8002A

μPROCESSOR LAB

SYSTEM USER'S MANUAL

This manual supports TEKDOS Version 1.

INSTRUCTION MANUAL

Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077

Serial Number ________________

070-2701-00

First Printing JUL 1978

WARRANTY

The 8002A μProcessor Lab System (including options) is warranted against defective materials and workmanship under normal use and service for a period of 90 days from date of initial shipment. CRTs found to be defective within 12 months from the date of shipment will be exchanged at no charge (this does not include installation).

On site warranty repair is provided during normal working hours (for the 90-day period). Travel to the site is confined to those areas in which Tektronix states it has service facilities available for this product.

Tektronix shall be under no obligation to furnish warranty service if:

a. Attempts to install, repair, or service the equipment are made by personnel other than Tektronix service representatives.

b. Modifications are made to the hardware or software by personnel other than Tektronix service representatives.

c. Damage results from connecting the 8002A μProcessor Lab System to incompatible equipment.

Specifications and price change privileges reserved.

Copyright © 1978 by Tektronix, Inc., Beaverton, Oregon. Printed in the United States of America. All rights reserved. Contents of this publication may not be reproduced in any form without permission of Tektronix, Inc.

U.S.A. and foreign Tektronix products covered by U.S. and foreign patents and/or patents pending.

TEKTRONIX is a registered trademark of Tektronix, Inc.

All software products including this document, all associated flexible discs and the programs they contain are the sole property of Tektronix, Inc., and may not be used outside the buyer's organization. The software products may not be copied or reproduced in any form without the express written permission of Tektronix, Inc. All copies and reproductions shall be the property of Tektronix and must bear this copyright notice and ownership statement in its entirety.

NOTE

IN THIS MANUAL, ALL REFERENCES TO THE "8002 μPROCESSOR LAB" APPLY EQUALLY TO THE 8002A μPROCESSOR LAB.

The TEKTRONIX 8002A μProcessor Lab, containing a standard 32k-byte program memory, replaces the TEKTRONIX 8002 μProcessor Lab with its standard 16k-byte program memory. At the time of this writing, the 8002A μProcessor Lab functions identically to the 8002 μProcessor Lab.

Contact your Tektronix field service representative to order manuals for the 8002A μProcessor Lab.

TEKTRONIX®

8002
μPROCESSOR LAB

SYSTEM USER'S MANUAL

This manual supports TEKDOS Version 1.

Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077
070-2313-01

Serial Number ____________________

First Printing April 1977

WARRANTY

The 8002 μProcessor Lab System (including options) is warranted against defective materials and workmanship under normal use and service for a period of 90 days from date of initial shipment. CRTs found to be defective within 12 months from the date of shipment will be exchanged at no charge. (This does not include installation.)

On site warranty repair is provided during normal working hours (for the 90-day period). Travel to the site is confined within the country of purchase.

Tektronix shall be under no obligation to furnish warranty service if:

a. Attempts to install, repair, or service the equipment are made by personnel other than Tektronix service representatives.

b. Modifications are made to the hardware or software by personnel other than Tektronix service representatives.

c. Damage results from connecting the 8002 μProcessor Lab System to incompatible equipment.

Specifications and price change privileges reserved.

Copyright © 1977 Tektronix, Inc.

All Rights Reserved.

All software products including this document, all associated flexible discs and the programs they contain are the sole property of Tektronix, Inc., and may not be used outside the buyer's organization. The software products may not be copied or reproduced in any form without the express written permission of Tektronix, Inc. All copies and reproductions shall be the property of Tektronix and must bear this copyright notice and ownership statement in its entirety.

DOCUMENTATION OVERVIEW

Introduction

The 8002 μPROCESSOR LAB support documentation consists of two groups of manuals; user's manuals and service manuals. User's manuals explain the procedures required to operate the 8002 μPROCESSOR LAB system and its peripheral devices. They are identified by their gray covers and are a standard part of the system package.

Service manuals provide the information necessary to perform routine maintenance and to make minor repairs to system components. The hardware test manuals, within this group, provides detailed trouble-shooting information beyond the scope of routine maintenance. Service manuals are identified by their blue covers and may be purchased as optional accessories.

User Manual Organization

The 8002 μPROCESSOR LAB user's manuals are incorporated into a series of user support packages. Each package contains a three-ring binder, a manual, a reference card that summarizes the contents of the manual, and one or more flexible discs. Some discs are blank and others contain coded programs.

The contents of the user support packages at the time of this writing are as follows:

8002 μPROCESSOR LAB System User's Package

Contents	Part Number
General purpose three-ring binder	016-0367-00
8002 μPROCESSOR LAB System User's Manual	070-2313-01
8002 μPROCESSOR LAB System Reference Card	070-2350-01
Two blank flexible discs	119-0848-01

Description

This package is a standard accessory to every 8002 μPROCESSOR LAB System. The System User's Manual is the fundamental documentation and contains information on how to use the 8002 μPROCESSOR LAB operating system. The System Reference Card summarizes the contents of the System User's Manual. The two blank flexible discs are provided so back-up copies of software can be safely stored. A blank disc may also be used to store user written programs.

8ØØ2 μPROCESSOR LAB Assembler & Emulator Support Package for 8Ø8Ø Microprocessor

Contents	Part Number
General purpose three-ring binder	016-0367-00
8ØØ2 μPROCESSOR LAB Assembler & Emulator User's Manual for 8Ø8Ø Microprocessor	070-2341-00
8ØØ2 μPROCESSOR LAB 8Ø8Ø Assembler and Emulator Reference Card	070-2351-00
8ØØ2 μPROCESSOR LAB System Disc for 8Ø8Ø Microprocessor	

Description

This package contains the necessary software and documentation to support 8Ø8Ø microprocessor program development. The system disc contains the TEKDOS operating system and the TEKTRONIX 8Ø8Ø Assembler. The manual explains how to operate the TEKTRONIX 8Ø8Ø Assembler and Emulator modules. This manual and the System User's Manual provide complete user information for 8Ø8Ø program development. The 8Ø8Ø Assembler and Emulator Reference Card, a summary of the commands in the 8Ø8Ø Assembler and Emulator User's manual, serves as a quick reference guide.

8ØØ2 μPROCESSOR LAB Assembler & Emulator Support Package for 68ØØ Microprocessor

Contents	Part Number
General purpose three-ring binder	016-0367-00
8ØØ2 μPROCESSOR LAB Assembler & Emulator User's Manual for 68ØØ Microprocessor	070-2349-00
8ØØ2 μPROCESSOR LAB 68ØØ Assembler and Emulator Reference Card	070-2352-00
8ØØ2 μPROCESSOR LAB System Disc for 68ØØ Microprocessor	

Description

This package supports program development for the 68ØØ microprocessor. The system disc contains the TEKDOS operating system and the TEKTRONIX 68ØØ Assembler. The manual contains the details necessary to operate the TEKTRONIX 68ØØ Assembler and Emulator

modules. This manual and the System User's Manual provide complete user information for 8Ø8Ø program development. The 8Ø8Ø Assembler and Emulator Reference Card is a summary of the commands in the 8Ø8Ø Assembler and Emulator User's manual and serves as a quick reference guide.

Future User Support Packages

Support packages similar to those packages described are planned for each microprocessor development module to be introduced in the future.

Service Manuals

The 8ØØ2 μPROCESSOR LAB consists of a main system service manual and supplementary service manuals for each plug-in module. The service manuals contain information pertinent to installation, servicing, and maintaining system components. Diagrams and circuit descriptions are provided, as are specifications and parts lists. Detailed information facilitates all necessary cleaning, lubrication, calibration, and diagnostic trouble-shooting.

Available service manuals with their respective part numbers and general content are as follows:

8ØØ2 μPROCESSOR LAB System Service Manual

Part No. 070-2312-00

. System Processor
. System Memory
. Program Memory
. Assembler Processor
. System Communications
. Debug and Front Panel I/O
. Flexible Disc Unit

8ØØ2 μPROCESSOR LAB 8Ø8Ø Emulator Processor Service Manual

Part No. 070-2353-00

. 8Ø8Ø Emulator Processor
. 8Ø8Ø Prototype Control Probe

8ØØ2 μPROCESSOR LAB 68ØØ Emulator Processor Service Manual

Part No. 070-2354-00

. 68ØØ Emulator Processor
. 68ØØ Prototype Control Probe

8ØØ2/8ØØ1 μPROCESSOR LAB Real-Time Prototype Analyzer System Service Manual

Part No. 070-2356-00

. Data Acquisition Interface
. P6451 Data Acquisition Probe

8ØØ2 μPROCESSOR LAB 27Ø4/27Ø8 PROM Programmer Service Manual

Part No. 070-2355-00

. Service Instructions

8ØØ2 μPROCESSOR LAB 17Ø2A PROM Programmer Service Manual

Part No. 070-2357-00

. Service Instructions

8ØØ2 μPROCESSOR LAB Maintenance Front Panel Instruction Manual

Part No. 070-2358-00

. Operating Instructions
. Service Instructions

8ØØ2 μPROCESSOR LAB Hardware Test Manual

Part No. 070-2375-00

Contains support documentation necessary to effectively troubleshoot the 8ØØ2 μPROCESSOR LAB System. The manual, together with diagnostic software and a test fixture, forms the 8ØØ2 μPROCESSOR Hardware Test Package, Part No. 067-0841-00.

TABLE OF CONTENTS

SECTION 1	8002 μPROCESSOR LAB SYSTEM INTRODUCTION	PAGE
	THE PURPOSE OF AN MDA	1-1
	KINDS OF MDAs	1-1
	IMPORTANT MDA FEATURES	1-2
	MICROPROCESSOR DEVELOPMENT CYCLE	1-4
	8002 μPROCESSOR LAB HARDWARE COMPONENTS	1-8
	8002 μPROCESSOR LAB SOFTWARE COMPONENTS	1-13
	SUMMARY	1-15
SECTION 2	BECOMING FAMILIAR WITH THE SYSTEM	
	INTRODUCTION	2-1
	SYSTEM POWER-UP PROCEDURE	2-3
	Turn on the Flexible Disc Unit	2-3
	Turn on the Terminal	2-4
	Turn on the 8002 μProcessor Lab	2-6
	Turn on the Line Printer	2-7
	Insert the System Flexible Disc	2-8
	TEKDOS Ready State	2-9
	LISTING THE FLEXIBLE DISC DIRECTORY	2-9
	SYSTEM POWER-DOWN PROCEDURE	2-9
	Remove the Flexible Discs	2-10
	Turn off the Terminal Power	2-10
	Turn off the Line Printer	2-10
	Turn off the 8002 μProcessor Lab	2-10
	Turn off the Flexible Disc Unit	2-11
	FLEXIBLE DISC INITIALIZATION	2-11
	Flexible Disc Formatting	2-11
	Flexible Disc Verification	2-12
	Disc Duplication	2-12
	TEXT EDITING	2-13
	Text Creation	2-13
	Text Storage	2-15
	Text Retrieval	2-15
	Text Alteration	2-16
SECTION 3	COMMAND CONVENTIONS	
	INTRODUCTION	3-1
	COMMAND NAME	3-1
	DELIMITERS	3-1
	PARAMETERS	3-2

TABLE OF CONTENTS (cont)

SECTION 4 **TEKTRONIX DISC OPERATING SYSTEM** PAGE

INTRODUCTION . . . 4-1
SYSTEM DESCRIPTION . . . 4-3
TEKDOS DISC AND FILE UTILITIES . . . 4-9
FORMAT . . . 4-10
VERIFY . . . 4-12
RENAME . . . 4-13
DUP . . . 4-15
COPYSYS . . . 4-17
LDIR . . . 4-18
DELETE . . . 4-19
CMPF . . . 4-20
COPY . . . 4-21
PRINT . . . 4-24
TEKDOS CONTROL COMMANDS . . . 4-25
Space Bar . . . 4-26
CTRL-Z . . . 4-27
RUB OUT Key . . . 4-28
ESC . . . 4-29
SUSPEND . . . 4-31
CONT . . . 4-32
ABORT . . . 4-33
TEKDOS OPTION COMMANDS . . . 4-34
SYSTEM . . . 4-35
DEVICE . . . 4-36
CLOCK . . . 4-37
ASSIGN . . . 4-38
CLOSE . . . 4-39
EMULATE . . . 4-40
COMMAND FILES . . . 4-41
Command Description . . . 4-42
* . . . 4-46
KILL . . . 4-47
TYPE . . . 4-48

TABLE OF CONTENTS (cont)

SECTION 5 TEXT EDITOR **PAGE**

INTRODUCTION TO THE TEXT EDITOR 5-3
INVOKING THE TEXT EDITOR 5-4
COMMAND CONVENTIONS 5-7
TEXT TRANSFER COMMANDS 5-10
- INPUT 5-11
- INSERT 5-13
- FILE 5-14
- GET 5-17
- PUT 5-21
- COPY 5-26
- TYPE 5-32
- LIST 5-33

SEARCHING AND ALTERATION COMMANDS 5-34
- N 5-35
- UP 5-36
- DOWN 5-38
- BEGIN 5-40
- END 5-41
- FIND 5-42
- SUBSTITUTE 5-44
- REPLACE 5-46
- KILL 5-48

UTILITY COMMANDS 5-50
- TAB 5-51
- TABS 5-53
- MACRO 5-54
- Space Bar 5-56
- ESC 5-57
- QUIT 5-58
- BRIEF 5-60
- ? 5-61
- AGAIN 5-62

SECTION 6 ASSEMBLING AND LINKING

INTRODUCTION 6-1
- ASM 6-2
- LINK 6-5

TABLE OF CONTENTS (cont)

SECTION 7 — EMULATOR ENVIRONMENT — PAGE

INTRODUCTION . . . 7-1
OPERATING MODES . . . 7-2
  EMULATE . . . 7-4
LOADING AND STORING . . . 7-5
  WHEX . . . 7-8
  RHEX . . . 7-9
  LOAD . . . 7-10
  MODULE . . . 7-11
  FETCH . . . 7-12
MEMORY CONTROL . . . 7-13
  DUMP . . . 7-14
  EXAM . . . 7-16
  PATCH . . . 7-18
  MAP . . . 7-19
  MOVE . . . 7-22
  FILL . . . 7-23
USER PROGRAM EXECUTION . . . 7-24
  GO . . . 7-26
  XEQ . . . 7-27
  STATUS . . . 7-28

SECTION 8 — DEBUG SYSTEM

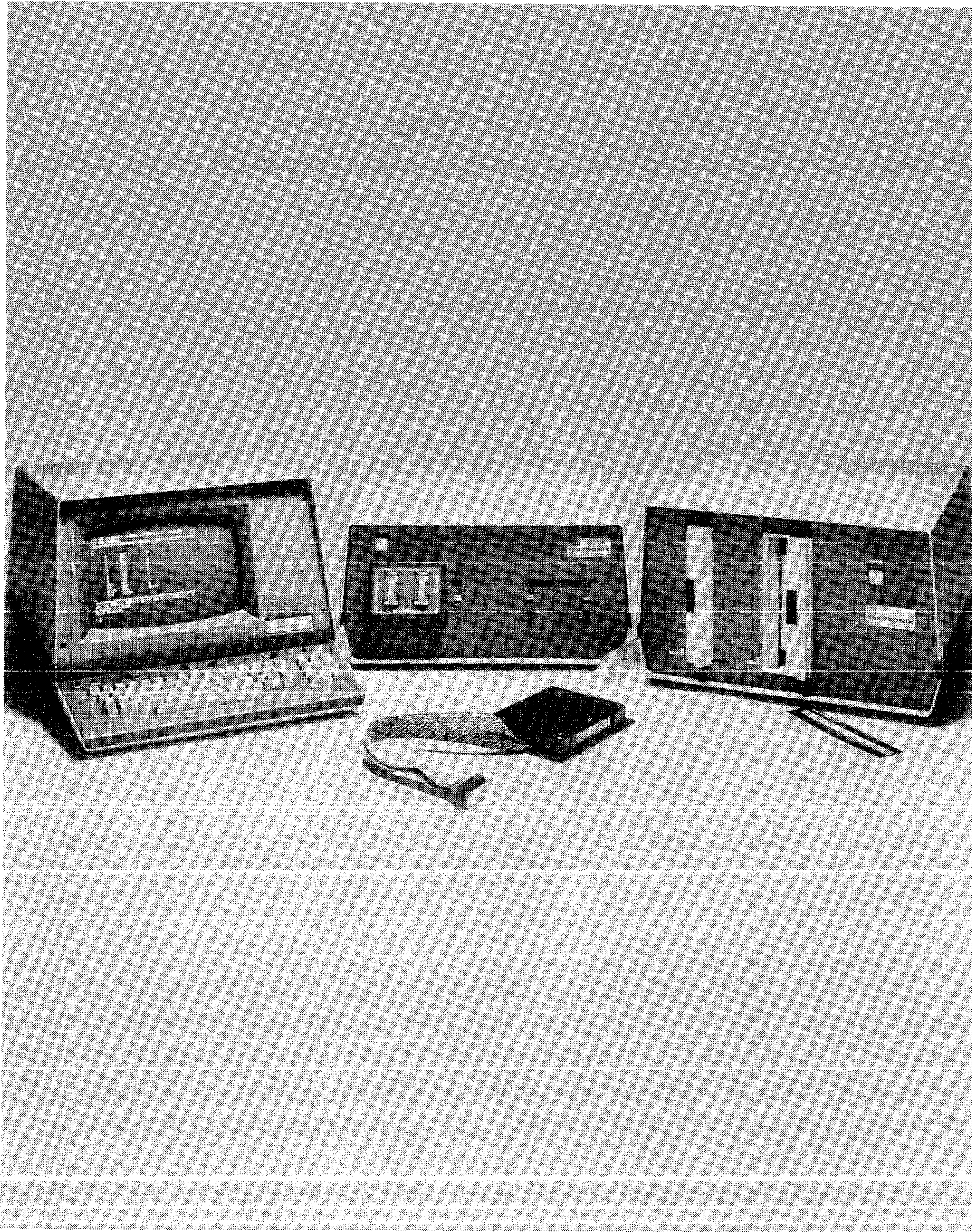
INTRODUCTION . . . 8-1
DEBUG SYSTEM STRUCTURE . . . 8-2
DEBUG SYSTEM FUNCTION . . . 8-6
DEBUG SYSTEM ENTRY AND EXIT . . . 8-8
COMMAND DESCRIPTIONS . . . 8-10
  DEBUG . . . 8-11
  TRACE . . . 8-12
  DSTAT . . . 8-18
  BKPT . . . 8-20
  CLBP . . . 8-23
  SET . . . 8-24
  RESET . . . 8-26

SECTION 9 — PROM PROGRAMMER

DOCUMENTATION NOTE . . . 9-1
INTRODUCTION . . . 9-2
PROM PROGRAMMER COMMANDS . . . 9-3
  RPROM . . . 9-5
  WPROM . . . 9-6
  CPROM . . . 9-7

TABLE OF CONTENTS (cont)

SECTION 9	**PROM PROGRAMMER (cont)**	**PAGE**
	HOW TO USE THE PROM PROGRAMMER	9-8
	SMS FORMAT COMMANDS	9-8
	CSMS	9-9
	RSMS	9-10
	WSMS	9-11
SECTION 10	**SERVICE CALLS**	
	INTRODUCTION	10-1
	SERVICE CALL DESCRIPTION	10-2
	SERVICE REQUEST BLOCK	10-3
	SRB Bytes	10-5
	SVC Functions	10-7
	SVC FUNCTION CODES	10-11
SECTION 11	**REAL-TIME PROTOTYPE ANALYZER**	
	INTRODUCTION	11-1
	DESCRIPTION	11-2
	COMMANDS	11-3
	EVT	11-4
	BIF	11-7
	RTT	11-8
	DRT	11-9
	CNT	11-10
SECTION 12	**INTER-SYSTEM COMMUNICATION**	
APPENDIX A	**TEKDOS ERROR CODES**	
	EDITOR ERROR MESSAGES	A-2
APPENDIX B	**TABLES**	
	HEXADECIMAL-DECIMAL CONVERSION	B-3
	HEXADECIMAL ADDITION	B-5
	HEXADECIMAL MULTIPLICATION	B-7
	POWERS OF 2	B-9
	ASCII CODE CONVERSION	B-11
APPENDIX C	**SYSTEM INSTALLATION**	
APPENDIX D	**COMMAND INDEX**	
APPENDIX E	**SOFTWARE ERROR REPORT FORMS**	
CHANGE INFORMATION		



2313-1

Fig. 1-1. The 8ØØ2 μPROCESSOR LAB System with Optional CT81ØØ CRT Terminal and Prototype Control Probe.

Section 1

8002 μPROCESSOR LAB SYSTEM INTRODUCTION

Microprocessor-based product development requires new and different design tools. In general, these design tools are called microprocessor development aids (MDAs). MDAs provide a total microprocessor design environment that closely approximates the actual environment of the product under development. The 8002 μPROCESSOR LAB provides these surroundings and allows all design team members to work within a common environment as they develop a product.

This section will introduce you to the terminology, concepts, and methods used in an MDA environment. First, we'll look at the features most desirable in an MDA system, and see how the 8002 μPROCESSOR LAB provides these features. Next, we'll look at the microprocessor development cycle. We'll point out typical microprocessor design problems and see how these problems are avoided while using the 8002 μPROCESSOR LAB. Finally, we'll discuss individual hardware and software modules within the 8002 μPROCESSOR LAB system and refer to documentation explaining their use.

THE PURPOSE OF AN MDA

An MDA, functioning as a design tool, is used to develop microprocessor software programs and to design microprocessor hardware circuits. The MDA then helps integrate the software and the hardware into a complete stand-alone microprocessor-based product.

KINDS OF MDAs

A variety of MDAs are available on the market today. They range from simple one-circuit board learning aids, to sophisticated multi-cabinet work stations. Most MDAs are tailored to support only one commercial microprocessor, and only a few can support the design activity for more than one. The TEKTRONIX 8002 μPROCESSOR LAB supports several.

IMPORTANT MDA FEATURES

Some of the many features that can make one MDA better than another are listed here.

System Programs Can Make a Difference

Most MDAs comprise three basic elements: a central processing unit (typically, a microprocessor); memory; and input/output (I/O) facilities. Most also contain support programs to perform supervisory functions. These "system programs" help enter microprocessor instructions into the MDA, and then assemble the instructions into a meaningful program for the prototype instrument under development. This resulting program is usually called a "user program."

System programs also monitor user program execution in the MDA emulator processor. The emulator processor, a microprocessor within the MDA, is identical to the microprocessor in the prototype. If errors are discovered, the system programs help make the corrections. One measure of an MDAs worth is the ease with which system programs perform user program entry, editing, assembly, and program debugging.

Memory and I/O Capability

Other differentiating MDA features are memory and I/O capability. An MDA with fast and convenient data handling capabilities can develop a program in minutes instead of hours. An MDA with resident random access memory (RAM) and on-line disc storage, allows information to be stored on disc until needed; then quickly transferred into RAM work space for processing. Clearly, efficient memory and I/O capability can decrease the development cycle turn-around time. The 8002 μPROCESSOR LAB features a fast, convenient, flexible disc operating system with 64K bytes of dynamic RAM storage in program memory. Approximately 630K bytes of on-line storage are available in the flexible disc unit.

Simulation Versus Emulation

MDAs can use either simulation or emulation to locate run-time errors and errors in program logic. When the simulation method is used, an MDA software program "acts" like the prototype microprocessor. The program interpreter reads a microprocessor user program instruction, then executes the instruction like the microprocessor in the prototype. Program logic flow can be checked in this manner. However, the simulation program often runs slower than the real microprocessor, and critical timing relationships between hardware and software are impossible to verify.

An emulation method MDA contains a hardware model of the prototype microprocessor. The model may be centered around discrete logic, another type microprocessor, or a microprocessor identical to the prototype microprocessor. When the processor is identical to the prototype microprocessor, the method is called "substitutive emulation." The 8002 μPROCESSOR LAB uses the substitutive emulation method. All user programs executed on the system can be checked for critical timing relationships between the software and the prototype hardware.

Value of In-Prototype Testing

Typically, the simplest MDAs do not have facilities for hardware development and testing. More complete MDAs provide limited signal monitoring functions, but most of these functions could be handled by conventional hardware test equipment. The most advanced MDAs (the 8002 μPROCESSOR LAB, for example), have the ability to swap known-good hardware elements into the prototype hardware, and can also swap known-good software programs. By connecting portions of the MDA circuitry to the prototype hardware in the early stages of development, the two parts can be exercised together as one complete microcomputer. The combined unit then runs under the control of the developmental software while being supervised by the MDAs debug system program. This technique, in-prototype testing, allows both hardware and software subcomponents to be tested, debugged, and verified as soon as they are complete. The entire prototype system is developed from the ground up, on known-good building blocks, and the chance of total system failure at the end of the development cycle is eliminated. The 8002 μPROCESSOR LAB supports in-prototype testing to the fullest extent.

MICROPROCESSOR DEVELOPMENT CYCLE

Unified hardware/software effort from conception to completion eliminates many problems, and hard to find system integration bugs can usually be avoided. A commonly beneficial environment is available to all design team members throughout the microprocessor product development cycle. Figure 1-2 illustrates the development cycle.

A New Product is Conceived

Management determines the need for a new product based on microprocessor technology. The hardware and software design teams are organized, time schedules are defined, and the funds are appropriated. The purchase of a TEKTRONIX 8002 μPROCESSOR LAB is included.

Microprocessors are Evaluated

Because the 8002 μPROCESSOR LAB supports several commercial microprocessors, the performance of each microprocessor can be evaluated and compared before a final selection is made. Using the emulator processors available in the 8002 μPROCESSOR LAB, the software team evaluates the different microprocessor instruction sets and software architecture facilities. The hardware team evaluates hardware features, execution speeds, and I/O handling facilities. Hardware/software trade-offs are discussed and a microprocessor selection is based on the overall requirements for the new product.

Prototype Functions are Defined

Both teams are now ready to define each function in the new product and to determine whether the function should be handled by the software or the hardware. Software flow charts are then drawn for each function. Since the design team is now familiar with the strengths and weaknesses of the selected microprocessor, the software architecture is structured appropriately.

Specification documents are written to ensure that every team member clearly understands the definition of each hardware and software function and how they are related. The 8002 μPROCESSOR LAB has a powerful text editor and convenient flexible disc storage facilities. Specification documents are entered and stored on the flexible discs. The documents are then easily updated as the product matures. Copies are quickly available from the optional LP8200 Line Printer.

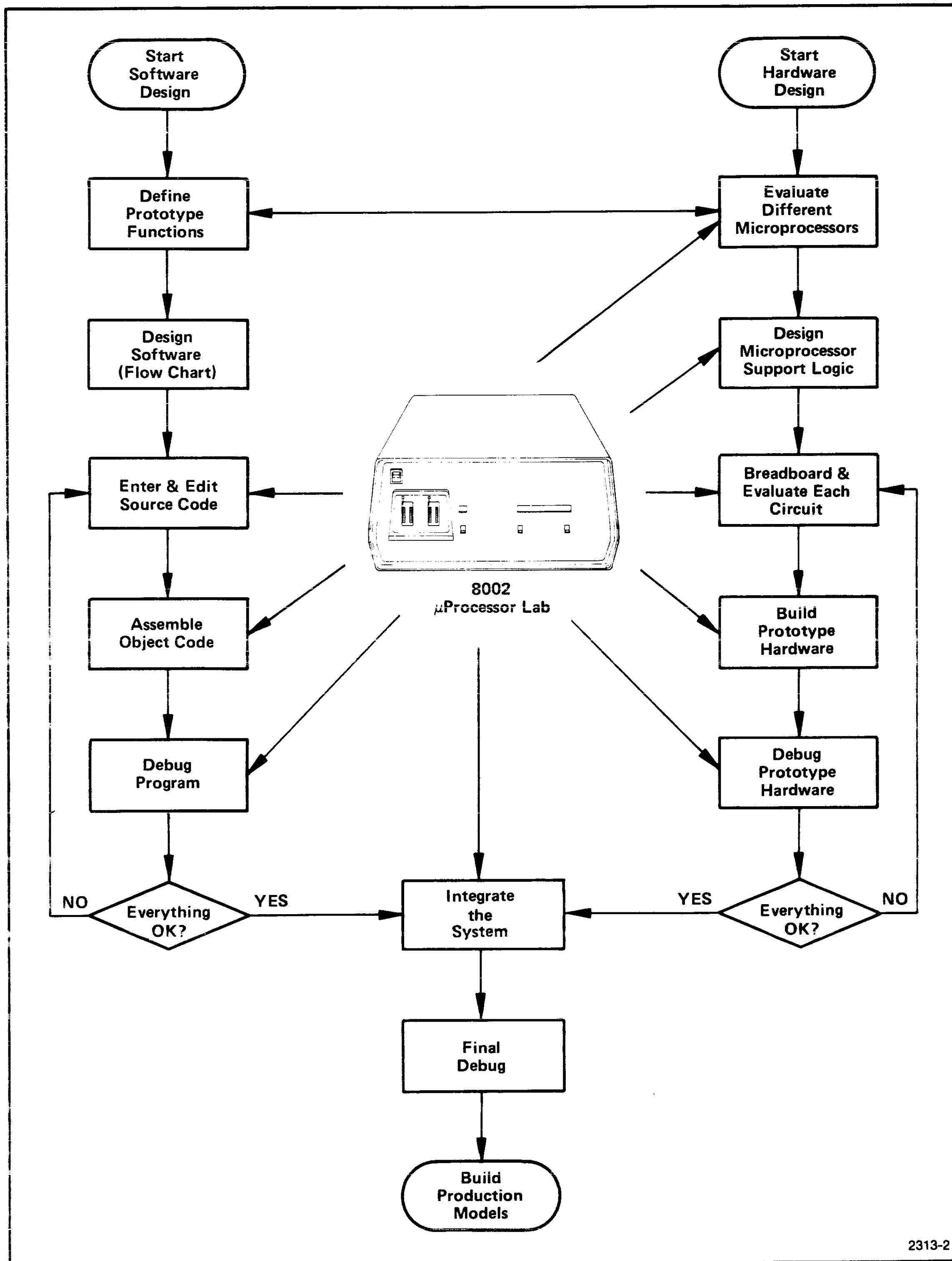


Fig. 1-2. Microprocessor Development Cycle with the 8002 μPROCESSOR LAB.

Hardware Team Draws Schematics and Breadboards the Circuits

While the software team is working on the software specifications, the hardware team designs the support logic circuits for the prototype microprocessor. As each circuit is built, it is connected to the emulator processor via the prototype microprocessor. Using their own test programs, stored on the flexible disc unit, the engineers test each circuit. Circuits are modified as required to improve their performance.

Software Team Starts Coding

Using assembly language, the software engineers code the prototype microprocessor program. Because the total program is large and complex, sub-modules are created. One software engineer is assigned to the software keyboard drive; another to the math routines, and so on. Each engineer enters his assembly language program via the system console, and uses the text editor to add, delete, or change code lines. After editing, the updated program is automatically stored on a flexible disc.

Software Team Assembles the Source Code

As each program sub-module becomes ready, the software engineer invokes the TEKTRONIX Assembler. The assembler assembles the source code into machine executable object code. Source code errors are then corrected by the engineer with the text editor.

The source code program is then re-assembled. This process is repeated until an error-free assembler listing is obtained. The program sub-module is then executed on the emulator processor.

