISITION AT ADDRESS >100 OR A MEMORY WRITE
     OF DATA VALUE >1A AT ADDRESS >330.
     AFTER 'N' NUMBER OF SUCH EVENTS HAVE OCCURED A 'BREAKPOINT'
     CONDITION IS GENERATED.  THE 'BREAKPOINT' SIGNAL REQUESTS THE
     EMULATOR TO HALT EXECUTION FOLLOWING COMPLETION OF THE CURRENTLY
```

Figure H-4. Listing of DEBUG Documentation File (Sheet 4 of 7)

```
EXECUTING INSTRUCTION. ALTERNATIVELY, THIS HALTING ACTION MAY
BE 'DELAYED' UNTIL A USER SPECIFIED NUMBER OF ADDTIONAL TRACE
SAMPLES HAVE BEEN TAKEN. FOR EXAMPLE, THE NEXT 100 INSTRUCTIONS
FOLLOWING THE 'BREAKPOINT' CONDITION.

THE COMMANDS DEFINED HERE REPLACE THE AMPL EMULATOR/TRACE MODULE
USER INTERFACE WITH A SIMPLE, UNIFORM YET POWERFULL COMMAND SET.

-------------------------------------------------------------------

TRACE SPECIFICATION:

       SPECIFY WHAT IS TO BE TRACE AND HOW MANY BEFORE HALTING
       THE EMULATION.  THIS COMMAND MAY ONLY BE USED WHEN THE
       EMULATION IS HALTED AND MUST BE STATED BEFORE USE OF
       ANY OF THE BREAKPOINT SPECIFICATION COMMANDS.

            TRACE [(COUNT [,QUAL])]

       IF NO ARGUMENTS ARE SPECIFIED ALL MEMORY CYCLES ARE TRACED.
       THE COUNT ARGUMENT INDICATES THE NUMBER OF TRACE SAMPLES
        TO TAKE BEFORE EMULATION IS HALTED.  THE VALUE MAY
        BE FROM 0 TO 256.  THE VALUE 0 INDICATES INFINITY - DO
        NOT HALT EMULATION (THE DEFAULT VALUE).
       THE QUAL ARGUMENT MUST BE: IAQ, DBIN, OR -DBIN TO SPECIFY
        TRACING ALL INSTRUCTION ACQUISITIONS, ALL READ CYCLES, OR
        ALL WRITE CYCLES RESPECTIVELY.  LEAVING THIS ARGUMENT OUT
        DEFAULTS THE TRACE SAMPLES TO ALL MEMORY CYCLES
        (INSTRUCTION ACQUISITIONS, READ AND WRITE CYCLES).

-------------------------------------------------------------------

BREAKPOINT SPECIFICATON:

       AN EVENT IS DEFINED AS A SPECIFIC TYPE OF MEMORY CYCLE,
        WITH A PARTICULAR DATA VALUE, AT A DEFINITE ADDRESS.
       A BREAKPOINT OCCURS AFTER N NUMBER OF EVENTS HAVE OCCURED.


TILL

       THE TILL COMMAND SPECIFIES THAT AN EVENT IS BEING DEFINED.
        IF NO OTHER EVENT SPECIFICATION IS DONE THE DEFAULT EVENT
        IS ANY MEMORY CYCLE.

            TILL [(COUNT)]

       COUNT - THE NUMBER OF EVENTS BEFORE A BREAKPOINT IS GENERATED.
            THE DEFAULT VALUE IS ONE (1).

-------------------------------------------------------------------

EVENT MEMORY CYCLE

       THE USER MAY SELECT ONE OF THE FOLLOWING COMMANDS:

            INSTR          - INSTRUCTION ACQUISITION CYCLE

            MREAD          - MEMORY READ CYCLE
```

Figure H-4. Listing of DEBUG Documentation File (Sheet 5 of 7)

```
            MWRITE          - MEMORY WRITE CYCLE

        IF NONE OF THE ABOVE ARE SPECIFIED THE EVENT IS
         DEFINED TO BE ANY MEMORY CYCLE.

-------------------------------------------------------------------------

EVENT DATA VALUE

        A DATA BUS VALUE MAY BE SPECIFIED BY USE OF THE VALUE COMMAND.

            VALUE <DATA [,MASK] >

        DATA - VALUE TO MATCH ON THE DATA BUS FOR AN EVENT
        MASK - INDICATES WHICH BITS OF 'DATA' ARE TO BE COMPARED
               AGAINST THE DATA BUS.  THE DEFAULT IS TO >FFFF (COMPARE
               ALL 16 BITS OF 'DATA')

-------------------------------------------------------------------------

EVENT ADDRESS

        AN ADDRESS SPECIFICATION REQUIRES THE EVENT TO OCCUR
        AT THAT ADDRESS.

            AT <ADDRESS>

        ADDRESS - A MEMORY CYCLE AT THIS ADDRESS DEFINES AN EVENT

-------------------------------------------------------------------------

BREAKPOINT DELAY

        HALTING EMULATION OF THE TARGET SYSTEM MAY BE DELAYED
        FOLLOWING THE LAST EVENT BY USE OF THE DELAY COMMAND.

            DELAY <COUNT>

        COUNT - NUMBER OF TRACE SAMPLES TO TAKE FOLLOWING THE
            LAST EVENT (BREAKPOINT CONDITION) BEFORE HALTING
            EMULATION.  COUNT MAY BE IN THE RANGE 0 TO 255.

*************************************************************************

TRACE/BREAKPOINT SPECIFICATION EXAMPLES:

    THE COMMANDS DESCRIBED ABOVE SHOULD BE INVOKED IN THE ORDER
    GIVEN.  THEIR USE IS ILLUSTRATED BELOW.  NOTE THAT THE OPTIONAL
    ARGUMENTS OF EACH COMMAND DEFAULT TO REASONABLE VALUES.

    TRACE           ..TRACE INFINITE NUMBER OF ALL MEMORY CYCLES
    TRACE(10,IAQ)   ..TRACE NEXT 10 INSTRUCTION ACQUISITIONS
    TRACE(0,IAQ)    ..TRACE INFINITE NUMBER OF INSTRUCTION ACQUISITIONS

    TILL            ..BREAKPOINT ON THE FIRST EVENT
    TILL(10)        ..BREAKPOINT ON THE TENTH EVENT
```

Figure H-4. Listing of DEBUG Documentation File (Sheet 6 of 7)

```
INSTR           . . AN EVENT MUST BE AN INSTRUCTION ACQUISITION
MREAD           . . AN EVENT MUST BE A MEMORY READ
MWRITE          . . AN EVENT MUST BE A MEMORY WRITE

VALUE(>10FF)        . . LOOK FOR >10FF ON THE DATA BUS
VALUE(>1000,>FF00) . . LOOK FOR >10XX ON THE DATA BUS (X-DON'T CARE)

AT(>100)        . . LOOK FOR >100 ON THE ADDRESS BUS

DELAY(10)       . . TAKE TEN MORE TRACE SAMPLES BEFORE HALTING
                . .  EMULATION FOLLOWING N EVENTS.
```

```
THE COMMANDS ARE USED IN CONJUNCTION TO DEFINE
  THE DEBUG CONFIGURATION.


  TRACE           . . TRACE ALL MEMORY CYCLES
  TILL            . . TILL THE FIRST
  INSTR           . . INSTRUCTION ACQUISITION


  TRACE(0,IAQ)    . . TRACE ALL INSTRUCTION ACQUISITIONS
  TILL (10)       . . TILL THE 10TH
  MREAD           . . MEMORY READ
  AT (WP)         . . AT WP (REGISTER ZERO)
  DELAY (128)     . . THEN CONTINUE TRACING FOR 128 MORE INSTRUCTIONS


THE COMMANDS MAY BE TYPED ON THE SAME LINE.

  TRACE TILL(2) MWRITE VALUE(0200) AT(01234) DELAY(10)

MEANS: TRACE ALL MEMORY CYCLES
       TILL THE SECOND
       MEMORY WRITE CYCLE OF VALUE HEX 200 AT ADDRESS HEX 1234
       DELAY 10 MEMORY CYCLES THEN HALT EMULATION.
```

Figure H-4. Listing of DEBUG Documentation File (Sheet 7 of 7)

```
... TITLE:          EDUMP:  DUMP ADDRESSES FROM EMULATOR TRACE BUFFER
... REVISION:       03/28/77
... REVISION:       09/09/77        9980 UPDATE3
...
... ABSTRACT:
...     EDUMP PRINTS THE CONTENTS OF A SPECIFIED RANGE OF THE EMULATOR'S
... TRACE BUFFER, INTERPRETED AS EVEN ADDRESSES TAKEN FROM THE EMULATOR'S
... MEMORY ADDRESS BUS.
...
... USAGE:
...     THE EMULATOR MUST BE SET UP TO TRACE ADDRESSES IN ITS BUFFER,
... BEFORE IT IS RUN.  AFTER IT HAS HALTED, FOR WHATEVER REASON, EDUMP
... MAY BE USED TO EXAMINE A SEQUENCE OF TRACED ADDRESSES.  EDUMP PRINTS
... ONE ADDRESS PER LINE.  AT THE LEFT MARGIN IT PRINTS THE INDEX INTO
... THE EMULATOR TRACE BUFFER.  NEXT IT PRINTS THE SAMPLED ADDRESS IN
... HEXADECIMAL.  LAST IT PRINTS THE CONTENTS OF THE ADDRESSED WORD OF
... TARGET MEMORY, IN HEXADECIMAL, AS TWO ASCII CHARACTERS, AND DISAS-
... SEMBLED AS AN INSTRUCTION WORD.  NOTICE THAT THE DISASSEMBLY MAY
... REQUIRE THE VALUES OF REGISTERS, AND THAT THESE VALUES WILL NOT
... BE LIKELY TO BE THE SAME AT THE TIME OF DISASSEMBLY AS AT THE TIME
... OF EXECUTION OF THE INSTRUCTION.  IF THE 9980 EMULATOR IS BEING
... USED, THEN THE ADDRESSES TRACED MAY BE BOTH EVEN AND ODD FOR
... EACH BYTE FETCHED OR, IF THE TRACE MODULE QUALIFIER Q0 IS BEING
... USED TO QUALIFY SAMPLES, THEN JUST ODD, OR JUST
... EVEN ADDRESSES MAY BE TRACED.  PLEASE REFER TO THE APPROPRIATE
... SECTION OF THE MANUAL FOR A DISCUSSION OF TRACING WHEN USING THE
... 9980 EMULATOR.
...     EDUMP HAS ONE REQUIRED AND ONE OPTIONAL ARGUMENT.  FIRST IS THE
... STARTING INDEX TO BE USED.  THE OPTIONAL SECOND ARGUMENT SPECIFIES
... THE ENDING INDEX TO BE USED.  IF IT IS OMITTED, THE STARTING INDEX
... PLUS NINETEEN IS USED, GIVING TWENTY SAMPLES.
...
'DEFINING EDUMP' NL
...
PROC EDUMP (0,2) BEGIN
  IF ARG 0 EQ 0
       THEN LOC 2 = ETBO
       ELSE LOC 2 = ARG 1;
  IF ARG 0 GE 2
       THEN LOC 1 = ARG 2
       ELSE LOC 1 = LOC 2 + 19;
  IF LOC 1 GT ETBN
       THEN LOC 1 = ETBN;
  'EMU INDEX   ADDR       CONTENTS(HEX,ASCII,INSTR)'; NL;
  WHILE LOC 2 LE LOC 1
       DO BEGIN
            LOC 2:ND;
            ETB(LOC 2):X4H '/  ';
            @ETB(LOC 2):HAI;
            LOC 2 = LOC 2 + 1
       END;
  END
... TITLE:          TDUMP:  DUMP DATA BUS SAMPLES FROM THE TRACE BUFFER
... REVISION:       03/28/77
...                 09/09/77        9980 UPDATES
...
... ABSTRACT:
...     TDUMP PRINTS THE CONTENTS OF THE TRACE MODULE TRACE BUFFER
... ON THE CONSOLE, ASSUMING THAT THE TRACE MODULE HAS BEEN SET TO
... TRACE THE EMULATOR DATA BUS.  IN THIS MODE THE LOW-ORDER 16 BITS
... OF EACH SAMPLE ARE TAKEN FROM THE DATA BUS AND THREE OF THE HIGH-
... ORDER 4 BITS ARE TAKEN FROM THE BUS CONTROL LINES AND THE EMULATOR
```

Figure H-5. Listing of DUMPS Procedure File (Sheet 1 of 5)