Software Team Debugs the Software

Programs loaded into program memory are executed under the supervision of the debug system. Each program can be single-stepped through execution, or executed in multiple step sequences, or executed continuously to completion. At any point, the debug system allows program execution to be suspended. Stack pointers, program counters, or general registers can be modified to correct errors. Execution can then continue.

Software and Hardware Sub-Systems are Debugged Together

As each software module and its associated hardware module becomes error-free, they are tested together. All design personnel are able to observe realistic results without the need to "second guess" actual conditions.

The software is loaded into the emulator processor, the emulator processor is then connected to the prototype hardware, and testing begins. Under the supervision of the debug software and the optional real-time prototype analyzer, prototype functions are brought to life. Logic errors are immediately detected and can be corrected quickly. System integration continues until all prototype components are joined. After all hardware circuits have been tested with their software counterparts, the prototype instruments are built.

Total System Integration Begins

When all prototype hardware is assembled, the emulator processor is connected to the prototype microprocessor socket via the optional prototype control probe. The total software program is loaded into program memory from the flexible disc files. Prototype hardware is activated, the emulator processor is turned on, and final system integration begins. Again, the debug system and the real-time prototype analyzer are used to monitor software/hardware activities. System integration proceeds rapidly and smoothly because each subsystem has already been debugged individually.

PROM Programming

When final tests are completed, the prototype program is transferred to Programmable Read Only Memory (PROM) chips, using the optional PROM Programmer. After being programmed, the PROMs are plugged into the prototype memory slots. The prototype control probe is removed from the prototype microprocessor socket and is replaced by the actual microprocessor. The prototype has now become a complete, thoroughly tested, stand-alone unit. Production can begin.

Using the 8002 μPROCESSOR LAB as a Manufacturing Test Device

After the new product is in production, the 8002 μPROCESSOR LAB is used to test production models before shipment to customers. If troubles exist, the 8002 μPROCESSOR LAB quickly isolates the problem. Troubleshooting time and troubleshooting costs are sharply reduced.

Conclusions

A microprocessor design effort centered on the 8002 μPROCESSOR LAB provides the necessary system integration tools from the start. Decisions are based on test results, not guess work. Each hardware/software module is tested before further decisions are based on its use, and system integration proceeds in an orderly manner.

The 8002 μPROCESSOR LAB removes the doubt from the microprocessor design process. Therefore, decisions are based on fact, and system integration results are clearly visible.

8002 μPROCESSOR LAB HARDWARE COMPONENTS

The 8002 μPROCESSOR LAB internal architecture centers around a system microprocessor that uses other microprocessors to perform different software and hardware support functions. The system contains 16K bytes of system random access memory (RAM) and up to 64K bytes of RAM program memory (depending on the options selected). The system also supports two flexible disc drives with approximately 315K bytes on each disc.

An 8002 μPROCESSOR LAB system block diagram is shown in Figure 1-3. The system contains three microprocessors—the system processor, the assembler processor, and the emulator processor. Each microprocessor resides on a separate plug-in circuit card in the system mainframe. These cards are connected to each other through a common system bus. Also residing in the mainframe is the optional PROM programmer, the RS-232-C interface with three I/O ports, the 16K byte system memory, and the standard 16K byte program memory (expandable to 64K).

The flexible disc unit is housed in a separate chassis and communicates with the other system components through the system processor. Other optional system peripherals such as the CT8100 CRT Terminal and the LP8200 Line Printer communicate with the system through the RS-232-C interface. The following is a brief description of each component in the system.

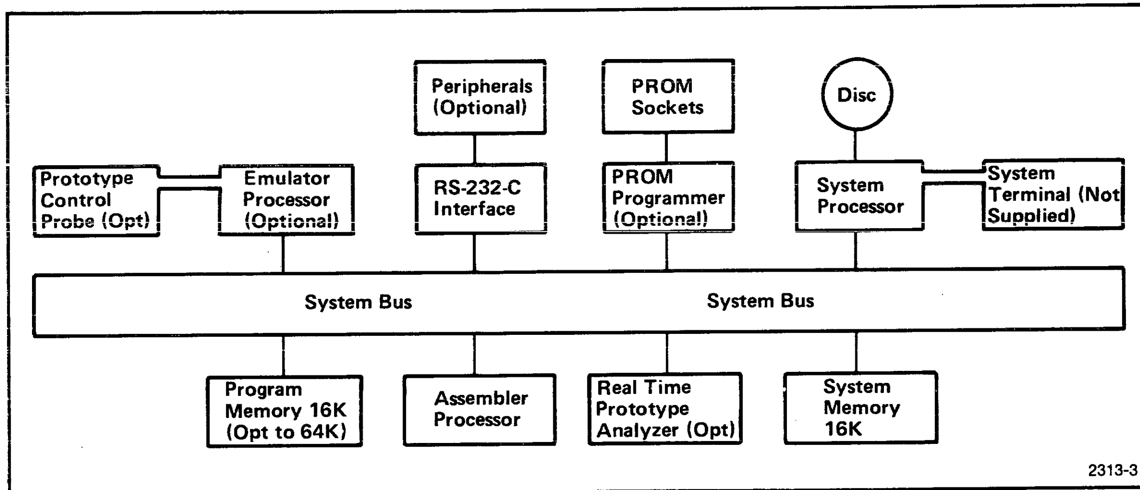


Fig. 1-3. 8002 μPROCESSOR LAB System Block Diagram.

System Processor

The system processor performs the following supervisory functions:

- System Input/Output — directs all I/O activity for the system peripherals, such as the flexible disc, the console, and the line printer.
- File Management — organizes, stores, and retrieves user programs and system programs from the disc drives.
- Text Editing — executes the text editor program and maintains text files on the flexible disc unit.
- Debugging — executes the debug program and controls the emulator processor through separate debug hardware.
- System Utilities — performs all system utility functions such as processing the messages between system peripheral devices.
- PROM Programming — monitors and controls all PROM (Programmable Read Only Memory) activity.

Assembler Processor

The assembler processor runs the TEKTRONIX Assembler program when the TEKDOS ASM command is executed. All assembler I/O activity to and from the flexible disc unit is handled by the system processor.

Emulator Processor

The emulator processor, a system option, runs and debugs user programs written for a particular commerical microprocessor. A separate processor is available for each commercial microprocessor you wish to emulate. Emulator processors available at the time of this writing are the 8080 and the 6800. More are planned for the near future.

The emulator processor serves two purposes. First, the emulator processor runs the user program while the system debugger program is active. This detects program run-time errors and program logic errors. Second, with the addition of an optional prototype control probe, the emulator processor takes the place of the actual microprocessor in the prototype under development. The user program can then drive and test the prototype hardware while under the supervision of the debug system.

System Memory

The system memory is a 16K-byte dynamic RAM located on a separate module within the main chassis. The system memory is accessed only by the system processor and is used to store TEKDOS programs while they are executing. The system memory also provides buffer space for all I/O activities.

Program Memory

The standard 16K byte program memory is located on a separate module within the main chassis. Additional 16K byte memory modules can be added to increase the total capacity to 64K bytes. The primary purpose of program memory is to store a user program while the program is being executed by the emulator processor. The system processor also uses program memory as a text buffer during text editing sessions.

Prototype Control Probe

The optional prototype control probe consists of cables, interface circuits, and a 40-pin connector. The connector plugs into the empty microprocessor socket on the prototype circuit board. The prototype control probe allows the emulator processor and program memory to take the place of the actual microprocessor and its associated memory in the prototype. Thus, the user program can be run, tested, and debugged in the prototype while under the supervision of the debug system.

The following three emulator operational modes are available with the prototype control probe plugged into the prototype:

- System Mode (Mode 0) — the emulator processor runs the program residing in program memory.
- Partial Emulation Mode (Mode 1) — the emulator processor runs the program residing in program memory and prototype memory. All I/O signals and data are supplied by the external prototype hardware.
- Full Emulation Mode (Mode 2) — the emulator processor runs the program resident in the external prototype memory. All I/O signals and data are also supplied by the prototype hardware.

Real-Time Prototype Analyzer

The optional real-time prototype analyzer enables you to dynamically monitor the prototype address bus, data bus, and up to eight other locations of your choice on the prototype circuit board. The analyzer's main function is to locate critical timing problems and hardware/software sequence problems in the prototype during the last stages of system integration and debugging. The analyzer monitors prototype activity while the prototype is running at full speed. The test results are printed on either the system console or the optional LP8200 Line Printer.

PROM Programmer

The PROM programmer option allows user programs to be transferred from program memory into Programmable Read Only Memory (PROM) chips. These PROM chips are then plugged into the prototype memory sockets and provide permanent program instructions for the prototype microprocessor. Not only can user programs be transferred from program memory into PROMs, but the reverse action can also take place; the contents of PROMs can be read into program memory. In addition, the user program residing in a PROM can be compared with the user program residing in program memory. The differences are displayed on the system console. This comparison technique is used to verify the contents of a PROM.

The 8ØØ2 μPROCESSOR LAB presently supports two different PROM programmers—one type for 17Ø2A PROMs and another type for 27Ø4/27Ø8 PROMS.

PROM programming is accomplished by plugging a PROM chip into the appropriate socket on the front panel. TEKDOS commands are then executed from the system keyboard to transfer program instructions back and forth between the PROM and program memory.

An 8ØØ2 μPROCESSOR LAB without the PROM programmer option still has the PROM sockets on the front panel. The PROM programmer circuit board, however, will not be in the mainframe. This renders the PROM programmer inactive. This circuit board can be ordered as a field installation kit at a later time and plugged into the 8ØØ2 μPROCESSOR LAB mainframe to activate the PROM programmer.

RS-232-C Interface

The RS-232-C interface board provides three I/O ports for connecting optional peripheral devices to the system. Any device that conforms to the EIA standard RS-232-C can be connected to the interface board. Typically, devices such as the LP82ØØ Line Printer are connected to the interface. A larger host computer can also be connected to the 8ØØ2 μPROCESSOR LAB. User programs can be transferred from the host and down-loaded into program memory for execution.

Flexible Disc Unit

A flexible disc unit is the on-line mass storage device for the 8ØØ2 μPROCESSOR LAB system. The flexible disc unit consists of two separate disc drive assemblies, a microprocessor controller, a power supply, and a cabinet. The flexible disc unit communicates directly with the system processor module through an interconnecting cable. Another flexible disc unit can be connected into the system to provide a four disc drives option.

System Terminal

The 8002 μPROCESSOR LAB System Terminal serves as the main communication channel between the system and the operator. (The system terminal is also referred to as the system console in this manual.)

Any terminal-like device can be used as the system terminal if the device has a keyboard, a display and an RS-232-C communications port. The terminal cable is connected directly to the system processor board in the mainframe.

Two TEKTRONIX System Terminals are available as options. The CT8100 CRT Terminal (Cathode Ray Tube) features a 9-inch refresh alphanumeric display. The CT8101 Printing Terminal features a paper print-out display instead of a refresh CRT display.

8002 μPROCESSOR LAB SOFTWARE COMPONENTS

TEKDOS (TEKTRONIX Disc Operating System)

TEKDOS is the operating system for the 8002 μPROCESSOR LAB and is loaded from the system disc in the flexible disc unit each time the system is powered up. TEKDOS contains the supervisory software programs for the system. The TEKDOS operation commands are described fully in the TEKtronix Disc Operating System section of this manual.

Text Editor

The text editor is invoked by the TEKDOS EDIT command and performs powerful text editing functions. The text editor is used to (1) enter new user programs into memory, then store the programs on disc and (2) correct user programs for errors detected during assembly. The text editor can also be used to store and update the support documentation for the prototype under development. Complete text editor instructions are given in the Text Editor section of this manual.

TEKTRONIX Assembler

After a source program has been entered and stored on a flexible disc unit by the text editor, the user program must be translated into machine-executable object code. This function is performed by the TEKTRONIX Assembler. The assembler then stores the assembled object code on disc in another file.

The assembler is loaded from disc into program memory and runs on the assembler processor. The assembler uses free space in program memory for I/O buffers and symbol tables. Versions of the TEKTRONIX Assembler exist for each microprocessor supported by the 8ØØ2 μPROCESSOR LAB. A separate disc is used for each version.

Instructions for calling the assembler from TEKDOS are given in the Assembling and Linking section of this manual. Instructions for the assembler are given in the Assembler and Emulator User's Manual specific to each microprocessor chip.

Linker

The linker software is considered a sub-module of the assembler software and is provided with each system disc. The linker is used to join several smaller user program modules into one large program. This feature allows several software engineers to work on program segments independently, and then join the segments into a large workable program.

Instructions for calling the linker from TEKDOS are given in the Assembling and Linking section of this manual. Instructions for the Linker are given in the Assembler and Emulator User's Manual.

Emulator

The emulator software allows user programs to be loaded into the optional emulator processor for operation, testing and debugging. Emulator software instructions are given in the Emulator Environment section of this manual.

Debug System

Since the assembler software detects syntax errors in the user program, a number of program logic errors usually remain undetected until the user program is executed on a real microprocessor. The debug system monitors user program execution on the emulator processor and the prototype microprocessor. The debug software allows you to examine, trace, modify, and change portions of your program as the program executes. This feature ensures that your program will be clean and free of "bugs" before it is placed in PROMs and plugged into the prototype instrument.

PROM Programmer

The PROM programmer software supervises and controls the transfer of user programs between program memory and PROM chip plugged into the front panel. Instructions for using the optional PROM programmer are given in the PROM Programmer section of this manual.

SUMMARY

The 8002 μPROCESSOR LAB is a design aid that integrates software and hardware into a microprocessor-based product. Efficient system programs, large memory, versatile I/O, and in-prototype testing are featured. Substitutive emulation permits an authentic microprocessor model to be tested in the prototype hardware. The 8002 μPROCESSOR LAB reduces microprocessor design guesswork by permitting system integration and debugging in the early phases of product development.

The 8002 μPROCESSOR LAB system centers around a system microprocessor using other microprocessors for support. The assembler runs on the assembler processor. User programs residing in program memory run on the emulator processor, under the supervision of the debug system. The optional prototype control probe connects the system to the prototype hardware for in-prototype testing. The flexible disc unit provides 630K bytes of on-line storage. The 8002 μPROCESSOR LAB software includes a disc operating system called TEKDOS, a text editor, an assembler, a linker, an optional emulator, a debug system, and an optional PROM programmer.

Section 2

BECOMING FAMILIAR WITH THE SYSTEM

INTRODUCTION

The procedures described in this section provide an initial overview of the 8002 μPROCESSOR LABs operation. Procedures demonstrated include system power up, directory listing, system power down, flexible disc initialization and a sample editing session. Since this section will guide you through a typical operating sequence for the first time, the emphasis is placed on the way the different procedures fit together. Detailed descriptions of each procedure and system command are provided in later sections of this manual. After completing this section, you should be able to:

1. Power up the system and load the Tektronix Disc Operating System (TEKDOS) from a flexible disc into system memory, as well as power down the system;

2. Display a directory listing on the console;

3. Format and verify a new flexible disc, as well as perform flexible disc file duplication;

CONTENTS

SECTION 2 BECOMING FAMILIAR WITH THE SYSTEM

INTRODUCTION 2-1
SYSTEM POWER-UP PROCEDURE 2-3
Turn on the Flexible Disc Unit 2-3
Turn on the Terminal 2-4
Turn on the 8002 μProcessor Lab 2-6
Turn on the Line Printer 2-7
Insert the System Flexible Disc 2-8
TEKDOS Ready State 2-9

LISTING THE FLEXIBLE DISC DIRECTORY 2-9
SYSTEM POWER-DOWN PROCEDURE 2-9
Remove the Flexible Discs 2-10
Turn off the Terminal Power 2-10
Turn off the Line Printer 2-10
Turn off the 8002 μProcessor Lab 2-10
Turn off the Flexible Disc Unit 2-11
FLEXIBLE DISC INITIALIZATION 2-11
Flexible Disc Formatting 2-11
Flexible Disc Verification 2-12
Disc Duplication 2-12
TEXT EDITING 2-13
Text Creation 2-13
Text Storage 2-15
Text Retrieval 2-15
Text Alteration 2-16

SYSTEM POWER-UP PROCEDURE

Turn on the Flexible Disc Unit

Always remove all flexible discs prior to turning the power on or off. If you do not, valuable data may be destroyed.

The flexible disc unit has a single front panel POWER rocker switch, as shown in Figure 2-1. Push the switch to its ON position. Allow a five minute warm-up time to permit the disc drive electronics to reach a stable temperature.

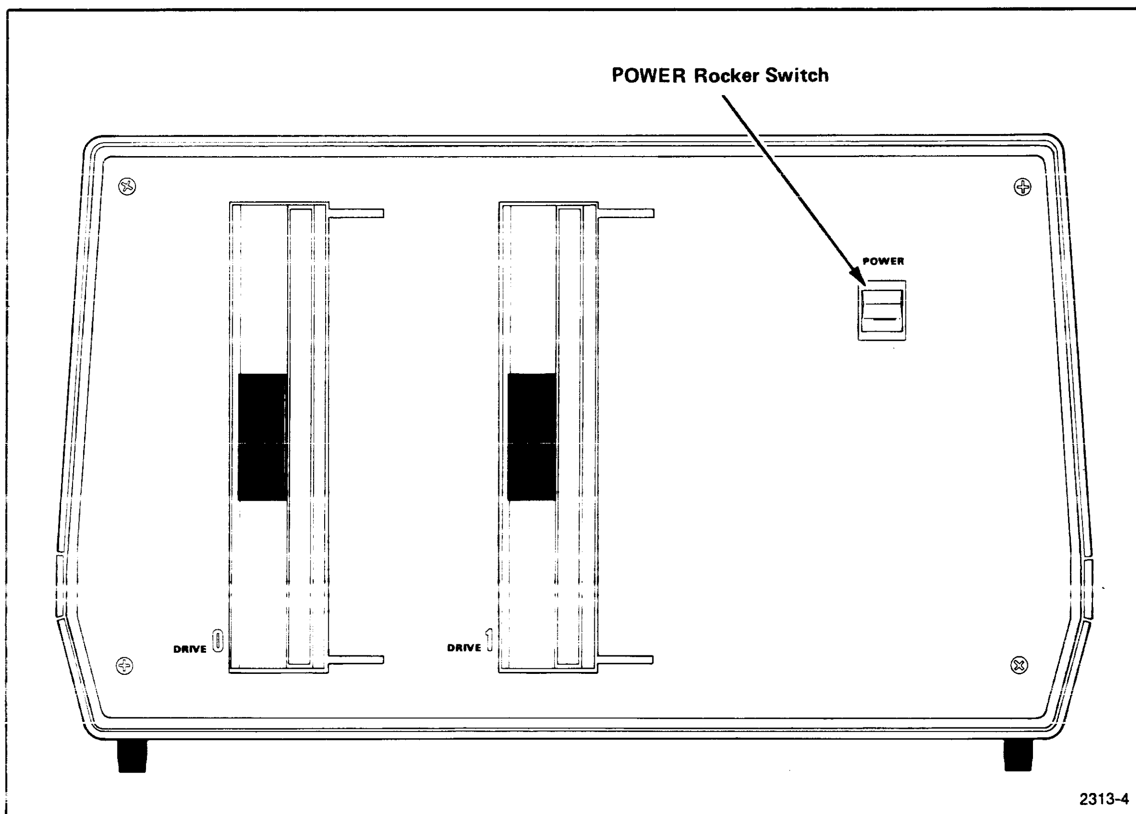


Fig. 2-1. Flexible Disc Unit Front panel.

Turn on the Terminal

If you are using the optional CT8100 CRT Terminal, a power ON/OFF rocker switch is located on the right side of the terminal. Push the switch to its ON position to activate the terminal. The green POWER indicator in the upper right corner of the front panel should light. Refer to Figure 2-2.

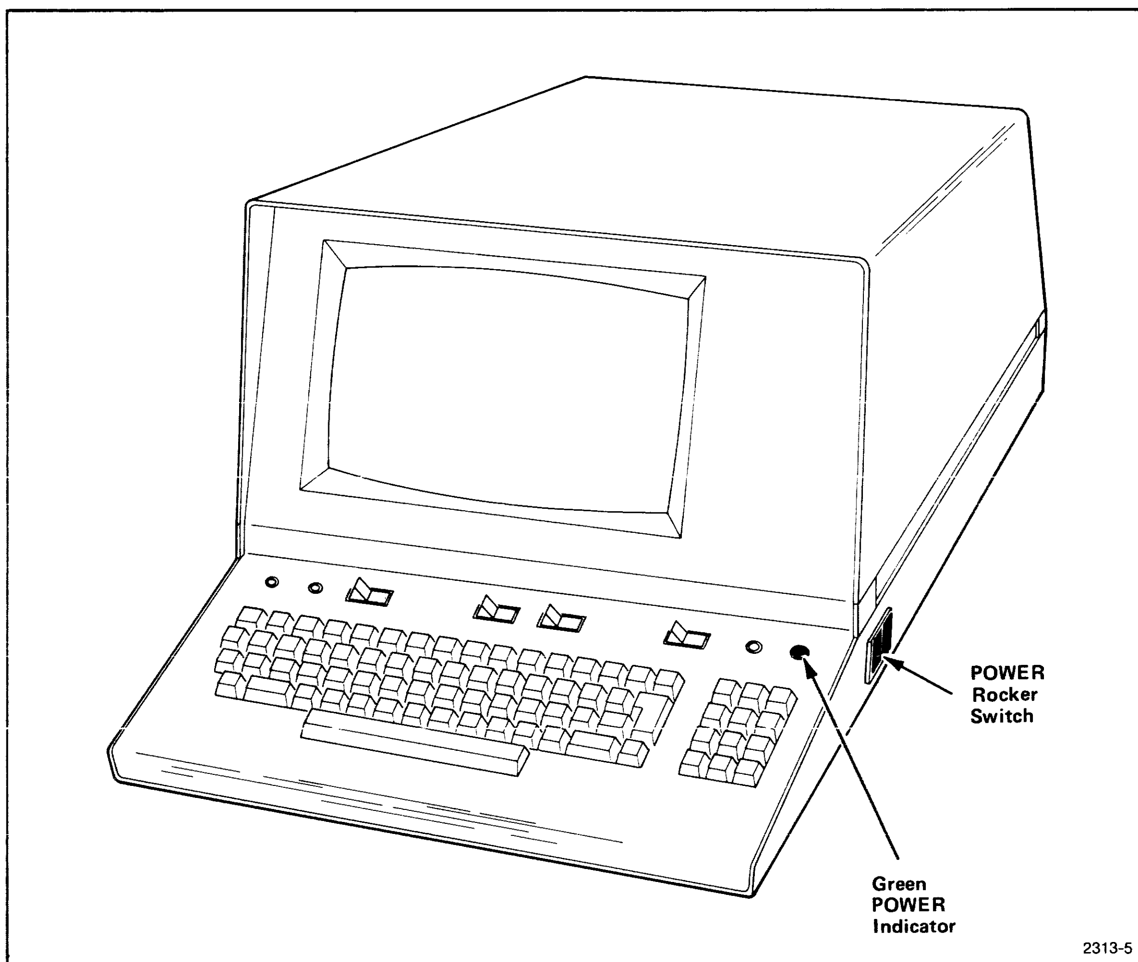


Fig. 2-2. Optional CT8100 CRT Terminal.

A slide switch is located on the top of the optional CT81Ø1 Printer Terminal, as shown in Figure 2-3. Slide the switch to the rear to turn the terminal power on. The green POWER indicator below the keyboard should light.

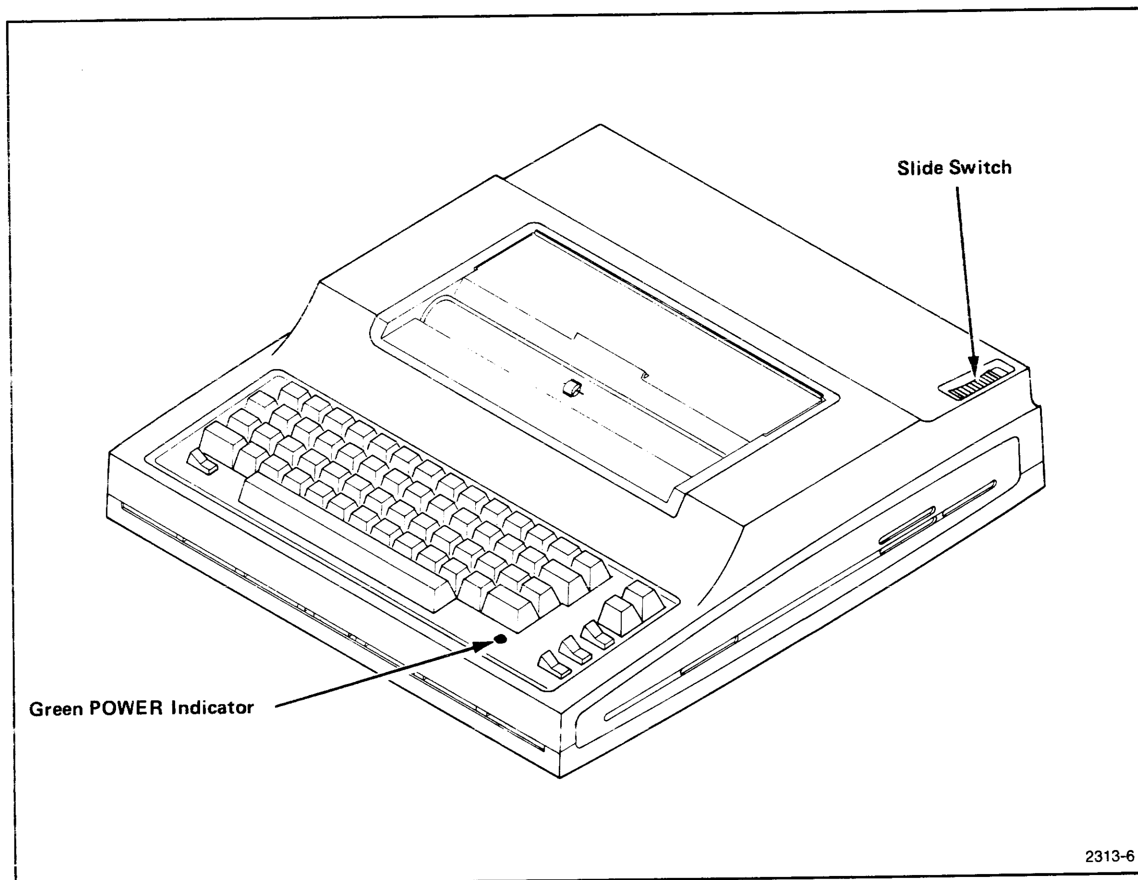


Fig. 2-3. Optional CT81Ø1 Printing Terminal.

Turn on the 8ØØ2 μPROCESSOR LAB

An ON/OFF rocker switch labeled POWER is located on the front panel of the 8ØØ2 μPROCESSOR LAB. Refer to Figure 2-6. Press this switch to its ON position. The SYS, RUN, and PWR indicator lights on the backlighted display should light. Applying power to the 8ØØ2 μPROCESSOR LAB causes an automatic read from drive Ø, and loads TEKDOS into System Memory.

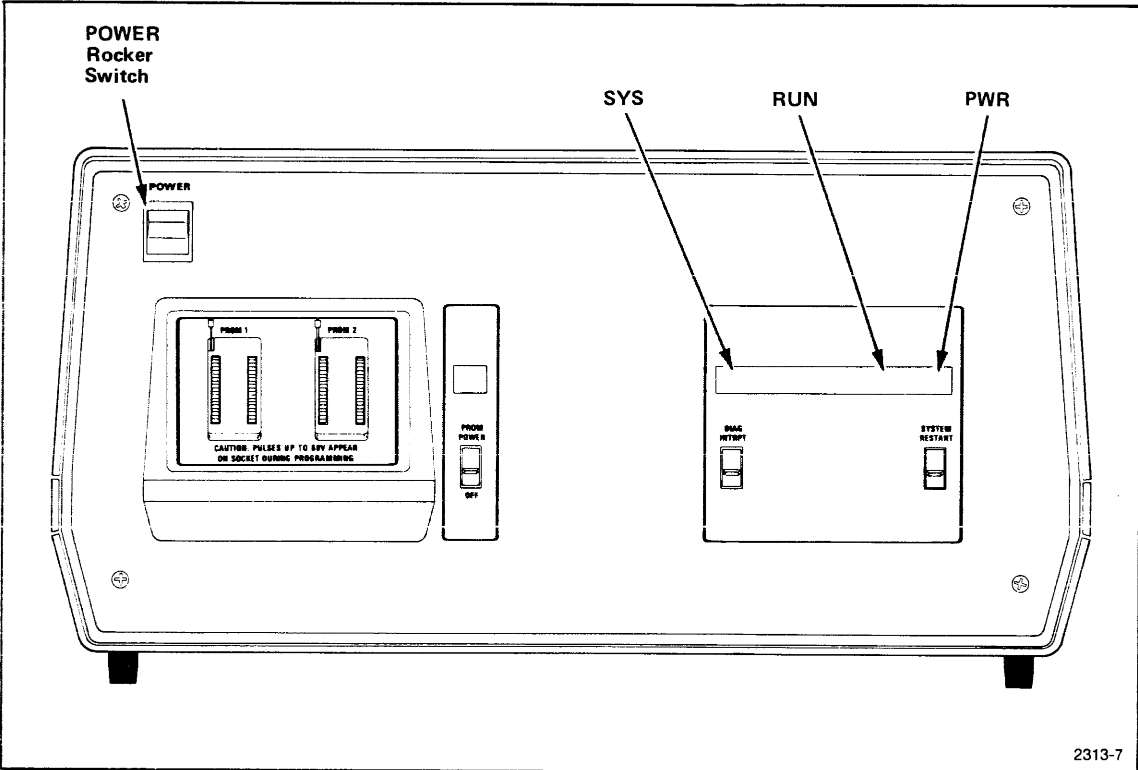


Fig. 2-6. 8ØØ2 μPROCESSOR LAB Front Panel.

Turn on the Line Printer

Apply power to the optional LP8200 Line Printer by pressing the POWER ON/OFF rocker switch to its ON position. Press the ON LINE/OFF LINE rocker switch to its ON LINE position. The LP8200 Line Printer is shown in Figure 2-5.

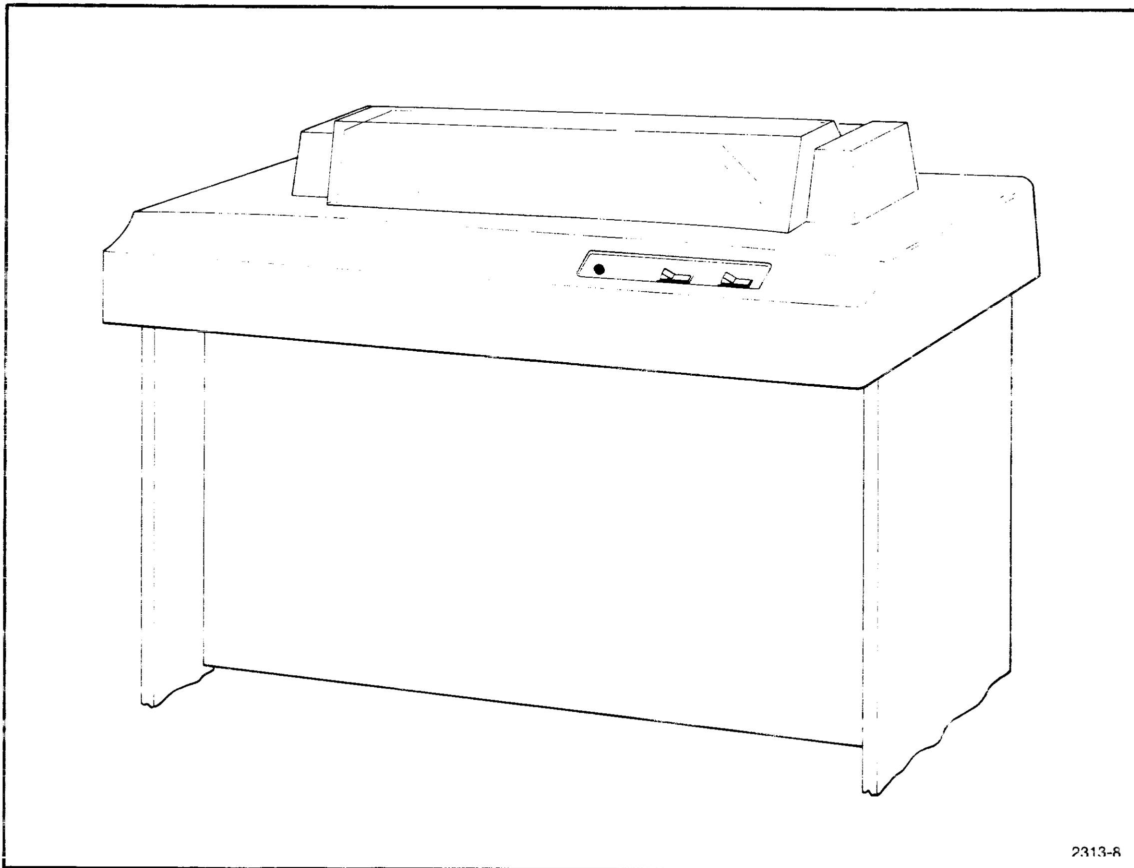


Fig. 2-5. Optional LP8200 Line Printer.

Insert the System Flexible Disc

Insert a system disc into drive Ø on the flexible disc unit. The correct method for inserting a flexible disc is shown in Figure 2-4. Make sure that the flexible disc's label is facing the POWER switch and that the label is the last part of the flexible disc to be inserted into the drive. Close the disc drive door with a firm push to the left, making sure the door snaps shut.

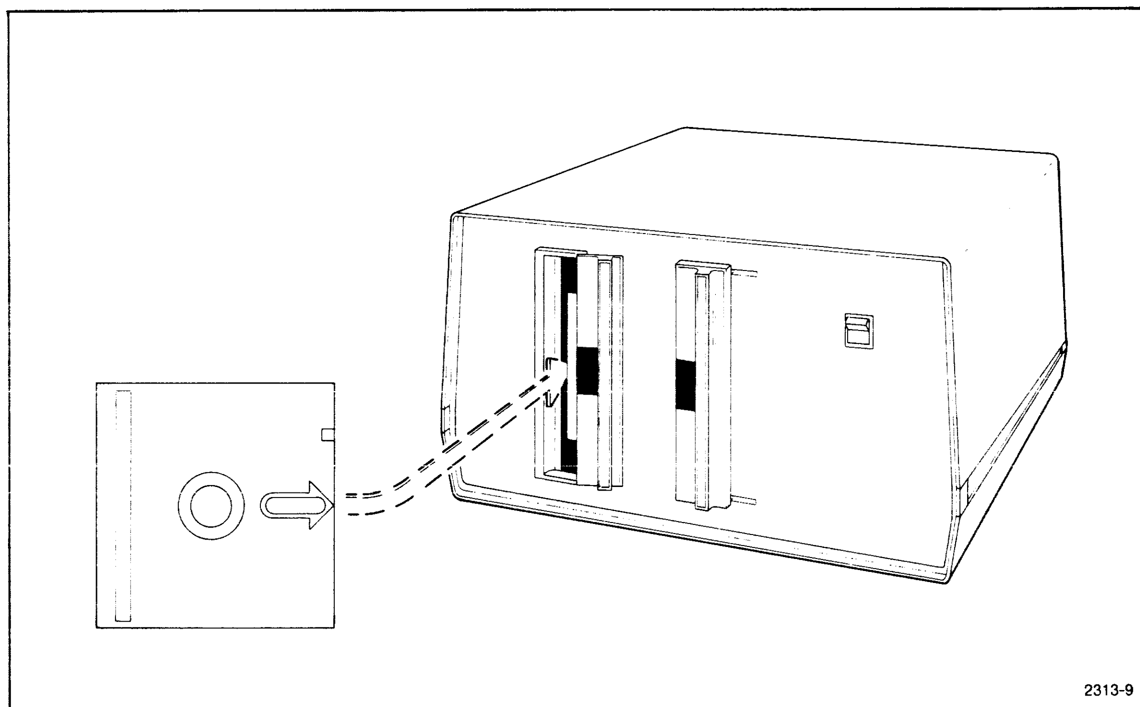


Fig. 2-4. Flexible Disc Insertion.

NOTE

Before the flexible disc can be formatted or have any data written on it, the write-protect notch must be covered with one of the opaque self-adhesive tabs that are provided with the discs. The write-protect notch is the largest (approximately .5 cm by .4 cm) one of the three notches in the bottom edge of the flexible disc. Any opaque adhesive-backed material may be used to cover this notch.

TEKDOS Ready State

As soon as TEKDOS is loaded, a bell rings and a welcoming message is displayed on the console as shown below:

```
>TEKDOS type VERSION 1.6
>
```

In the above example, TEKDOS is an acronym for Tektronix Disc Operating System. TYPE indicates the type of emulator processor enabled, such as 8Ø8Ø, 68ØØ, or Z8Ø. VERSION 1.6 is the version and release number. Your number may be a later release due to software improvements. The ">" character following the welcoming message is the TEKDOS prompt character, which indicates that TEKDOS is ready to accept commands. If the welcoming message does not appear within 15 seconds, toggle the RESET switch on the right side of the 8ØØ2 μPROCESSOR LAB. If the system again does not respond correctly, a damaged or improperly loaded flexible disc or faulty disc drive may be causing the problem. If trouble persists, request service assistance from your Tektronix Customer Service Representative.

LISTING THE FLEXIBLE DISC DIRECTORY

After TEKDOS has been successfully loaded into system memory, you may wish to display the directory contents of your system disc. This is implemented by entering the LDIR command shown below.

```
>LDIR Ø


        SYSTEM FLEXIBLE DISC

        FILE NAME               BLOCKS

        TEKDOS                    16
        COPYSYS                    1

TOTAL FILES                       42
TOTAL BLOCKS USED                 78
BLOCKS AVAILABLE                 226
TOTAL BAD BLOCKS                   Ø

>
```

The "Ø" following the LDIR command indicates that the system disc drive (or drive Ø) is the drive containing the desired directory listing. A carriage return follows this line (and all command lines) and is entered by pressing the RETURN key. The carriage return cues the system to execute the LDIR command line, thus causing a directory listing to be displayed.

The two files that appear in your directory listing are TEKDOS and COPYSYS. As you know, TEKDOS is Tektronix Disc Operating System. COPYSYS is a command file that copies TEKDOS from one flexible disc to another.

SYSTEM POWER-DOWN PROCEDURE

Remove the Flexible Discs

Suppose you now wish to shut down the power on your 8ØØ2 μPROCESSOR LAB. Remove your flexible disc from the flexible disc unit. This again eliminates the danger of possible data destruction when turning off the power. To remove your flexible disc, squeeze the drive door handle and slide the door to the right. Pull out the flexible disc and place the disc back in its storage envelope.

Turn off the Terminal Power

Slide the optional CT81Ø1 Printer Terminal POWER slide switch toward you. The green power indicator below the keyboard should turn off. If you are using the optional CT81ØØ CRT Terminal, push the rocker switch on the terminal's right side to the OFF position. The green POWER indicator below the keyboard should turn off.

Turn off the Line Printer

For the LP82ØØ Line Printer, press the ON LINE/OFF LINE rocker switch to its OFF LINE position. Press the POWER ON/OFF rocker switch toward its OFF position.

Turn off the 8ØØ2 μPROCESSOR LAB

Press the ON/OFF POWER rocker switch on the front panel of the 8ØØ2 μPROCESSOR LAB System to its OFF position. At this point, all the indicator lights on the backlighted display should turn off.

Turn off the Flexible Disc Unit

Press the POWER rocker switch on the flexible disc unit's front panel to its OFF position.

The 8002 μPROCESSOR LAB is now shut down.

FLEXIBLE DISC INITIALIZATION

Included with each 8002 μPROCESSOR LAB is one system disc for each emulator processor ordered. Also two blank flexible discs are included to be used for creating system back ups and storing user programs. Before a blank flexible disc is used, it must be formatted and verified. After formatting and verifying the blank disc, files may be duplicated from a system disc to a blank flexible disc.

NOTE

Before the flexible disc can be formatted or have any data written on it, the write-protect notch must be covered with one of the opaque self-adhesive tabs that are provided with the discs. The write-protect notch is the largest (approximately .5 cm by .4 cm) one of the three notches in the bottom edge of the flexible disc. Any opaque adhesive-backed material may be used to cover this notch.

Flexible Disc Formatting

First, power up the 8002 μPROCESSOR LAB, as previously described in the SYSTEM STARTUP PROCEDURE. Insert one of the system flexible discs into drive 0, one of the blank flexible discs into drive 1. Enter the following command line to the keyboard to invoke the flexible disc formatting:

```
>FORMAT 1, NEW FLEXIBLE DISC
```

In this FORMAT command the 1 refers to the disc drive location of the flexible disc to be formatted. NEW FLEXIBLE DISC is an arbitrary flexible disc identifier that names the newly formatted flexible disc. Execution of this command should take approximately three minutes. When the flexible disc formatting process is complete, the following system response should be displayed on the console.

```
*FMT* EOJ
>
```

This response indicates that an End of Job status is reached for the FORMAT command.

Flexible Disc Verification

The flexible disc verification process is necessary for detecting possible bad sectors that might exist on a newly formatted flexible disc. To verify a flexible disc enter:

```
>VERIFY 1
```

In this VERIFY command 1 refers to the disc drive location of the newly formatted flexible disc. Execution of this command should take approximately three minutes. When flexible disc verification is complete, the following system response is displayed on the console:

```
*VER* EOJ
>
```

This response indicates that an End of Job status is reached for the VERIFY command.

Disc Duplication

You may now duplicate your system disc files onto your newly formatted and verified disc. Disc duplication is implemented with the DUP command as follows:

```
>DUP Ø 1 NEW FLEXIBLE DISC
```

The Ø in this DUP command refers to the drive which contains the disc source of the files to be copied. The 1 refers to the destination drive. NEW FLEXIBLE DISC is an arbitrary flexible disc identifier for the flexible disc in drive 1. The duplication process should take approximately five minutes, depending on the quantity of information to be duplicated. When completed, the following system response should appear on the console.

```
*DUP* EOJ
>
```

This response indicates that an End of Job status is reached for the DUP command. The entire contents of the system flexible disc is now duplicated onto the new flexible disc.

In order to verify that a copy of all system disc files is loaded onto the flexible disc in drive 1, you may request a directory listing for drive 1 as follows:

```
>LDIR 1
NEW FLEXIBLE DISC
FILE NAME                BLOCKS
    TEKDOS                 16
    COPYSYS                 1
TOTAL FILES                42
TOTAL BLOCKS USED          78
BLOCKS AVAILABLE          226
TOTAL BAD BLOCKS            Ø
```

TEXT EDITING

The 8ØØ2 μPROCESSOR LAB Text Editor is used to create new source programs or to change existing ones. The text editor is discussed here through the use of the examples of text creation, text storage, text retrieval, and text alteration.

Text Creation

Suppose you conceive and code a program to run on an 8Ø8Ø microprocessor. The program computes the average of four numbers and stores the result in a particular register. You now wish to create a new file, AVERAGE, that contains the source program. Start the text editor by entering:

```
>EDIT AVERAGE/Ø
```

AVERAGE is the source file name and "/Ø" indicates the disc drive where the file is to be output. This command string loads the text editor into program memory and causes command execution. The text editor displays the following:

```
* * EDIT VERSION 1.6 * *
* * NEW FILE * *
*
```

The ending " * " character is the text editor prompt character, indicating that the text editor is ready to accept commands.

The INPUT command allows source code to be entered into your file. Upon entering INPUT, followed by a carriage return, the system response INPUT: should appear as shown below:

```
* INPUT
INPUT:
```

Begin typing source code as follows:

```
START        ORG     ØØ
;
             XRA     A          ;CLEAR ACC
             MOV     B,A
             MOV     H,A
             LXI     D,13FFH    ;LOAD TOP OF MEMORY
             LDAX    D
             DCX     D
             MOV     C,A
             LDAX    D          ;LOAD SECOND NUMBER
             DCX     D          ;DECREMENT POINTER
             MOV     L,A
;
             DAD     B
             LDAX    D          ;LOAD THIRD NUMBER
             DCX     D          ;DECREMENT POINTER
             MOV     C,A
             DAD     B          ;DOUBLE PRECISION ADD
             LDAX    D          ;LOAD FOURTH NUMBER
             MOV     C,A
             DAD     B
             STC                ;SET CARRY
             CMC                ;COMPLEMENT CARRY. I.E. CLEAR IT
;
             MOV     A,H        ;MOVE HIGH ORDER BYTE
             RAR                ;DIVIDE BY TWO
             MOV     H,A        ;SWAP REG. 4
             MOV     A,L
             RAR
             MOV     L,A        ;SWAP REG.
             MOV     A,H
             RAR                ;DIVIDE BY TWO UPPER BYTE
             MOV     A,L        ;LOAD LOWER BYTE
             RAR                ;DIVIDE BY TWO ANSWER IN ACC.
;
             LXI     D,13FFH
             STAX    D
             HALT
             END
```

Text Storage

After the last line of text "END" has been entered, press two carriage returns. The text editor prompt character " * " appears as indicated in the previous program.

Entering FILE transfers all text entered in program memory to a permanent output file. The command FILE then closes the output file and terminates the editing session with a display of the system response * PGM * EOJ. The TEKDOS prompt character ">" announces the return of TEKDOS.

```
*FILE
*PGM* EOJ
>
```

The permanent file AVERAGE has now been created with the text editor.

A disc directory listing verifies that the new file AVERAGE is stored:

```
>LDIR Ø
NEW FLEXIBLE DISC
FILE NAME                 BLOCKS
    TEKDOS                  16
    COPYSYS                  1
    AVERAGE                  2
TOTAL FILES                 43
TOTAL BLOCKS USED           8Ø
BLOCKS AVAILABLE           224
TOTAL BAD BLOCKS             Ø
```

Text Retrieval

Suppose you wish to retrieve your source code and make corrections with the text editor. Again enter the following:

```
>EDIT AVERAGE/Ø
```

The text editor displays the following:

```
* * EDIT VER 1.6 * *
*
```

To read the source code into program memory enter GET, followed by a number large enough to read in all the lines of code. In this case it is clear that there are under 1ØØ lines of code. Enter the following:

```
*GET 1ØØ
* * EOF * *
*
```

* * EOF * * is a text editor response indicating that all lines of text have been read into program memory and the end of the file has been found. The contents of AVERAGE are now ready for alteration.

Text Alteration

Suppose the correction you wish to make is the addition of a text line just below the first line. First, indicate that you wish to reference the beginning of the text:

```
*BEGIN
START     ORG     ØØ
 *
```

Press the N key followed by a carriage return to verify the current line of text. The system response should be:

```
LINE = 1
*
```

To refer to the line just below the beginning line of text, enter DOWN 1 as shown below:

```
*DOWN 1
              MOV H,A

*
```

The text editor responds by displaying the line of text that is presently below the first line of text, MOV H,A. Text may now be inserted above the line where the current line pointer is positioned.

To insert the needed line of text enter INSERT followed by a space and the line of text. Enter the necessary spaces before the characters MOV B,A to align this text with the other lines.

```
* INSERT     MOV B,A
*
```

Now display the contents of program memory to verify the correction. Again type BEGIN followed by a carriage return. Follow with the TYPE command and a number to indicate the desired number of lines of code to be displayed. The following list should appear on the console:

```
* BEGIN
* TYPE 100

START       ORG     00
;
            XRA     A           ;CLEAR ACC
            MOV     B,A
            MOV     H,A
            LXI     D,13FFH     ;LOAD TOP OF MEMORY
            LDAX    D
            DCX     D
            MOV     C,A
            LDAX    D           ;LOAD SECOND NUMBER
            DCX     D           ;DECREMENT POINTER
            MOV     L,A
;
            DAD     B
            LDAX    D           ;LOAD THIRD NUMBER
            DCX     D           ;DECREMENT POINTER
            MOV     C,A
            DAD     B           ;DOUBLE PRECISION ADD
            LDAX    D           ;LOAD FOURTH NUMBER
            MOV     C,A
            DAD     B
            STC                 ;SET CARRY
            CMC                 ;COMPLEMENT CARRY. I.E. CLEAR IT
```

```
;
                MOV     A,H         ;MOVE HIGH ORDER BYTE
                RAR                 ;DIVIDE BY TWO
                MOV     H,A         ;SWAP REG. 4
                MOV     A,L
                RAR
                MOV     L,A         ;SWAP REG.
                MOV     A,H
                RAR                 ;DIVIDE BY TWO UPPER BYTE
                MOV     A,L         ;LOAD LOWER BYTE
                RAR                 ;DIVIDE BY TWO ANSWER IN ACC.
;
                LXI     D,13FFH
                STAX    D
                HALT
                END
 * * EOF * *

 *
```

Again enter FILE to transfer the text in program memory to the permanent output file.

The text editing session is again terminated with a display of the system response * PGM * EOJ, followed by the TEKDOS prompt character ">" as shown below:

```
*FILE
*PGM* EOJ
>
```

Section 3

COMMAND CONVENTIONS

INTRODUCTION

A command line contains a command and in most cases, one or more parameters with delimiting characters. In this section you will find descriptions of the conventions used in describing the command line structure.

COMMAND NAME

A minimum set of characters is required for each command. This minimum set of characters is underlined in the syntactical description. In the page heading for the command the exact spelling of the command name is given with the short form capitalized.

In addition to the minimum set of characters in the command name, a maximum set (long form) is also given for each command name. Any number of characters in the command name ranging from the short form spelling to the long form spelling may be used as long as the exact spelling is followed.

DELIMITERS

The components in the command line must be separated by delimiters when entered into the computer. A space is used as the main delimiter. The slash "/" is used to delimit a file name and the disc drive number.

The comma may be used as a delimiter in most cases. In the text editor a comma may not be used as a delimiter between a command and the parameters. Two commas are used to specify null or empty fields in a parameter list. Three commas are used to specify two adjacent null fields.

Special delimiters may be specified in some text editor commands. The two text editor commands FIND and SUBSTITUTE use special delimiters that are specified by you. The delimiters you specify in these cases must not be any of the characters in the string being sought or replaced. For example, if you are trying to find the string $15 in the text, you might use the ampersand "&" character as the delimiter in this way:

```
* FIND &$15&
```

SYNTAX

NUDGE { file name } [device / file name [/disc drive]] [{ line number 1 } { line number 2 }] . . .

PARAMETERS

The parameters or controlling conditions of each command line are shown in the syntactical description above. These parameters may be names, numbers, characters or symbols. When the parameter is shown capitalized it must be entered exactly as shown. A parameter shown in lower case letters is a descriptive term to signify the type of entry, as shown above.

Braces and Brackets

When the parameter is enclosed in braces, the parameter must be present in the command line. Parameters enclosed in brackets are optional. Brackets and braces may be nested. The following is an example of braces nested in brackets:

[{ line number 1 } { line number 2 }]

The use of braces and brackets are for syntactical representation and should not be entered as part of the command line.

Stacked Item

Parameters stacked within either braces or brackets indicate that only one of the enclosed items should be selected. In the example below a peripheral device name may be selected or a file name with a disc drive number, but not both.

[device / file name [/disc drive]]

Trailing Dots

A line of dots following a parameter indicate that the parameter may be repeated a number of times. Usually the number of times cannot exceed the length of the display console field. In the example below the line number parameters can be repeated:

[{line number 1} {line number 2}] ...

Numeric Values

A parameter calling for a numeric value may be referenced in the explanation by "n". A parameter range may be referenced by "a" and "b". Multiple numeric parameters may be referenced in order by "a", "b", "c", etc.

Section 4

TEKTRONIX DISC OPERATING SYSTEM

INTRODUCTION

This section describes the Tektronix Disc Operating System (TEKDOS). Topics covered include descriptions of the system, flexible disc and file utilities, control commands, system option commands, and command files. A command summary is also given at the beginning of each subsection.

CONTENTS

SECTION 4 TEKTRONIX DISC OPERATING SYSTEM

INTRODUCTION 4-1
SYSTEM DESCRIPTION 4-3
TEKDOS DISC AND FILE UTILITIES 4-9
FORMAT 4-10
VERIFY 4-12
RENAME 4-13
DUP 4-15
COPYSYS 4-17
LDIR 4-18
DELETE 4-19
CMPF 4-20
COPY 4-21
PRINT 4-24
TEKDOS CONTROL COMMANDS 4-25
Space Bar 4-26
CTRL-Z 4-27
RUB OUT Key 4-28
ESC 4-29
SUSPEND 4-31
CONT 4-32
ABORT 4-33

TEKDOS OPTION COMMANDS 4-34
SYSTEM 4-35
DEVICE 4-36
CLOCK 4-37
ASSIGN 4-38
CLOSE 4-39
EMULATE 4-40
COMMAND FILES 4-41
Command Description 4-42
* 4-46
KILL 4-47
TYPE 4-48

SYSTEM DESCRIPTION

The Tektronix Disc Operating System (TEKDOS) performs flexible disc and file utility functions, data transfer functions, and system and peripheral device control functions. The Tektronix Disc Operating System (TEKDOS) executes in system memory. However, due to the size of TEKDOS only a portion of the system is resident in memory. Other portions are brought into memory from the system disc when needed to execute system commands.

System Memory

The system memory contains both PROM (Programmable Read Only Memory) and RAM (Random Access Memory). The PROM holds the bootstrap loader that initially loads TEKDOS from the flexible disc into the RAM system memory when the POWER switch on the 8002 μPROCESSOR LAB is turned on. This PROM resident bootstrap also loads TEKDOS when the RESTART switch on the front panel of the 8002 μPROCESSOR LAB is toggled.

Some of the Tektronix Disc Operating System is resident on the RAM portion of the system memory during operation. The remainder is loaded from the system disc as needed. The resident part of TEKDOS includes some system commands and the following modules:

- Command Line Processor
- Service Call Processor
- Job Dispatcher
- File Manager
- Device Drivers

A more complete description of these modules will be found in the next section.

The system commands in the system memory include SYSTEM, LOAD, GO, and XEQ. Most of the remainder of the system commands are brought into system memory as needed. System commands EDIT, ASM, and LINK are executed in program memory of the 8002 μPROCESSOR LAB and thus are only invoked through TEKDOS.

RESIDENT TEKDOS MODULES

Command Line Processor: The command line processor operates on commands entered from the system control console or from a command file stored on the flexible disc. The command line processor interprets the commands and prepares a parameter list. Then the function is performed by transferring control to the appropriate resident procedure or by loading and executing a system command.

Service Call Processor: The service call processor operates on internal requests for input and output (I/O) or a TEKDOS function. All of the I/O communication between the emulator processor and system peripherals is performed by the service call processor.

Job Dispatcher: The job dispatcher controls execution of the active jobs in the system. The job dispatcher transfers control to the highest priority job whose I/O operation has been completed or which is ready to run.

File Manager and Device Drivers: The flexible disc drive file manager and other device drivers control operation of the peripheral devices in the system.

MEMORY AREA ASSIGNMENT

Most TEKDOS commands are brought into system memory as needed. System memory contains two memory areas into which the commands are loaded prior to execution. These areas are referred to as memory area 1 and memory area 2. Some system commands are executed in memory area 1 and some in memory area 2. Some system commands require both memory areas for execution.

In the following list the TEKDOS commands are categorized by the memory area in which they are executed:

Memory Area 1		Memory Area 2			Memory Area 1 & 2
COPY	RHEX	ABORT	DEVICE	MOVE	COMM
CSMS	RSMS	ASSIGN	DRT	PATCH	CPROM
DEBUG	VERIFY	BIF	DSTAT	RENAME	LDIR
DUP	WHEX	BKPT	DUMP	RESET	MODULE
FORMAT	WSMS	CLBP	EVT	RTT	RPROM
PRINT		CLOSE	EMULATE	SET	WPROM
		CMPF	EXAM	STATUS	
		CNT	FILL	SUSPEND	
		CONT	KILL	TRACE	
		DELETE	MAP	TYPE	

TEKDOS commands can be executed concurrently as long as their associated programs do not occupy the same memory area. In addition the concurrent execution must be consistent with the current state of the peripheral devices and must not cause any system conflicts.

As an example of concurrent execution of commands, a paper tape is being read into program memory while disc files are being deleted. The following dialog shows how this is carried out:

```
>   RHEX  PPTR
    ESC
>>  DELETE  FILE/1  DATA/1  SOURCE/1
```

The command RHEX PPTR starts the system reading the paper tape into program memory. Striking the ESC key suspends execution of the RHEX command and displays the TEKDOS prompt character >>. The DELETE command is then entered. When you strike the RETURN key the RHEX command continues execution and the DELETE command starts. Note that RHEX executes in memory area 1 while DELETE executes in memory area 2. This allows execution of both commands at the same time.

Files, Devices, and Channels

The Tektronix Disc Operating System (TEKDOS) is a file-oriented system. Understanding a file-oriented system is greatly enhanced by understanding the concepts of a file, a device, and a channel.

A file is a discrete set of data with a logical beginning and a logical end. The files used for the 8002 μPROCESSOR LAB are stored on flexible discs. A file can be accessed through its logical beginning address, a map that indicates the location of the data on the flexible disc and a logical ending address.

A file name must have the following properties:

- The file name must contain at least one character but not more than eight characters.
- The characters in the file name must come from the following set:
  The alphabetic characters (A-Z)
  The numeric characters (Ø-9)
  The special characters ! " # % & ' ( ) * ; = ?
- The file name may not begin with a numeric character.
- The file name must not be one of the reserved names that identify the following physical devices: CONO, CONI, PPTP, PPTR, REMI, REMO, TTYR, and LPT1. (See Table 4-1 for a definition of these names.)
- The file name must be unique to the flexible disc containing the file.

Every flexible disc has a system area called the directory, where system information is kept concerning all the files on the flexible disc. This information includes the file name, disc sectors used, beginning and ending disc addresses, etc. The directory also includes system information that prevents bad disc sectors from being allocated for file usage.

Devices are physical peripherals that provide input and output services for TEKDOS. The eight standard devices are the console output and input devices, the teletypewriter reader, the paper tape reader, the paper tape punch, the line printer, and the remote input and output data communication lines (RS-232-C). These devices have reserved names that you must specify in order to access them. These names appear in Table 4-1.

Table 4-1

List of TEKDOS Device Names

DEVICE NAME	
CONI	Console Terminal Input
CONO	Console Terminal Output
LPT1	Line Printer
PPTP	Paper Tape Punch
PPTR	Paper Tape Reader
REMI	Remote Input (RS-232-C)
REMO	Remote Output (RS-232-C)
TTYR	Teletypewriter Reader

For example, the command:

```
>COPY PPTR LPT1
```

copies the information from the paper tape reader (PPTR) to the line printer (LPT1).

Files may also be viewed as devices and specified either as input or output devices. You refer to a file as a device by using the file name and the disc drive on which the flexible disc with that file is located.

For example:

```
>COPY NAVERAGE/1 LPT1
```

In this example the file named NAVERAGE on disc drive 1 is to be copied to the line printer LPT1. If the file is located on the system disc drive, the drive number usually does not have to be specified.

TEKDOS is only aware of flexible discs that are loaded in the available disc drives. For this reason flexible discs are referred to by drive number and not by name. As an example, flexible discs are loaded in drives Ø and 1 with drive Ø designated as the system drive. A file named NAVERAGE is on drive 1. Drive Ø also contains a file named NAVERAGE. The following command string shows how to copy the file NAVERAGE on disc 1 to the line printer:

```
>COPY NAVERAGE/1 LPT1
```

This command specifies flexible disc drive 1 with the /1 after the file name. TEKDOS assumes that a file resides on the system disc if a number is not appended to the file name and searches only the system disc for that file.

Channels are used by the program running on the emulator processor. A channel can be assigned to a device to enable the emulator processor to perform input and output operations to the device through that channel. The device specified in the assignment may be a physical device or a file.

TEKDOS Commands

The TEKDOS command line consists of the command name and its parameters. Most commands require that parameters be specified. The command is always separated from its parameter by one or more spaces or by a comma. When two or more parameters are present in a command line, the parameters must also be separated by spaces or a comma. The following two command lines are interpreted by TEKDOS in the same way:

```
> LDIR Ø /
> LDIR,Ø,/
```

The command line is entered after the prompt character > is displayed. In the example above, each command line is preceded by the prompt character. LDIR is the command to be executed, the zero "Ø" is the first parameter, and the "/" is the second parameter.

Most TEKDOS commands indicate that they have completed their function by displaying an End-Of-Job message. The form of this message is * id * EOJ where 'id' is the TEKDOS system command identifier and EOJ is the end of job message. Completion of any command causes the TEKDOS prompt character > to be displayed.

TEKDOS DISC AND FILE UTILITIES

You can perform flexible disc and file utilities and move data around the 8002 μPROCESSOR LAB System with these commands.

Command Name	Description	Page
FORMAT	Initializes a flexible disc.	4-10
VERIFY	Finds and catalogs defective blocks on a flexible disc.	4-12
RENAME	Changes the name of a file or flexible disc.	4-13
DUP	Duplicates all files on a flexible disc.	4-15
COPYSYS	Copies the operating system to a blank flexible disc.	4-17
LDIR	Lists directory of a flexible disc.	4-18
DELETE	Removes a file from a flexible disc.	4-19
CMPF	Compares two files.	4-20
COPY	Moves data between system devices.	4-21
PRINT and PRINTL	Prints out lines of data to devices.	4-24

These commands are explained in detail on the following pages.

SYNTAX

FORMAT {disc drive} [name]

PURPOSE

A new flexible disc must be formatted and verified before it can be used by TEKDOS.

The formatting process prepares a blank flexible disc for use with TEKDOS by writing such information as clock bits, sync patterns, track and sector numbers, data patterns and CRC characters on the flexible disc. The formatting process also presets the flexible disc director to reserve tracks 1 through 4 for TEKDOS.

EXPLANATION

The FORMAT command causes the flexible disc on the specified drive to be formatted. The drive specified must not be the designated system drive. For example, if the designated system drive is drive 1, the appropriate command format is FORMAT Ø. In this case the flexible disc on drive Ø is formatted.

The NAME portion of the command is optional but serves to identify the flexible disc. This identification is always displayed when the disc directory is listed. If the NAME is not specified at the time of formatting, a string of blanks is used to identify the flexible disc. The NAME is truncated to 48 characters if more than that are entered.

If the flexible disc is not used for storage of system software, the area reserved for TEKDOS may be freed for other uses after formatting by entering the DELETE TEKDOS/n command. This prevents the use of this flexible disc for system programs unless it is again formatted, in which case the files on the flexible disc are destroyed.

CAUTION

The DELETE TEKDOS command can delete TEKDOS from your system disc. If this happens, the system is non-recoverable and you will need to obtain a new system disc from Tektronix.