```
... COMPARISON CIRCUITRY.  IF THE TRACE SAMPLES ARE NOT ACTUALLY FROM
... THE EMULATOR'S DATA BUS, THE INTERPRETATION PROVIDED BY THIS
... PROCEDURE WILL BE MEANINGLESS.  THE FOLLOWING TRACE MODULE
... SETUP SHOULD BE USED TO PROPERLY SET UP THE 9900 EMULATOR
... TO TRACE 16 BIT DATA VALUES (OR THE 9980 EMULATOR TO TRACE 8
... BIT DATA VALUES).
...
...          TTRC(DATA,256,EXT)
...
... THE TRACING QUALIFIER "Q0" IS NECESSARY FOR TRACING 16 BIT DATA
... VALUES IN THE 9980 EMULATOR.  THE FOLLOWING TRACING SETUP IS USED.
...
...          TTRC(DATA-Q0,256,EXT)
...
... PLEASE REFER TO THE APPROPRIATE SECTION OF THE MANUAL FOR A
... DISCUSSION OF TRACING WHILE USING THE 9980 EMULATOR.  THE ABOVE
... COMMANDS SET THE TRACE MODULE TO STORE UP TO 256 SAMPLES BEFORE "FULL"
... BECOMES TRUE.  THE NUMBER MAY BE ANYTHING BETWEEN 1 AND 256, AND THE
... TERMINATION MODE MAY BE ANYTHING.  THE IMPORTANT POINTS ARE THAT
... "DATA" MODE IS SELECTED AND "EXT" CLOCK IS USED (EMULATOR'S MEMORY
... CYCLE CLOCK).  OF COURSE, THE DATA CABLE MUST BE CONNECTED FOR THE
... DATA BUS TO BE AVAILABLE FOR THE TRACE MODULE.
...
... USAGE:
...
...     AFTER THE TRACE MODULE HAS HALTED (BY ITSELF, UNDER CONTROL OF
... THE EMULATOR, OR BY THE USER WITH "THLT") THE TRACE BUFFER MAY BE
... DUMPED WITH TDUMP.  THERE IS ONE REQUIRED ARGUMENT AND ONE OPTIONAL
... ARGUMENT.  THE FIRST IS THE STARTING INDEX TO BE USED.  THE SECOND
... IS THE ENDING INDEX TO BE USED.  IF IT IS OMITTED, THE START-
... ING INDEX PLUS 19 IS USED, TO DUMP 20 SAMPLES.
...
... EACH SAMPLE IS PRINTED ON A SEPARATE LINE.  IN THE LEFT MARGIN,
... THE INDEX IS PRINTED.  NEXT, IF THE EMULATOR COMPARE CIRCUIT WAS
... TRUE WHEN THE SAMPLE WAS TAKEN, *ECMP IS PRINTED.  NEXT THE LOW-
... ORDER 16 BITS OF THE SAMPLE ARE PRINTED IN ONE OF THREE FORMATS.
... IF THE INSTRUCTION ACQUISITION LINE WAS TRUE WHEN THE SAMPLE WAS
... TAKEN, THEN THE SAMPLE IS DISASSEMBLED AS AN INSTRUCTION.  IF
... NOT, THEN IF THE DATA BUS IN LINE WAS TRUE THE WORD "READ" IS
... PRINTED, OTHERWISE THE WORD "WRITE", AND FINALLY THE SAMPLE IS
... PRINTED IN HEXADECIMAL AND AS TWO ASCII CHARACTERS.  IF THE
... TRACE MODULE WAS SET TO TRACE 8 BIT VALUES ON THE 9980 EMULATOR
... DATA BUS, THEN THE INSTRUCTION INTERPRETATION OF A SAMPLE
... ABOVE WILL BE MEANINGLESS.
...
'DEFINING TDUMP' NL
...
PROC TDUMP(0,2) BEGIN
  IF ARG 0 EQ 0
       THEN LOC 2 = TTBO
       ELSE LOC 2 = ARG 1
  IF ARG 0 GE 2
       THEN LOC 1 = ARG 2
       ELSE LOC 1 = LOC 2 +19;
  IF LOC 1 GT TTBN
       THEN LOC 1 = TTBN;
  'TRA INDEX   ECMP?   DATA(HEX,ASCII)    INTERPRETATION'; NL;
  WHILE LOC 2 LE LOC 1 DO'BEGIN
       LOC 2:ND;
       IF TTBH(LOC 2,EMU)
            THEN '     *ECMP    '
            ELSE '              ';
```

Figure H-5. Listing of DUMPS Procedure File (Sheet 2 of 5)

```
        TTB(LOC 2):HA;
        IF TTBH(LOC 2,IAQ)
             THEN TTB(LOC 2):X7I
             ELSE BEGIN
                  IF TTBH(LOC 2,DBIN)
                       THEN '            READ  '
                       ELSE '            WRITE ';
                  TTB(LOC 2):H
             END;
        LOC 2 = LOC 2 + 1
   END
END
...TITLE:          TEDUMP:  DUMP EMULATOR AND TRACE BUFFERS TOGETHER
...REVISION:       03/28/77
...                09/09/77        9980 UPDATES
...
...ABSTRACT:
...     TEDUMP PRINTS THE CONTENTS OF THE EMULATOR AND TRACE MODULE
...TRACE BUFFERS TOGETHER.  TEDUMP ASSUMES THAT THE EMULATOR HAS SAMPLED
...ADDRESSES AND THE TRACE MODULE DATA, AND THAT THE QUALIFIERS FOR THE
...TWO TRACINGS WERE THE SAME.  THIS IMPLIES THAT ON EVERY QUALIFIED
...MEMORY CYCLE BOTH THE EMULATOR AND THE TRACE MODULE SAMPLED SIMUL-
...TANEOUSLY.  THUS THE SAMPLES IN THE TWO BUFFERS ARE SYNCHRONIZED,
...AND MAY BE CONSIDERED TOGETHER MEANINGFULLY.  THERE ARE VARIOUS
...WAYS TO ACHIEVE THIS, ONE OF WHICH IS TO TRACE ALL MEMORY CYCLES
...IN BOTH BUFFERS.
...
...USAGE:
...     TEDUMP MAY ONLY BE CALLED WHEN THE TRACE MODULE AND THE EMULATOR
...ARE BOTH HALTED.
...     BEFORE TRACING, THE USER SHOULD ENSURE THAT THE DATA CABLE AND
...THE CONTROL CABLE ARE BOTH CONNECTED.  HE SHOULD SET UP TRACING IN
...EACH MODULE SO THAT BOTH SAMPLE SIMULTANEOUSLY.  THE EASIEST WAY TO
...ACCOMPLISH THIS IS TO USE THE EXTERNAL CLOCK SELECTION IN BOTH ETRC
...AND TTRC.  FURTHER, THE EMULATOR SHOULD TRACE ADDR AND THE TRACE
...MODULE SHOULD TRACE DATA+X, WHERE X IS ANY DESIRED QUALIFICATION.
...THE TRACE MODULE SHOULD NOT HALT ITSELF, BUT MAY HALT THE EMULATOR
...AND THEN HALT AS A RESULT OF THE EMULATOR HALTING.  WHEN USING
...THE 9980 EMULATOR, THE QUALIFIER Q0 SHOULD BE USED FORCE FULL
...16 BIT (RATHER THAT 8 BIT ) DATA TRACES TO OCCUR IN THE TRACE MODULE.
...     THE FOLLOWING SETUP WILL TRACE THE EXECUTION OF 10 INST-
...RUCTIONS IN THE EMULATOR.  THE EMULATOR AND TRACE COMMANDS NOT
...MENTIONED (ECMP,EEVT,TCMP,TEVT) MAY BE SET HOWEVER THE USER WISHES.
...THE SETUP IS CORRECT FOR TRACING 16 BIT DATA VALUES WHEN USING
...THE 9900 EMULATOR OR 8 BIT DATA VALUES IN THE 9980 EMULATOR.  THE
...QUALIFIER "-Q0" SHOULD BE USED ON THE "DATA" PARAMETER OF THE
...TTRC COMMAND IF THE USER WISHES TO TRACE 16 BIT DATA VALUES
...FETCHED BY THE 9980.  PLEASE REFER TO THE APPROPRIATE SECTION OF
...THE MANUAL FOR A DISCUSSION OF TRACING WITH THE 9980 EMULATOR.
...
...          ETRC(ADDR,10,EXT)
...          TTRC(DATA,256,EXT)
...          EBRK(FULL,SELF)
...          TBRK(OFF,OFF)
...
...THE TRACE MODULE MAY BREAK ON EITHER FULL OR EVT, BUT MUST CONTINUE
...TRACING UNTIL THE EMULATOR HAS STOPPED TRACING.
...     AFTER THE EMULATOR HAS HALTED, HALTING THE TRACE MODULE WITH
...IT VIA THE CONTROL CABLE, THE TWO BUFFERS CONTAIN THE SAME NUMBER
...OF SAMPLES, TAKEN AT THE SAME TIMES.  TEDUMP MAY BE USED TO DUMP
...THEM IN PARALLEL, SHOWING BOTH THE ADDRESS AND DATA LINES OF THE
...EMULATOR'S MEMORY BUS.
```

Figure H-5. Listing of DUMPS Procedure File (Sheet 3 of 5)

```
...     TEDUMP HAS TWO ARGUMENTS:  START AND END.  THE RANGE OF INDICES
...START...END IS USED TO DISPLAY SAMPLES FROM THE EMULATOR BUFFER.
...IF END IS OMITTED, START PLUS 19 SAMPLES ARE DISPLAYED GIVING
...A 20 SAMPLE DEFAULT DISPLAY.
...CORRESPONDING SAMPLES ARE TAKEN FROM THE TRACE MODULE BUFFER BUT
...WITH POSSIBLY DIFFERENT INDICES, IF TCMP AND TEVT HAVE BEEN SET TO
...COUNT EVENTS AND DELAYS.  THESE ARE DISPLAYED IN THE APPROPRIATE
...FORMAT ACCORDING TO THE CONTROL SIGNALS TRACED IN THE HIGH FOUR
...BITS OF EACH TRACE BUFFER WORD.  REMEMBER THAT IF A 9980
...EMULATOR IS BEING USED, THE QUALIFIER Q0 DETERMINES WHETHER
...(A) BYTE ADDRESSES AND DATA ARE BEING TRACED OR (B) ODD WORD
...ADDRESSES AND 16 BIT DATA VALUES ARE BEING TRACED.  REFER
...TO THE APPROPRIATE SECTION OF THE MANUAL FOR DISCUSSION
...OF TRACING WITH THE 9980 EMULATOR.
...
'DEFINING TEDUMP' NL
...
PROC TEDUMP(0,4) BEGIN
 'EMU INDEX  ECMP?  ADDR  TRA INDEX  DATA(HEX,ASC)  INTERPRETATION';
 NL;
  IF ARG 0 EQ 0
       THEN LOC 1 = ETBO;
       ELSE IF ARG 1 LT ETBO
            THEN LOC 1 = ETBO
            ELSE LOC 1 = ARG 1;
  IF ARG 0 GE 2
       THEN LOC 3 = ARG 2                    ...END WAS SUPPLIED
       ELSE LOC 3 = LOC 1 + 19;              ...END WAS NOT SU PLIED
  LOC 4 = TTBN - (TTBO-ETBO)
  IF ETBN GT LOC 4
       THEN LOC 4 = ETBN;
  IF LOC 3 GT LOC 4
       THEN LOC 3 = LOC 4;
  LOC 2 = TTBO + LOC 1 - ETBO;               ...INITIAL TRACE BUFFER INDEX
  WHILE LOC 1 LE LOC 3
       DO BEGIN
          IF LOC 2 GE TTBO AND LOC 2 LE TTBN
            THEN LOC 4 = -1 ELSE LOC 4 = 0;
          IF LOC 1 GE ETBO AND LOC 1 LE ETBN
          THEN BEGIN
               LOC 1:ND;                        ...PRINT INDEX IN ETB
               IF LOC 4 EQ 0 THEN '  --N/A-- '
                   ELSE IF TTBH(LOC 2,EMU)
                        THEN '   *ECMP  '  ...EMULATOR COMPARE TRUE
                        ELSE '          ';
               ETB(LOC 1):H;                      ...ADDRESS
               END;
          IF LOC 4 NE 0 THEN
            BEGIN
            LOC 2:D4;                       ...PRINT THE INDEX IN TTB
            TTB(LOC 2):X4HA;                     ...PRINT VALUE IN HEX
            IF TTBH(LOC 2,IAQ)
                 THEN TTB(LOC 2):X4I          ...DISPLAY THE INSTRUCTION
                 ELSE BEGIN                   ...ITS A DATA ACCESS
                             IF TTBH(LOC 2,DBIN)
                          THEN 'READ  '     ...IT'S A READ
                          ELSE 'WRITE '; ...IT'S A WRITE
                      TTB(LOC 2):H;         ...DISPLAY THE DATA
                 END;
            END;
            LOC 2 = LOC 2 + 1;                ...STEP TTB INDEX
            LOC 1 = LOC 1 + 1;                ...STEP ETB INDEX
```

Figure H-5. Listing of DUMPS Procedure File (Sheet 4 of 5)

```
        END
END
```

Figure H-5. Listing of DUMPS Procedure File (Sheet 5 of 5)

```
...TITLE:          FNDBYT:  SEARCH MEMORY FOR A SPECIFIED SUBBYTE
...REVISION:       06/07/77
...
...ABSTRACT:
...     FNDBYT SEARCHES A SPECIFIED RANGE OF MEMORY IN THE TARGET
...SYSTEM TO FIND A BYTE CONTAINING A SPECIFIED VALUE IN A SPECIFIED
...FIELD.
...
...USAGE:
...     FNDBYT CAN BE CALLED WHENEVER THE EMULATOR IS HALTED.
...FNDBYT CAN HAVE ONE TO FOUR ARGUMENTS.  THE POSSIBLE FORMS ARE
...GIVEN BELOW:
...          FNDBYT(START, END, VALUE, MASK)
...          FNDBYT(START, END, VALUE)
...          FNDBYT(VALUE, MASK)
...          FNDBYT(VALUE)
...THE ARGUMENTS, THEIR MEANINGS, AND THEIR DEFAULT VALUES ARE
...DESCRIBED BELOW.
...          NAME      MEANING                                  DEFAULT
...          ------    ---------------------------------        -------
...          START     LOW END OF ADDRESS RANGE TO SEARCH       0
...          END       HIGH END OF ADDRESS RANGE                >FFFF
...          VALUE     INTERESTING FIELD TO SEARCH FOR          -
...          MASK      ONE BITS IN INTERESTING POSTITIONS       >00FF
...
...     THE RANGE OF BYTES ADDRESSES START...END IS SEARCHED FOR THE
...FIRST OCCURRENCE OF VALUE IN A BYTE.  EACH BYTE IS MASKED WITH MASK
...AND THEN COMPARED TO VALUE, SO THE INTERESTING FIELD MUST BE IN THE
...CORRECT POSITION IN VALUE.  IF SUCH AN OCCURRENCE IS FOUND, FNDBYT
...RETURNS THE ADDRESS OF THE BYTE.  IF NONE IS FOUND, FNDBYT RETURNS
...>FFFF.  THIS UNFORTUNATE AMBIGUITY CANNOT BE RESOLVED SIMPLY, SO
...IT IS LEFT TO THE USER OF FNDBYT TO DETERMINE WHETHER >FFFF IS IN
...FACT A VALID ANSWER OR JUST A FAILURE FLAG.
...
'DEFINING FNDBYT'
...
FUNC FNDBYT(1,8) BEGIN
  IF ARG 0 GE 3
       THEN BEGIN                              ...RANGE WAS SPECIFIED
            LOC 1 = ARG 1;                     ...START
            LOC 2 = ARG 2;                     ...END
            LOC 3 = ARG 3;                     ...VALUE
       END ELSE BEGIN                          ...RANGE WAS NOT SPECIFIED
            LOC 1 = 0;
            LOC 2 = >FFFF;
            LOC 3 = ARG 1;
       END;
  IF ARG 0 MOD 2
       THEN LOC 4 = >FF                        ...MASK WAS NOT SPECIFIED
       ELSE LOC 4 = ARG ARG 0;                 ...MASK WAS SPECIFIED
  LOC 5 = MPY(LOC 3,256);                      ...HIGH BYTE VERSION OF VALUE
  LOC 6 = MPY(LOC 4,256);                      ...HIGH BYTE VERSION OF MASK
  LOC 7 = LOC 1;                               ...SAVE COPY OF START
  REPEAT BEGIN
       LOC 8 = @LOC 1;                         ...FETCH A WORD
       IF (LOC 8 AND LOC 6) EQ LOC 5
            THEN RETURN LOC 1;                 ...FOUND IT IN HIGH BYTE
       IF (LOC 8 AND LOC 4) EQ LOC 3
            THEN RETURN LOC 1 + 1;             ...FOUND IT IN LOW BYTE
       LOC 1 = LOC 1 + 2;                      ...STEP TO NEXT WORD
  END UNTIL LOC 1 HI LOC 2 OR LOC 1 LOE LOC 7;
  RETURN -1


  END
```

Figure H-6. Listing of FNDBYT Procedure File

```
...TITLE:          FNDWRD:  SEARCH TARGET MEMORY FOR A SUBWORD
...REVISION:       03/21/77
...
...ABSTRACT:
...     FNDWRD SEARCHES A SPECIFIED RANGE OF TARGET MEMORY FOR A WORD
...CONTAINING A SPECIFIED VALUE IN A SPECIFIED FIELD.   IT RETURNS THE
...ADDRESS OF THE FIRST SUCH WORD FOUND IN THAT RANGE.
...
...USAGE:
...     FNDWRD MAY BE CALLED WHENEVER THE EMULATOR IS HALTED.   IT MAY
...HAVE AS MANY AS FOUR ARGUMENTS, OR AS FEW AS ONE.   EACH FORMAT
...IS LISTED BELOW.   THE RANGE IS INCLUSIVE.
...             FNDWRD(START, END, VALUE, MASK)
...             FNDWRD(START, END, VALUE)
...-            FNDWRD(VALUE, MASK)
...             FNDWRD(VALUE)
...IN EACH FORM, THE ARGUMENTS AND THEIR DEFAULT VALUES ARE AS FOLLOWS:
...             NAME        MEANING                              DEFAULT
...             -----       -------------------------------      -------
...             START       LOW END OF ADDRESS RANGE             0
...             END         HIGH END OF ADDRESS RANGE            0FFFE
...             VALUE       THE INTERESTING VALUE, IN ITS        -
...                         PROPER FIELD
...             MASK        THE INTERESTING FIELD HAS ONE        0FFFF
...                         BITS IN IT
...
'DEFINING FNDWRD'
...
FUNC FNDWRD(1,5) BEGIN
  IF ARG 0 GE 3                          ...WAS RANGE GIVEN?
       THEN BEGIN                        ...RANGE WAS GIVEN.
            LOC 3 = ARG 3;               ...VALUE ARGUMENT IS THIRD
            LOC 1 = ARG 1;               ...LOW END OF RANGE
            LOC 2 = ARG 2;               ...HIGH END OF RANGE
       END ELSE BEGIN                    ...RANGE WAS NOT GIVEN
            LOC 3 = ARG 1;               ...VALUE ARGUMENT IS FIRST
            LOC 1 = 0;                   ...USE RANGE 0...
            LOC 2 = 0FFFE;               ...    0FFFE
       END;
  IF ARG 0 MOD 2                         ...WAS MASK GIVEN?
       THEN LOC 4 = -1                   ...NO, USE WHOLE WORD
       ELSE LOC 4 = ARG ARG 0;           ...YES, USE SPECIFIED MASK
  LOC 5 = LOC 1;                         ...SAVE LOW END FOR CHECKING
  REPEAT BEGIN
       IF (@LOC 1 AND LOC 4) EQ (LOC 3 AND LOC 4)
            THEN RETURN LOC 1;
       LOC 1 = LOC 1 + 2;
  END UNTIL LOC 1 HI LOC 2 OR LOC 1 LOE LOC 5;
END
```

Figure H-7. Listing of FNDWRD Procedure File

```
... TITLE:          HIST:      GENERAL PURPOSE HISTOGRAM ROUTINE
... REVISION:       04/06/77
...
... ABSTRACT:
...     ONE PROCEDURE DEFINITION IS CONTAINED IN THIS FILE.
... IN ADDITION AN ARRAY (HISTAR) IS DEFINED AND THE USER
... MUST PROVIDE A FUNCTION (SAMPLE) AND ONE USER VARIABLE (S).
... THE HIST ROUTINE COLLECTS SAMPLES TO EVENTUALLY PLOT A
... HISTOGRAM OF FREQUENCY OF SAMPLES OVER A SPECIFIED RANGE.
...
... USAGE:
...     HIST IS CALLED WITH TWO ARGUMENTS WHICH DEFINE THE
... RANGE OF SAMPLES OVERWHICH TO BUILD THE HISTOGRAM.  FOR
... EXAMPLE 'HIST(>1000,>2000)' WILL CONSTRUCT A HISTOGRAM
... OF SAMPLES BETWEEN THE RANGE OF >1000 TO >2000 (SAMPLES
... OUTSIDE OF THIS RANGE ARE IGNORED).  IF NO ARGUMENTS ARE
... SPECIFIED THE RANGE DEFAULTS TO >0000 TO >FFFF.  'SAMPLES'
... ARE OBTAINED THROUGH A FUNCTION WHICH MUST ALREADY BE
... DEFINED AT THE TIME HIST IS COPIED IN.  IN ADDITION THE USER
... SYMBOL 'S' MUST BE DEFINED.  AS LONG AS 'S' IS NON ZERO
... SAMPLES WILL BE TAKEN BY THE 'HIST' ROUTINE.  FOR EXAMPLE:
...     PROC SAMPLE BEGIN
...             ... STUFF TO OBTAIN THE SAMPLE VALUE
...         IF <CONDITION> THEN S=0  ...SIGNAL TO STOP TAKING SAMPLES
...         RETURN <SAMPLE VALUE>
...         END
...
'DEFINING HISTAR - HISTOGRAM ARRAY'
DELE ('HISTAR')        ..DELETE ANY EXISTING HISTAR ARRAY
ARRAY HISTAR(64)       ..DEFINE HISTOGRAM ARRAY
...
'DEFINING HIST(RANGE) - HISTOGRAM ROUTINE'
PROC HIST (0,7) BEGIN
......HISTOGRAM OF SAMPLES OVER A GIVEN RANGE
...
...     NOTE :: TWO ADDITIONAL ITEMS ARE NECESSARY TO USE
...         THIS PROCEDURE: 1) A VARIABLE 'S' - AS LONG AS
...         THIS VARIABLE IS NON ZERO SAMPLES ARE TAKEN.
...         2) A FUNCTION 'SAMPLE' WHICH RETURNS A SAMPLE VALUE.
...
... ARGUMENT 1 IS THE LOWER LIMIT OF THE VALUE.  (0)
... ARGUMENT 2 IS THE UPPER LIMIT OF THE VALUE.  (>FFFF)
... LOC 1 IS THE LOW END OF THE RANGE.
... LOC 2 IS THE HIGH END OF THE RANGE.
... LOC 3 IS THE BUCKET WIDTH
...
...  DETERMINE THE DEFAULT VALUES IF NECESSARY
...
IF ARG 0 GE 1 THEN LOC 1 = ARG 1 ELSE LOC 1 = 0;
IF ARG 0 GE 2 THEN LOC 2 = ARG 2 ELSE LOC 2 = 0FFFF;
...
... ZERO THE HISTOGRAM ARRAY (LOC 10...LOC 75)
...
FOR LOC 4 = 1 TO 64 DO HISTAR(LOC 4) = 0
...
...  DETERMINE THE BUCKED WIDTH
...
MDR = 0
LOC 3 = DIV(64,LOC 2 - LOC 1 + 63)  ...BUCKET WIDTH
...
'THE BUCKET WIDTH IS '; LOC 3:HN;
...
```

Figure H-8. Listing of HIST Procedure File (Sheet 1 of 2)