Procedure for Formatting a Flexible Disc

NOTE

Before the flexible disc can be formatted or have any data written on it, the write-protect notch must covered with one of the opaque self-adhesive tabs that are provided with the discs. The write-protect notch is the largest (approximately .5 cm by .4 cm) one of the three notches in the bottom edge of the flexible disc. Any opaque adhesive-backed material may be used to cover this notch.

1. Power up the 8ØØ2 μPROCESSOR LAB as previously described in Section 2 of this manual.

2. Insert a system disc into drive Ø of the Flexible Disc Unit.

3. Insert a blank flexible disc into drive 1.

4. Enter the following command string after the prompt character:

```
> FORMAT 1 NAME
```

5. Execution of this command takes approximately three minutes. When the formatting process is completed, TEKDOS responds with:

```
*FMT* EOJ
```

FMT Error Responses

2—Directory write error
9—Invalid drive number
17—Output device assign failure
18—Device in use
47—System area bad

SYNTAX

Verify {disc drive}

PURPOSE

The verification process determines if any sectors on a flexible disc are defective, then records the location of the defective track on the block bit map.

EXPLANATION

This command causes the flexible disc on the specified disc drive to be verified.

This verification process consists of reading every sector on the flexible disc and noting any errors that occur. When an error on a sector is found, the four blocks on the track in which the defective sector resides are recorded in the block bit map. In addition, the track and sector number of the defective sector are printed on the console. When all the sectors have been read, the block bit map is written on the flexible disc.

Whenever files are created and disc space allocation for the file is performed, reference is made to the block bit map and any defective blocks are not allocated.

If a defective sector is detected on tracks Ø through 4 (the TEKDOS area) during the verification process, the process is aborted and an appropriate message displayed on the console.

VER Error Responses:
1—Directory read error
2—Directory write error
9—Invalid drive number
16—Input device assign failure
18—Device in use
47—System area bad

SYNTAX

RENAME {file name 1/disc drive} {file name 2}
or
RENAME {disc drive} {flexible disc identifier}

PURPOSE

The RENAME command has two forms. The first form is used to rename a file and the second form is used to rename a flexible disc.

EXPLANATION

Renaming a File

The first form of the RENAME command causes the name of a file on the specified disc drive to be changed. This form requires that a disc drive number be specified with the OLDFILE name. A disc drive number may be specified with the NEWFILE name, however, it must be the same as the drive number specified for the OLDFILE name. The following is a typical transaction:

```
> REN TEST/Ø TEXT
*REN* EOJ
```

Renaming a Flexible Disc

The second form renames the flexible disc on the specified disc drive with the character string NAME. The identifier NAME is truncated if longer than 48 characters. The following is a typical transaction:

```
> REN Ø MASTER SYSTEM DISC
 *REN* EOJ
```

REN Error Responses

1—Directory read error
2—Directory write error
8—Drive not specified
9—Invalid drive number
12—Invalid file name
13—Input file not found
16—Input device assign failure
18—Device in use
3Ø—Invalid parameter
31—Parameter required
32—Too many parameters
57—File name in use

SYNTAX

DUP { disc drive 1 } { disc drive 2 } [flexible disc identifier]

PURPOSE

The DUP command causes an exact duplicate of the contents of one flexible disc to be created on another flexible disc.

EXPLANATION

The DUP command causes the files stored on the flexible disc on disc drive 1 to be copied onto the flexible disc on disc drive 2. The disc drive number entered for 1 may not be the same as the number used for 2. In addition, the number used for 2 may not specify the system drive.

The NAME portion of the command is optional but serves to identify the flexible disc. This identification is always displayed when the flexible disc directory is listed. If the NAME is not specified at the time of formatting, a string of blanks is used to identify the flexible disc. The NAME is truncated to 48 characters if more than that are entered.

If a disc read or write error occurs during a file copy, the output file is deleted from the flexible disc on drive 2, a warning message is displayed and the DUP process continues with the next file.

The flexible disc on drive 2 should be verified before the DUP command is executed in order to establish the block bit map for the disc.

Uses for the DUP command include making backup discs for your system. At times you may want to make a backup disc for a non-system disc. If your system has more than two disc drives, the DUP command can be used if you have the system disc in the system drive and the other two flexible discs in other drives. The system disc is needed to provide the DUP command.

If you have the minimum system you have only two disc drives and it might seem to be impossible to DUP a non-system disc. The following procedure allows you to use your system to duplicate non-system discs.

Procedure for Duplicating Non-system Discs

1. Insert the system disc into drive zero and toggle the RESET switch.

2. After the prompt character is displayed on the console, enter the following command:

```
>DUP 1 Ø
```

3. The following error will be displayed along with an End-Of-Job message:

```
*DUP* ERROR 9
*DUP* EOJ
```

   Now the DUP command program is resident in system memory and the system disc is not required for execution of the DUP command.

4. Insert the non-system disc to be copied into disc drive one.

5. Remove the system disc from drive zero and insert a blank flexible disc into drive zero.

6. Enter the following command:

```
>SYS 1
```

7. Next enter the command:

```
>DUP 1 Ø
```

8. When the files on the flexible disc in drive one have been copied onto the blank disc drive Ø, the following message will be displayed on the console:

```
*DUP* EOJ
```

DUP Error Responses:

1—Directory read error
2—Directory write error
6—Read error, DUP continues
7—Write error, DUP continues
9—Invalid drive number
16—Input device assign failure
17—Output device assign failure
21—Channel assign failure

SYNTAX

COPYSYS {disc drive 1} {disc drive 2}

PURPOSE

COPYSYS is a command file that copies the TEKDOS operating system from one flexible disc to another.

EXPLANATION

The COPYSYS command causes the system files on the flexible disc mounted on disc drive 1 to be copied to the flexible disc on disc drive 2. The COPYSYS command is entered after the TEKDOS prompt character > is displayed. The following example illustrates the use of this command to copy the TEKDOS system from a disc on disc drive Ø to a disc on disc drive 1:

```
>COPYSYS Ø 1
TYPE OFF
COPYSYS COMPLETED
>
```

The underlined portion is entered from the keyboard. When execution of the command has been completed, the following files have been copied over:

- The resident TEKDOS binary load file.
- All TEKDOS overlays including the assembler and the editor.
- The COPYSYS command file.

The operating system should be copied onto a flexible disc before any other files to achieve the most rapid system response to commands. This allocates the system files to the tracks on the outside of the disc and minimizes read head movement when the commands are brought into memory.

SYNTAX

LDIR [disc drive] [.] [/] [device name / file name/disc drive]

PURPOSE

The LDIR command causes the contents of a flexible disc directory to be sent to the specified device.

EXPLANATION

The LDIR command program lists the contents of the disc directory that is mounted on the specified disc drive. If the disc drive is not specified, the system disc directory is listed. When a decimal point "." is specified as a parameter, the command program includes all of the TEKDOS system files. When a slash "/" is specified as one of the parameters, space allocation information and file identification information are included in the listing.

Any one of the four output device names CONO, LPT1, PPTP or REMO is a valid entry (see Table 4-1). In addition, any valid file may be named as an output device; however, the operator must specify the disc drive on which the flexible disc with that file is located. The listing of the directory to a file will overlay any data in that file. A new file will be created if a file with the specified name does not exist.

* DIR * Error Responses

1—Directory read error
7—Device write error
10—Overlay load failure
15—Invalid output device
17—Output device assign failure

SYNTAX

DELETE { file name/disc drive } [file name/disc drive] . . .

PURPOSE

The DELETE command is used to delete specified files from a flexible disc.

EXPLANATION

The DELETE command causes the file named to be deleted from the specified disc drive. [file name] ... indicates that more than one file can be specified for deletion in one command line, however each file specified must have a disc drive number associated with it.

Upon execution of the DELETE command, each file specified in the parameter list is deleted from the directory of the flexible disc on which it resides. The sector blocks allocated to the deleted file are released for reallocation.

DEL Error Responses

2—Directory write error
8—Drive not specified
9—Invalid drive number
12—Invalid file name
13—File not found
18—Device in use
21—Channel assign failure
3Ø—Invalid parameter
31—Parameter required
61—File in use

SYNTAX

CMPF {file name 1 [/disc drive]} {file name 2 [/disc drive]} [output device name | output file name/disc drive]

PURPOSE

The CMPF command is used to compare two files and list their differences on the specified device.

EXPLANATION

The CMPF command compares the first file specified on a byte for byte basis with the second file specified. Any differences are printed out on the device named. If a device is not named, the output is sent to the console (CONO).

The disc drive number for either file is optional, but if either file is not stored on the system disc then the drive number must be specified.

Any one of the four output device names CONO, LPT1, PPTP, or REMO is a valid entry. In addition, any valid file may be named as an output device. The output to a file overlays any data in that file. A new file is created if a file with the specified name does not exist.

If the first file named is an ASCII file, the output will be line oriented; otherwise the output will be block oriented. When differences are found, the line or block number is printed out, followed by the first file name and the line or block of text. Then the second file name is printed out with its corresponding line or block of text. The comparison continues until an end-of-file marker is reached on either file.

CMP Error Responses

6—Device read error
1Ø—Overlay load failure
13—Input file not found
14—Invalid input device
15—Invalid output device
16—Input device assign failure
17—Output device assign failure
31—Parameter required

SYNTAX

```
      { input device name          }  [ input device name          ]
COPY  { input file name/disc drive }  [ input file name/disc drive ] . . .

      { output device name          }
      { output file name/disc drive }
```

PURPOSE

The COPY command is used to transfer data from one device or file to another.

EXPLANATION

The COPY command transfers data from the input device or file to an output device or file. More than one input device or file may be specified; however, the output device or file may not be used as an input device or file.

Data is transferred from the input device or file to the specified output device until an end-of-file condition is encountered on the input. If more than one input device or file is specified, the data is transferred to the output device or file in the following manner:

1. The data from the first input device or file is transferred to the output device or file until an end-of-file condition is reached.

2. The data from the second input device or file is transferred to the output device or file and concatenated directly to the first set of data.

3. The data from the third input file or device is then transferred to the output device or file, etc.

4. When the end-of-file condition is encountered in the last input device or file the output device or file is closed.

When an ASCII file is being input from one of the system devices, (CONI, PPTR, or REMI), the CTRL-Z character is interpreted as the end-of-file condition.

EXAMPLE

A programmer used the COPY command to create a file directly from the console as shown below. In this case a partial 68ØØ Assembly program was created from the keyboard. It was named AVER.

```
        > COPY CONI AVER
 1      START   LDX       13FCH   ;LOAD LOCATION OF NUMBER
 2              CLR   B           ;ZERO B
 3              LDD   A  X        ;LOAD A INDEXED
 4              ADD   A  1,X      ;ADD 2ND NUMBER
 5              ADC   B  Ø
 6              ADD   A  2,X      ;ADD 3RD NUMBER
 7              ADC   B  Ø
 8              ADD   A  3,X      ;ADD 4TH NUMBER
 9              ADC   B  Ø
 1Ø
        * COP * EOJ
```

After entering line 9, the programmer entered a carriage return. At this point, wishing to stop, he entered a CTRL-Z and terminated the COPY transaction. If he had entered the CTRL-Z at the end of line 9 instead of the carriage return, the blank line would not have been added to the file at line 10.

Later, the programmer had another segment of 68ØØ code on paper tape. The following transaction shows how he entered the data into a file called AGE.

```
         > COPY PPTR AGE
          *COP* EOJ
         >COPY AGE CONO

        AVG     ROR   B           ;DIVIDE BY TWO
                ROR   A
                ROR   B           ;DIVIDE BY TWO
                ROR   A           ;RESULT IN ACC A
                STA   A  3,X
         *COP* EOJ
```

At this point the programmer wanted to combine these two programs and store the result in a file named AVER68Ø. The transaction follows:

```
>COPY AVER AGE AVER68Ø
*COP* EOJ
```

The programmer then used the COPY command to print out the AVER68Ø file contents on the console display as shown below:

```
      >COPY AVER68Ø CONO
1     START    LDX      13FCH     ;LOAD LOCATION OF NUMBER
2              CLR   B            ;ZERO B
3              LDD   A  X         ;LOAD A INDEXED
4              ADD   A  1,X       ;ADD 2ND NUMBER
5              ADC   B  Ø
6              ADD   A  2,X       ;ADD 3RD NUMBER
7              ADC   B  Ø
8              ADD   A  3,X       ;ADD 4TH NUMBER
9              ADC   B  Ø
1Ø
11    AVG      ROR   B            ;DIVIDE BY TWO
12             ROR   A
13             ROR   B            ;DIVIDE BY TWO
14             ROR   A            ;RESULT IN ACC A
15             STA   A  3,X
      *COP* EOJ
```

COP Error Responses

6—Input read error
7—Output write error
13—Input file not found
14—Invalid input device
15—Invalid output device
16—Input device assign failure
17—Output device assign failure
3Ø—Parameter error

SYNTAX

{PRINT / PRINTL} {file name} [device / file name/disc drive] [{line number 1} {line number 2}]

PURPOSE

The PRINT and PRINTL commands transfer lines of data from an input file to an output device or file. In addition, the PRINTL command also numbers each line in succession.

EXPLANATION

The PRINT or PRINTL commands transfer lines of data from the specified input file to the specified output device or file. When the output device or file is not specified, the data lines are printed to the line printer LPT1.

The line numbers must be greater than or equal to one and less than 32,768. Line number 2 must be greater than or equal to line number 1. When the line range is specified using 1 and 2, only the lines from 1 through 2 are transferred. For example, when the first number is 4 and the second number is 7, then lines 4, 5, 6, and 7 are transferred from the specified file. If only the first line number is specified, all lines from the first line in the file through the specified line are transferred. When a line range is not specified the entire file is transferred.

When the PRINTL command form is used the lines are numbered as they are transferred.

PRN Error Response

6—Input read error
7—Output write error
13—Input file not found
14—Invalid input device
15—Invalid output device
16—Input device assign failure
17—Output device assign failure
3Ø—Invalid parameter

TEKDOS CONTROL COMMANDS

The following control commands and special characters are used to control execution of both system and emulation programs.

Command Name	Description of Command	Page
Space Bar	Suspends console display.	4-26
CTRL Z	Issues an end-of-file character during an ASCII read operation.	4-27
RUBOUT	Deletes the last character from the line buffer.	4-28
ESC	Suspends or terminates any action.	4-29
SUSPEND	Suspends program execution.	4-31
CONT	Continues execution of suspended programs.	4-32
ABORT	Terminates program execution.	4-33

SYNTAX

Space Bar

PURPOSE

The space bar is used to halt the display output to the console or cause the display to continue.

EXPLANATION

Striking the space bar during console display temporarily halts the display. Striking the space bar again causes the display to continue. The display may be halted and continued as many times as needed.

SYNTAX

CTRL Z

PURPOSE

The CRTL-Z (control-Z) provides an end-of-file character during as ASCII read operation.

EXPLANATION

A CRTL-Z is sent by holding down the CTRL (control) key while striking the Z key.

The CTRL-Z is treated as an end-of-file character when an ASCII read is being performed from the console or other system input device.

CTRL-Z does not send a visual character to the console.

RUB OUT Key

SYNTAX

RUB OUT Key

PURPOSE

The RUB OUT key is used to delete an incorrectly entered character.

EXPLANATION

Striking the RUB OUT key deletes the last character input from the console keyboard. This function deletes the last character in the line buffer and echoes that character to the display. If more than one character has been entered incorrectly, the RUB OUT key may be used to delete each character in the string. The entry can then be completed as if the incorrect characters were never entered.

Example

If when entering the command line to copy the file AVER to the console, you enter the device name by mistake before the file name, the entry may be corrected with the RUB OUT key as follows:

```
> COPY,CONOONOCAVER,CONO
```

The underlined portion of the command line above shows the incorrect entry and the effect of using the RUB OUT key four times followed by the correct entry.

SYNTAX

ESC Key

PURPOSE

Striking the ESC key (escape) causes suspension or termination of program execution and returns control to TEKDOS.

EXPLANATION

Striking the ESC key twice (ESC ESC) suspends all active programs. A program suspended by this means (ESC ESC) does not resume execution unless a CONT (continue execution) command is entered from the console.

The response to striking the ESC key once varies depending on whether input is being performed or a program is being executed.

ESC During Console Input

Striking the ESC key during an input operation produces differing results as follows:

- If a TEKDOS command line is being entered, the system deletes the command line and responds to the display console with a double prompt character >>.
- If the EXAM command is being executed, the command is terminated and a double prompt character is displayed. Any memory locations that were altered prior to striking the ESC key remain altered.
- If the EDIT command is being executed, the Editor prompt character * is displayed. When the Editor is in the Input mode, striking the ESC key deletes the current line being entered and moves the display cursor to the next line. In this case a prompt character is not displayed.
- If a user program is being executed, the response depends on the function that has been programmed for the ESC key.

ESC During Program Execution

Striking the ESC key during the execution of a TEKDOS command causes the execution of that command to pause. Execution of the command may be continued either by striking the RETURN key or by entering the CONT * command.

The execution of the TEKDOS commands LDIR, CMPF, TRACE, STATUS, EXAM, and DUMP will be terminated by striking the ESC key instead of just being paused.

Suspend

SYNTAX

```
           ( program name )
SUSPEND    <  *           >
           ( /            )
```

PURPOSE

The SUSPEND command suspends the execution of active programs.

EXPLANATION

The SUSPEND command may be used with any active program except DEBUG.

This command is mainly used in conjunction with the command file capability. Inserting the SUSPEND command in a command file suspends system operation and allows some required user action, such as inserting a special flexible disc into one of the drives.

The SUSPEND command must be accompanied by one of the parameters.

- SUSPEND,* suspends all active programs.
- SUSPEND,/ suspends any active user program.
- SUSPEND,program name suspends the specified program.

SUS Error Responses

24—Job not active
26—Job not suspended
31—Parameter required

SYNTAX

```
        { program name }
CONT    { *            }
        { /            }
```

PURPOSE

The CONT command continues the execution of a suspended program.

EXPLANATION

A suspended program may be continued by entering the CONT command. TEKDOS returns execution to the suspended program at the point where execution ceased.

When the ESC key has been used to terminate execution of the TEKDOS commands LDIR, CMPF, TRACE, STATUS, EXAM and DUMP, they cannot be continued.

The CONT command must be accompanied by one of the following parameters:

- CONT, * causes execution of all active programs to be resumed.
- CONT,/ causes execution of an active user program to be resumed.
- CONT,program name causes execution of the specified program to be continued.

* CON * Error Respones

24—Job not active
25—Job not suspended
31—Parameter required

SYNTAX

```
           { program name }
ABORT      { *            }
           { /            }
```

PURPOSE

The ABORT command terminates execution of an active program.

EXPLANATION

The ABORT command causes the execution of an active program to be terminated. The ABORT command must be accompanied by one of the parameters.

- ABORT, * causes all active programs to be terminated.
- ABORT,/ causes an active user program to be terminated.
- ABORT,program name causes the named program to be terminated.

ABT Error Responses

24—Job not active
31—Parameter required

TEKDOS OPTION COMMANDS

You may set the value of various system options. These options remain in effect during all subsequent operations until removed or changed.

Command Name	Description of Command	Page
SYSTEM	Designates the system drive.	4-35
DEVICE	Specifies the device status to TEKDOS.	4-36
CLOCK	Specifies the real time clock status.	4-37
ASSIGN	Connects user channels for I/O devices.	4-38
CLOSE	Disconnects user channels from I/O devices.	4-39
EMULATE	Activates the emulator processor and sets the emulation mode.	4-40

SYNTAX

SYSTEM {disc drive}

PURPOSE

The SYSTEM command is used to specify the disc drive to be used as the system drive.

EXPLANATION

The SYSTEM command allows you to designate any disc drive as the system drive. The default value for the system drive is Ø.

At power-up or on reset the system selects the system drive as the first disc drive that contains a flexible disc. This search starts with drive Ø.

DOS Error Responses

9—Invalid drive number

SYNTAX

DEVICE {device name} {U / D}

PURPOSE

The DEVICE command informs TEKDOS of the availability of a peripheral device.

EXPLANATION

The DEVICE command specifies the availability of the named device. The device named must be one of the system device names (CONI, CONO, LPT1, PPTP, PPTR, REMI, or REMO; see Table 4-1).

The second parameter, either U or D, must be specified. If U is specified, the system is informed that the device is up and available for use. If D is specified, the system is informed that the device is down and not available for use.

DEV Error Responses

30—Invalid parameter
31—Parameter required
52—Invalid device

SYNTAX

CLOCK { ON | OFF }

PURPOSE

The CLOCK command enables or disables the real time clock.

EXPLANATION

The command CLOCK,ON enables the 100 millisecond real time clock. The real time clock is synchronized with the system clock and is available out of system memory for use by user programs.

The power-up default value of the CLOCK command is OFF.

NOTE

The real time clock should be disabled when running a prototype system with the emulator processor in either emulation mode 1 or mode 2.

CLK Error Responses

30—Invalid parameter
31—Parameter required

SYNTAX

```
                                 { device name          }
ASSIGN   { channel number }      { file name/disc drive }

         [                 { device name          } ]
         [ channel number  { file name/disc drive } ] . . .
```

PURPOSE

The ASSIGN command causes the connection of an I/O channel to the specified device.

EXPLANATION

The ASSIGN command causes an emulator I/O channel to be connected to a device. The channel number must be in the range from 0 to 7. The device named may be a file on the flexible disc or one of the system device names (CONI, CONO, LPT1, PPTP, PPTR, REMI, or REMO; see Table 4-1).

Any legitimate file may be named as a device. A new file is created if a file with the specified name does not exist.

ASN Error Responses

1—Directory read error
9—Invalid drive number
12—Invalid file name
19—Invalid channel number
2Ø—Channel in use
21—Channel assign failure
31—Parameter required

SYNTAX

CLOSE {channel number} [channel number] . . .

PURPOSE

The CLOSE command disconnects the specified I/O channel from the specified device.

EXPLANATION

The CLOSE command disconnects the channel from the device that was created by the ASSIGN command.

CLS Error Responses

2—Directory write error
7—Device write error
19—Invalid channel number
31—Parameter required
62—Device not operational
64—Invalid flexible disc

SYNTAX

EMULATE { operational mode }

PURPOSE

The EMULATE command activates the emulator processor and sets the mode of operation.

EXPLANATION

The EMULATE command activates the emulator processor and sets the mode in which it will operate. The possible values for the operational mode are:

Ø — System mode. Uses program memory, system I/O, and system clock.

1 — Partial emulation mode. Uses program memory, user prototype memory, prototype I/O and user clock.

2 — Full emulation mode. Uses user prototype memory, prototype I/O and user clock.

(Note that in mode 2 the TRACE JUMP option is not available.)

The emulation mode may be changed while the DEBUG command is active. However, changing the emulation mode while a user program is being executed causes execution to be aborted.

EMU Error Responses

31—Parameter required
32—Too many parameters
54—Invalid mode
56—Invalid device address

COMMAND FILES

Command files provide you with the capability of multiple command execution by entering a single command. You store the desired TEKDOS commands in the desired order on one file. Then enter that file name to TEKDOS as if the file name were a system command. The commands are executed in sequence using the specified parameters.

This section contains the following:

Command Description		4-42
*	Prefaces comment statements.	4-46
KILL	Aborts a system command on error condition.	4-47
TYPE	Prints the system command being executed.	4-48

SYNTAX

{file name} [/disc drive] [parameter [parameter] . . .]

PURPOSE

Command files provide you with the capability of executing a sequence of system commands by entering a single command.

EXPLANATION

The command file is identified by a single name that must conform to the file naming conventions specified in the section under SYSTEM DESCRIPTION.

When the command file is not resident on the system disc, the disc drive must also be specified. This value defaults to the system disc.

Parameters specified in the command line may be used by the system commands that make up the command file. For example, you may specify a program file name as a parameter in the command line and then have several system commands within the command file use or modify that file. Later you can have that same command file perform the same action on another program file.

Command files cannot be nested but they can be chained. That is, if the last system command in a command file is the name of another command file, the command file being executed is terminated and the next command file is started. The parameters are passed from one command file to another in the same way they are passed to the system commands within the command file.

A command file may be created either using the text editor or using the COPY command.

Examples

A command file may be created by using either the TEKDOS command COPY or the text editor. The following is an example of creating a file with the COPY command:

```
> COPY CONI LISTALL
LDIR,Ø,LPT1
LDIR,1,LPT1
LDIR,2,LPT1
LDIR,3,LPT1 CTRL-Z
```

The COPY command line says to COPY input from the console input CONI to the file named LISTALL. The CTRL-Z is the end-of-file marker and returns control to the TEKDOS monitor.

The following is an example of creating a file with the Text Editor:

```
>EDIT LISTALL
* * EDIT VERSION 1.6 * *
* * NEW FILE * *
* INPUT
INPUT:
LDIR,Ø,LPT1
LDIR,1,LPT1
LDIR,2,LPT1
LDIR,3,LPT1

*FILE
*PGM* EOJ
```

The underlined portions above are entered from the console keyboard, all else is a computer response. The EDIT LISTALL command creates a temporary file which has data put into it with the INPUT command.

After the command file has been entered, striking the RETURN key a second time returns control to the text editor from INPUT. Then issuing the command FILE stores the entered data onto the file LISTALL and returns control to the TEKDOS monitor.

In the following example, the command file named LISTALL contains the following lines:

```
LDIR,Ø,LPT1
LDIR,1,LPT1
LDIR,2,LPT1
LDIR,3,LPT1
```

The command line that invokes this command file is:

```
>LISTALL
```

Execution of this command file results in the directories of each flexible disc mounted on disc drive Ø through 3 being printed on the line printer in order. If a flexible disc is not mounted on one of the disc drives an error results and execution of the command file is halted.

The following change in the command file LISTALL allows the execution to continue after the error is detected:

```
KILL OFF
LDIR,Ø,LPT1
LDIR,1,LPT1
LDIR,2,LPT1
LDIR,3,LPT1
```

If the system drive is Ø but the command file LISTALL is on the flexible disc mounted on disc drive 2, you enter the following command line for execution:

```
>LISTALL/2
```

Parameters may be entered in the command line; however, the order of entry is important. Each parameter entered is referenced by its position in the command line. When the following command line is entered:

```
>LISTALL,.,/
```

and the command file contains:

```
KILL OFF
LDIR,Ø,LPT1,$1,$2
LDIR,1,LPT1,$1,$2
LDIR,2,LPT1,$2
LDIR,3,LPT1,$1
```

The result is that the "." (the first parameter) replaces each $1 and the "/" (the second parameter) replaces each $2 in the system commands within the command file. The result is the same as when the following system commands are executed:

```
KILL OFF
LDIR,Ø,LPT1,.,/
LDIR,1,LPT1,.,/
LDIR,2,LPT1,/
LDIR,3,LPT1,.
```

SYNTAX

* (The Asterisk)

PURPOSE

The asterisk * is used to insert comments into the job flow of the command file.

EXPLANATION

The asterisk is used to insert comments into the job flow of the command file. The asterisk is entered into the first character position and must be followed by a space. The comment inserted is printed on the console as its turn comes up in the job flow. The print out of the comment may be inhibited by invoking the TYPE OFF command.

Comment lines must be entered on their own lines and not entered on a line with a system command.

SYNTAX

KILL { ON / OFF }

PURPOSE

The KILL command causes termination of command file execution upon detection of an error.

EXPLANATION

After the KILL ON command has been invoked, a command file will be terminated if an error is encountered during the execution of any of the system commands within that file.

The KILL OFF command allows execution of a command file to be continued starting with the next system command in the file.

The KILL command defaults to ON.

KIL Error Responses

30—Invalid parameter
31—Parameter required

SYNTAX

```
      { ON  }
TYPE  { OFF }
```

PURPOSE

The TYPE command causes each system command in the command file to be printed on the console as the command is interpreted.

EXPLANATION

After the TYPE ON command has been invoked, either from the keyboard or within the command file, the command line for each system command is printed on the console at the start of the command execution.

The TYPE OFF command inhibits the printing of the command line and only error messages are output.

The system defaults to TYPE ON at power-up and reset.

* TYP * Error Responses

30—Invalid parameter
31—Parameter required.

Section 5

TEXT EDITOR

INTRODUCTION

This section describes the capabilities of the 8002 μPROCESSOR LAB Text Editor. General topics of discussion include an introduction to the text editor, text transfer commands, searching and alteration commands, and system utility commands. Detailed descriptions and examples of each command are given to provide a full grasp of text editor capabilities.

CONTENTS

SECTION 5 **TEXT EDITOR**

INTRODUCTION TO THE TEXT EDITOR 5-3
INVOKING THE TEXT EDITOR 5-4
COMMAND CONVENTIONS 5-7
TEXT TRANSFER COMMANDS 5-10
- INPUT 5-11
- INSERT 5-13
- FILE 5-14
- GET 5-17
- PUT 5-21
- COPY 5-26
- TYPE 5-32
- LIST 5-33

SEARCHING AND ALTERATION COMMAND 5-34
- N 5-35
- UP 5-36
- DOWN 5-38
- BEGIN 5-40
- END 5-41
- FIND 5-42
- SUBSTITUTE 5-44
- REPLACE 5-46
- KILL 5-48

UTILITY COMMANDS 5-50
TAB 5-51
TABS 5-53
MACRO 5-54
Space Bar 5-56
ESC 5-57
QUIT 5-58
BRIEF 5-59
? 5-61
AGAIN 5-62

INTRODUCTION TO THE TEXT EDITOR

The 8002 μPROCESSOR LAB Text Editor aids in the creation and modification of programs written in source code. The editor performs these functions by processing key-entered command lines in program memory and storing the resulting data on flexible discs. Each command line specifies one action or series of actions for the text editor to undertake.

The text editor resides in program memory and occupies approximately seven thousand bytes of the memory. This allows approximately 150 60-character lines of program memory to remain for the creation and modification of text in a 16k system.

Two important terms are used throughout this discussion. These are:

Program Memory: Program memory is the storage area that contains the text on which the editor operates. Lines of text are written into and read from program memory by the editor. Text stored in program memory can be viewed in terms of having a beginning and ending line.

Line Pointer: The text editor can operate on any line in program memory. Program code is edited by first locating and examining a line, and then changing, inserting or replacing the line. The text editor keeps track of the line presently being worked on by keeping a pointer at the line to be edited. The line pointer can be positioned upward or downward, depending on the desired program memory location.

This section describes how to invoke the text editor and outlines the editor command description conventions.

INVOKING THE TEXT EDITOR

The text editor is invoked with the TEKDOS command EDIT. Although EDIT is part of the TEKDOS command set, the EDIT command executes out of program memory after being loaded from the system disc. The EDIT command has three forms:

1) EDIT INFILE OUTFILE
2) EDIT OUTFILE
3) EDIT

EDIT INFILE OUTFILE

When EDIT INFILE OUTFILE is used, INFILE is the name of a file that has previously been created and stored on flexible disc. A copy of INFILE is read into program memory for modification, and the original INFILE remains unchanged.

After modification the edited data must be stored back onto the flexible disc. OUTFILE becomes the name of the new file on the flexible disc where the edited text is written after the editing session is terminated.

This form of invoking the editor enables you to modify a file without disturbing the original text. An example of this application is shown in the command line below. A carriage return follows the last character of the command line, EDIT OLD NEW. * *EDIT VER 1.6* * is the editor welcoming message that indicates successful command line loading. * *NEW FILE* * indicates that a new file (in this case NEW) will be created as a result of the editing session. These messages are entered by the editor, not the user. The final prompt character "*" indicates the editor's readiness to accept commands.

```
> EDIT OLD NEW
* * EDIT VER 1.6 * *
* * NEW FILE * *
.
```

In this general form of the text editor command line (EDIT INFILE OUTFILE) INFILE may be more descriptively referred to as the Primary Input file. The final output filename, OUTFILE, is the Primary Output file. To illustrate this concept:

EDIT OUTFILE

When EDIT OUTFILE is used, the editor interpretation is based on whether OUTFILE is an existing file or a new file.

If OUTFILE is an existing file, the file is edited onto itself at termination of the editing session. OUTFILE is both the primary input and the primary output file. This is accomplished by the creation of a temporary file. The temporary file is identified by a filename with the first letter replaced by an *. In the example below, OLD is read into program memory and given the name, *LD. After modification takes place and the editing session is terminated, the new data held in *LD is stored back on the flexible disc. The * is replaced by the first letter of the original file name. In the example below, *LD becomes OLD again. In effect, the original text has been deleted. Since a modification to the original file has occurred, a new file name has not been stored, as in the previous example. Therefore, the editor message * * NEW FILE * * does not appear.

```
> EDIT OLD
* * EDIT VER 1.6 * *
*
```

If OUTFILE is a new file, the name designates the disc file where text is to be written after the editing session is terminated. OUTFILE is the primary output file. There is no primary input file since the process creates a new file. An example is shown below:

```
> EDIT OLD
* * EDIT VER 1.6 * *
* * NEW FILE * *
*
```

The editor response * * NEW FILE * * indicates the creation of a new file.

EDIT

After entering EDIT, lines of text may be read into program memory from a file not named in the EDIT command line. This file type may be referred to as an Alternate Input file. Lines of text in program memory may also be written to a file not named in the EDIT command line. This file type may be referred to as an Alternate Output file.

When entering EDIT, there is no primary input and no primary output file. This form is primarily used for reading text into program memory from alternate input files and for writing text from program memory into alternate output files. In the example below, the editor response * * NO FILES SPECIFIED * * is a reminder of this operational mode:

```
> EDIT
* * EDIT VER 1.6 * *
* * NO FILES SPECIFIED * *
*
```

COMMAND CONVENTIONS

As a prelude to the description of each text editor command, this subsection outlines the conventions and terms used in each command description.

Command Lines

After the editor is invoked in its proper form, the prompt character " * " is displayed to indicate the editor's readiness to accept command lines. The command line has two basic components—the command name and the parameter list. The command identifies the particular action the editor executes. The parameter list identifies the necessary variables for the command to act upon. In some cases there may not be a parameter list. Command lines are followed by a carriage return that signifies execution of the command line. A command line may not exceed 128 characters.

The Space as a Delimiter

The components of the command line are generally separated by delimiters to aid in execution. A space between the command and each parameter separates the two components in most cases. Unlike TEKDOS commands, commas may not be used in place of spaces between commands and parameters.

The n Parameter

The exception to the above convention occurs when a parameter is represented by the symbol "n". The parameter represented by "n" can be directly appended (without a space) to any command.

The symbol "n" generally means "number" and takes two possible forms. These forms include an absolute number form "a" and line range form "b-c". When invoking the KILL command, for example, you can delete the next "a" lines with the command KILL a or delete lines "b" through "c" with KILL b-c. Parameters a, b, and c must be in the range 1 to 32,767.

When using the b-c form, the letters B, E, and C may be used where applicable to refer to the Beginning, Ending, and Current lines in program memory. If used, these letters may not be directly appended to the command. A space after the command is required. A default value "n = 1" is assumed if "n" is omitted except in cases where otherwise specified.

Special Command Characters

The text editor has three special delimiting characters and three special parameter characters. The special delimiting characters are :, <, and >.

The colon character ":" allows several commands to be stacked in one command line. For example, the following command allows you to advance the line pointer to the beginning of program memory and then move the line pointer downward two lines:

```
* BEGIN:DOWN 2
```

When used as a pair, the angle bracket characters "<" and ">" execute a command line repetitively. Implementation of this capability requires the syntax, "n<command>". "n" is the number of times the command line is performed, and the command entered between the angle brackets is the command line to be performed. In the following example the command line between the angle brackets is executed twice:

```
*2<FIND $WD$:SUBSTITUTE $WD$WC$>
```

In this case the first occurrence of the characters WD is found in program memory and then WD is replaced with the characters WC. This action is then repeated a second time for the next occurrence of the characters WD. If the 2 had been omitted in the previous command line, the commands inside the angle brackets would have been performed once. Repetitive commands may be nested to a depth of sixteen levels.

Three other special characters, B, E and C, may be used as command parameters. B, E and C represent the Beginning line, the Ending line and the Current line. Any one of these characters may be entered to define the range parameter in place of the actual line numbers. A space must be used as a delimiter between the command and the range parameter pair.

Line Pointer

The editor maintains a line pointer that is positioned at the line in program memory that is currently being referenced. The line is known as the current line. The line pointer will be designated in this discussion as: →. For example, suppose the text in program memory appears as:

```
  LDAX D
→ DCR E
  MOV C,R
```

The line pointer is currently positioned at the line of text containing DCR E. At the completion of most editor commands the system responds by displaying the line pointed to by the line pointer.

TEXT TRANSFER COMMANDS

The transfer of text in and out of program memory can be accomplished with these commands.

COMMAND NAME	DESCRIPTION	PAGE
INPUT	Allows any number of text lines to be entered into program memory.	5-11
INSERT	Allows single lines to be entered into program memory.	5-13
FILE	Transfers all text in program memory to a permanent file on flexible disc. Terminates the EDIT session and returns control to TEKDOS.	5-14
GET	Read text into program memory from an input file.	5-17
PUT	Writes text from program memory to an output file.	5-21
COPY	Copies the specified number of text lines from the input file to the output file.	5-26
TYPE	Displays lines of text in program memory on the console.	5-32
LIST	Lists the specified text lines on the line printer.	5-33

SYNTAX

INPUT

PURPOSE

The INPUT command allows any number of text lines to be entered into program memory. INPUT is used both when creating new text and when adding to existing text.

EXPLANATION

The INPUT command is entered, followed by a carriage return. The editor response "INPUT:" welcomes you to enter text. Text lines are then entered, each one ending with a carriage return. If a new file is being created, the text entered makes up the entire file on termination of INPUT. On INPUT termination, if a file has been modified, all newly entered text lines precede the line of text where the line pointer was previously located. The original line pointer position remains unaltered, regardless of the number of lines entered. A line of text may not exceed 127 characters. The INPUT command is terminated by entering two carriage returns at the end of the last line of text.

Examples

The editor may be placed in INPUT mode by entering:

```
* INPUT
```

The editor responds with:

```
INPUT:
```

This response indicates that the editor has entered the INPUT mode. Lines of text may now be entered into program memory.

Suppose you wish to modify an already existing file. The text in program memory may appear as follows:

```
   LDAX D
→  DAD B
```

Perform the following sequence and terminate INPUT by pressing two carriage returns after the last line.

```
*INPUT
INPUT:
DCR E
MOV C,A
```

The editor prompt character returns. The text in program memory has now been altered to:

```
   LDAX D
   DCR E
   MOV C,A
→  DAD B
```

To view the text now in program memory advance the line pointer to the beginning and print all lines to the console with the command line:

```
BEGIN:TYPE 1ØØ
```

The text in program memory should be displayed as indicated above.

SYNTAX

INSERT {command line to be entered}

PURPOSE

The INSERT command allows single text lines to be entered into program memory. The command is generally used when adding text lines to existing files.

EXPLANATION

A text line is entered into program memory and precedes the current line where the line pointer is located. The INSERT command is entered, followed by the single delimiting space and the text line to be added. The carriage return terminates the inserted line and the editor prompt character " * " returns. If a null line of text is attempted by entering a carriage return after the single delimiting space, the editor enters INPUT mode rather than INSERT mode.

Examples

Suppose the buffer appears as:

```
  LDAX D
→ MOV C,A
```

The command line below is entered, followed by a carriage return to end the text line as well as terminating the INSERT command. Control returns to the editor with the prompt character " * ".

```
*INSERT DCR E
*
```

The text in program memory is now altered to:

```
  LDAX D
  DCR E
→ MOV C,A
```

SYNTAX

FILE

PURPOSE

Temporary text is transferred from program memory to the primary output file with the FILE command. The command creates a permanent file, terminates the editing session, and returns control to TEKDOS.

EXPLANATION

Temporary text in program memory is transferred to the primary output file beginning at the primary output file pointer. The primary output pointer is then repositioned to the end of the transferred text. Any text remaining between the primary input pointer and the end of the primary input file is then transferred to the primary output file and is positioned below any previously transferred text. Both files are then closed. The editing session is terminated and system control returns to TEKDOS.

Examples

The file NEW is used throughout this example. NEW contains the following lines of text:

```
NEW1
NEW2
NEW3
NEW4
NEW5
```

Suppose you wish to add three lines of code, starting at the third line of text in NEW. You then wish to save this modified text in a file named NEWER. NEWER has never been created before. The command below names NEW as the primary input file and NEWER as the primary output file:

```
> EDIT NEW NEWER
```

All five lines of text in NEW are called into the buffer with the command:

```
*GET 5
```

The line pointer is positioned to the third line of text in the primary input file with the command string,

```
* BEGIN:DOWN 2
```

In this command string BEGIN positions the line pointer to the beginning line in program memory and DOWN 3 moves the pointer downward three lines. The text in program memory now exists as,

```
   NEW 1
   NEW 2
→  NEW 3
   NEW 4
   NEW 5
```

You may now input three lines of text:

```
* INPUT
INPUT:
LINE 1
LINE 2
LINE 3
```

Two carriage returns follow the last text line input to the file. Text in program memory now exists as:

```
   NEW 1
   NEW 2
   LINE 1
   LINE 2
   LINE 3
→  NEW 3
   NEW 4
   NEW 5
```

The new text lines are inserted prior to the line containing NEW 3, where the line pointer was positioned.

The line pointer has moved downward, relative to the number of lines entered. You may now terminate the editing session and create a permanent file named NEWER by entering the command:

```
*FiLE
```

This transfers all text lines that reside in program memory to the primary output file NEWER and then transfers any data remaining between the primary input file pointer and the end of the primary input file to the end of NEWER. After the FILE command is entered, the system response *PGM* EOJ is displayed. System control returns to TEKDOS and the prompt character ">" returns.

NEWER is now a permanent file that contains the text:

```
NEW 1
NEW 2
LINE 1
LINE 2
LINE 3
NEW 3
NEW 4
NEW 5
```

SYNTAX

GET [absolute number of lines / range of lines] [{ PRIMARY INPUT file name / ALTERNATE INPUT file name } [/disc drive]]

PURPOSE

The GET command reads text into program memory from an input file. This command is used when calling lines of text into program memory from a primary input file or an alternate input file for modification purposes.

EXPLANATION

The GET command is followed by an absolute number of lines or range of lines "n". The GET command reads text into program memory from an input file INFILE. INFILE may be either the primary input or alternate input file name. The disc drive number identifies where INFILE is located if the file does not reside on the system disc.

"n" lines of text or an "n" range of text may be read into program memory. "n" must be specified when an alternate input file is specified and does not default to n = 1. "n" need not be specified with a primary input file if a default of n = 1 is desired. A space between GET and "n" is optional.

INFILE specifies the input file that is accessed to provide text for program memory manipulation. INFILE is optional and can be either a primary input or alternate input file. The primary output file may not be used as INFILE.

If INFILE is omitted, text is read into program memory from the primary input file. In this case, the primary input file line pointer position is altered downward according to the number of lines read into program memory. Each GET command causes text to be inserted in program memory just prior to the line pointer.

If the primary input file is specified as INFILE, text is also inserted in program memory just prior to the line pointer.The primary input file line pointer position remains unaltered when the primary input file name is specified in the command.

If an alternate input file is specified, text is inserted just prior to the line pointer. The alternate input file line pointer position remains unaltered when the alternate input file is specified.

The disc drive number where the disc of the primary input or alternate input file resides, must be specified if the input file does not reside on the system disc.

Examples

The examples to follow relate to the PRIMARY INPUT file NEW. NEW contains the following lines of text:

NEW 1
NEW 2
NEW 3
NEW 4
NEW 5

Reading Text Into Program Memory from Primary Input File with Filename Omitted

Suppose the primary input file NEW has been specified with the EDIT command. Presently text has not been read into program memory. When program memory is empty, the line pointer points to line 1. Three lines of text are read into program memory with the command:

```
* GET 3
```

The primary input file line pointer is now positioned at line 4, which contains the text, NEW 4. Subsequent GET commands of this kind would read lines of text into program memory beginning at line 4 in the primary input file. After the GET command above is entered, the text in program memory exists as shown below:

```
  NEW 1
  NEW 2
  NEW 3
→
```

In this case three lines were read. Since the line pointer was originally positioned at line 1, it is now positioned three lines below line 1 to the fourth line in program memory.

Reading Text Into Program Memory from Primary Input File with Filename Specified

Suppose the primary input file NEW has again been specified with the EDIT command. The first three lines have been read into program memory with the GET command. The text and line pointers in both program memory and the primary input file exist as follows:

Program Memory	Primary Input File NEW
NEW 1	NEW 1
NEW 2	NEW 2
NEW 3	NEW 3
→	→ NEW 4
	NEW 5

When the primary input file name, NEW, is specified with the GET command, the line pointer in the primary input file does not move. The data read into program memory is stored ahead of the line pointer in program memory.

The command below reads the two lines NEW 3 and NEW 4 from input file NEW. These lines are stored after the third line in program memory as shown below:

```
*GET 3-4 NEW
```

Program Memory	Input File NEW
NEW 1	NEW 1
NEW 2	NEW 2
NEW 3	NEW 3
NEW 3	→ NEW 4
NEW 4	NEW 5
→	

Now the command GET 2 would read the two lines NEW 4 and NEW 5 from the primary input file NEW and append these lines to the lines already in program memory, as shown below:

Program Memory	Input File NEW
NEW 1	NEW 1
NEW 2	NEW 2
NEW 3	NEW 3
NEW 3	NEW 4
NEW 4	NEW 5
NEW 4	
NEW 5	
→	

Reading Text Into Program Memory from an Alternate Input File

Suppose the primary input file NEW has again been specified with the EDIT command. The first three lines have been read into program memory and the line pointer is positioned below the last line of text. The text in program memory exists as follows:

```
  NEW 1
  NEW 2
  NEW 3
→
```

Suppose you wish to read two lines of text into program memory from an alternate input file named OLD. OLD is not located on the system disc in drive Ø but on a work disc in drive 1.

The file OLD contains the five lines:

```
OLD 1
OLD 2
OLD 3
OLD 4
OLD 5
```

Enter the command:

```
*GET 4-5 OLD/1
```

This action causes the text in program memory to be altered to:

```
NEW 1
NEW 2
NEW 3
OLD 4
OLD 5
```

Note that the text retrieved with the GET command is inserted between the line pointer and the previously entered text. Since the alternate input file OLD was specified in the above GET command, subsequent GET commands from OLD would cause the line pointer to remain positioned at line 1 in the alternate input file.

SYNTAX

```
      [ absolute number of lines ]  [ { PRIMATRY OUTPUT file name      }              ]
PUT   [ range of lines           ]  [ { ALTERNATE OUTPUT file name     }              ]
                                    [ { ALTERNATE OUTPUT device        } [/disc drive] ]

or

      [ absolute number of lines ]  [ { PRIMARY OUTPUT file name       }              ]
PUTK  [ range of lines           ]  [ { ALTERNATE OUTPUT file name     }              ]
                                    [ { ALTERNATE OUTPUT device        } [/disc drive] ]
```

PURPOSE

The PUT command writes text in program memory to an output file or device. This command is implemented in two forms, PUT and PUTK. These forms are valuable when used in conjunction with the GET command for manipulative transfer of text between program memory and output files or devices.

EXPLANATION

The PUT command writes lines of copied text from program memory to an output file or device. The PUTK command writes lines of copied text from program memory to an output file or device and then deletes the corresponding text from program memory. PUT and PUTK write "n" lines of text to an output file or device OUTFILE. OUTFILE may be either an alternate output file or device or the primary output file. The disc drive number identifies where OUTFILE is located if the file does not reside on the system disc.

"n" lines of text or an "n" range of text may be written to OUTFILE. "n" must be specified when an alternate output file is specified and does not default to n = 1. "n" need not be specified when the primary output file is specified if a default of n = 1 is desired. Lines located prior to the current line pointer position may be written to OUTFILE if they are specified as a range of lines. A space between PUT and "n" and PUTK and "n" is optional.

OUTFILE specifies the file where text is to be written. OUTFILE is optional. If OUTFILE is specified, text is written to an alternate output file or device. If OUTFILE is not specified, text is written to the primary output file. If OUTFILE is specified, text copying begins at the line pointer in program memory and is written to the alternate output file or device. In this case text is written starting at the beginning of the alternate output file or device. The file or device is closed when the write is complete. Thus, if OUTFILE already contains text, the old text in the file or device is lost.

If OUTFILE is not specified, text writing defaults to the primary output file. Text is written, beginning at the line pointer in program memory to the primary output file. Each time this occurs text is inserted between the line pointer in the primary output file and any previously inserted text. This allows text to be continually written to the output file in sequential order.

Each text line left in program memory on termination of the editing session is written to the primary output file unless killed prior to termination with the PUTK command. Text lines written to the primary output file with the PUT command are not killed. Therefore, all text lines left in program memory are repeated at the end of the primary output file on termination of the editing session. Text lines written to the primary output file with the PUTK command are killed. Thus with the PUTK command none of the text lines written to the primary output file are repeated on termination of the editing session.

The disc drive number, where the flexible disc of the primary output or alternate output file resides, must be specified if the output file does not reside on the system disc.

Examples

The examples below relate to the file name NEW that contains the following lines of text:

```
NEW 1
NEW 2
NEW 3
NEW 4
NEW 5
```

Writing Text from Primary Input File to Alternate Output File

Suppose the primary input file NEW has been specified with the EDIT command. All text in NEW has been read into program memory with the GET command. The line pointer is positioned at the second line in program memory by entering:

```
*BEGIN:DOWN 1
```

The text in program memory exists as:

```
  NEW 1
→ NEW 2
  NEW 3
  NEW 4
  NEW 5
```

Suppose you wish to create an alternate output file ALT to contain the two lines of text below the line pointer, NEW 2 and NEW 3. Enter the following command line:

```
*PUT 2 ALT
```

This creates the alternate output file ALT on the system disc and writes the following text in the file on termination of the editing session.

```
NEW 2
NEW 3
```

If ALT had been created previously, all old text within ALT would now be lost. All subsequent PUTs to ALT from NEW or other primary input files destroy text.

Text may be read back into NEW from ALT or any other alternate output file with the GET command. When PUT and GET are used together in this manner, they provide a powerful data manipulation tool.

Writing Text from Primary Input to Alternate Output Device

The above two lines of text could have been written to a device, such as the console, with the command:

```
* PUT 2 CONO
```

This command line treats the console as the alternate output device, and the lines are typed on the console display as follows:

```
NEW 2
NEW 3
```

Writing Text from Primary Input to Primary Output File

Suppose the primary input file NEW has again been specified with the EDIT command. All text in NEW has been read into program memory with the GET command. The line pointer is positioned to the fourth line in program memory by entering:

```
* BEGIN:DOWN 3
```

The text in program memory exists as:

```
  NEW 1
  NEW 2
  NEW 3
→ NEW 4
  NEW 5
```

Suppose you now wish to repeat the fourth line at the beginning of the primary output file. The primary output file line pointer always points to line 1 on the first PUT execution. The command:

```
* PUT
```

writes one line of text (in this case NEW 4) above the primary output file line pointer. The primary output file now contains:

```
  NEW 4
→
```

Subsequent PUT commands would write any new text between the last text written and the line pointer.

Since the text written to the primary output file is not killed with each PUT, all text remaining in program memory on termination of the editing session is also written to the primary output file. On termination and saving of the above editing session the text stored in the primary output file is:

```
NEW 4
NEW 1
NEW 2
NEW 3
NEW 4
NEW 5
```

The PUTK Command

If the PUTK command had been used in the above example, the resulting primary output file would contain the following text:

```
NEW 4
NEW 1
NEW 2
NEW 3
NEW 5
```

NEW 4 was not repeated when the remaining text in program memory was written to the primary output file at the end of the session. The original line in program memory was killed when NEW 4 was written to the beginning of the primary output file.

SYNTAX

```
        ( absolute number of lines )   ( ( PRIMARY INPUT file name   )
COPY   <                           >  < <                            >
        ( range of lines           )   ( ( ALTERNATE INPUT file name )

                        )  [ ( PRIMARY OUTPUT file name   )                 ]
                        >  [< ALTERNATE OUTPUT file name  > [ /disc drive ] ]
       [ /disc drive ]  )  [ ( ALTERNATE OUTPUT device    )                 ]
```

PURPOSE

A specified number of text lines are copied from an input file to an output file or device with the COPY command.

EXPLANATION

The COPY command copies a specified number of lines "n" from an input file INFILE to an output file or device OUTFILE. INFILE may be either a primary input or alternate input file. OUTFILE may be either a primary output or alternate output file or device.

"n" may be an absolute number of lines or a range of lines. Lines prior to the line pointer may be copied to OUTFILE if they are specified as a range of lines. A space between COPY and "n" is optional. "n" must always be specified and does not default to n = 1.

INFILE is the file name from which lines of text are copied. INFILE must always be specified in the COPY statement. INFILE may be either the primary input or the alternate input file. Unlike other editor commands, text is always copied directly from the beginning of INFILE, rather than from a line pointer position in program memory. More than one INFILE is disallowed.

OUTFILE is the file name or device name where lines of text are sent from INFILE. OUTFILE is optional and may be either an alternate output file or device, or the primary output file.

If OUTFILE is specified, text is copied from INFILE to an alternate output file or device. When this occurs, lines of text are copied directly to the alternate output file or device. The file or device is then closed when the write is complete. Thus, if OUTFILE already contains text, the old text in the file is lost.

If OUTFILE is not specified, text is copied from INFILE to the primary output file. Each subsequent time this occurs, text is inserted below the previously inserted text. This allows text to be continually written to the output file in sequential order. When all files are closed, text residing in INFILE is written below the copied text. The disc drive number where INFILE or OUTFILE reside, must be specified if either do not reside on the system disc.

Examples

All examples to follow in this subsection relate to the primary input file, NEW, which contains the following lines of text:

NEW 1
NEW 2
NEW 3
NEW 4
NEW 5

Copying Primary Infile to Alternate Outfile

Suppose the primary input file NEW has been specified with the EDIT command below:

> EDIT NEW

Perhaps you wish to copy three lines of text from NEW to an alternate file ALT. In this example NEW is the INFILE and ALT is the OUTFILE. The following command is entered:

* COPY 3 NEW ALT

ALT is closed and saved at this point and exists as:

NEW 1
NEW 2
NEW 3

Any subsequent COPY statements to ALT at this time would cause newly copied text to be overlayed in place of old text. Old text in ALT would be lost with each subsequent COPY statement.

Copying a Primary Infile to an Alternate Device

In the preceding example a device such as CONO could have been used in place of ALT. In this case the command below is typed at the keyboard:

```
*COPY 3 NEW CONO
```

The following lines of data are output to the console:

```
NEW 1
NEW 2
NEW 3
```

Copying Primary Infile to Primary Outfile

Suppose the primary input file, NEW, has again been specified with the EDIT command. Remember that NEW contains:

```
NEW 1
NEW 2
NEW 3
NEW 4
NEW 5
```

This time a primary output file, OLD, is specified as well.

```
> EDIT NEW OLD
```

Perhaps you wish to copy lines 3 through 5 of NEW to the primary output file, OLD. The command below is entered:

```
* COPY 3-5 NEW
```

If OLD has not been previously created and all files are not closed, the following text is loaded into a temporary file, OLD.

```
NEW 3
NEW 4
NEW 5
```

Subsequent lines of text could be copied to the temporary file, OLD. Perhaps the following command is entered:

```
* COPY 1 NEW
```

The first line of text in NEW is now loaded into the primary output file. OLD contains the text:

```
NEW 3
NEW 4
NEW 5
NEW 1
```

If all files are closed and saved at this point, the primary output file, OLD, contains the text:

```
NEW 3
NEW 4
NEW 5
NEW 1
NEW 1
NEW 2
NEW 3
NEW 4
NEW 5
```

All copied text is inserted, followed by all text located in the primary input file, NEW.

Copying Alternate Infile to Primary Outfile

Suppose the primary input file, NEW, has been specified with the EDIT command. Remember that NEW contains:

```
NEW 1
NEW 2
NEW 3
NEW 4
NEW 5
```

Since a name is not specified for the primary output file, NEW also defaults to the name of the primary output file.

> EDIT NEW

Perhaps you wish to copy three lines of text from the alternate input file, ADD, to the primary output file NEW. ADD contains the following lines of text:

ADD 1
ADD 2
ADD 3
ADD 4
ADD 5

The command below is entered:

*COPY 3 ADD

If all files are closed and saved, text in the primary output file, NEW, now appears as:

ADD 1
ADD 2
ADD 3
NEW 1
NEW 2
NEW 3
NEW 4
NEW 5

Copying an Alternate Infile to an Alternate Outfile

Suppose the primary input file NEW has been specified with the EDIT command. Perhaps you wish to copy three lines of text from an alternate input file, ALT, to an empty alternate output file to be named SUBALT. ALT contains the following lines of text:

ALT 1
ALT 2
ALT 3
ALT 4
ALT 5

The command below is entered:

```
* COPY 3 ALT SUBALT
```

All files are closed. Data in SUBALT now exists as:

```
ALT 1
ALT 2
ALT 3
```

Any subsequent COPY statements at this time would cause newly copied text to be overlayed in place of old text in SUBALT. Old text in the file is lost when the file is closed.

SYNTAX

TYPE [absolute number of lines / range of lines]

PURPOSE

Lines of text in program memory may be displayed on the console with the TYPE command.

EXPLANATION

The TYPE command displays "n" lines of text on the console. The current line pointer position remains unaltered after TYPE execution. "n" may be an absolute number of lines or a range of lines. Lines prior to the current line pointer position may be displayed if they are specified in a range of lines. "n" need not be specified if a default of n = 1 is desired. In this case the current line is displayed. A space between TYPE and "n" is optional.

EXAMPLES

Suppose the text in program memory appears as follows:

```
  LINE 1
  LINE 2
→ LINE 3
  LINE 4
  LINE 5
```

The following command is entered:

```
*TYPE 2-4
```

The following lines are displayed on the console:

```
LINE 2
LINE 3
LINE 4
```

The line pointer position remains unaltered.

SYNTAX

```
        [ absolute number of lines ]
LIST    [ range of lines           ]
```

PURPOSE

A specified number of lines of text are listed on the line printer with the LIST command.

EXPLANATION

This command lists "n" lines of text on the line printer. "n" may be an absolute number of lines or a range of lines. Lines prior to the line pointer may be listed if they are specified as a range of lines. A space between LIST and "n" is optional. "n" need not be specified if a default of n = 1 is desired. The current line pointer position in program memory remains unchanged throughout the LIST execution.

Examples

Suppose the primary input file, NEW, has been specified with the EDIT command. All text in NEW has been read into program memory with the GET command. The line pointer is positioned to the third line in program memory after the command, BEGIN:DOWN 2, is entered:

```
  NEW 1
  NEW 2
→ NEW 3
  NEW 4
  NEW 5
```

The command below is entered:

```
* LIST 1-2
```

The following lines are output to the line printer:

```
NEW 1
NEW 2
```

The line pointer in program memory continues to be positioned at the line containing NEW 3.

SEARCHING AND ALTERATION COMMANDS

The line pointer position in program memory can be repositioned upward and downward in order to allow access to specific lines of text. The editor has five commands that facilitate line pointer repositioning so as to aid in text searching. Characters within lines of text, as well as entire lines of text, can then be altered by way of three other editor commands.

COMMAND NAME	DESCRIPTION OF COMMAND	PAGE
N	Displays number of line where line pointer is positioned.	5-35
UP	Moves line pointer upward a specified number of lines.	5-36
DOWN	Moves line pointer downward a specified number of lines.	5-38
BEGIN	Positions line pointer to the first line in program memory.	5-40
END	Positions line pointer one line below last line in program memory.	5-41
FIND	Searches the program memory for the first line that contains a specified string of characters and positions the line pointer to that line.	5-42
SUBSTITUTE	Finds the first occurrence of a specified character string within the current line and replaces the string with a newly specified string.	5-44
REPLACE	Replaces the current line with a new line of text.	5-46
KILL	Deletes a specified number of lines or range of lines.	5-48

SYNTAX

N key

PURPOSE

The number of the line pointed to by the current line pointer is displayed on the console after pressing the N key. This command allows you to keep track of the current line pointer position without taking time to print out lines of text.

EXPLANATION

When the N key is pressed and followed by a carriage return, the number of the current line in program memory is displayed. This number is always relative to the first line in program memory. The system responds with LINE = n, where "n" is the line number of the current line.

Examples

Suppose you have several text lines in program memory and want to know where the line pointer is positioned. Press the N key as follows:

```
*N
```

The command invokes the following system response when the line pointer is positioned at the seventh line in the program memory.

```
LINE = 7
```

SYNTAX

UP [specified number of lines]

PURPOSE

This command moves the line pointer upward a specified number of lines of text.

EXPLANATION

The UP command moves the line pointer upward "n" lines. "n" may be a specified number of lines and may not be a range of lines. "n" defaults to n = 1 when not specified. If the current line in program memory is "q" and "q" minus "n" is less than 1, the line pointer is positioned to the first line in program memory. A space between UP and "n" is optional.

After the UP command is entered, the system responds by displaying the current text line residing where the line pointer has been positioned.

Examples

Suppose the primary input file, NEW, has been specified with the EDIT command. NEW contains the following text.

```
NEW 1
NEW 2
NEW 3
NEW 4
NEW 5
```

Read all five lines of text into program memory with the GET command.

```
* GET 5
```

The line pointer is now positioned to the line below the last line of text in program memory. The N key affirms the position:

LINE = 6

Suppose you now wish to make changes to line 3 in program memory. The command below is entered:

*UP 3

This command moves the line pointer upward in program memory three lines and displays the line residing where the line pointer is now positioned as follows:

NEW 3

Suppose you now enter the command:

*UP 289

Text located at line 1 is displayed as:

NEW 1

SYNTAX

DOWN [specified number of lines]

PURPOSE

This command moves the line pointer downward a specified number of lines of text.

EXPLANATION

The DOWN command moves the line pointer downward "n" lines. "n" may be a specified number of lines and may not be a range of lines. "n" defaults to n = 1 when not specified. If the current line in program memory is "q" and "q" plus "n" is greater than the number of lines in program memory, the line pointer is positioned to the line below the last line in program memory. In this case the system response * * EOF * * is displayed to indicate an end of file condition. A space between DOWN and "n" is optional.

After the DOWN command is entered, the system responds by typing the current line of text residing where the line pointer has been positioned.