```
... COLLECT SAMPLES
...
WHILE S DO BEGIN
      LOC 5 = SAMPLE;                    ..GATHER A SAMPLE
      IF (LOC 5 HIE LOC 1) AND (LOC 5 LO LOC 2) THEN BEGIN
          LOC 6 = LOC 5 - LOC 1
          MDR = 0
          LOC 5 = 1+DIV(LOC 3,LOC 6)            ..COMPUTE BUCKET INDEX
          HISTAR(LOC 5)=HISTAR(LOC 5) + 1;      ..INCREMENT COUNT
          END;
       END;
...
... PRINT OUT THE VALUES IN THE HISTOGRAM ARRAY
...
FOR LOC 4 = 1 TO 64 DO BEGIN
       IF LOC 4 MOD 8 EQ 0 THEN NL;
       HISTAR(LOC 4):D;
       END; NL;
...
... FIND THE MAXIMUM VALUE IN HISTOGRAM ARRAY
...
LOC 4 = HISTAR(1)                  ..ASSUMED MAXIMUM
FOR LOC 5 = 1 TO 64 DO                        ...ARRAY INDEX
       IF HISTAR(LOC 5) GT LOC 4 THEN LOC 4 = HISTAR(LOC 5);
...
'THE MAXIMUM IS '; LOC 4:D; NL;
IF LOC 4 EQ 0 THEN BEGIN 'QUITTING'; RETURN; END;
...
... FIND THE SCALING FACTOR
...
LOC 4 = LOC 4 +7
MDR=0
LOC 4 = DIV(8,LOC 4);
...
'THE SCALE IS '; LOC 4:D;.... ' TYPE A NUMBER TO CONTINUE';
...
LOC 5=8;                                      .. INITIALIZE LINE COUNT
WHILE LOC 5 GE 1 DO BEGIN                     ...LOOP TO SCAN A PAGE
  NL;
  LOC 6=LOC 5*LOC 4
  FOR LOC 7 = 1 TO 64 DO          ..SCAN A LINE
    IF HISTAR(LOC 7) GE LOC 6 THEN '*'; ELSE ' ';
  LOC 5=LOC 5-1;
  END;
...
... PRINT HISTOGRAM 'HEADER'
...
NL;
'+---------------+---------------+---------------+---------------+';
NL;
MDR=0
LOC 4=DIV(4,LOC 2 - LOC 1 + 1);
LOC 1:HX7;LOC 1+LOC 4:HX9;       ...LABEL AXIS
LOC 1+(2*LOC 4):HX9;LOC 2-LOC 4:HX9;
LOC 2:HN;
'                  HISTOGRAM OF SAMPLES OVER A RANGE'; ...PRINT TITLE
END
```

Figure H-8. Listing of HIST Procedure File (Sheet 2 of 2)

```
...TITLE:          BL:  SIMULATE A BRANCH AND LINK INSTRUCTION
...REVISION:       05/02/77
...
...ABSTRACT:
...     BL SIMULATES THE EXECUTION OF A BL INSTRUCTION IN THE
...TARGET SYSTEM.  BL MAY BE CALLED WHENEVER THE EMULATOR IS HALTED.
...
...USAGE:
...     BL HAS ONE ARGUMENT, THE ADDRESS TO TRANSFER IN THE TARGET
...SYSTEM.  THE CURRENT PC IS STORED INTO REGISTER ELEVEN (R11)
...AND THE PC CHANGED TO THE ADDRESS TO TRANSFER TO.  THE CURRENT
...WORKSPACE POINTER IS NOT CHANGED.    NOTE THAT THE PC IS NOT
...INCREMENTED BEFORE IT IS STORED.
...
'DEFINING BL' NL
...
PROC BL(1) BEGIN
  R11 = PC;                                  ...THE OLD PC
  PC = ARG 1                                 ...NEW PC
END
...TITLE:          BLWP:  SIMULATE A BLWP INSTRUCTION
...REVISION:       03/21/77
...
...ABSTRACT:
...     BLWP SIMULATES THE EXECUTION OF A BLWP INSTRUCTION IN THE
...TARGET SYSTEM.  BLWP MAY BE CALLED WHENEVER THE EMULATOR IS HALTED.
...
...USAGE:
...     BLWP HAS ONE ARGUMENT, THE ADDRESS OF A TRANSFER VECTOR IN
...TARGET MEMORY.  THE FIRST WORD OF THE TRANSFER VECTOR IS LOADED INTO
...THE WORKSPACE POINTER AND THE SECOND WORD INTO THE PROGRAM COUNTER.
...THE PREVIOUS WP, PC, AND ST ARE STORED IN R13, R14, AND R15, RESPEC-
...TIVELY, OF THE NEW WORKSPACE.  NOTE THAT THE PC IS NOT INCREMENTED
...BEFORE IT IS STORED.
...
'DEFINING BLWP' NL
...
PROC BLWP(1,1) BEGIN
  LOC 1 = WP;                                ...THE CURRENT WP
  WP = @ARG 1;                               ...THE NEW WP
  R13 = LOC 1;                               ...THE OLD WP
  R14 = PC;                                  ...THE OLD PC
  R15 = ST;                                  ...THE OLD ST
  PC = @(ARG 1 + 2);                         ...THE NEW`PC
END
...TITLE:          INTR:  SIMULATE AN INTERRUPT IN TARGET SYSTEM
...REVISION:       03/21/77
...
...ABSTRACT:
...     INTR SIMULATES THE OCCURRENCE OF AN INTERRUPT ON A PARTICULAR
...INTERRUPT PRIORITY LEVEL.  INTR FUNCTIONS THE SAME WAY THAT THE
...INTERRUPT ARBITRATION HARDWARE DOES.  THAT IS, THE INTERRUPT IS ONLY
...HONORED IF THE CPU'S PRIORITY IS HIGHER THAN THE LEVEL ON WHICH THE
...INTERRUPT IS RAISED.  IF THE PRIORITY ALLOWS THE INTERRUPT (DETER-
...MINED BY CHECKING ST) TO OCCUR, THEN A BLWP TRANSFER IS PERFORMED
...THROUGH THE VECTOR STORED AT THE APPROPRIATE INTERRUPT LEVEL.
...ALSO, THE PRIORITY OF THE CPU IS SET TO THE LEVEL OF THE INTERRUPT
...MINUS ONE.
...     INTR HAS ONE ARGUMENT, THE LEVEL OF THE INTERRUPT REQUEST.
...INTR MAY BE CALLED WHENEVER THE EMULATOR IS HALTED.  INTR RETURNS
...0 IF THE INTERRUPT WAS NOT ALLOWED, AND -1 IF IT WAS.
...
```

Figure H-9. Listing of INSTR Procedure File (Sheet 1 of 3)

```
'DEFINING FUNCTION INTR' NL
...
FUNC INTR(1,12) BEGIN
  IF (ST AND 15) GE ARG 1
      THEN BEGIN                          ...INTERRUPT IS ALLOWED
           LOC 1 = WP;                    ...SAVE OLD WORKSPACE POINTER
           WP = @(ARG 1 * 4);             ...NEW WP
           R13 = LOC 1;                   ...OLD WP
           R14 = PC;                      ...OLD PC
           R15 = ST;                      ...OLD ST
           PC = @(ARG 1 * 4  +  2);       ...NEW PC
           LOC 1 = ARG 1 - 1;             ...NEW PRIORITY
           IF LOC 1 LT 0
                THEN LOC 1 = 0;           ...PATCH FOR LEVEL ZERO
           ST = (ST AND 0FFF0) + LOC 1;   ...NEW ST
           RETURN -1;                     ...INTERRUPT SUCCEEDED
      END ELSE
           RETURN 0;                      ...INTERRUPT FAILED
END
...TITLE:        RT:  SIMULATE AN RT INSTRUCTION IN TARGET SYSTEM
...REVISION:     05/02/77
...
...ABSTRACT:
...     RT SIMULATES THE EXECUTION OF AN RT INSTRUCTION IN THE TARGET
...SYSTEM.  THE CURRENT PROGRAM COUNTER (PC) IS CHANGED TO THE CONTENTS
...OF REGISTER ELEVEN (THE RETURN ADDRESS).
...
...USAGE:
...     RT HAS NO ARGUMENTS.  RT MAY BE CALLED WHENEVER THE EMULATOR
...IS HALTED.
...
'DEFINING RT' NL
...
PROC RT(0) BEGIN
  PC = R11;
END
...TITLE:        RTWP:  SIMULATE A RTWP INSTRUCTION IN TARGET SYSTEM
...REVISION:     03/21/77
...
...ABSTRACT:
...     RTWP SIMULATES THE EXECUTION OF AN RTWP INSTRUCTION IN THE TARGET
...SYSTEM.  WP, PC, AND ST ARE LOADED FROM R13, R14, AND R15, RESPEC-
...TIVELY.
...
...USAGE:
...     RTWP HAS NO ARGUMENTS.  RTWP MAY BE CALLED WHENEVER THE EMULATOR
...IS HALTED.
...
'DEFINING RTWP' NL
...
PROC RTWP(0) BEGIN
  ST = R15;
  PC = R14;
  WP = R13;
END
...TITLE:        XOP:  SIMULATE AN XOP IN TARGET SYSTEM
...REVISION:     05/02/77
...
...ABSTRACT:
...     XOP SIMULATES AN EXTENDED OPERATION INSTRUCTION IN THE
...TARGET SYSTEM.  TWO ARGUMENTS ARE EXPECTED:
...     1) THE 'VALUE' TO PASS TO THE XOP HANDLER ROUTINE.
```

Figure H-9. Listing of INSTR Procedure File (Sheet 2 of 3)

```
...     2) THE XOP LEVEL NUMBER.
...A BLWP OPERATION IS PERFORMED TO TRANSFER THROUGH THE
...APPROPRIATE XOP LEVEL.  REGISTER ELEVEN (R11) WILL CONTAIN
...THE 'VALUE' AS SPECIFIED BY THE USER.
...
'DEFINING XOP' NL
...
PROC XOP(2,1) BEGIN
       LOC 1 = WP;                      ...SAVE OLD WORKSPACE POINTER
       WP = @(ARG 2 * 4 + >40);         ...NEW WP
       R13 = LOC 1;                     ...OLD WP
       R14 = PC;                        ...OLD PC
       R15 = ST;                        ...OLD ST
       PC = @(ARG 2 * 4 + >42);         ...NEW PC
       R11 = ARG 1                      ...VALUE TO PASS IN REGISTER 11
       ST = ST OR <0000001000000000     ...'XOP' STATUS BIT
END
```

Figure H-9. Listing of INSTR Procedure File (Sheet 3 of 3)

```
... TITLE:          SOFTWARE BREAKPOINTS
... REVISION:       07/15/77
...
... ABSTRACT:
...     THIS MODULE DEFINES USER PROCEDURES TO IMPLEMENT
... A MULTIPLE BREAKPOINT SOFTWARE BREAKPOINT CAPABILITY.
...
... USAGE:
...     THREE (3) PROCEDURES ARE AVAILABLE:
...
...                 SB      - SET SOFTWARE BREAKPOINT
...                 CB      - CLEAR SOFTWARE BREAKPOINT
...                 EB      - EXECUTE FROM SOFTWARE BREAKPOINT
...
... EACH OF THESE ARE DESCRIBED INDIVIDUALLY BELOW.
...
...     SOFTWARE BREAKPOINTS ARE IMPLEMENTED IN THE
... TARGET SYSTEM AS SPIN INSTRUCTIONS.  THE SPINS ARE PLACED
... IN MEMORY USING THE 'SB' PROCEDURE.  EXAMINATION OF TARGET
... SYSTEM MEMORY AT SOFTWARE BREAKPOINT LOCATIONS
... WILL REVEAL SPIN INSTRUCTIONS ( >10FF / JMP $ ).
... THE 'CB' PROCEDURE IS USED TO CLEAR A SOFTWARE BREAKPOINT.
... THE EXECUTE FROM SOFTWARE BREAKPOINT ('EB') PROCEDURE STARTS THE
... TARGET SYSTEM EXECUTING AND WILL HALT THE TARGET SYSTEM
... AND RETURN TO THE USER WHEN A SOFTWARE BREAKPOINT IS HIT.
... THE SOFTWARE BREAKPOINT PROCEDURES USE TWO ARRAYS DEFINED
... IN THE USER SYMBOL TABLE.  THESE ARRAYS SHOULD NOT BE
... USED WHEN USING THE SOFTWARE BREAKPOINT PROCEDURES.
...                 SBA(8)
...                 SBI(8)
...
.....................................................................
...
NL
'DEFINING: SOFTWARE BREAKPOINT PROCEDURES'
...
.....................................................................
...
...  DEFINE THE TWO ARRAYS TO BE USED:
ARRAY SBA(8)      ...SOFTWARE BREAKPOINT 'ADDRESS' ARRAY
ARRAY SBI(8)      ...SOFTWARE BREAKPOINT 'INSTRUCTION' ARRAY
...
...  INITIALIZE THE ADDRESS ARRAY TO 'NO BREAKPOINTS SET'
PROC INIT (0,1) FOR LOC 1 = 1 TO 8 DO SBA(LOC 1) = >FFFF
INIT            ..INITIALIZE THE ADDRESS ARRAY
DELE ('INIT') ..DELETE THIS TEMPORARY PROCEDURE
...
.....................................................................
...
NL
'DEFINING: SB - SET BREAKPOINT'
...
... ABSTRACT: SET A BREAKPOINT OR INSPECT ALL OF THE CURRENT
...           BREAKPOINTS.
... INTERFACE:SYNTAX:   SB [(<ADDRESS> [,<ADDRESS>]...)]
...            IF THE ARGUMENT IS NOT SUPPLIED THEN SB
...           LIST THE ADDRESS OF EACH SOFTWARE
...           BREAKPOINT AND THE INSTRUCTION AT THAT ADDRESS.
...           IF ARGUMENTS (<ADDRESS>) ARE INCLUDED, THEN
...           A SOFTWARE BREAKPOINT IS SET AT EACH ADDRESS.
... DIAGNOSTICS: 'BREAKPOINT SET AT HHHH' IS PRINTED FOR
...            EACH BREAKPOINT SET.  IF THIS MESSAGE IS NOT
```

Figure H-10. Listing of SB Procedure File (Sheet 1 of 4)