Examples

Suppose the primary input file NEW has been specified with the EDIT command. NEW contains the following text:

```
NEW 1
NEW 2
NEW 3
NEW 4
NEW 5
```

All five lines of text are read into program memory with the GET command:

```
*GET 5
```

The line pointer now points to the line below the last line in program memory. Pressing the N key and following with a carriage return confirms the position:

LINE = 6

Suppose you now wish to make changes to lines 2 and 5 in program memory. The following command is entered:

```
* UP 4
```

The system response below indicates the line pointer position:

```
NEW 2
```

The line pointer is positioned and ready for changes to be made to line 2. Now enter the command:

```
* DOWN 3
```

The following system response indicates the line pointer position.

```
NEW 5
```

Changes may now be made to line 5.

Entering the following command results in the system response * * EOF * *, since the current line number plus 289 is greater than the number of lines in program memory.

```
* DOWN 289
* * EOF * *
```

SYNTAX

BEGIN

PURPOSE

This command positions the line pointer to the first line in program memory and displays the first line on the console terminal.

Examples

Suppose the primary input file, NEW, has been specified with the EDIT command. New contains the following text:

```
NEW 1
NEW 2
NEW 3
NEW 4
NEW 5
```

Read all lines of text into program memory with the GET command:

```
*GET 5
```

The line pointer is now positioned to the line below the last line ot text. This can be affirmed by pressing the N key. The following system response appears:

```
LINE = 6
```

Enter the following command:

```
*BEGIN
```

If the N key is now pressed and followed by a carriage return, the system responds with:

```
LINE = 1
```

SYNTAX

END

PURPOSE

This command positions the line pointer one line below the last line in program memory.

EXPLANATION

When the END command is entered, the system response * * EOF * * is displayed to indicate that the line pointer is positioned to the end of the file.

Examples

Suppose the primary input file, NEW, has been specified with the EDIT command. NEW contains the following text:

```
NEW 1
NEW 2
NEW 3
NEW 4
NEW 5
```

Read all lines of text into program memory with the GET command. Suppose you wish to make changes to text in the middle of program memory.

After these changes are made, you may decide to add lines of text to the end of the text in program memory. The following command advances the line pointer to the line immediately below the last line of text:

```
*END
```

The following system response indicates that the line pointer is positioned at the end of the text in program memory.

```
* * EOF * *
```

You may now add new text lines to the end of the text in program memory.

SYNTAX

FIND {delimiter, string of text, delimiter}

PURPOSE

The FIND command searches downward into text in program memory for the first line containing a specified string of characters and positions the line pointer to that line.

EXPLANATION

Beginning with the current line, the FIND command searches program memory, for the first line containing STRING. STRING may be any combination of characters and spaces found in the text. If STRING is found, the line pointer is repositioned to the line where STRING occurs. If STRING is not found, the system response * * NOT FOUND * * is displayed and the line pointer remains unaltered.

There must be at least one space between the FIND command and the first delimiter surrounding STRING. The delimiters surrounding STRING may be represented by any character on the keyboard except a space. Caution must be used when using delimiters. They must not be the same as any characters appearing within STRING. The first delimiter marks the beginning of STRING, and the second marks the end. The second delimiter must always be the same character as the first.

If the FIND command is invoked by using the AGAIN command, the search starts at the current line plus one. The AGAIN command repeats the last command entered.

Examples

Suppose the primary input file NEW has been specified with the EDIT command. NEW contains the text:

```
NEW 1
NEW 2
NEW 3
NEW 4
NEW 5
```

All text lines are read into program memory with the GET command. The line pointer is positioned at the beginning of the text in program memory with the BEGIN command.

```
→ NEW 1
  NEW 2
  NEW 3
  NEW 4
  NEW 5
```

The following command is executed, and the line pointer moves to the first line in program memory containing a 4:

```
*FIND $4$
```

The line pointer now appears as follows:

```
  NEW 1
  NEW 2
  NEW 3
→ NEW 4
  NEW 5
```

The command below is entered.

```
*FIND $1$
```

The following system response is displayed on the console since the specified STRING is above the current line pointer:

```
* * NOT FOUND * *
```

SYNTAX

SUBSTITUTE { delimiter, string of text to be searched and substituted, delimiter, string of text to replace, delimiter }

PURPOSE

The SUBSTITUTE command performs both searching and alteration functions. The command finds the first occurrence of a specified character string within the current line and replaces the string with a newly specified character string.

EXPLANATION

The SUBSTITUTE command searches the current line for the first occurrence of a specified string of characters OLDSTRING. OLDSTRING is then replaced with NEWSTRING. OLDSTRING may be any combination of characters found in the line of text. If OLDSTRING is found, the substitution takes place. If OLDSTRING is not found, the system response, * * NOT FOUND * *, is displayed on the console.

NEWSTRING may be any desired combination of characters. NEWSTRING may contain TAB characters. TAB characters are used to pre-define and simplify column spacing in program memory. The conversion of the TAB characters to spaces in program memory depends on the column in which the substitution occurs. The substitution of spaces for TAB characters is always in accordance with the current TAB positions.

There must be one space between the SUBSTITUTE command and the first delimiter to surround OLDSTRING. All delimiters surrounding the two text strings may be represented by any single character on the keyboard. Caution should be taken when using delimiters. Delimiters may not be the same as any characters appearing within the two text strings. Delimiters may not be the same as the current TAB character. The first delimiter marks the beginning of the text string to be searched and substituted. The second delimiter separates the two text strings. The third delimiter marks the end of the substitution string. The second and third delimiters must always be the same character as the first.

If a substitution causes a line to exceed 127 characters, the message * *TRUNCATED* *, is displayed on the console to indicate that only the first 127 characters are accepted into program memory.

The line pointer position remains unaltered, no matter what the result of a substitution.

Examples

Suppose the text in the current line is:

→ MOV C,A

In order to change the "A" in this line to a "B", the following command is entered:

*SUBSTITUTE AB$

The line now appears as follows:

→ MOV C,B

SYNTAX

REPLACE { specified line of text to replace current line }

PURPOSE

The REPLACE command replaces the current line of text in its entirety with a newly specified line of text.

EXPLANATION

This command replaces the current line of text with a new line, STRING. There must be one space between the replace command and STRING. A blank line of text is not allowed as STRING. The line pointer position remains unaltered after the REPLACE command is executed.

The delimiters :, <, and > are special command delimiting characters. The ":" character allows several commands to be stacked on the one command line. The < and > characters are used to execute a command line repetitively. If the REPLACE command is used in a command line that contains more than one command, these characters may also be used to indicate the end of STRING.

Examples

Suppose the current line in program memory appears as:

→ MOV C,A

Entering the following command string replaces this line with a new line of text:

*REPLACE LDAX D

The text in program memory is now altered to:

→ LDAX D

Suppose you wish to perform a group of commands on the same line. The following command moves the line pointer downward four lines, and replaces that line with a new line of text ADD 1, and then inserts a new line of text SUB 1 just before the line containing ADD 1.

```
* DOWN 4:REPLACE ADD 1:I SUB 1
```

In addition to allowing multiple commands per line, the ":" character indicates the end of the replacement string ADD 1.

SYNTAX

KILL [absolute number of lines / range of lines]

PURPOSE

A specified number of lines are deleted from program memory with the KILL command.

EXPLANATION

The KILL command deletes a specified number of lines "n". "n" need not be specified if the default n = 1 is desired. A space between KILL and "n" is optional. "n" may take two forms in the KILL command. The first occurs when "n" is an absolute number of lines; the second when "n" is a range of lines.

The first form begins with the current text line in program memory and deletes the next "n" lines. In this case "n" is represented by an absolute number of lines. If the line pointer is originally positioned at line "a" and KILL n is executed, the line pointer is repositioned to the line that was equal to "a" plus "n" before the deletion took place.

When the second form of "n" is used with the KILL command, a range of lines "b" through "c" are deleted from program memory. This form allows for two possible positionings of the line pointer:

- The first positioning occurs when the line pointer points to a line between lines "b" and "c". After execution of the KILL command, the line pointer is positioned at what was originally line "c" plus 1.

- The second positioning occurs when the line pointer points to a line that is not between lines "b" and "c". Lines may be deleted that reside before line "b" or after line "c". The position of the line pointer remains unaltered after execution occurs.

Examples

Suppose the text in program memory exists as follows:

```
→ LINE 1
  LINE 2
  LINE 3
  LINE 4
  LINE 5
```

The following command is performed:

```
*KILL 4
```

The text in program memory now appears as follows:

```
→ LINE 5
```

In another example, suppose the text in program memory appears as:

```
  TEXT 1
  TEXT 2
→ TEXT 3
  TEXT 4
  TEXT 5
  TEXT 6
  TEXT 7
```

The following command is performed:

```
*KILL 2-5
```

The text in program memory now appears as follows:

```
  TEXT 1
→ TEXT 6
  TEXT 7
```

UTILITY COMMANDS

The Text Editor has several utility commands that perform service or program maintenance functions.

TAB	Defines a single character as the tab character.	5-51
TABS	Assigns non-standard tab positions.	5-53
MACRO	Defines and executes an integer that is assigned to represent a command line.	5-54
SPACE BAR	Temporarily halts the system display to the console and then resumes the display.	5-56
ESC	Suspends the editing session or rubs out the current input line.	5-57
QUIT	Closes all files and terminates the editing session.	5-58
BRIEF	Disables and enables display of current line after searching and alteration.	5-59
?	Displays editor I/O status.	5-61
AGAIN	Repeats the previous command.	5-62

SYNTAX

TAB { character defined as tab character }

PURPOSE

The TAB command defines a single character as the tab character. Using tab characters in this way speeds up keyboard input and increases column spacing accuracy much more efficiently than the use of spaces.

EXPLANATION

This command defines the single character CHAR as the tab character for the current editing session. If a character is defined as the tab character, the character is not displayed with the text in program memory after execution. Whenever the tab character is entered in input mode, the next character to follow starts at the next tab position.

The tab character must be defined prior to entering the INPUT mode. The character being defined as CHAR must not be a part of the text to be input. The tab character cannot be the :, <, −, or > characters. A space between TAB and CHAR is required. The default tab character is CTRL-I, which is produced by holding down the CTRL key while pressing the I key. When CTRL-I is used, a character is not echoed to the console during input. An additional tab key labeled TAB is located on the keyboard of the CT8100 CRT Terminal and performs the same function as CTRL-I. Tab column positions default to 8, 16, 24, 32, 40, 48, 56, and 64 unless defined with the TABS command. A maximum of eight tabs are allowed per text line.

Examples

Suppose the character # is defined as the TAB character as in the following command line:

```
*TAB #
```

Enter the following lines of text after entering the INPUT command:

```
THIS#IS#A#LINE#OF#TEXT
HERE#IS#ANOTHER
#HERE#IS#ONE#MORE
A#BIG#WORD#OVERRIDES#TABS
```

The result of the preceding execution produces the following lines of text in program memory:

```
THIS    IS      A       LINE    OF      TEXT
HERE    IS      ANOTHER
        HERE    IS      ONE     MORE
A       BIG     WORD    OVERRIDES       TABS
```

SYNTAX

TABS { column1 } [column2] [column3] ... [column8]

PURPOSE

Non-standard tab positions are assigned with the TABS command.

EXPLANATION

This command sets non-standard tab positions to the given columns 1 through 8. TABS must be defined prior to entering INPUT mode. A space between TABS and each column assignment is required. If TABS is not specified, the default column positions are 8, 16, 24, 32, 4Ø, 48, 56, and 64. A maximum of eight tab positions may be defined.

Examples

The tab positions for the text could be altered as follows:

```
*TABS  6  12  18  24  3Ø  36  42
```

Suppose the $ character is defined as the TAB character in the following command line:

```
* TAB $
```

Enter the following lines after entering the INPUT command:

```
THIS$IS$A$LINE$OF$TEXT$WITH$TABS
THIS$IS$ONE$MORE
$HERE$IS$ONE$MORE
```

The result of the preceding execution produces the following lines of text in program memory:

```
THIS IS   A    LINE  OF    TEXT  WITH  TABS
THIS IS   ONE  MORE
     HERE IS   ONE   MORE
```

The first column of text begins at column 1, the second at column 6, the third at column 12, the fourth at column 18, the fifth at column 24, the sixth at column 3Ø, the seventh at column 36, and the eighth at column 42.

SYNTAX

MACRO { integer = a desired command line }
or
MACRO { integer that represents command line to be executed }

PURPOSE

An editor command line may be defined and executed as an integer, rather than being entered as an entire command line. This capability is implemented with the MACRO command and provides a time-saving function when editing.

EXPLANATION

The MACRO command takes two forms. One form defines an integer as a command line. The second form executes the command line when you enter its integer form.

An integer may be defined as a command line if MACRO n = COMMANDLINE is performed. Defining an integer as a command line requires that each identifying integer "n" be greater than Ø and less than 128. COMMAND LINE can be any legal editor command line but cannot contain a MACRO execution or definition command. If "n" is already defined and a new definition for "n" is performed, the new COMMANDLINE is identified as "n". When defining an integer to represent a command line, there need not be spaces between MACRO and the parameters in the line.

Executing a command line that has been entered in its integer form is implemented with the MACRO n form of this command. The effect of executing the integer is equivalent to executing COMMANDLINE in its normal form. There need not be a space between MACRO and "n" when entering the two components.

Examples

Suppose the text in program memory appears as follows:

```
  LINE 1
  LINE 2
→ LINE 3
  LINE 4
  LINE 5
```

The following line defines the integer 99 as a command line whose function advances the line pointer to the beginning of the text in program memory and then displays all lines of text on the console:

```
* MACRO 99 = BEGIN:TYPE 1ØØ
```

Anytime you wish to perform the above function, simply enter:

```
* MACRO 99
```

The results of this execution are the same as if the long form of the command line were entered:

```
   LINE 1
→  LINE 1
   LINE 2
   LINE 3
   LINE 4
   LINE 5
   **EOF**
```

SYNTAX

Space Bar

PURPOSE

The space bar is used to halt the display on the console and then cause the display to continue.

EXPLANATION

Pressing the space bar during console display temporarily halts the display. Pressing the space bar again causes the display to continue. The display may be halted and continued as many times as is needed.

SYNTAX

ESC key

PURPOSE

Pressing the ESC key (escape) once causes deletion of the line currently being input to program memory. Pressing the ESC key twice causes suspension of program execution and returns control to TEKDOS.

EXPLANATION

Pressing the ESC key once causes deletion of the current input line. The system reponse after ESC execution is the editor prompt character " * " unless the editor is in the INPUT mode. If in the INPUT mode, the current input line is deleted and the display cursor is moved to the next line to await further input.

Pressing the ESC key twice (ESC ESC) results in the deletion of the current input line and suspension of all active editor programs. Control returns to TEKDOS. A program suspended by this means will not resume execution unless you issue a CONT * (continue execution) command.

SYNTAX

QUIT

PURPOSE

The QUIT command closes all files, terminates the editing session, and returns control to TEKDOS.

EXPLANATION

This command closes the primary input and primary output files, terminates the editing session, and returns control to TEKDOS. Text input to program memory during the current editing session is not saved in the primary output file. If the primary output file is a new file, the file is deleted before control returns to TEKDOS. Text written to an alternate output file during the current editing session is not affected since alternate output files are closed immediately after receiving data.

SYNTAX

BRIEF
or
.

PURPOSE

The BRIEF command suppresses the display of the text located at the current line pointer after most searching and alteration commands. Entering the BRIEF command once again reinstates the display. When a period "." is appended to a command, the BRIEF status for the current line is canceled.

EXPLANATION

At completion of the commands, END, UP, DOWN, FIND, SUBSTITUTE, and REPLACE, the editor responds by displaying the text line located at the line pointer. Entering the BRIEF command suppresses this display action. The BRIEF command is said to be on when in this state. Entering the BRIEF command a second time reinstates the display activity. The BRIEF command is said to be off when in this state.

If the BRIEF switch is off (display is enabled), display can be suppressed on a line-by-line basis if a "." is appended to the command. If the BRIEF switch is on (display is suppressed), display can also be reinstated on a line-by-line basis by appending a "." to the command.

Examples

Suppose the text in program memory exists as shown below:

```
→ LINE 1
  LINE 2
  LINE 3
  LINE 4
```

The following command is performed:

```
*DOWN 1
```

The editor moves the line pointer downward one line and displays the text located there as follows:

```
LINE 2
```

Now suppose the following commands are entered:

```
*BRIEF
*DOWN 1
```

The text editor again moves the line pointer downward one line, but does not display that line. The BRIEF command is in the on state.

Suppose the line pointer is moved downward once again. This time a display of the current line may be viewed after command execution by entering:

```
*DOWN.1
```

The editor moves the line pointer downward one line and displays the text located there as follows:

```
LINE 4
```

SYNTAX

?

PURPOSE

The ? displays the editor I/O status.

EXPLANATION

Entering the ? character results in the following informative messages being displayed on the console:

```
STATUS
PI  = primary input file name
        LINE    next line to "GET" from the primary input file
PO  = primary output file name
        LINE    next line to "PUT" to the primary output file
LAST AI   = Last alternate input file referenced
LAST AO   = Last alternate output file referenced
```

This command provides a helpful record-keeping service for referencing the status of files accessed in the current text editing session.

SYNTAX

AGAIN

PURPOSE

This command repeats execution of the previous repeatable command. AGAIN provides time-saving capabilities when entering editor commands.

EXPLANATION

The AGAIN command repeats the execution of the last editor command with the following exceptions:

- AGAIN
- BRIEF
- QUIT
- FILE
- TAB
- TABS
- MACRO (AGAIN does not repeat the definition of a MACRO, but does repeat the execution of a MACRO.)

CAUTION

If a non-repeatable command was the last command specified and the AGAIN command is entered, the AGAIN command continues to search backward until a repeatable command is found. That command line is then executed.

Examples

Suppose the text in program memory exists as follows:

```
  LINE 1
→ LINE 2
  LINE 3
  LINE 4
  LINE 5
  LINE 6
```

The command below is entered:

```
*KILL 2
```

The text in program memory now exists as:

```
  LINE 1
→ LINE 4
  LINE 5
  LINE 6
```

The next command performed is:

```
*AGAIN
```

The text in program memory is now altered to:

```
  LINE 1
→ LINE 6
```

Section 6

ASSEMBLING AND LINKING

INTRODUCTION

This section describes the syntax required to translate source code into absolute binary object code. In addition, the resulting assembler output is described. Further information pertaining to the assembler for a specific emulator microprocessor is found in the corresponding Assembler and Emulation User's Manual.

This section also describes the syntax required to merge several independently assembled modules into one executable program with the linker. In addition, the resulting linker output is described. Further information pertaining to the linker is found in the Assembler and Emulation User's Manual.

COMMAND NAME	DESCRIPTION	
ASM	Translates the source code relating to the microprocessor being used into absolute binary object code.	6-2
LINK	Merges several independently assembled modules into one executable program.	6-5

SYNTAX

ASM [object file name / object device] [list device / list file name] {source file name / source device} [source file name / source device]...

PURPOSE

The ASM command translates the source code of the microprocessor being used into absolute binary object code. The object code is then executable by the emulator processor.

EXPLANATION

The ASM command invokes the assembler when the 8002 µPROCESSOR LAB is under TEKDOS control. The OBJECT parameter causes the assembler to output the absolute binary object code to the specified disc file or device. The LIST parameter causes the assembler to output a listing of the assembled code to the specified device or disc file. SOURCE is the name of the disc file or device that contains the symbolic code pertaining to the particular emulator processor being used.

All parameters within the ASM command line may be separated by spaces or commas. The OBJECT parameter is optional and may be replaced by two commas in this manner:

```
ASM,,LIST SOURCE>
```

In this case an object file is not generated. The LIST parameter is also optional and may be replaced by two commas in this manner:

```
ASM OBJECT,,SOURCE
```

In this case an assembled listing is not generated. If OBJECT and LIST are both omitted, they must be replaced by three commas in this manner:

```
ASM,,,SOURCE
```

If the OBJECT or LIST files are not intended to reside on the system disc, the appropriate disc drive number must follow the "/" character in this manner:

ASM OBJECT/1 LIST/1 SOURCE

At least one SOURCE file must be specified in the ASM command line. Multiple SOURCE files are acceptable as long as the ASM command line is not longer than one line. If the SOURCE file is not stored on the system disc, the appropriate disc drive must be specified after the "/" character in this manner:

ASM OBJECT LIST SOURCE/1

After the assembler has completed execution, an assembler message is displayed. This message indicates the number of source code lines assembled, the number of errors, and the number of undefined symbols. The TEKDOS prompt character ">" appears below the assembler message to indicate assembly completion.

If you specified the OBJECT parameter in the ASM command line, your assembled program is stored in the form of binary object code. A correctly assembled object file may be executed, linked or debugged.

If you specified the LIST parameter, information is output to a device or file in the form of an assembled listing. LIST is composed of two parts — the assembled listing and the symbol table.

Each page of the assembled listing contains a header. The header is followed by a blank line and the listing information. The header includes the assembler version on the left side of the page and the page number on the right side of the page, as shown below:

TEKTRONIX 8Ø8Ø ASM Vn.m   PAGE XXXX

Each instruction in your source program is converted into an assembled line. The assembled line includes the line number of the instruction, the instruction's location in program memory, the object code, and the source code. The assembled line may be a maximum of 72 characters. The line number is five characters wide and the location is four characters wide. Both are right justified and are separated by one space. The object code follows the location and a space. The object code is left justified and may be a maximum of eight characters wide. The source code follows the object code and a space. There are 51 characters remaining in the assembled line for the source code. Any source line exceeding the 51-character limit is truncated. Any non-printing character, other than the space and tab characters, is represented by a "?" in the assembled listing. The assembled line appears as shown below:

```
XXXXX LLLL DDDDDDDD SOURCE CODE ----
        location
  line         object
  number       code
```

If an error occurs in a particular instruction, the appropriate assembled line is followed by an error response. The error response takes the form shown below:

****E

Following the assembled listing is the symbol table. The symbol table indicates those labels, defined, or undefined, and the location in program memory that the labels represent. This table may also contain other special symbols.

The symbol table contains a header. The header includes the words, "SYMBOL TABLE" in the center of the page and the page number on the right, as shown below:

```
TEKTRONIX           M68ØØ ASM V1.ØA   SYMBOL TABLE LISTING          PAGE 1Ø
```

The header is followed by a blank line and the symbol table information. Up to four symbols and their hexadecimal values are specified on each line. If undefined, the value field is indicated in this manner: ****. Symbols may be a maximum of six characters and are followed by at least one space. The value field may be a maximum of four characters and is right justified. If a value field is under four characters wide, the remainder of the field is filled with leading zeroes. Each value field is followed by four spaces and the next symbol. A sample symbol table format is shown below:

```
SYM1    ØØØ9    SYM2    Ø1A1    SYM3    ****    SYM4
```

SYNTAX

LINK [[load module file name] [list file name] {input file name} [input file name] ...]

PURPOSE

The LINK command merges several independently assembled modules generated by the assembler into one executable program. The linked file may be either in hexadecimal or binary format.

EXPLANATION

The linker is invoked in one of two ways:

- Simple invocation
- Interactive command invocation

Simple invocation is performed by entering LINK, followed by all file name parameters on one line. The linker assumes a set of default options for parameters that are not specified in this linker form.

All parameters within the LINK command line may be separated by spaces or commas. The LODMOD parameter is the name of the binary formatted load module created by the linker. LODMOD is optional and may be replaced by two commas in this manner:

```
LINK,,LIST,INFILE
```

LIST is the name of the file or device where the linker listing is output. LIST is optional and may be replaced by two commas in this manner:

LINK LODMOD,,INFILE

If LODMOD and LIST are both omitted, the parameters are replaced by three commas in this manner:

LINK,,,INFILE

INFILE is the name of the object file to be linked. Multiple INFILE parameters are acceptable in a command line, however, all file names must fit on one line. At least one INFILE must be specified in the simple invocation linking form.

Upon correct completion of the LINK command line execution LODMOD is created in a binary format. A binary LODMOD file may be read into program memory with LOAD. A hexadecimal LODMOD file may be read into program memory with RHEX. Data is checksummed at the time of linking and when read into program memory.

If LIST is specified in the simple invocation form of the linker, error messages, a memory map, and linker statistics are output to the specified file or device.

Interactive command invocation allows you to enter a series of commands to the linker. To invoke this form enter LINK, followed by a carriage return. The linker responds with the asterisk character, prompting you to enter a linker command. The linker continues to prompt for commands until an "END" command is entered.

Interactive command invocation allows you to do the following things: specify modules to be linked, configure memory, override section configurations set at assembly time, and control and format the contents of linker output files. For a more descriptive view of interactive command invocation, refer to the Assembler and Emulation User's manual corresponding to your particular microprocessor.

Section 7

EMULATOR ENVIRONMENT

INTRODUCTION

This section describes the operating environment of the emulator processor and the user program. The topics covered include emulator operating modes, user program loading and storing, memory control, and execution of user programs.

CONTENTS

SECTION 7 EMULATOR ENVIRONMENT

INTRODUCTION 7-1
OPERATING MODES 7-2
  EMULATE 7-4
LOADING AND STORING 7-5
  WHEX 7-8
  RHEX 7-9
  LOAD 7-10
  MODULE 7-11
  FETCH 7-12
MEMORY CONTROL 7-13
  DUMP 7-14
  EXAM 7-16
  PATCH 7-18
  MAP 7-19
  MOVE 7-22
  FILL 7-23
USER PROGRAM EXECUTION 7-24
  GO 7-26
  XEQ 7-27
  STATUS 7-28

OPERATING MODES

User programs are executed by the emulator processor using the same microprocessor chip as you are using in your prototype system. The program may be residing either in program memory or in your prototype system memory during execution. The mode of operation (either in program memory or in prototype system memory) must be set by the EMULATE command before starting user program execution. Setting the operational mode activates the emulator processor, defines the memory that holds the user program, and defines other operational parameters that will be defined later.

The system processor controls the emulator processor. However, the two processors can not talk directly to each other. The system processor controls the emulator processor through the interrupt utility module, as shown in Fig. 7-1. You can also see in Fig. 7-1 that both processors can access program memory. Program memory serves as a link over which the system processor and the emulator processor can pass data. Thus full communication is accomplished through the use of these two routes (control through the interrupt utility module and data passed by storage in program memory).

The emulator processor is completely controlled by the system processor. The emulator processor can execute (mode Ø only) the user program at full speed until a breakpoint or an operator console command stops it. The system processor using the forced jump sequencer can start the user program at any location or cause the emulator processor to jump to a trace routine to dump the register contents.

The emulator processor can operate in three modes: a system mode, a partial emulation mode, and a full emulation mode. Setting the emulator's operating mode determines more than just the location where the emulator processor will look for the user program. The mode also specifies the input/output devices to use and the clocking sources. When the mode is set with the value of zero, the emulator processor executes the program in program memory and uses the system input and output. Emulation mode Ø is the system mode.

Setting the mode to a value of one causes the emulator processor to execute the user program in the program memory and prototype memory. However, emulation mode 1 uses the user's prototype system input/output and clocking signals.

Emulation mode 2 is the full emulation mode using the user's prototype system memory, input/output and clocking.

The emulation mode may be changed while the DEBUG command is active. However, changing the emulation mode while a user program is being executed will cause execution to be aborted.

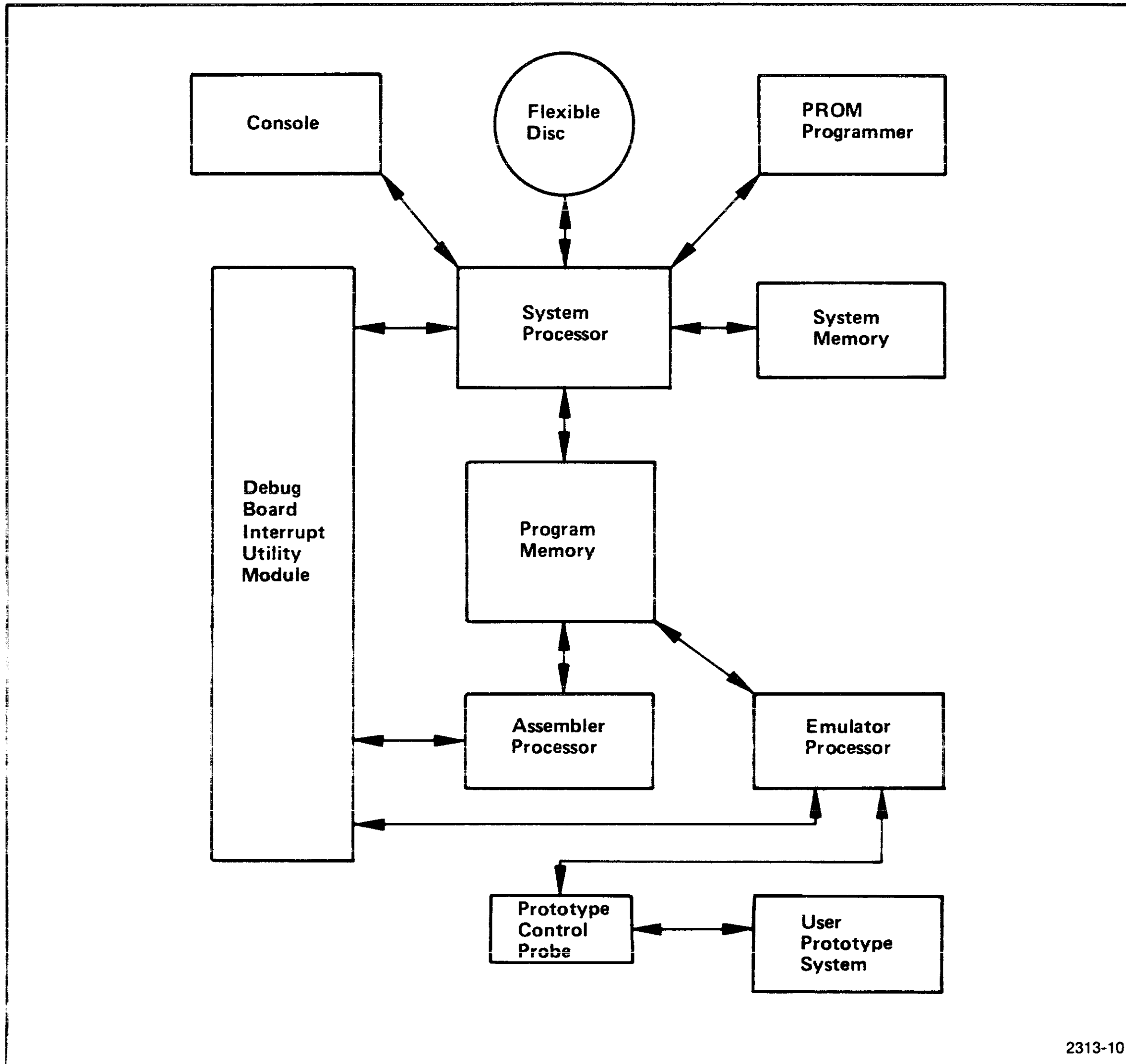


Fig. 7-1. System Functional Overview.

SYNTAX

EMULATE {operational mode}

PURPOSE

The EMULATE command activates the emulator processor and sets the mode of operation.

EXPLANATION

The EMULATE command activates the emulator processor and sets the mode in which it operates. The possible values for the operational mode are:

Ø - System mode. Uses program memory and system I/O.

1 - Partial emulation mode. Uses program memory, user prototype memory; prototype I/O and user clock.

2 - Full emulation mode. Uses user prototype memory, prototype I/O and clock. (Note that in mode 2 the TRACE JUMP option is not available.)

The emulation mode may be changed while the DEBUG command is active. However changing the emulation mode while a user program is being executed will cause execution to be aborted.

EMU Error Response

31—Parameter required
32—Too many parameters
54—Invalid mode
56—Invalid device address

LOADING AND STORING

The commands in this section are used to move object code between program memory and flexible disc storage or a peripheral device. The object code may be stored on a flexible disc either in binary or hexadecimal format. The object code is loaded into program memory in binary format.

COMMAND NAME	DESCRIPTION	PAGE
WHEX	Converts binary code to hexadecimal format and writes it on the flexible disc or to a device	7-8
RHEX	Reads hexadecimal formatted code converts it to binary code and loads the binary code in program memory	7-9
LOAD	Loads assembler and linker object files into program memory	7-10
MODULE	Moves binary code from program memory to a flexible disc or to a device	7-11
FETCH	Loads absolute object code from a flexible disc into program memory	7-12

INTRODUCTION

The output of the TEKTRONIX Assembler is stored on a flexible disc in absolute binary code. This absolute binary code is then loaded into program memory for execution and debugging. Absolute code means that each memory address reference is an actual address, not a relative address that may not be determined until used.

Hexadecimal Code

Binary code cannot be directly displayed on the terminal. The console considers the code to be ASCII and many ASCII characters are unprintable or are control characters to the console. When you want the code printed out on the console use hexadecimal format. That is, use two hexadecimal digits to represent the eight binary digits in each byte.

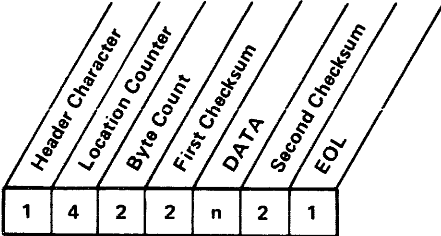
The WHEX command converts the binary code in program memory to hexadecimal format, then stores the hexadecimal form on flexible disc. This hexadecimal form of the code is read back into program memory from the flexible disc with the RHEX command.

Hexadecimal Loading Format

The hexadecimal object code is stored as an absolute hexadecimal file. This file is composed of one or more data blocks and a terminating block. The terminating block is the last block in the file. The following example is a listing of a hexadecimal file.

```
/ØØØØ1EØFAF47671 1FF131A1D4F1A1DCFØ91A1D4FØ91A4FØ9373F761F677D1F6F7C1FB6
/ØØ1FØ6167D1F11FF13124B
/ØØØØØØØØ
```

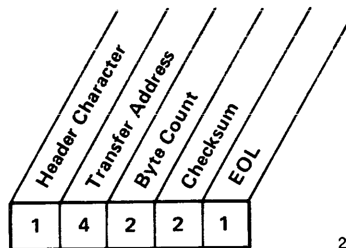
The format of a normal data block is prescribed in digits as:



2313-11

All entries in the normal data block are hexadecimal digits except the header byte and the EOL byte. The following list describes each entry in the normal data block:

- HEADER CHARACTER is always a slash "/".
- LOCATION COUNTER is four hexadecimal digits and gives the starting location of the block in program memory.
- BYTE COUNT is the number of bytes in this block. The byte count uses two hexadecimal digits .
- FIRST CHECKSUM is the sum of the hexadecimal values of the six digits that make up the location counter and the byte count. The first checksum uses two hexadecimal digits.
- DATA consists of two hexadecimal digits per data byte. The maximum number of data bytes (n) per block is thirty. This means that the maximum number of hexadecimal digits in the data section is sixty.
- SECOND CHECKSUM is the sum of the hexadecimal values of the digits that make up the "n" bytes of data. The second checksum uses two hexadecimal digits (modulo 256).
- EOL is a carriage return.



All entries in the terminating block are hexadecimal except the header byte and the EOL byte. The following list describes each entry in the terminating block.

- HEADER CHARACTER is always a slash "/".
- TRANSFER ADDRESS, four hexadecimal digits, is the location where the code begins executing. The transfer address can be supplied by the linker or as a parameter to the WHEX command. If the transfer address is not supplied to WHEX, a ØØØØ is entered here. The RHEX command ignores this field and returns to TEKDOS when the terminating block is encountered.
- BYTE COUNT is set to zero to indicate a terminating block.
- CHECKSUM is the sum of the hexadecimal values of the six digits that make up the transfer address and the byte count. The checksum uses two hexadecimal digits (modulo 256).
- EOL is a carriage return.

SYNTAX

WHEX {address 1} {address 2} [{,,address 1} {address 2}]. . .
[{address 3} {device / file name [/disc drive]}]

PURPOSE

The WHEX command converts binary code to hexadecimal format and writes the hexadecimal code blocks on the flexible disc.

EXPLANATION

The WHEX command program causes an absolute hexadecimal format file to be written from the binary code in program memory to the flexible disc. Address 1 and address 2 are the addresses of the lower and upper bounds respectively of the user program in program memory. The addresses 1, 2, and 3, are to be entered in hexadecimal, not decimal. Address 3 is an optional starting address.

The device is an optional output device or file. When the device is specified, the starting address 3 must be specified. When the device is not specified, the output is to the console output device, CONO.

The WHEX command writes in hexadecimal ASCII format, the data from address 1 to address 2 for each 1, 2 pair present in the parameter list. Note that two commas are required between address pairs if multiple address pairs are specified.

WHX Error Responses

7—Device write error
15—Invalid output device
17—Output device assign failure
30—Invalid parameter

SYNTAX

RHEX [/bias amount] [device / file name [/disc drive]]

PURPOSE

The RHEX command reads the hexadecimal formatted code from the flexible disc or from a device, converts it to binary code and loads the binary code into program memory.

EXPLANATION

The RHEX command loads the absolute hexadecimal code into program memory. The absolute hexadecimal code is read into memory from the specified device or file. The device defaults to the paper tape reader PPTR.

The bias amount is used to alter the absolute load address for the file. The default value for the bias amount is zero. The initial load address is altered by the bias amount that is entered as a signed hexadecimal value. When the sign is not specified, of the bias amount is assumed to be positive.

RHX Error Responses

6—Device read error
14—Invalid input device
16—Input device assign failure
33—Bias parameter error
4Ø—Invalid input format

SYNTAX

LOAD { file name [/disc drive] } [file name [/disc drive]]

PURPOSE

The LOAD command program loads assembler and linker object files into program memory.

EXPLANATION

The specified file name is loaded into program memory by the LOAD command. The file must have been previously created by the assembler or the linker.

The file named is loaded into program memory starting at the location specified in the source code.

DOS Error Responses

6—Device read error
14—Invalid input device
48—Load file not found
49—Load file assign failure
51—Invalid load request

SYNTAX

MODULE {file name} [/disc drive] {address 1} {address 2} {address 3}
[identifying string]

PURPOSE

The MODULE command program writes binary code onto the flexible disc from program memory.

EXPLANATION

The MODULE command writes binary code into the specified file from program memory. Address 1 and address 2 are the addresses of the lower and upper bound respectively of the user program in program memory. Address 2 must be greater than or equal to address 1. Address 3 is the starting address of the program. The addresses, 1, 2, and 3, are to be entered in hexadecimal, not decimal. The load module is preceded by a header which contains the addresses 1, 2, and 3.

The identifying string is an optional character string used to identify the module. The identifying string is truncated after the first 21 characters.

MOD Error Responses

7—Device write error
1Ø—Overlay load failure
12—Invalid file name
32—Too many parameters
34—Invalid address

SYNTAX

FETCH {file name [/disc drive]}

PURPOSE

The FETCH command loads binary code into program memory from a flexible disc.

EXPLANATION

The binary code specified by the file name is loaded into program memory by the FETCH command.

The file named is loaded into program memory starting at the location specified at the time the code was created.

* DOS * Error Responses

6—Device read error
14—Invalid input device
48—Load file not found
49—Load file assign failure
5Ø—File not a load module
51—Invalid load request

MEMORY CONTROL

The commands in this section are used for manipulating the contents of program memory and the user's prototype memory. The contents of memory may be examined, altered, or shifted in memory location. Depending on emulation mode, program memory or user's prototype memory may also be filled with a repeating hexadecimal pattern.

COMMAND NAME	DESCRIPTION	PAGE
DUMP	Copies program memory contents to the specified device	7-14
EXAM	Displays a data byte and permits that byte to be altered	7-16
PATCH	Alters program memory with the specified hexadecimal constants	7-18
MAP	Sets and displays the memory map assignments.	7-19
MOVE	Copies the specified data block to a new location in program memory or prototype memory	7-22
FILL	Fills program memory or prototype memory with the specified hexadecimal constants	7-23

SYNTAX

DUMP {address 1} [address 2] [device / file name [/disc drive]]

PURPOSE

The DUMP command copies the specified contents of program memory to the device named in the command line.

EXPLANATION

The DUMP command copies the contents of program memory starting with address 1 to the device named. Two hexadecimal characters are used to represent each data byte. Address 1 and address 2 must be in hexadecimal form.

When address 2 is not specified, only sixteen data bytes are copied. If the output device is not specified, the data is displayed on the system console.

Addresses 1 and 2 are automatically adjusted in the following manner. The least significant digit of both addresses is replaced with a zero. For example, the address 3F4E is altered to 3F4Ø. Then address 2 is replaced by the sum of it and 1Ø base 16. This automatic change results in address 1 being reduced to the next lowest multiple of 1Ø base 16, and address 2 being raised to the next higher multiple of 1Ø base 16.

For example, if you specify the block of data between 7D and B4 the DUMP command reduces 7D to 7Ø and raises B4 to BF and the display follows:

```
> DUMP 7D B4
ØØ7Ø   =Ø3  ØC  19  18  E4  2Ø  18  7Ø  E4  ØD  98  2E  ØC  19  15  18
ØØ8Ø   =17  2Ø  CC  19  15  ØC  19  14  18  18  ØE  19  4E  ØF  19  4F
ØØ9Ø   =CE  19  4A  CF  19  4B  1B  4D  ØC  19  14  1C  4Ø  Ø3  2Ø  CC
ØØAØ   =19  14  ØE  19  4C  ØF  19  4D  1B  66  E4  3A  98  ØC  CC  19
ØØBØ   =2B  ØC  19  15  1C  4Ø  65  1F  4Ø  81  ØC  19  19  1C  41  ØØ
```

Each line in the display of the block of data that includes the data between 7D and B4 begins with the address of the first byte on that line. Each line displays sixteen bytes of data (i.e., 1Ø base 16).

* DMP * Error Responses

17—Output device assign failures
31—Parameter required
35—Invalid starting address 1
36—Invalid ending address 2

SYNTAX

EXAM {address}

PURPOSE

The EXAM command displays the data byte at the specified address in program memory and permits that data byte to be altered.

EXPLANATION

The EXAM command causes a single data byte from program memory to be displayed on the system console. The data byte displayed is the byte at the specified address. The address must be given in hexadecimal form. The data byte is displayed as two hexadecimal digits.

After the first byte is displayed, further display and altering functions may be achieved by using the following keys:

- SPACE BAR — Causes the display of the next byte in memory.
- LINE FEED or RUB OUT key — Moves the display cursor to the next line, displays the address of the currently referenced byte and displays the byte.
- RETURN key — Terminates the EXAM command.
- Entering a hexadecimal data pair replaces the current data byte with the entered line. Then the next data byte is displayed.

When the address of the data byte being displayed is a multiple of 1Ø base 16, the display cursor moves down to the beginning of the next line.

If you strike the ESC key while EXAM is being performed, the memory locations that were altered before you struck the ESC key remain altered.

The following is an example of replacing a data byte. In this example the data byte D7 located at address 3723 is to be changed to C2.

```
>EXAM 3723
3723=D7-C2 Ø8
>
```

The response 3723=D7 is displayed on the system console. When you enter C2 the system responds with -C2 and then displays the next data byte in memory. Note that the system provides the hyphen (-) when you enter the new data.

* EXM * Error Responses

31—Parameters required
35—Invalid start address
39—Invalid hexadecimal character

SYNTAX

PATCH {address} {hexadecimal string}

PURPOSE

The PATCH command alters program memory with the specified hexadecimal string of constants.

EXPLANATION

The PATCH command is used to alter program memory starting at the specified address (n). Address n is a hexadecimal value. The contents of program memory starting at address n is replaced by the hexadecimal string specified. The data in the hexadecimal string directly replaces the data in memory, the hexadecimal string data is not inserted between the currently existing data.

The hexadecimal string specified may be from 1 to 58 digits long.

The following is an example of using the PATCH command to replace the data beginning at address 7A with 49 27 CC:

```
> DUMP 7A
ØØ7Ø=Ø3 ØC 19 18 E4 2Ø 18 7Ø E4 ØD 98 2E ØC 19 15 18

> PATCH 7A 4927CC                     Address 7A

> DUMP 7A
ØØ7Ø=Ø3 ØC 19 18 E4 2Ø 18 7Ø E4 ØD 49 27 CC 19 15 18
```

* PAT * Error Responses

31—Parameter required
34—Invalid address
39—Invalid hexadecimal character

SYNTAX

```
      ┌ P ┐
      │ U │  ┌ address       ┐
      │ M │  │               │
MAP   └ R ┘  └ address range ┘...

MAP ┌ M ┐
    └ R ┘

MAP ┌ P ┐  ┌ address       ┐
    └ U ┘  └ address range ┘...
```

PURPOSE

The MAP command is used to display or set memory map assignments.

EXPLANATION

Display

The MAP command is used to display memory map assignments either in tabular form or in graphic form.

Using either an "R" parameter or no parameter with the map command will cause the memory map assignments to be displayed in tabular form. Each displayed item includes the address range (hexadecimal) and a symbol indicating assignment to prototype memory (U) or to program memory (P). The following is an example of MAP command with the R parameter:

```
>MAP R
ØØØØ-Ø7FF=U     Ø8ØØ-3FFF=P
```

This response says that memory locations ØØØØ through Ø7FF are assigned to the User prototype memory. The memory from Ø8ØØ through 3FFF is assigned to program memory.

Entering an "M" parameter with the MAP command will cause the memory map assignments to be displayed in graphic form. The graphic display is a matrix with each element representing 128 bytes of prototype or program memory. (128 base 1Ø is 8Ø base 16.) An asterisk (*) is used to represent the user prototype memory assignment of each 128 bytes. A hyphen (-) represents each 128 bytes of program memory assignment.

The following is a sample printout of MAP R and MAP M commands:

```
>MAP R
ØØØØ-7FFF=P     8ØØØ-BFFF=U
CØØØ-FBFF=P     FCØØ-FC7F=U
FC8Ø-FFFF=P
```

```
>MAP M
ØXXX  -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
1XXX  -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
2XXX  -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
3XXX  -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --

4XXX  -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
5XXX  -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
6XXX  -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
7XXX  -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --

8XXX  ** ** ** **   ** ** ** **   ** ** ** **   ** ** ** **
9XXX  ** ** ** **   ** ** ** **   ** ** ** **   ** ** ** **
AXXX  ** ** ** **   ** ** ** **   ** ** ** **   ** ** ** **
BXXX  ** ** ** **   ** ** ** **   ** ** ** **   ** ** ** **

CXXX  -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
DXXX  -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
EXXX  -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
FXXX  -- -- -- --   -- -- -- --   -- -- -- --   *- -- -- --

       Ø  1  2  3    4  5  6  7    8  9  A  B    C  D  E  F
```

Each double column above has a number typed in below it to identify the column. Column Ø is composed of two $8Ø_{16}$ byte blocks. If you look at row 4XXX, the address range of the first block is $4ØØØ_{16}$ through $4Ø7F_{16}$. The address range of the next block is $4Ø8Ø_{16}$ through $4ØFF_{16}$.

The memory represented by one line is 4K bytes (4K = 4Ø96 bytes, base 1Ø). The memory represented on the map from ØØØØ to $3FFF_{16}$ is the memory on the first 16K memory module.

Assignment

The MAP command is used to set memory map assignments either to program memory (P) or to user prototype memory (U). Either type of memory is mapped in blocks of 128 bytes (128 base 1Ø equals 8Ø base 16).

You may specify an address or an address range. More than one specification may be included in a command line. Also the addresses or address ranges do not need to be in ascending numeric order. However, the second address given in an address range must be within the same address block as the first address or greater than the first address. As an example: 947E-9427 is valid, but 9481-9427 results in an error message.

The following are examples of setting memory map assignments:

```
> MAP P ØØØØ-7FFF   CØØØ-FFFF
> MAP U FCØØ
> MAP U 8ØØØ-BFFF
```

The result of setting these memory map assignments can be seen below:

```
ØXXX   -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
1XXX   -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
2XXX   -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
3XXX   -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --

4XXX   -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
5XXX   -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
6XXX   -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
7XXX   -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --

8XXX   ** ** ** **   ** ** ** **   ** ** ** **   ** ** ** **
9XXX   ** ** ** **   ** ** ** **   ** ** ** **   ** ** ** **
AXXX   ** ** ** **   ** ** ** **   ** ** ** **   ** ** ** **
BXXX   ** ** ** **   ** ** ** **   ** ** ** **   ** ** ** **

CXXX   -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
DXXX   -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
EXXX   -- -- -- --   -- -- -- --   -- -- -- --   -- -- -- --
FXXX   -- -- -- --   -- -- -- --   -- -- -- --   *- -- -- --

        0  1  2  3    4  5  6  7    8  9  A  B    C  D  E  F
```

SYNTAX

```
              ( PP )
              ) PU (
              ) UU (
              ( UP )

MOVE  {U} {P}  {address 1}  {address 2}  {address 3}
      {P} {U}
```

PURPOSE

The MOVE command copies the specified data block from either program memory or user prototype memory to a new location in either program memory or user prototype memory.

EXPLANATION

The MOVE command program copies the data at address 1 through address 2 to memory starting at address 3. The data source may be either program memory or user prototype memory. The destination may also be either program memory or user prototype memory.

The letter "P" designates program memory.

The letter "U" designates user prototype memory.

Do not use a delimiter between the P-U parameters.

MOV Error Responses

30—Invalid parameter (P or U)
31—Parameter required
32—Too many parameters
34—Address 1>2 or invalid address 3
35—Invalid address 1
36—Invalid address 2
59—Memory write error

SYNTAX

FILL {address n1} {address n2} {hexadecimal data string}

PURPOSE

The FILL command fills the specified memory area in either program memory or user prototype memory with the repeating hexadecimal data string.

EXPLANATION

The FILL command causes the hexadecimal data string that is entered to be repeatedly placed into the memory area specified by address 1 and address 2. The area filled may be either in program memory, in prototype memory, or in both.

When the system is set to emulation mode Ø, then the program memory is filled with the hexadecimal data string. When in emulation mode 1, program memory or user prototype memory is filled in accordance with the user prototype memory map. When emulation mode 2 is invoked, user prototype memory is filled with the hexadecimal data string.

The hexadecimal data string must be an even number of hexadecimal characters, i.e., composed of hexadecimal pairs. For example, if the value B is to be used, you enter ØB. A maximum of sixty digits may be entered (thirty hexadecimal pairs).

When the memory area to be filled is not an exact multiple of the hexadecimal data string length, the hexadecimal data string is truncated at address 2 and a warning message * FIL * ERROR 36 is displayed on the console.

* FIL * Error Responses

3Ø—Invalid parameter
31—Missing parameter
32—Too many parameters
34—Invalid address - 2<1
36—Fill string truncated at address 2
59—Memory write error

USER PROGRAM EXECUTION

The commands in this section invoke execution or provide execution status information.

COMMAND	DESCRIPTION	PAGE
GO	Causes the emulator processor to begin execution of the user program	7-26
XEQ	Loads a binary load module and causes execution to begin	7-27
STATUS	Causes emulator processor status to be displayed on the system console	7-28

INTRODUCTION

The steps required to execute an assembly language program include:

- Assemble the user program and store on flexible disc
- Set the emulation mode:
  - Ø - operate entirely within the system using program memory.
  - 1 - operate with program memory, or user prototype memory, using the prototype timing and I/O.
  - 2 - operate with the prototype memory, timing and I/O.
- Set the system clock on or off based on the emulation mode.
- Load program into program memory. If in emulation mode 1 or 2 operating with prototype memory, move program into prototype memory.
- Enter the GO command.

In the following example the assembly program IRSPEC is to be executed in emulation mode Ø. The commands needed to achieve execution are:

```
>ASM IRPROG LPT1 IRSPEC
>EMULATE Ø
>CLOCK ON
>LOAD IRPROG
>GO Ø
```

In the ASM command IRPROG is the designated object file. The assembly listing is written to the line printer LPT1. The object file IRPROG is read into program memory with the LOAD command because the file is stored by the assembler on the flexible disc.

SYNTAX

GO [address]

PURPOSE

The GO command causes execution control to be passed to the emulator processor.

EXPLANATION

The GO command causes execution control to be passed to the emulator processor with execution to begin at the specified address. When the address is not specified, execution begins at the start address of a previously loaded module or execution continues from the last stop point.

The GO command is a forced jump and will supercede a RESET command.

* DOS * Error Responses

37—Invalid GO address

SYNTAX

XEQ { load module file name [/disc drive] }

PURPOSE

The XEQ command causes a binary module to be loaded into program memory and then be executed.

EXPLANATION

The XEQ command is equivalent to the commands LOAD FILENAME and GO.

* DOS * Error Responses

6—Device read error
14—Invalid input device
48—Load file not found
49—Load file assign failure
5Ø—File not a load module
51—Invalid load request

SYNTAX

STATUS

PURPOSE

The STATUS command causes the status of the emulator processor to be displayed on the system console.

EXPLANATION

The STATUS command provides the status of the emulator processor and the program being executed. The status is displayed on the system console. Also the status of any command file in progress is displayed.

The status information and possible parameters are:

(Chip Name) EMULATOR IS	ACTIVE IDLE
(user program name) IS	LOADED EXECUTING IN I/O WAIT SUSPENDED UNDER DEBUG CONTROL
CHAN (n) ASSIGNED TO (device)	OPEN READ WRITE EOF
COMMAND FILE (name) IS	IN PROGRESS SUSPENDED

Section 8

DEBUG SYSTEM

INTRODUCTION

This section describes the methods you may use to monitor your software and hardware execution flow with the debug system. General discussion topics include an overall debug system description, a debug entry and exit sequence, and debug command descriptions. This section provides a working knowledge of debugging capabilities.

The basic operation of the debug and the DEBUG command functions are the same for all emulator processors. However, some command parameter formats and some display formats generated in response to commands vary, depending on the specific microprocessor version. Further information on particular versions of the debug system, related to each type of emulator processor, is contained in the corresponding 8002: Assembler and Emulator User's Manual.

CONTENTS

SECTION 8 DEBUG SYSTEM

INTRODUCTION 8-1
DEBUG SYSTEM STRUCTURE 8-2
DEBUG SYSTEM FUNCTION 8-6
DEBUG SYSTEM ENTRY AND EXIT 8-8
COMMAND DESCRIPTIONS 8-10
- DEBUG 8-11
- TRACE 8-12
- DSTAT 8-18
- BKPT 8-20
- CLBP 8-23
- SET 8-24
- RESET 8-26

DEBUG SYSTEM STRUCTURE

The debug system is a subsystem of TEKDOS. When you invoke the TEKDOS command DEBUG the debug system is transferred from the flexible disc and stored in system memory. This transfer is controlled by the system processor. The debugging routines appropriate to the type of emulator processor being used are loaded into system memory.

Debug System Communications Route

Program memory contents, emulator processor register contents, and user prototype memory contents are made available for your examination through a unique communications route. The communications route that is enabled by the debug system in accomplishing its functions is illustrated in Fig. 8-1.

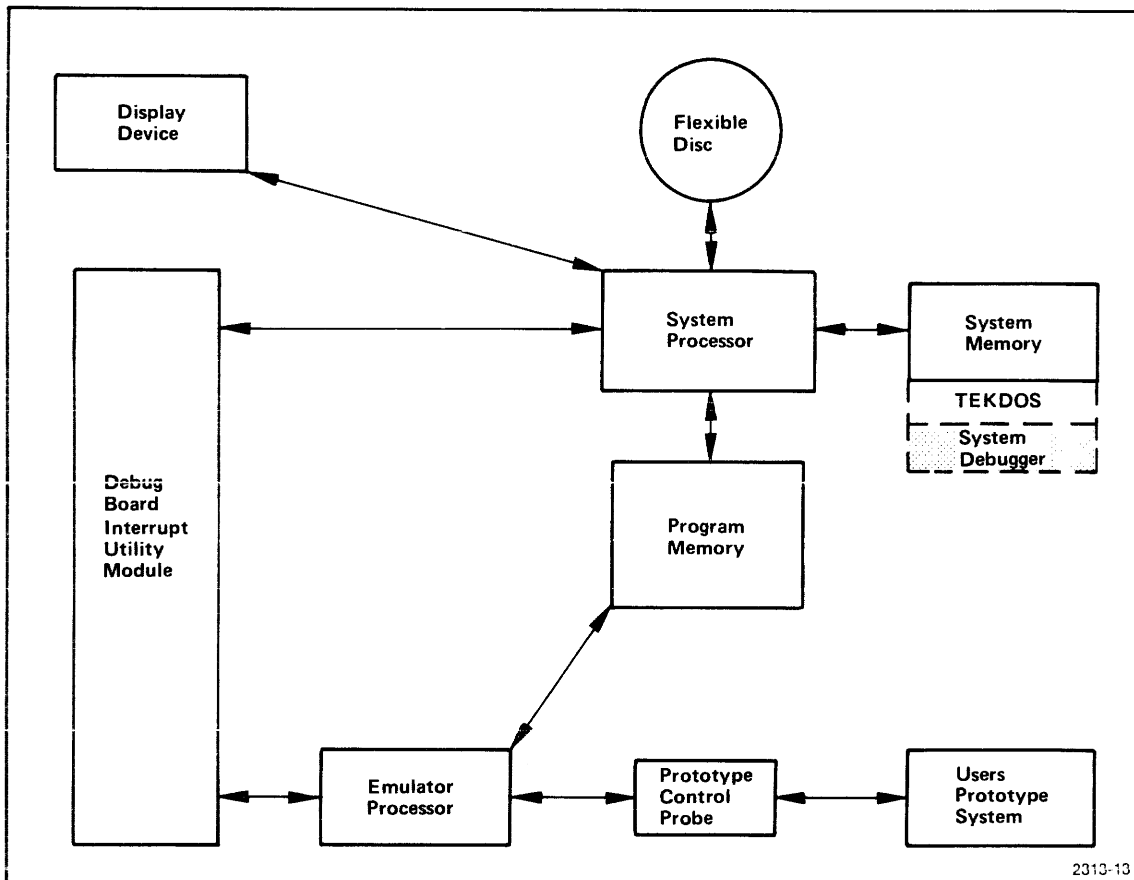
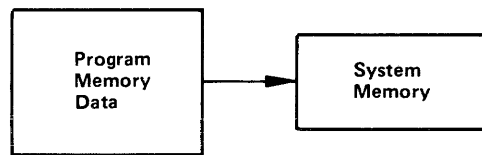


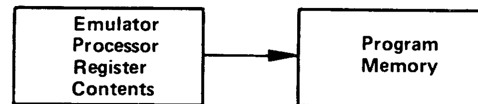
Fig. 8-1. Debug System Communication Route.

Remember that the debug system is controlled through the system processor. The system processor cannot directly access data located in the emulator processor. The system processor can, however, directly access program memory. The communications link between the debug system and the data in the emulator processor registers is achieved by way of program memory. Object program data located in a region of program memory is saved in system memory by the debug system as shown below. This transfer reserves space in program memory for the emulator processor register contents to be stored.



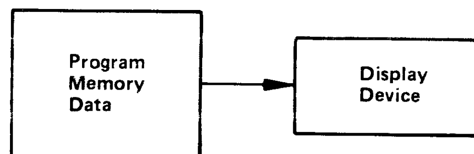
2313-14

The system processor then transfers control to the emulator processor. The emulator processor stores its register contents in the space left vacant in program memory, as shown below.



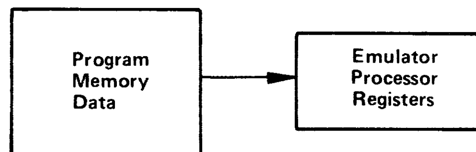
2313-15

The system processor again takes control and accesses the data in program memory, thus allowing you to examine or modify the data by invoking the debugging commands.



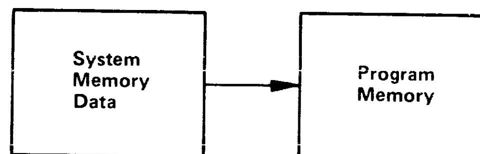
2313-16

Control is passed back to the emulator processor, which restores the data from program memory that has been examined or modified.



2313-17

The system processor again takes control and restores the original object program data back into program memory from system memory.

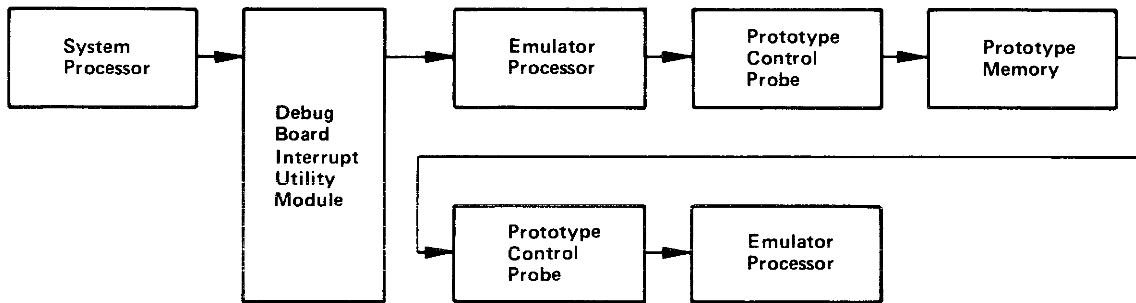


2313-18

Thus, the communications link between the debug system and the data in the emulator processor registers is achieved.

Accessing Prototype Memory

Your prototype's memory contents are directly accessible by only the emulator processor. The system processor transfers control to the emulator processor via the Debug Board's Interrupt Utility Module. The Prototype Control Probe is the cable used to link the emulator processor to your prototype system. When accessing prototype memory, the emulator processor moves data in prototype memory across the Prototype Control Probe. This route is illustrated below:



After the prototype memory is accessed by the emulator processor in the manner above, the debug system achieves a communications link with data in the emulator processor registers in its usual manner.

Summary

As a result of the debug system's communications route, program memory, emulator processor register contents, and prototype memory are available to the system processor to facilitate your examination and modification.

DEBUG SYSTEM FUNCTION

The debug system's structure allows the various examination and modification functions described below:

Examination Commands

The debug system includes several commands that allow memory and register content examination. The following debugging commands are useful when monitoring the effects of program execution.

The TRACE command monitors the progress and state of object program execution and displays pertinent information about desired program locations. Such information pertains to the last instruction executed in the sequence, the instruction mnemonic, index register states, the operand, addresses, and register contents.

The DSTAT command displays information pertaining to the general debugging status such as the emulator processor's last instruction address, the active breakpoints, and the emulator processor's register contents.

Breakpoints may be used to control program execution at specified locations. Breakpoints suspend program execution and allow you to view the effects of read and write executions at specified addresses in program memory. The BKPT and CLBP commands are used to assign and clear breakpoints, respectively.

User program execution is not intended to work in real-time when the TRACE or BKPT commands are invoked. Only the Real-time Prototype Analyzer provides real-time tracing.

Modification Commands

Register content modification may be implemented by way of the SET command.

The emulator processor's hardware sequencing may be reset to a known beginning state with the RESET command.