```
...          PRINTED THEN THE BREAKPOINT TABLE IS FULL AND
...          THE BREAKPOINT WAS NOT SET.
...
.....................................................................
..
PROC SB (0,3) BEGIN ...VARIABLE ARGUMENTS, THREE LOCAL VARIABLES;
            IF ARG 0 EQ 0 THEN BEGIN
..           PRINT OUT ALL CURRENTLY SET BREAKPOINTS
                FOR LOC 1 = 1 TO 8 DO BEGIN
                    LOC 2 = SBA(LOC 1);
                    IF LOC 2 NE >FFFF THEN BEGIN
                        @LOC 2 = SBI(LOC 1) ..RESTORE THE INSTRUCTION
                        LOC 2:NS;'/ ';@LOC 2:HI
                        @LOC 2 = >10FF      ..RESTORE THE SPIN
                        END;
                    END;
                END;
            ELSE FOR LOC 2 = 1 TO ARG 0 DO BEGIN
..          SET SOFTWARE BREAKPOINT
                FOR LOC 1 = 1 TO 8 DO
                    IF SBA(LOC 1) EQ >FFFF THEN BEGIN
                        LOC 3 = ARG (LOC 2) ..ADDRESS TO BREAKPOINT
                        SBA(LOC 1) = LOC 3  ..SAVE THE ADDRESS
                        SBI(LOC 1) = @LOC 3 ..SAVE THE INSTRUCTION
                        @LOC 3 = >10FF      ..SET THE SPIN
                        NL;'BREAKPOINT SET AT ';LOC 3:S
                        ESCAPE
                        END;
                END;
        END

.....................................................................
..
NL
'DEFINING: CB - CLEAR SOFTWARE BREAKPOINT'
..
..ABSTRACT: CLEAR GIVEN OR ALL SOFTWARE BREAKPOINTS.
..INTERFACE: SYNTAX:  CB [(<ADDRESS> [,<ADDRESS>]...)]
..          IF <ADDRESS> IS INCLUDED THEN CLEAR THE
..          BREAKPOINT AT THAT ADDRESS.  IF THER IS NO
...         ARGUMENT THEN CLEAR ALL OF THE BREAKPOINTS.
...DIAGNOSTICS: 'BREAKPOINT CLEARED AT HHHH' MESSAGE WILL
...         BE PRINTED FOR EACH BREAKPOINT CLEARED.  IF THIS
...         MESSAGE IS NOT PRINTED A SOFTWARE BREAKPOINT
...         WAS NOT FOUND AT THE GIVEN ADDRESS.
...
.....................................................................
..
PROC CB (0,2) BEGIN ..VARIABLE ARGUMENTS, TWO LOCAL VARIABLE;
            IF ARG 0 EQ 0 THEN BEGIN
..          CLEAR ALL SOFTWARE BREAKPOINTS
                NL;'CLEARING ALL BREAKPOINTS'
                FOR LOC 1 = 1 TO 8 DO BEGIN
                    IF SBA(LOC 1) NE >FFFF THEN BEGIN
                        @SBA(LOC 1) = SBI(LOC 1)
                        SBA(LOC 1) = >FFFF
                        END
                    END,
                END;
            ELSE FOR LOC 2 = 1 TO ARG 0 DO BEGIN
..          CLEAR SPECIFIED SOFTWARE BREAKPOINT(S)
                FOR LOC 1 = 1 TO 8 DO
```

Figure H-10. Listing of SB Procedure File (Sheet 2 of 4)

```
                    IF SBA(LOC 1) EQ ARG (LOC 2) THEN BEGIN
                        @SBA(LOC 1) = SBI(LOC 1)
                        SBA(LOC 1) = >FFFF . . CLEAR THE BREAKPOINT
                        NL; 'BREAKPOINT CLEARED AT ';
                        ARG(LOC 2):S
                        ESCAPE
                        END;
                END;
            END;
...
.....................................................................
...
NL
'DEFINING: EB - EXECUTE FROM SOFTWARE BREAKPOINT'
...
... ABSTRACT: EXECUTE FROM SOFTWARE BREAKPOINT
... INTERFACE:SYNTAX:  EB
...     THE EB COMMAND STARTS TARGET SYSTEM EXECUTION FROM
...     A SOFTWARE BREAKPOINT AND CONTINUES EXECUTION UP
...     TO THE NEXT SOFTWARE BREAKPOINT ENCOUNTERED.  THE
...     INSTRUCTION AT THE ENCOUNTERED SOFTWARE BREAKPOINT
...     IS NOT EXECUTED.  SOFTWARE BREAKPOINTS ARE SET BY
...     THE SB (SET SOFTWARE BREAKPOINT) COMMAND AND CLEARED
...     BY THE CB (CLEAR SOFTWARE BREAKPOINT) COMMAND.
...
.....................................................................
...
... NOTE::NOTE::NOTE::NOTE::NOTE::NOTE::NOTE::NOTE::NOTE::NOTE
...     EBRUN IS A LOCAL PROCEDURE FOR THE 'EB' PROCEDURE.
...     THIS ROUTINE DETERMINES HOW THE HOST SYSTEM STOPS THE
...     TARGET SYSTEM ONCE A SOFTWARE BREAKPOINT IS HIT.  TWO
...     IMPLEMENTATIONS ARE DEFINED BELOW - CHOOSE THE ONE
...     WHICH IS APPLICABLE TO THE AMPS HARDWARE YOU HAVE OR
...     DEFINE YOUR OWN ROUTINE.  VERSION #1 IS INITIALLY
...     COMMENTED OUT, ASSUMING THAT THE HAS A TRACE MODULE.
... NOTE::NOTE::NOTE::NOTE::NOTE::NOTE::NOTE::NOTE::NOTE::NOTE
...
...  EBRUN - VERSION #1
...     - USES ONLY THE EMULATOR MODULE
...     - PERIODICALLY STOPS THE TARGET SYSTEM TO SEE IF
...       IT IS ON A SOFTWARE BREAKPOINT.  TRACE CAPABILITY
...       OF THE EMULATOR CANNOT BE USED (WILL TRACE SPINS)
...
.. PROC EBRUN (0,2) BEGIN . . RUN THE TARGET UNTIL SOFTWARE BREAKPOINT
..       ETRC(OFF)      . . TURN OFF THE TRACE CAPABILITY
..       EBRK(OFF,OFF) . . MAKE SURE NOTHING ELSE STOPS US
..       LOC 2 = 0      . . BREAKPOINT ENCOUNTERED FLAG
..       . . PERIODICALLY STOP THE TARGET UNTIL ON A BREAKPOINT
..       REPEAT BEGIN
..            ERUN          . . START THE TARGET SYSTEM
..            WAIT(20)      . . DELAY ONE SECOND
..            EHLT          . . STOP THE TARGET SYSTEM
..            FOR LOC 1 = 1 TO 8 DO
..                IF SBA(LOC 1) EQ PC THEN LOC 2 = 1 . . SET FLAG TO TRUE
..       END UNTIL LOC 2
..       . . RETURN TO CALLER
..       END
...
...
...  EBRUN - VERSION #2
...     - USES BOTH THE EMULATOR AND LOGIC STATE TRACE MODULES
...     - THESE MODULES ARE INITIALIZED TO TRACE ALL ADDRESS
```

Figure H-10. Listing of SB Procedure File (Sheet 3 of 4)

```
...       AND DATA PLACED ON THE RESPECTIVE BUSSES OF THE TARGET.
...       (PROCEDURE TEDUMP MAY BE USED TO EXAMINE THE RESULTS)
...
PROC EBRUN (0,0) BEGIN . . EXECUTE UNTIL SOFTWARE BREAKPOINT HIT
     . . PREPARE THE AMPS HARDWARE - 'USING TRACE ANALYZER'
       TTRC(DATA+(EMT-1)*Q0,256,EXT)   . . TRACE ALL DATA IN TM
       TCMP(DATA+IAQ,>10FF,>FFFF) . . DEFINE 'SPIN' AS AN EVENT
       TEVT(1,0,NORM+EACH+INT) . . COUNT ONE EVENT, ZERO DELAYS
       TBRK(EVT,EMU)          . . HALT EMULATOR ON SATISFACTION OF EVT/DELAY
       ETRC(ADDR,256,EXT)   . . TRACE ALL ADDRESS IN THE EMULATOR MODULE
       EBRK(OFF,SELF)         . . HALT EMULATOR ON SIGNAL FROM TRACE ANALYZER
     . . START THE TARGET SYSTEM
       TRUN                   . . START THE TRACE ANALYZER MODULE
       ERUN                   . . START THE EMULATOR MODULE
     . . WAIT UNTIL EMULATOR HALTS
       WHILE EST AND 1 DO NULL
     . . RETURN TO CALLER
       END
...
.....................................................................
...
PROC EB (0,1) BEGIN    . . EXECUTE FROM SOFTWARE BREAKPOINT
     . . SEE IF THE PC IS CURRENTLY AT A SOFTWARE BREAKPOINT
       FOR LOC 1 = 1 TO 8 DO
            IF SBA(LOC 1) EQ PC THEN BEGIN
..               PC IS CURRENTLY AT A SOFTWARE BREAKPOINT
                 @SBA(LOC 1) = SBI(LOC 1)   . . RESTORE THE INSTRUCTION
..               'SINGLE STEP' THE TARGET SYSTEM PAST THIS BREAKPOINT
                      ECMP(IAQ,PC)
                      EEVT(INT)
                      EBRK(EVT,SELF)
                      ERUN
                 @SBA(LOC 1) = >10FF       . . RESTORE THE SPIN INSTRUCTION
                 ESCAPE
                 END
     . . EXECUTE THE TARGET SYSTEM UNTIL A SOFTWARE BREAKPOINT IS FOUND
       NL; 'STARTING THE EMULATOR'; NL
       EBRUN
       NL; 'BREAKPOINT ENCOUNTERED AT ADDRESS: '; PC:H
     . . RETURN TO CALLER
       END
```

Figure H-10. Listing of SB Procedure File (Sheet 4 of 4)

```
...TITLE:          SETMEM:  SET A RANGE OF MEMORY TO CONSTANT VALUES
...REVISION:       03/24/77
...
...ABSTRACT:
...     SETMEM WRITES A SPECIFIED VALUE INTO EVERY WORD OF A SPECIFIED
...RANGE OF TARGET MEMORY.
...
...     SETMEM HAS THREE ARGUMENTS.  THE FIRST TWO SPECIFY THE RANGE OF
...MEMORY TO BE FILLED WITH THE CONSTANT.  THE OPTIONAL THIRD ARGUMENT
...IS THE VALUE TO BE WRITTEN INTO MEMORY.  IF IT IS OMITTED, ZERO IS
...USED FOR THE VALUE.  SETMEM MAY BE CALLED ONLY WHEN THE EMULATOR IS
...HALTED.
...
'DEFINING SETMEM'
...
PROC SETMEM(2,1) BEGIN
  IF ARG 0 EQ 3
       THEN LOC 1 = ARG 3                    ...VALUE WAS SPECIFIED
       ELSE LOC 1 = 0;                       ...NOT SPECIFIED
  REPEAT BEGIN
       @ARG 1 = LOC 1;                       ...WRITE A WORD
       ARG 1 = ARG 1 + 2;                    ...INCREMENT ADDRESS
  END UNTIL ARG 1 HI ARG 2 OR ARG 1 EQ 0;
END
```

Figure H-11. Listing of SETMEM Procedure File

```
...TITLE:        SIE:  SINGLE INSTRUCTION EXECUTION
...REVISION:     06/02/77
...
...ABSTRACT:
...     SIE LETS THE EMULATOR EXECUTE ONE INSTRUCTION AND HALT.  BEFORE
...IT EXECUTES THE INSTRUCTION, SIE PRINTS THE PC AND THE INSTRUCTION
...ABOUT TO BE EXECUTED.  IF THE INSTRUCTION HAS A SOURCE (DESTINATION)
...OPERAND, SIE PRINTS ITS ADDRESS AND VALUE BEFORE AND AFTER THE
...INSTRUCTION IS EXECUTED.
...
...USAGE:
...     WP, PC, AND ST MUST BE SET CORRECTLY FOR THE INSTRUCTION.
...NORMALLY, SIE MAY BE USED REPEATEDLY TO STEP THE EMULATOR THROUGH
...A PROGRAM ONE INSTRUCTION AT A TIME.
...     SIE HAS ONE OPTIONAL ARGUMENT, WHICH SPECIFIES HOW MANY
...INSTRUCTIONS TO STEP THROUGH, ONE AT A TIME, BEFORE SIE RETURNS.
...SIE PRINTS THE INSTRUCTION ABOUT TO BE EXECUTED ON THE FIRST LINE.
...ON TWO MORE LINES SIE PRINTS THE VALUES OF WP, PC, AND ST, AND
...OF THE SOURCE AND THE DESTINATION OPERANDS, BEFORE AND AFTER THE
...INSTRUCTION IS EXECUTED, RESPECTIVELY.  IF AN INSTRUCTION HAS
...NO SOURCE OR NO DESTINATION OPERAND, THE DISASSEMBLER SETS THE
...SYSTEM VARIABLE SRC OR DST, RESPECTIVELY, TO -1.  IF SIE FINDS
...SRC (DST) EQUAL TO -1 IT ASSUMES THERE IS NO SOURCE (DESTINATION)
...OPERAND AND DOES NOT PRINT THE ADDRESS (-1) OR VALUE BEFORE OR
...AFTER THE INSTRUCTION IS EXECUTED.  ON THE FOURTH LINE SIE PRINTS
...THE NEXT INSTRUCTION TO BE EXECUTED, AT THE NEW PC.
...     WHEN SIE IS USED, THE EMULATOR COMPARISON LOGIC MAY NOT BE
...USED.  THE EMULATOR TRACE LOGIC AND THE TRACE MODULE MAY BE USED,
...BUT ONLY ONE INSTRUCTION WILL BE RECORDED.
...
'DEFINING LOCAL PROC ZSIESD'
...
PROC ZSIESD (0) BEGIN                  ...LOCAL PROCEDURE FOR SIE
       NL '    WP=' WP:H ' PC=' PC:H ' ST=' ST:H
       IF SRC NE -1
            THEN BEGIN
                 ' SRC=' SRC:H '/ ' @SRC:H
            END;
       IF DST NE -1
            THEN BEGIN
                 ' DST=' DST:H '/ ' @DST:H
            END;
  END
...
NL
'DEFINING SIE'
...
PROC SIE (0,1) BEGIN
       EEVT(INT);                ...USE INTERNAL EVENT LOGIC
       EBRK(EVT,SELF);           ...HALT WHEN NEXT PC EXECUTED
       IF ARG 0 EQ 1             ...HOW MANY TIMES TO REPEAT
            THEN LOC 1 = ARG 1
            ELSE LOC 1 = 1;
       WHILE LOC 1 GT 0
            DO BEGIN
                 @PC:NI;                  ...CURRENT INSTRUCTION
                 ZSIESD;                  ...LOCAL PROC PRINTS INFO
                 ECMP(IAQ,PC);            ...BREAK ON THE NEXT PC
                 ERUN;                    ...SINGLE-STEP EMULATOR
                 WHILE (EST AND 1) DO NULL
                 ZSIESD;
                 LOC 1 = LOC 1 - 1        ...ITERATE THE PROCESS
```

Figure H-12. Listing of SIE Procedure File (Sheet 1 of 2)

```
        END;
    @PC:NXXXXXI;                    ...NEXT INSTRUCTION
    EBRK(OFF,OFF)                   ...TURN SIE STUFF OFF
END
```

Figure H-12. Listing of SIE Procedure File (Sheet 2 of 2)

```
... TITLE:          ESTAT
... REVISION:       05/02/77
...
... ABSTRACT:
...     ESTAT USES SYSTEM VARIABLES EST, ENI, ETBO, AND ETBN TO PRINT
... THE STATUS OF THE EMULATOR MODULE.
...
... USAGE:
...     ESTAT HAS NO ARGUMENTS AND MAY BE CALLED AT ANY TIME.
...
'DEFINING ESTAT' NL
...
PROC ESTAT(0) BEGIN
  'EMULATOR IS '
  IF (EST AND  1) EQ 0
       THEN 'NOT '
  'EXECUTING'
  IF EST AND 8
       THEN ' AN IDLE INSTRUCTION'
  '. '
  NL 'EMULATOR TRACE BUFFER IS '
  IF (EST AND 2) EQ 0
       THEN 'NOT '
  'FULL. '
  NL 'EVENT CONDITIONS ARE '
  IF (EST AND 4) EQ 0
       THEN 'NOT '
  'SATISFIED. '
  ENI:NU 'BREAKPOINTS COUNTED. '
  IF (EST AND 1) EQ 0
       THEN BEGIN
            ETBN-ETBO+1:ND 'ADDRESSES TRACED'
            IF ETBN GE ETBO
                 THEN BEGIN
                      ': ' ETBO:D '... ' ETBN:D;
                 END
       END
END
... TITLE:          TSTAT:  PRINT TRACE MODULE STATUS
... REVISION:       05/02/77
...
... ABSTRACT:
...     TSTAT USES SYSTEM VARIABLES TST, TNE, TNI, TTBO, AND TTBN TO
... PRINT AN ENGLISH DESCRIPTION OF THE STATUS OF THE TRACE MODULE.
...
... USAGE:
...     TSTAT HAS NO ARGUMENTS.   IT MAY BE CALLED AT ANY TIME.
...
'DEFINING TSTAT' NL
...
PROC TSTAT(0) BEGIN
  'TRACE MODULE IS '
  IF (TST AND 1) EQ 0
       THEN 'NOT ';
  'TRACING. '
  NL 'TRACE BUFFER IS '
  IF (TST AND 2) EQ 0
       THEN 'NOT ';
  'FULL. '
  NL 'EVENT CONDITIONS ARE '
  IF (TST AND 4) EQ 0
       THEN 'NOT ';
```

Figure H-13. Listing of STAT Procedure File (Sheet 1 of 2)

```
  'SATISFIED.'
  TNI:NU 'BREAKPOINTS COUNTED.'
  IF (TST AND 1) EQ 0
       THEN BEGIN
            TTBN-TTBO+1:ND 'SAMPLES IN BUFFER';
            IF TTBN GE TTBO
                 THEN BEGIN
                      ':' TTBO:D '...' TTBN:D;
                 END;
            TNE:NU 'EVENTS COUNTED.'
       END
END
```

Figure H-13. Listing of STAT Procedure File (Sheet 2 of 2)

```
...TITLE:          TDATA:  PRINT TRACE MODULE SAMPLES
...REVISION:       03/24/77
...
...ABSTRACT:
...     TDATA PRINTS A SPECIFIED RANGE OF SAMPLES FROM THE TRACE MODULE
...TRACE BUFFER.  THE SAMPLES ARE NOT INTERPRETED; ALL TWENTY BITS ARE
...PRINTED FOR EACH SAMPLE.
...
...USAGE:
...     TDATA HAS ONE REQUIRED AND ONE OPTIONAL ARGUMENT.  THE FIRST
...ARGUMENT IS THE STARTING INDEX TO BE USED.  THE OPTIONAL SECOND
...ARGUMENT IS THE ENDING INDEX.  IF IT IS OMITTED, THE ENDING INDEX
...IS THE STARTING INDEX PLUS 29, GIVING 30 SAMPLES, IF THERE ARE
...THAT MANY IN THE TRACE BUFFER.  TDATA MAY BE CALLED WHENEVER THE
...TRACE MODULE IS HALTED.
...
'DEFINING TDATA' NL
...
PROC TDATA(1,2) BEGIN
  IF ARG 0 EQ 2
       THEN LOC 1 = ARG 2                    ...ENDING INDEX WAS GIVEN
       ELSE LOC 1 = ARG 1 + 29;              ...NOT GIVEN
  IF LOC 1 GT TTBN
       THEN LOC 1 = TTBN;                    ...LIMIT ENDING INDEX
  LOC 2 = 0;                                 ...SET NEWLINE FLAG
  ' INDEX     HIGH   LOW            HIGH   LOW'
  '            HIGH   LOW';
  WHILE ARG 1 LE LOC 1
       DO BEGIN
            IF LOC 2 MOD 3 EQ 0
                 THEN BEGIN
                      ARG 1:ND '/ '          ...START A NEW LINE
                      END
                 ELSE '    '                 ...TAB TO NEXT COLUMN
            LOC 2 = LOC 2 + 1;               ...STEP THE FLAG COUNTER
            TTBH(ARG 1):H;                   ...DISPLAY SAMPLE
            TTB(ARG 1):HA;
            ARG 1 = ARG 1 + 1;
       END;
END
...TITLE:          TFOUR:  DISPLAY TRACED SAMPLES IN 4-BIT GROUPS
...REVISION:       05/06/77
...
...ABSTRACT:
...     TFOUR DISPLAYS A SPECIFIED RANGE OF TRACE BUFFER SAMPLES
...IN GROUPS OF FOUR BITS.
...
...USAGE:
...     THE TRACE MODULE MUST BE HALTED AND CONTAIN SAMPLES.  TFOUR
...HAS ONE REQUIRED AND ONE OPTIONAL ARGUMENT.  THE FIRST ARGUMENT IS
...REQUIRED AND SPECIFIES THE STARTING INDEX.  THE SECOND ARGUMENT IS
...OPTIONAL AND SPECIFIES THE ENDING INDEX.  IF IT IS OMITTED, START+
...9 IS USED.
...
'DEFINING TFOUR' NL
...
PROC TFOUR(1,2) BEGIN
       IF ARG 0 GE 2
            THEN LOC 1 = ARG 2
            ELSE LOC 1 = ARG 1 + 9;
       'INDEX   0123   4567   8901   2345   6789   0   4826' NL;
       FOR ARG 1 = ARG 1 TO LOC 1 DO BEGIN
```

Figure H-14. Listing of TDUMPS Procedure File (Sheet 1 of 2)

```
        ARG 1:D5;  LOC 2 = TTB(ARG 1);
        TTBH(ARG 1):B4;
        MDR = 0; DIV(>1000,LOC 2):B4;
        (LOC 2 AND >0F00) / >100:B4;
        (LOC 2 AND >00F0) / >10:B4;
        LOC 2 AND >000F:B4;
        TTBH(ARG 1):H1;
        LOC 2:HN;
        END;
END
```

Figure H-14. Listing of TDUMPS Procedure File (Sheet 2 of 2)

```
...TITLE:   TIMER:  TRACE MODULE TIMING OF EMULATOR
...REVISION:03/24/77
...
...ABSTRACT:
...     THIS ROUTINE SETS THE TRACE MODULE TO START TIMING WHEN
...THE EMULATOR IS NEXT STARTED.  IT MAY BE CALLED AGAIN WHILE
...THE EMULATOR IS RUNNING OR AFTER THE EMULATOR HAS HALTED TO
...PRINT THE ELAPSED TIME SINCE THE EMULATOR WAS STARTED,
...COMPUTED FROM THE EVENT COUNTER, THE DELAY COUNTER, AND
...THE NUMBER OF INTERRUPTS GENERATED BY THE TRACE MODULE.
...THE TRACE MODULE MAY NOT BE USED FOR TRACING WHILE IT
...IS BEING USED FOR TIMING, BUT THE EMULATOR MAY TRACE.
...THE TIMING IS ACCURATE TO 100 NANOSECONDS WHEN THE EMULATOR
...HAS HALTED, AND TO 6.5 MILLISECONDS WHILE THE EMULATOR IS
...RUNNING.
...
...USAGE:
...     THE CONTROL CABLE MUST CONNECT THE EMULATOR AND TRACE MODULE.
...NO OTHER TRACE MODULE COMMANDS SHOULD BE USED WHEN TIMER IS
...BEING USED.  IF THE TRACE MODULE IS NOT RUNNING, TIMER WILL
...SET IT UP FOR TIMING AND START IT.
...
...THE PRESENCE OF THE OPTIONAL ARGUMENT SUPPRESSES THE
...PRINTING OF ELAPSED TIME.  THIS ARGUMENT (ANY VALUE
...WILL DO) CAN BE USED WHEN TIMER IS BEING CALLED FOR THE
...FIRST TIME IN A SERIES OF TIMINGS, TO SET THE TRACE
...MODULE UP WITHOUT PRINTING MEANINGLESS TIMING INFORMATION.
...FOR EXAMPLE,
...
...     ? TIMER(0)            (ARGUMENT SUPPRESSES PRINTING)
...       ...                 (COMMANDS TO SET UP EMULATOR)
...     ? ERUN                (AFTER SETTING UP EMULATOR)
...     ? TIMER               (EMULATOR MAY STILL BE RUNNING)
...       1   .   234     567     800  SECONDS, STILL TIMING
...       ...                 (EMULATOR HALTS ITSELF, OR EHLT IS GIVEN)
...     ? TIMER               (NOW EMULATOR IS NOT RUNNING)
...       1   .   789      12     300  SECONDS
...
'DEFINING TIMER'
...
PROC TIMER(0,7) BEGIN
...
...VAR
...     LOC 1:    FIRST SCRATCH, THEN NUMBER OF SECONDS ELAPSED.
...     LOC 2:    MILLESECONDS.
...     LOC 3:    MICROSECONDS.
...     LOC 4:    NANOSECONDS (ACCURATE TO 100 NS).
...     LOC 5:    SCRATCH.
...     LOC 6:    SCRATCH.
...     LOC 7:    NUMBER OF EVENTS, IF TRACE IS HALTED;
...               0 IF TRACE IS RUNNING.
...
...IF THE OPTIONAL ARGUMENT IS NOT PRESENT, COMPUTE ELAPSED
...TIME AND PRINT IT.
...
        IF ARG 0 EQ 0
            THEN BEGIN
...
...IF THE TRACE MODULE IS ,HALTED, THEN USE NUMBER OF EVENTS.
...OTHERWISE, USE ZERO AND GET APPROXIMATE ELAPSED TIME ONLY.
...
                IF TST AND 1
```

Figure H-15. Listing of TIMER Procedure File (Sheet 1 of 3)

```
                    THEN
                         LOC 7 = 0
                    ELSE
                         LOC 7 = TNE;
...
...EACH THREE DIGITS OF THE ELAPSED TIME RIGHT OF THE DECIMAL
...POINT ARE COMPUTED SEPARATELY, STARTING WITH NANOSECONDS.
...THE WHOLE FORMULA FOR THE ELAPSED TIME IS
...          T  =  (65536 * TNI  +  TNE)  *  1.0E-07
...BECAUSE TNE IS INCREMENTED EACH 100 NS, AND TNI IS INCREMENTED
...EACH 65,536 TIMES TNE IS INCREMENTED.
...
...THE NANOSECONDS POSITIONS ARE COMPUTED BY
...          D9  =  600 * TNI  +  100 * TNE
...AND PRINTED WITH
...          D9  (MOD 1000)            (ONLY THE LOW-ORDER THREE DIGITS).
...
...THE MICROSECONDS POSITIONS ARE COMPUTED BY
...          D6  =  553 * TNI  +  (D9 DIV 1000)
...WHERE "DIV" IS INTEGER DIVISION, SO (D9 DIV 1000) IS THE CARRY
...FROM THE NANOSECONDS COMPUTATION INTO THE MICROSECONDS POSITION.
...AND PRINTED WITH
...          D6  (MOD 1000).
...
...THE MILLISECONDS POSITIONS ARE COMPUTED BY
...          D3  =  6 * TNI  +  (D6 DIV 1000)
...AND PRINTED WITH
...          D3  (MOD 1000).
...
...THE SECONDS POSITIONS, TO THE LEFT OF THE DECIMAL POINT, ARE JUST
...THE EXCESS IN D3 OVER 1000, VIZ.,
...          D3  DIV  1000,
...AND MAY BE ABOUT AS LARGE AS 400 (6 OR 7 MINUTES) BEFORE OVERFLOW
...AND INFORMATION LOSS OCCURS.    THUS THIS ROUTINE MAY BE USED TO
...TIME EMULATOR RUNS OF MOST INTERESTING LENGTHS.
...
                    LOC 4 = MPY(100,LOC 7);
                    LOC 5 = MDR;
                    LOC 6 = MPY(600,TNI);
                    LOC 5 = MDR + LOC 5;
                    LOC 6 = LOC 4 + LOC 6;
                    MDR = MDR + LOC 5;
                    LOC 5 = DIV(1000,LOC 6); ...CARRY INTO MICROSECS
                    LOC 4 = MDR;             ...NANOSECS
...
                    LOC 3 = MPY(553,TNI);
                    LOC 6 = MDR;
                    LOC 3 = LOC 3 + LOC 5;
                    MDR = MDR + LOC 6;
                    LOC 6 = DIV(1000,LOC 3); ...CARRY INTO MILLISECS
                    LOC 3 = MDR;             ...MICROSECS
...
                    LOC 2 = MPY(6,TNI);
                    LOC 1 = MDR;
                    LOC 2 = LOC 2 + LOC 6;
                    MDR = LOC 1 + MDR;
                    LOC 1 = DIV(1000,LOC 2); ...CARRY INTO SECS
                    LOC 2 = MDR;             ...MILLISECS
...
...PRINT THE RESULT, AT LAST.
...
                LOC 1:D;                     ...SECONDS
```

Figure H-15. Listing of TIMER Procedure File (Sheet 2 of 3)

```
                '.';                        ...DECIMAL POINT
               LOC 2:D;                     ...MILLISECONDS
               LOC 3:D;                     ...MICROSECONDS
               LOC 4:D;                     ...NANOSECONDS
                'SECONDS'
...
...IF THE TRACE MODULE IS NOT HALTED, INDICATE SO.
...
                IF TST AND 1
                     THEN BEGIN
                          ', STILL TIMING';
                          RETURN
                     END;
           END
...
...WHETHER THE OPTIONAL ARGUMENT IS THERE OR NOT, IF THE TRACE
...MODULE IS HALTED, SET IT UP FOR THE NEXT TIMING RUN.
...
           IF (TST AND 1) EQ 0
                THEN BEGIN
                     TTRC(OFF,256,INT);
                     TCMP(OFF,0,0);
                     TEVT(0,0,INT+EACH+NORM);
                     TBRK(EVT,OFF);
                     TRUN;
                END
      END
```

Figure H-15. Listing of TIMER Procedure File (Sheet 3 of 3)

```
...TITLE:          TRACE:  DISPLAY TRACE DATA IN TIME DOMAIN
...REVISION:       03/25/77
...
...ABSTRACT:
...     TRACE DISPLAYS THE DATA IN THE TRACE BUFFER IN A
...FORMAT SIMILIAR TO A LOGIC STATE ANALYZER
...
...USAGE:
...ARGUMENT 1 IS THE INDEX TO TTB TO PUT ON THE LEFT OF THE DISPLAY
.....THE DEFAULT VALUE IS TTB0
...ARGUMENT 2 IS THE FIRST BIT TO BE DISPLAYED
.....THE DEFAULT VALUE IS 0
...ARGUMENT 3 IS THE LAST BIT TO BE DISPLAYED
.....THE DEFAULT VALUE IS 19
...ARGUMENT 4 IS THE SCALE FACTOR
.....THE SCALE WILL DEFAULT TO 1 MARK/BIT NEGATIVE SCALES
.....ARE TAKEN AS BITS/MARK
...
'DEFINING TRACE PROCEDURE PLEASE WAIT';
...
PROC TRACE (0,11) BEGIN  ...PRINT OUT TIMING DIAGRAM
...LOC 1 IS THE FIRST INDEX TO TTB
...LOC 2 IS THE FIRST BIT TO DISPLAY
...LOC 3 IS THE LAST BIT TO DISPLAY
...LOC 4 IS THE SCALE FACTOR IN MARKS/BIT
...LOC 5 IS THE INDEX TO TTB
...LOC 6 IS THE MASK FOR EACH BIT
...LOC 7 IS THE SCALE COUNTER
...LOC 8 IS THE MAXIMUM INDEX TO TTB
...LOC 9 IS THE VALUE TO MASK OFF FOR EACH BIT
...LOC 10 IS THE PREVIOUS VALUE OF LOC 9
...LOC 11 IS A FLAG FOR NEGATIVE SCALES
.....................................................................
...SET UP DEFAULT VALUES
IF ARG 0 GE 1 THEN LOC 1=ARG 1; ELSE LOC 1=TTB0;
IF ARG 0 GE 2 THEN LOC 2=ARG 2; ELSE LOC 2=0;
IF ARG 0 GE 3 THEN LOC 3=ARG 3; ELSE LOC 3=19;
IF ARG 0 GE 4 THEN LOC 4=ARG 4; ELSE LOC 4=1;
IF LOC 4 LE 0 THEN
LOC 8=LOC 1+(60*(-LOC 4)); ELSE
LOC 8=LOC 1+(60/LOC 4);     ...INDEX ON RIGHT SIDE OF DISPLAY
...PRINT HEADER
NL
'TRACE BIT    * AMPL LOGIC STATE ANALYZER *    INDEX';LOC 1:D;'TO';
IF LOC 8 LE TTBN THEN LOC 8:D; ELSE TTBN:D;
'  EMU BIT';NL;
IF LOC 2 LT 4 THEN LOC 5=4-LOC 2; ELSE LOC 5=16-(LOC 2-4);
LOC 6=1
WHILE LOC 5 GT 1 DO BEGIN
LOC 6=LOC 6*2;          ...INITIALIZE MASK TO PROPER VALUE
LOC 5=LOC 5-1;
END;
...SCAN A PAGE
.....................................................................
WHILE LOC 2 LE LOC 3 DO BEGIN
    LOC 2:D       ...PRINT OUT BIT NUMBER
    ':';
    LOC 5=LOC 1;
    LOC 7=1;
    ...SCAN A LINE
    .................................................................
  IF LOC 4 GT 0 THEN
```

Figure H-16. Listing of TRACE Procedure File (Sheet 1 of 2)

```
    WHILE LOC 5 LT LOC 8 DO BEGIN
        IF (LOC 5 GE TTBO) AND (LOC 5 LE TTBN) THEN
        IF LOC 2 LT 4 THEN LOC 9=TTBH(LOC 5);ELSE
        LOC 9=TTB(LOC 5);
        ...PRINT OUT BIT STATE
        IF (LOC 6 AND LOC 9) EQ 0 THEN        ...HIGH OR LOW?
        '_';                            ...LOW STATE
        ELSE
        '-';                            ...HIGH STATE
        ...POSITIVE SCALE FACTOR
        IF LOC 7 GE LOC 4 THEN BEGIN
           LOC 5=LOC 5+1;   ...INCREMENT WORD INDEX
           LOC 7=0;         ...RESET SCALE COUNTER
        END;
        LOC 7=LOC 7+1;      ...INCREMENT SCALE COUNTER
    END;
  ELSE      ...NEGATIVE SCALE FACTOR
    WHILE LOC 5 LT LOC 8 DO BEGIN
        LOC 7=-1;
        LOC 11=0;           ...RESET HIGH AND LOW FLAG
        WHILE LOC 7 GE LOC 4 DO BEGIN
            LOC 10=LOC 9
            IF (LOC 5 GE TTBO) AND (LOC 5 LE TTBN) THEN
            IF LOC 2 LT 4 THEN LOC 9=TTBH(LOC 5);ELSE
            LOC 9=TTB(LOC 5);
            ELSE LOC 11=2;
            LOC 9=LOC 9 AND LOC 6;
            IF LOC 7 EQ -1 THEN LOC 10=LOC 9
            IF LOC 10 NE LOC 9 THEN LOC 11=1
            LOC 7=LOC 7-1;      ...DECREMENT SCALE COUNT
            LOC 5=LOC 5+1;      ...INCREMENT INDEX
        END;
        ...PRINT OUT BIT STATE
        IF LOC 11 EQ 1 THEN 'X'; ELSE ...NOT CONSTANT HIGH OR LOW
        IF LOC 11 EQ 2 THEN ' '; ELSE ...OUT OF ARRAY BOUNDS
        IF LOC 9 EQ 0 THEN      ...HIGH OR LOW?
        '_';                    ...LOW STATE
        ELSE
        '-';                    ...HIGH STATE
    END;

    ..........................................................................
    ':';
    IF LOC 2 GE 4 THEN
    LOC 2-4:D;      ...PRINT OUT BIT NUMBER WITH RESPECT TO EMULATOR
    ELSE IF LOC 2 EQ 01 THEN '   IAQ';     ...LABEL STATUS BITS
    ELSE IF LOC 2 EQ 02 THEN '   DBIN';
    ELSE IF LOC 2 EQ 03 THEN ' EMU CMP';
    IF LOC 2 NE LOC 3 THEN BEGIN
        LOC 6=LOC 6/2;  ...MASK FOR NEXT LINE
        NL;             ...NEW LINE
    END;
    LOC 2=LOC 2+1;                  ...INCREMENT LINE COUNT
    IF LOC 2 EQ 4 THEN LOC 6=>8000;     ...RESET BIT MASK
END;
..........................................................................
END
```

Figure H-16. Listing of TRACE Procedure File (Sheet 2 of 2)

```
... TITLE:          TRACE MODULE SAMPLE VERIFICATION
... REVISION:       05/06/77
...
... ABSTRACT:
...     THESE TWO ROUTINES RECORD A TRACE MODULE TRACE BUFFER FULL OF
... SAMPLES ON A FILE AND THEN COMPARE THEM TO A SUBSEQUENT BUFFER FULL
... OF SAMPLES FOR CONSISTENCY.  THE COMPARISON IS SUBJECT TO AN OPTIONAL
... BIT MASK.
...
... USAGE:
...     THE TRACE MODULE MUST BE HALTED AND CONTAIN SAMPLES IN ITS
... BUFFER TO USE EITHER OF THESE PROCEDURES.  TSAVE HAS ONE
... ARGUMENT, THE ACCESS NAME OF THE FILE ON WHICH THE CONTENTS OF THE
... TRACE BUFFER WILL BE SAVED.  TVRFY HAS ONE REQUIRED ARGUMENT AND
... TWO OPTIONAL ARGUMENTS.  THE FIRST ARGUMENT MUST BE THE ACCESS NAME
... OF THE FILE CONTAINING THE TRACE SAMPLES SAVED BY TSAVE.  THE SECOND
... AND THIRD ARGUMENTS GIVE THE HIGH AND LOW-ORDER BIT MASKS.
... IF THE SECOND ARGUMENT IS OMITTED THE MASK >F IS USED.  IF THE THIRD
... ARGUMENT IS OMITTED, THE MASK >FFFF IS USED FOR THE LOW-ORDER PART
... COMPARISION.  THE SAMPLES ARE COMPRED IN SEQUENCE.  ANY DIFFERENCES
... ARE PRINTED.
...
'DEFINING TSAVE' NL
PROC TSAVE (1,1) BEGIN
...
... TURN OFF THE CONSOLE SO THE LISTING TO THE FILE WILL GO FASTER.
...
        CNSL(OFF);
...
... OPEN THE FILE FOR LISTING AND WRITE THE TRACE BUFFER LIMITS.
...
        LIST(ARG 1)
        TTBO:DN; TTBN:DN;
...
... WRITE ALL THE SAMPLES FROM THE BUFFER, HIGH FOUR BITS FIRST,
... FOLLOWED BY LOW SIXTEEN BITS OF EACH SAMPLE.
...
        FOR LOC 1 = TTBO TO TTBN DO BEGIN
            TTBH(LOC 1):H1N; TTB(LOC 1):HN;
            END;
...
... TURN THE CONSOLE BACK ON AND THE LISTING FILE OFF.
...
        CNSL(ON);
        LIST('DUMY');
        LIST(OFF);
        END
...
'DEFINING TVRFY' NL
...
PROC TVRFY (1,10) BEGIN
        CLSE;
...
... OPEN THE FILE AND SET UP THE MASKS IN LOC 9 AND LOC 10.
...
        OPEN(ARG 1)
        IF ARG 0 GE 2
            THEN LOC 9 = ARG 2
            ELSE LOC 9 = >F;
        IF ARG 0 GE 3
            THEN LOC 10 = ARG 3
            ELSE LOC 10 = >FFFF;
```

Figure H-17. Listing of TRSAVR Procedure File (Sheet 1 of 2)

```
...
...READ THE TRACE BUFFER LIMITS AND COMPARE TO THE CURRENT LIMITS.
...
      LOC 1 = READ;
      LOC 2 = READ;
      IF LOC 1 NE TTBO OR LOC 2 NE TTBN
           THEN BEGIN
                'FILE HAS ' LOC 2-LOC 1+1:D3 'SAMPLES, '
                LOC 1:D3 '...   ' LOC 2:D3 NL
                'TRACE BUFFER HAS ' TTBN-TTBO+1:D3 'SAMPLES, '
                TTBO:D3 '...   ' TTBN:D3 NL
                END;
...
...LOOP THROUGH THE FILE AND THE TRACE BUFFER COMPARING SAMPLES.
...LOC 1 AND 3 WILL BE USED FOR INDICES.
...
      LOC 3 = TTBO; LOC 8 = 0;
      WHILE LOC 1 LE LOC 2 AND LOC 3 LE TTBN DO BEGIN
           LOC 4 = READ; LOC 5 = READ;
           LOC 6 = TTBH(LOC 3); LOC 7 = TTB(LOC 3);
           IF (LOC 4 AND LOC 9) NE (LOC 6 AND LOC 9)
           OR (LOC 5 AND LOC 10) NE (LOC 7 AND LOC 10)
                THEN BEGIN
                     IF LOC 8 EQ 0
                          THEN BEGIN
                               NL '      FILE        TRACE BUFFER  ';
                               '   DIFFERENCE' NL;
                               LOC 8 = 1;
                               END;
                     LOC 1:D3; LOC 4:H1; LOC 5:H;
                     LOC 3:D3; LOC 6:H1; LOC 7:HXX;
                     ((LOC 4 AND NOT LOC 6) OR (NOT LOC 4 AND LOC 6))
                          AND LOC 9:H1;
                     ((LOC 5 AND NOT LOC 7) OR (NOT LOC 5 AND LOC 7))
                          AND LOC 10:HN;
                     END;
           LOC 1 = LOC 1 + 1; LOC 3 = LOC 3 + 1;
           END;
      CLSE;
      END
```

Figure H-17. Listing of TRSAVR Procedure File (Sheet 2 of 2)

APPENDIX I

GLOSSARY

APPENDIX I

GLOSSARY

This glossary contains definitions of terms used in the manual, some of which are defined in greater detail in other manuals.

AMPL Microprocessor Prototyping Laboratory – A combination of hardware and software that executes in an FS990 or DS990 to develop prototype hardware and software.

AMPL Microprocessor Prototyping Language – The language with which the user communicates with the AMPL system. When host memory consists of more than 16K words, an extended AMPL may be used, providing additional capabilities.

Assembler – A computer program that develops a computer program by assembling machine instructions from source program statements. The computer program output by the assembler is called the object program.

Breakpoint – The result of a computer reaching a predefined program condition or set of program conditions. Typically (but not necessarily) a breakpoint halts the computer.

Buffer Module – The hardware module that interfaces between the emulator module and the target computer. A buffer module adapts the emulator module to emulate a specific microprocessor, i.e., a TMS 9900 buffer module is required to interface the emulator to a target computer that uses a TMS 9900 microprocessor.

Data Probe – A connector that connects a single line to a point on a module or printed circuit board for test purposes. The trace module may be connected to desired points on the target system using a set of data probes.

Delay Counter – A counter on the trace module that counts a specified number of traced items to delay the breakpoint signal. The delay counter begins to count when the event counter has counted the specified number of events.

Emulator – A module that connects to a host computer at the CRU interface and to a target computer through the appropriate buffer module. The host computer controls the emulator as a peripheral device causing the emulator to control the target system for test purposes. The emulator includes memory that can be used as target system memory.

Event – In the emulator, an event is a valid comparison between an address and a specified value. Alternatively, an external signal may be defined as an event. In the trace module, an event is a valid comparison between a value being stored in the trace memory and a specified value. The trace module may also use an external signal as an event.

Event Counter – A counter in the trace module that counts events. The event counter in conjunction with the delay counter allows the user to specify a count of events and a delay count after which a breakpoint may occur.

Floppy Disc – A data storage device that stores data magnetically on the surface of a flexible disc (diskette). The floppy disc accesses data randomly.

Format Specification – A set of one or more characters (table 4-2) preceded by a colon (:) used in a display statement to specify the format of the display. Format specification characters are effective in the order in which they are entered; they control conversions to decimal, hexadecimal, or octal values, or display in binary form or in ASCII character form. Other format specification characters insert spaces or carriage returns.

Function – An AMPL subroutine that is called by entering the function name and arguments as a variable in an AMPL expression or statement, and which returns a value that is used as the value of the variable.

Host Computer – The Model 990 Computer in which the AMPL Microprocessing Prototyping Laboratory software executes to control the microprocessor of a target system.

Host Memory – The memory of the host computer.

Host System – The AMPL Microprocessor Prototyping Laboratory system as it functions to control and test a target system.

Link Utility – A computer program (TXLINK) that integrates object modules into a linked object module (load module). Typically each module requires addresses of labels defined in other modules (external references) and defines labels required in other modules (external definitions). Linking consists of satisfying the external references in a set of modules from the external definitions of these modules.

Mask – A mask is used in a comparison between words or groups of bits to select the bits to be compared. The mask of a word comparison contains 16 bits and the mask of a comparison of a group of bits contains the number of bits in the group to be compared. Comparisons of qualifiers and stored values in the trace module are masked, allowing the user to select the bits for comparison.

Procedure – An AMPL subroutine that is called by entering the procedure name and arguments as an AMPL statement. All transfer of data to or from the procedure is through arguments.

Program Counter – An internal register in a computer or microprocessor of the 990 family which contains the address of the next instruction to be executed.

Programmable Read-Only Memory – A memory device that may only be read which is programmed with the information (instructions and constants) it requires.

PROM – Programmable Read-Only Memory.

Qualifiers – A set of four lines connected to the trace module that selects the clocks at which values are stored. The user specifies bit values to which the qualifiers are compared and a mask that selects which qualifiers are compared.

RAM – Random Access Memory.

Random Access Memory – A memory device that may be accessed for either reading or writing data. Addresses may be accessed randomly.

Read-Only Memory – A memory device that may only be read. The information (instructions and constants) is stored in memory when it is manufactured.

Status Register – An internal register in a computer or microprocessor of the 99Ø family that contains status bits set and reset to show the results of executing instructions, and the interrupt mask.

Target Program – The program to be executed by the target system computer.

Target System – The prototype system under test by the AMPL system.

Target System Memory – The memory of the target system. The emulator can map either the program memory or the trace memory, or both, into target system memory.

Terminal Executive Development System – A program development system that supports an assembler (TXMIRA), a Text Editor (TXEDIT), a Link Utility (TXLINK), and other utilities and executes in the FS99Ø.

Text Editor – A computer program (TXEDIT) that writes source files from user input, and allows the user to interactively edit the source files.

TMS 99ØØ – A microprocessor of the 99Ø family, using a single chip, and N-channel silicon-gate MOS technology. A 16-bit processor using 16-bit memory access, with byte addressing.

TMS 998Ø – A microprocessor of the 99Ø family, using a single chip, and N-channel silicon-gate MOS technology. A 16-bit processor using 8-bit memory access, with byte addressing.

Trace – In the emulator module, an operation that stores up to 256 addresses during emulation as specified by the user. In the trace module, an operation that stores up to 256 2Ø-bit values as specified by the user. The 2Ø bits may be data from selected points on a prototype module, or may be either memory addresses or data from the emulator. The user may display the values stored in either the trace memory of the emulator or in the memory of the trace module.

Trace Memory – A 256-word memory in the emulator which is used by the emulator to store traced addresses. Trace memory is also available to the target system. When the target system uses trace memory, addresses $FEØØ_{16}$ through $FFFF_{16}$ of target system memory space address the trace memory.

Trace Module – A module that connects to a host computer at the CRU interface and to the target computer or the emulator. The trace module may operate in conjunction with or independently of the emulator to store selected data and to compare stored data to specified criteria. The trace module may stop the trace when the specified number of values has been stored, or when a specified number of valid comparisons have been made, and a specified number of additional values has been stored. Alternatively, either of these conditions may cause the trace module to request the emulator to halt, or may interrupt the host computer.

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 AMPL Microprocessor Prototyping Laboratory Systems Operation Guide (946244-9701)

Manual Date: 15 January 1978 **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.

CUT ALONG LINE

Location in Manual	Comment/Suggestion
_____	_____
	_____
	_____
	_____
_____	_____
	_____
	_____
	_____
_____	_____
	_____
	_____
	_____

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