Other Commands Available While in Debug Mode

In addition to the debug system's command repertoire, a collection of TEKDOS commands is also available while debug is active. The TEKDOS commands that are legal when in debug mode include:

ABORT	GO
ASSIGN	KILL
CLOSE	LOAD
CONT	MAP
DEBUG	MOVE
DELETE	PATCH
DEVICE	RENAME
DSTAT	STATUS
DUMP	SYSTEM
EMULATE	TYPE
EXAM	XEQ
FILL	

The following TEKDOS commands may NOT be used when the debug system is active:

ASM	MODULE
CMPF	PRINT
COMM	RHEX
COPY	RPROM
CPROM	SUSPEND
DUP	VERIFY
EDIT	WHEX
FORMAT	WPROM
LDIR	

DEBUG SYSTEM ENTRY AND EXIT

The sequence below describes the steps to be taken when entering or exiting the debug system.

1. As a precautionary measure, you may choose to write-protect the system disc that contains TEKDOS and supports the emulator processor being used for microprocessor development. Remove the write-enable tab in the lower right corner of the system disc. Insert the system disc into disc drive Ø. Insert the work disc containing your source mode program into disc drive 1.

2. Assign the appropriate emulation mode with the TEKDOS command EMULATE.

3. If you wish to enable the 1ØØ millisecond real-time clock interrupt in emulation mode Ø, enter the TEKDOS command CLOCK ON. If you later wish to operate in emulation modes 1 or 2, you may disable real-time clock interrupt with the TEKDOS command CLOCK OFF. The clock interrupt may be undesirable when operating in emulation modes 1 or 2 since the system processor updates the clock every 1ØØ milliseconds and takes time away from program execution.

4. Invoke the assembler, thus converting the source code to absolute binary object code and storing the code on the specified storage device.

5. Read the absolute binary object code into program memory with the TEKDOS command LOAD.

6. Invoke the debug system with the DEBUG command. The debug system is now loaded into system memory. The TEKDOS prompt character ">" is displayed on the display device. The prompt character indicates the debug system's readiness to accept commands.

7. Invoke any desired debug system commands such as TRACE, DSTAT, BKPT, CLBP, SET, or RESET.

8. Initiate program execution by entering the TEKDOS command GO.

9. If the TEKDOS prompt character is not displayed on the display device during program execution and you wish to enter TEKDOS, the following procedure should be used.

   a. Press the ESC key once.

   b. When the TEKDOS prompt character ">" appears, you may enter the TEKDOS commands that are legal while in DEBUG mode.

   c. When you desire to resume your program's execution, enter the TEKDOS command GO. This action continues your program from the point at which it was interrupted.

10. When program execution is stopped or suspended, the TEKDOS prompt character ">" is displayed and the system awaits your TEKDOS input. Your program is stopped or suspended under the following conditions:

    a. you request control by pressing the ESC key;

    b. your program has encountered a breakpoint that suspends program execution;

    c. your program has executed a halt instruction in emulation mode Ø;

    d. your program has executed one instruction after the TRACE STEP command is invoked; or

    e. your program has reached an End-of-Job condition.

11. The only way you may terminate the debug system is to use the TEKDOS command, ABORT. This may be accomplished by entering ABORT DEBUG or ABORT*. In either case both the debug system and program execution are terminated.

12. When you have completed program or prototype debugging, you may wish to store your modified program code on the specified storage device in hexadecimal format with the TEKDOS command WHEX.

COMMAND DESCRIPTIONS

This section describes the DEBUG command, as well as the commands that are unique to the debug system.

COMMAND NAME	DESCRIPTION	PAGE
DEBUG	Causes the debug system to be loaded into system memory and initialized.	8-11
TRACE	Enables or disables program execution monitoring.	8-12
DSTAT	Displays the current status of the debugging session.	8-18
BKPT	Sets breakpoints in program memory that suspend program execution when read and write operations are performed.	8-20
CLBP	Clears breakpoints previously set in program memory.	8-23
SET	Reassigns hexadecimal values of the emulator processor's registers.	8-24
RESET	Resets the emulator processor hardware to a known beginning state.	8-26

SYNTAX

DEBUG [output device or flexible disc file name]

PURPOSE

The DEBUG command causes the debug system to be loaded into system memory.

EXPLANATION

Entering DEBUG invokes the debug system. DEVICE is the output device or flexible disc file where the debug system's output is written. If DEVICE is not specified, the console output device CONO is assumed. After invoking the debug system the TEKDOS prompt character ">" awaits your debug or TEKDOS commands.

Example

Invoking the DEBUG command below directs any output resulting from the debug commands to the line printer.

```
>DEBUG LPT1
```

SYNTAX

```
TRACE ALL [STEP] [[start address]          {stop address}]
or
TRACE JMP [STEP] [[start address]          {stop address}]
or
TRACE OFF
```

PURPOSE

The debug system allows you to enable or disable program execution monitoring with the TRACE command.

EXPLANATION

The Trace Line

When the appropriate TRACE command mode is invoked, a trace line is displayed to the display device specified in the DEBUG command. The trace line contains one line of program execution along with information pertaining to the executed line. The trace line display format varies with the type of microprocessor under development. This difference is outlined in the appropriate 8002: Assembler and Emulation User's Manual.

User program execution is not intended to work in real-time when the TRACE command is invoked.

For the debug system pertaining to the 8Ø8Ø Emulator Processor, the trace line follows the format below:

LOC INST MNEM OPER SP RF RA RB RC RD RE RH RL

A description of this trace line is outlined below:

LOC	— The location of the last instruction executed.
INST	— The hexadecimal representation of the instruction executed.
MNEM	— The instruction mnemonic.
OPER	— The value or address of the operand.
SP	— The value of stack pointer.
RF	— The value of the flag register.

RA	— The value of register A.
RB	— The value of register B.
RC	— The value of register C.
RD	— The value of register D.
RE	— The value of register E.
RH	— The value of register H.
RL	— The value of register L.

All values are displayed in hexadecimal format.

The Trace Command Forms

The TRACE command may be invoked in three forms: TRACE ALL, TRACE JMP, and TRACE OFF. TRACE ALL and TRACE JMP cause trace lines to be displayed during program execution. The third TRACE command form, TRACE OFF, disables all trace displays.

If TRACE ALL is specified, all instructions executed by the emulator processor have their trace information displayed on the DEBUG display device.

If TRACE JMP is specified, only jump instructions in the program's execution sequence have their trace information displayed on the DEBUG display device.

Following TRACE ALL or TRACE JMP, the TEKDOS command GO may be entered to begin program execution and to display the appropriate trace lines. The appropriate trace lines are then displayed to the DEBUG display device until you suspend program execution or an End of Job condition is reached. Pressing the ESC key once is sufficient to suspend program execution and display during the TRACE command mode. If program execution is suspended in this manner before an End of Job condition is reached, entering GO resumes the execution and display from the point in execution where the suspension occurred.

If STEP is specified when monitoring program execution in either the TRACE ALL or TRACE JMP modes, control is returned to the DEBUG display device after every instruction's trace line is displayed. If the STEP option is used, the TEKDOS command GO must be entered to continue program execution after each trace line is displayed.

If address range 1 through 2 is specified, only the instructions executed between address locations 1 and 2 have their trace information displayed. Addresses 1 and 2 are hexadecimal address constants in the range Ø to FFFF. Address 2 must be equal to or greater than address 1. The default value for address 1 is Ø. The default value for address 2 is FFFF. If an address range is specified and STEP is not, the address range must be preceded by two commas in this manner:

TRACE ALL,,address 1, address 2

The TRACE OFF form disables all trace display. Instruction traces are not displayed on the display device after program execution is invoked with the GO command.

* DEB * Error Responses:

31—Parameter required
35—Invalid start address
36—Invalid end address
44—Invalid trace mode parameter

Example

Suppose the following 8Ø8Ø Assembly Language user program resides on your work disc:

```
START          ORG     ØØ
;
               XRA     A          ;CLEAR ACC
               MOV     B,A
               MOV     H,A
               LXI     D,13FFH    ;LOAD TOP OF MEMORY
               LDAX    D
               DCX     D
               MOV     C,A
               LDAX    D          ;LOAD SECOND NUMBER
               DCX     D          ;DECREMENT POINTER
               MOV     L,A
```

```
;
                DAD     B
                LDAX    D           ;LOAD THIRD NUMBER
                DCX     D           ;DECREMENT POINTER
                MOV     C,A
                DAD     B           ;DOUBLE PRECISION ADD
                LDAX    D           ;LOAD FOURTH NUMBER
                MOV     C,A
                DAD     B
                STC                 ;SET CARRY
                CMC                 ;COMPLEMENT CARRY. I.E. CLEAR IT
;
                MOV     A,H         ;MOVE HIGH ORDER BYTE
                RAR                 ;DIVIDE BY TWO
                MOV     H,A         ;SWAP REG. 4
                MOV     A,L
                RAR
                MOV     L,A         ;SWAP REG.
                MOV     A,H
                RAR                 ;DIVIDE BY TWO UPPER BYTE
                MOV     A,L         ;LOAD LOWER BYTE
                RAR                 ;DIVIDE BY TWO ANSWER IN ACC.
;
                LXI     D,13FFH
                STAX    D
                HALT
                END
```

The program's function is to calculate the average of four numbers and store the result in a specified location. The program is assembled. Emulation Mode Ø is assigned. The absolute binary object code is read into program memory with LOAD. Entering DEBUG puts you in debug mode.

```
>DEBUG
```

Your program's execution may now be traced. Suppose you suspect a logic error between address locations ØØØB and ØØØE in your program's execution. You wish to examine the register contents resulting from the execution of four lines of code at address locations ØØØB through ØØØE. You wish to examine all types of instructions executed. You also want to return control to the console after the trace line of every instruction is displayed. Enter the following sequence:

```
>TRACE ALL STEP ØØØB ØØØE
>GO Ø
```

When the instruction at address location ØØØB is executed, the following system response appears on the display device:

```
LOC   INST  MNEM OPER SP    RF  RA  RB  RC  RD  RE  RH  RL
ØØØB  6F    MOV  L,A  ØØØØ  92  ØØ  ØØ  1Ø  13  FD  ØØ  ØØ
```

Entering GO again executes the next instruction and causes the next trace line to be displayed:

```
>GO
ØØØC  Ø9    DAD  B    ØØØØ  92  ØØ  ØØ  1Ø  13  FD  ØØ  1Ø
```

Continuing this sequence displays the next two trace lines:

```
>GO
ØØØD  1A    LDAX D    ØØØØ  92  41  ØØ  1Ø  13  FD  ØØ  1Ø
>GO
ØØØE  1D    DCR  E    ØØØØ  96  41  ØØ  1Ø  13  FC  ØØ  1Ø
```

All instructions within the range ØØØB through ØØØE have been displayed. Entering GO at this point continues program execution, until the program is suspended or reaches an End of Job condition.

```
>GO
```

Now, suppose you wish to run a continuous trace of all instructions in your program's execution. Enter the sequence below:

```
>TRACE ALL
>GO Ø
```

LOC	INST	MNEM	OPER	SP	RF	RA	RB	RC	RD	RE	RH	RL
ØØØØ	AF	XRA	A	ØØØØ	46	ØØ	ØØ	ØØ	ØØ	ØØ	ØØ	ØØ
ØØØ1	47	MOV	B,A	ØØØØ	46	ØØ	ØØ	ØØ	ØØ	ØØ	ØØ	ØØ
ØØØ2	67	MOV	H,A	ØØØØ	46	ØØ	ØØ	ØØ	ØØ	ØØ	ØØ	ØØ
ØØØ3	11FF13	LXI	D,13FF	ØØØØ	46	ØØ	ØØ	ØØ	13	FF	ØØ	ØØ
ØØØ6	1A	LDAX	D	ØØØØ	46	12	ØØ	ØØ	13	FF	ØØ	ØØ
ØØØ7	1D	DCX	D	ØØØØ	92	12	ØØ	ØØ	13	FE	ØØ	ØØ
ØØØ8	4F	MOV	C,A	ØØØØ	92	12	ØØ	12	13	FE	ØØ	ØØ
ØØØ9	1A	LDAX	D	ØØØØ	92	ØC	ØØ	12	13	FE	ØØ	ØØ
ØØØA	1D	DCX	D	ØØØØ	92	ØC	ØØ	12	13	FD	ØØ	ØØ
ØØØB	6F	MOV	L,A	ØØØØ	92	ØC	ØØ	12	13	FD	ØØ	ØC
ØØØC	Ø9	DAD	B	ØØØØ	92	ØC	ØØ	12	13	FD	ØØ	1E
ØØØD	1A	LDAX	D	ØØØØ	92	B1	ØØ	12	13	FD	ØØ	1E
ØØØE	1D	DCX	D	ØØØØ	96	B1	ØØ	12	13	FC	ØØ	1E
ØØØF	4F	MOV	C,A	ØØØØ	96	B1	ØØ	B1	13	FC	ØØ	1E
ØØ1Ø	Ø9	DAD	B	ØØØØ	96	B1	ØØ	B1	13	FC	ØØ	CF
ØØ11	1A	LDAX	D	ØØØØ	96	54	ØØ	B1	13	FC	ØØ	CF

Trace lines of all instructions are continuously displayed as your program executes until an End of Job condition is reached, a suspending breakpoint is encountered, the space bar is entered to suspend the display, or the ESC key is entered to suspend program execution.

Now, suppose you wish to trace only the jump instructions in your program's execution. You want to display the trace lines resulting from each jump instruction in a step-by-step fashion. Enter the following sequence:

```
>TRACE JMP STEP
>GO Ø
```

The following trace line is displayed showing a jump instruction at address location ØØ24 in your program's execution:

LOC	INST	MNEM	OPER	SP	RF	RA	RB	RC	RD	RE	RH	RL
ØØ24	FE	RST		FFFE	97	15	ØØ	ØØ	13	FF	ØØ	2B

SYNTAX

DSTAT

PURPOSE

The DSTAT command displays the current status of the debugging session.

EXPLANATION

This command sends a display line to the DEBUG display device. The display line includes:

1. The emulator processor's last instruction address;
2. The active breakpoints and the breakpoint parameters; and
3. The emulator processor's stack pointer, flag register, and register contents.

The DSTAT display format varies somewhat, depending on the type of microprocessor used in your prototype.

Example

Suppose breakpoints are set at address locations ØØØ9 and ØØØA in an 8Ø8Ø program. Whenever an attempt is made to read (specified by "R") from either of those address locations, a breakpoint will occur. The following command lines set those breakpoints.

```
>BKPT ØØØ9 R
>BKPT ØØØA R
```

When the program is executed with the GO command, the first breakpoint occurs at address location ØØØ9.

```
>GO
LOC   INST   MNEM OPER SP    RF  RA  RB  RC  RD  RE  RH  RL
ØØØ9  1A     LDAX D    ØØØØ  92  ØØ  ØØ  ØØ  13  FE  ØØ  ØØ
ØØØ9  BREAK
>
```

The second breakpoint occurs at address location ØØØA:

```
>GO
ØØØA   1D     DCR   E       ØØØØ   92   ØØ   ØØ   ØØ   13   FD  ØØ   ØØ
ØØØA   BREAK
```

At this point, a debug status line may be useful in order to examine the effects of the above breakpoints:

```
>DSTAT
P=ØØØA  BP=ØØØ9 R ØØØA R  SP=ØØØØ   RF=92   RA=ØØ  ØØ   ØØ   13   FD  ØØ   ØØ
>
```

The debug status line displays the emulator processor's last instruction address (ØØØA), the active breakpoints and their parameters (ØØØ9 R and ØØØA R), the stack pointer contents (ØØØØ), the flag register contents (92), and the emulator processor register contents (ØØ ØØ ØØ 13 FD ØØ ØØ).

SYNTAX

BKPT {address in program memory} [R/W] [C/S]

PURPOSE

When read and write operations are performed within program execution, you may monitor their effects by suspending execution with breakpoints. Breakpoints are set with the debug system command BKPT.

EXPLANATION

The BKPT command suspends program exection after a read and write operation is performed at a specified address location, ADDRESS. If R is specified, program execution is suspended after an attempt is made to read from ADDRESS. If W is specified, program execution is suspended after an attempt is made to write to ADDRESS. If neither R nor W is specified, program execution is suspended whenever an attempt is made to read from or write to ADDRESS.

If C is specified in the BKPT command line, execution continues after each breakpoint is encountered. If S is specified, execution is suspended after each breakpoint is encountered. If neither C nor S is specified, execution is suspended after each breakpoint is encountered. If R or W are not specified, C or S must be preceded by two commas, in this manner:

BKPT address,,C

When a breakpoint is encountered during program execution, a trace line of the instruction where the break occurred is output to the DEBUG display device. The trace line is followed by a breakpoint message that indicates the current address location followed by the word, BREAK.

User program execution is not intended to work in real-time when the BKPT command is invoked.

Up to two breakpoints may be set at one time. If you attempt to set more than two breakpoints, the system response TOO MANY BREAKPOINTS occurs.

* DEB * Error Responses

30—Invalid Parameter
34—Invalid Address

Example

Suppose the following user program written in 8080 Assembly Language resides on your work disc:

```
START           ORG     00
;
                XRA     A           ;CLEAR ACC
                MOV     B,A
                MOV     H,A
                LXI     D,13FFH     ;LOAD TOP OF MEMORY
                LDAX    D
                DCX     D
                MOV     C,A
                LDAX    D           ;LOAD SECOND NUMBER
                DCX     D           ;DECREMENT POINTER
                MOV     L,A
;
                DAD     B
                LDAX    D           ;LOAD THIRD NUMBER
                DCX     D           ;DECREMENT POINTER
                MOV     C,A
                DAD     B           ;DOUBLE PRECISION ADD
                LDAX    D           ;LOAD FOURTH NUMBER
                MOV     C,A
                DAD     B
                STC                 ;SET CARRY
                CMC                 ;COMPLEMENT CARRY. I.E. CLEAR IT
;
                MOV     A,H         ;MOVE HIGH ORDER BYTE
                RAR                 ;DIVIDE BY TWO
                MOV     H,A         ;SWAP REG. 4
                MOV     A,L
                RAR
                MOV     L,A         ;SWAP REG.
                MOV     A,H
                RAR                 ;DIVIDE BY TWO UPPER BYTE
                MOV     A,L         ;LOAD LOWER BYTE
                RAR                 ;DIVIDE BY TWO ANSWER IN ACC.
```

```
;
                    LXI       D,13FFH
                    STAX      D
                    HALT
                    END
```

The program is assembled and emulation mode Ø is selected. The absolute binary object code is read into program memory with LOAD.

Now suppose you wish to set a breakpoint at address location 13FF. Each time an attempt is made to read from address location 13FF, you wish for the emulator processor execution to halt. Enter the TEKDOS command, DEBUG, to enter debug mode. Enter the following BKPT command line. Then begin program execution starting at location Ø with GO Ø

```
>DEBUG
>BKPT 13FF R
>GO Ø
```

The trace line of the instruction at address location ØØØ6 is displayed, indicating the first attempt to read from memory location 13FF. The address location and the breakpoint message BREAK follow the trace line.

```
LOC   INST  MNEM OPER SP     RF  RA  RB  RC  RD  RE  RH  RL

ØØØ6  1A    LDAX D    ØØØØ   46  4E  ØØ  ØØ  13  FF  ØØ  ØØ
ØØØ6  BREAK
```

Program execution is again suspended after any subsequent execution attempts are made to read from address location 13FF.

SYNTAX

CLBP [address in program memory]

PURPOSE

The CLBP command clears breakpoints that were previously set in program memory.

EXPLANATION

The CLBP command may be used to clear breakpoints set at specified addresses in program memory or to clear all breakpoints. If ADDRESS is specified in the CLBP command line, an active breakpoint set at ADDRESS is cleared. If the address specified has not been previously assigned a breakpoint, the error response BREAKPOINT NOT ACTIVE is displayed. If ADDRESS is not specified, all breakpoints set in program memory are cleared.

DEB Error Responses

32—Too many parameters
34—Invalid address

Example

Suppose breakpoints have previously been set in program memory at address locations 0009 and 000A. An examination of the debug status as shown below verifies these breakpoints:

```
>DSTAT
P=000A  BP=0009 R  000A R   SP=FDFA  RF=92 RA=00  00   00   13   FD  00   00
```

Now suppose you wish to clear the breakpoint set at address location 0009. Enter the command below:

```
>CLBP 0009
```

Further examination of the Debug status shows that the breakpoint previously set at address location 0009 is now cleared.

```
>DSTAT
P=000A  BP=         000A R   SP=FDFA  RF=92 RA=00  00   00   00   13   FD  00
```

SYNTAX

SET R { initial register } { first hex value } {second hex value }

PURPOSE

The SET command allows you to reassign hexadecimal values in the emulator processor's registers.

EXPLANATION

Hexadecimal values "V" are reassigned to the emulator processor registers, beginning with the register specified "Rm". A series of one or more hexadecimal values "V1" through "Vn" is specified after Rm. All registers or any continuous group of registers may be reassigned values. Only those values specified change the register contents. The series of values must not exceed the registers available.

The command "SET Rm V..." causes the emulator processor registers beginning with Rm to be reassigned the values specified. Rm is set to V1. If V2 is specified, Rm + 1 is set to V2, and so forth.

* DEB * Error Responses:

30—Invalid parameter
43—Invalid data parameter
32—Too many parameters

Example

Suppose you observe the register contents below with the DSTAT command:

```
>DSTAT
P=0009  BP=      SP=FDFA  RF=92    RA=00  00    00  13  13  FD  00
```

You wish to reassign registers B through D with the values 1A, 33 and 7. The command line below performs this function:

```
>SET RB 1A 33 7
```

Another look at the register contents shows the change:

```
>DSTAT
P=ØØØ9   BP=      SP=FDFA   RF=92    RA=ØØ   1A    33   Ø7   13   FD   ØØ
```

SYNTAX

RESET

PURPOSE

The RESET command allows you to enable a reset pulse to the emulator processor hardware. Enabling the reset pulse allows the emulator processor's hardware to be reset to a known beginning state. This command is useful if the emulator processor hardware enters an unknown execution state.

EXPLANATION

Invoking the RESET command allows you to arm the debug system with a reset capability. When the emulator processor is activated, a pulse is sent to the reset pin. This allows the emulator processor hardware to be put in an initial state.

The RESET command has no immediate visible effect. After the RESET command is invoked, the GO command overrides the operation of the RESET command and resumes program execution from the point of suspension. If you wish to simulate the effect of the microprocessor's reset signal, enter GO Ø.

Section 9

PROM PROGRAMMER

DOCUMENTATION NOTE

At the time of this writing two PROM Programmer options are available. Option 47 is the 17Ø2A PROM Programmer, and option 48 is the 27Ø4/27Ø8 PROM Programmer. Additional PROM Programmer options may be available in the future.

CONTENTS

SECTION 9 PROM PROGRAMMER

DOCUMENTATION NOTE 9-1
INTRODUCTION 9-2
PROM PROGRAMMER COMMANDS 9-3
  RPROM 9-5
  WPROM 9-6
  CPROM 9-7
HOW TO USE THE PROM PROGRAMMER 9-8
SMS FORMAT COMMANDS 9-8
  CSMS 9-9
  RSMS 9-10
  WSMS 9-11

INTRODUCTION

The 17Ø2A and 27Ø4/27Ø8 PROM Programmer options provide the ability to program PROM (programmable read only memory) chips. Each option consists of an appropriate circuit card module and necessary support software. When the module is installed in the 8ØØ2 μPROCESSOR LAB mainframe, the PROM Programmer software allows object code to be written to or read from a PROM. Object code residing on a programmed PROM can also be compared to object code in program memory.

Because unprogrammed 27Ø4 and 27Ø8 PROMs have all bits set to 1, the PROM program writes Ø's to selected bits. After programming, the PROM can only be erased by exposure to an ultraviolet light source. The transparent PROM cover allows the ultraviolet light to reset all bits to 1. The PROM can then be programmed again.

1702A PROM programming and erasure procedure is identical to that just described for the 2704 and 2708 except that bit settings are reversed. An unprogrammed or erased 1702A has all bits set to Ø. Writing to the 1702A consists of setting selected bits to 1.

PROM Programmer software transfers one data byte at a time. This method permits command parameters to read, write, or compare the total PROM or any contiguous portion.

PROM data byte capacity is shown in the following table:

PROM	BYTES (DECIMAL)	HIGHEST ADDRESS (HEX)
17Ø2A	256	FF
27Ø4	512	1FF
27Ø8	1Ø24	3FF

PROM Programmer Precautions

- You should use care in handling PROMs. Static electricity discharges may destroy the microcircuit. You should ground yourself, preferably through the Logic Ground terminal on the rear panel, while handling the PROM.
- Front panel PROM Power switch must be turned off while inserting or removing a PROM from the socket.
- If more than one PROM Programmer Module is resident in the system insertion of a PROM into the wrong socket, whether the power is on or off, may destroy the PROM. Pay attention to the stick-on label that is attached to the programmer porch alongside the correct socket.

PROM PROGRAMMER COMMANDS

Three PROM programmer commands are available to read, write, or compare PROM data. The respective operating system commands are RPROM, WPROM, and CPROM.

Parameters have identical significance for the three commands, and the default values are the same. The command entry sequence is as follows:

COMMAND n1 type n2 n3 n4

The parameter values are defined in the following table.

PARAMETER VALUE TABLE

PARAMETER	PARAMETER USAGE	DEFAULT VALUE
n1	The first program memory location to be written to, read from, or compared.	Ø
type	The RPOM type to be used is designated as 17Ø2, 27Ø4, or 27Ø8.	27Ø8
n2	The first PROM location to be read from, written to, or compared.	Ø
n3	The last PROM location to be read from, written to, or compared.	highest addressable PROM location
n4	If PROM data is complemented, n4 is set to 1. When set to Ø, or if omitted, PROM data is not complemented.	Ø

If a command is entered without parameter values, default parameters are assumed. When parameter values are entered, a space or a comma is an acceptable delimiter between values.

If parameters are a combination of default and entered values, a comma can be entered for each omitted (default) parameter. A comma is not required for trailing default values.

Examples of command entries are shown in the following table.

ENTRY	=	COMMAND	n1	type	n2	n3	n4
RPROM	=	RPROM	Ø	27Ø8	Ø	Highest Address	Ø
WP,, 17Ø2,,1	=	WPROM	Ø	17Ø2	Ø	Highest Address	1
CP,,,,7F	=	CPROM	Ø	27Ø8	Ø	7F	Ø

As shown here, the command RPROM, with no parameters entered, causes 1Ø24 bytes to be read from a 27Ø8 PROM. The data is written to memory, starting at location Ø.

SYNTAX

RPROM [memory start address] [1702 / 2704 / 2708] [PROM start address] [PROM stop address] [0 / 1]

PURPOSE

The RPROM command transfers data from PROM to program memory.

EXPLANATION

RPROM reads data from the "PROM start address" through the "PROM stop address". The data is stored in program memory beginning at the "memory start address".

If the complement parameter is set to 1, the data is complemented before storage. The End-of-Job message appears on the console when the specified PROM data has been successfully stored in memory.

SYNTAX

WPROM [memory start address] [1702 / 2704 / 2708] [PROM start address] [PROM stop address] [0 / 1]

PURPOSE

The WPROM command writes data from program memory to a PROM in a microprocessor PROM socket.

EXPLANATION

WPROM reads data beginning at the "memory start address" and writes data to the PROM, beginning at the "PROM start address". If the complement parameter is set to 1, the data is complemented before writing occurs.

Writing terminates after the data is written to the "PROM stop address".

For the 1702A, each data byte is read immediately after being written and is compared to the equivalent byte in memory. If the bytes are not equal, the procedure is repeated. If the comparison is still unequal after 16 attempts, the PROM address, the PROM byte, and the memory byte are displayed on the console. The job is then terminated.

For the 2704 and the 2708, all data bytes are programmed 117 times. A comparison is then made between each byte in the PROM and the equivalent byte in program memory. The PROM is programmed once more during this operation for a total of 118 times. If the comparison is unequal, the PROM address, the PROM byte, and the memory byte are displayed on the console. A 2704 or 2708 must have all addresses programmed. If only a part of the PROM is to be programmed, read the PROM into program memory with RPROM. Then change the desired section in program memory and program the PROM using the WPROM command.

A console End-of-Job message indicates successful completion of PROM programming.

SYNTAX

CPROM [memory start address] [1702 / 2704 / 2708] [PROM start address] [PROM stop address] [0 / 1]

PURPOSE

The CPROM command compares PROM data to data in program memory.

EXPLANATION

The CPROM command causes PROM data content to be compared, byte by byte, to data in memory. If the complement parameter is set to 1, data in memory is complemented before comparison. The comparison begins by comparing the data byte at the "PROM start address" with the memory byte at the "memory start address". If the bytes are equal, each address is incremented by one and the procedure is repeated. The byte at the "PROM stop address" is the last PROM location compared.

Inequality between PROM and memory bytes causes the memory address, memory byte content, and PROM byte content to be displayed at the console. The job is then terminated.

Successful comparison between designated PROM and memory bytes is indicated by a displayed End of Job message at the console.

HOW TO USE THE PROM PROGRAMMER

When the TEKDOS prompt character appears:

1. Turn the PROM POWER switch OFF.
2. Push the PROM socket locking lever DOWN.
3. Insert a PROM chip in the socket.
4. Push the locking lever UP.
5. Turn the PROM POWER switch ON.
6. Enter the appropriate PROM command.

SMS FORMAT COMMANDS

Data files in SMS format can be read, written, and compared by the PROM software. The RSMS, WSMS, and CSMS commands, described here, may be used to input or output coded data to peripheral devices and to compare data to program memory.

The SMS format consists of a block of data preceded by a tape-on character and terminated by a tape-off character. The tape-on character is a control R which has the hexadecimal value of 12. As you know, control R is sent by holding down the CTRL key on the terminal while striking the R key. The tape-on character causes the address counter to be set to zero and as a result the first data byte will be stored at location Ø.

An apostrophe is used as a delimiter between data bytes. Each data byte is represented by its hexadecimal value. The last data byte is followed by an apostrophe and then the tape-off character. The tape-off character is control T which has the hexadecimal value of 14.

SYNTAX

CSMS [memory start address] [device or file containing SMS data]

PURPOSE

The CSMS command compares data from an SMS device of file to data in program memory.

EXPLANATION

CSMS reads an SMS file or device, translates to binary, and compares the binary data to program memory data. The default address value is Ø, and the default device is TTYR. The console cannot be the device from which data is compared. If the comparison is not equal, memory location, SMS value, and memory location value are displayed.

SYNTAX

RSMS [memory start address] [device or file containing SMS data]

PURPOSE

The RSMS command reads data from an SMS device or file into program memory.

EXPLANATION

RSMS reads data from an SMS device or file, translates to binary and stores the binary data in program memory. The default address value is Ø, and the default device is TTYR, the console cannot be the device from which data is read.

SYNTAX

WSMS [memory start address] [device or file to be written to]

PURPOSE

The WSMS command writes data from program memory to an SMS output device or file.

EXPLANATION

WSMS reads a 512-byte data block from memory, translates the data to SMS format, and writes the data to an output device or disc file. The default address is Ø, and the default device is the console.

Section 10

SERVICE CALLS

INTRODUCTION

Service calls are used by the emulator processor to obtain input and output service from the system peripherals including the console terminal and the flexible disc drives. This section contains the following:

INTRODUCTION 10-1
SERVICE CALL DESCRIPTION 10-2
SERVICE REQUEST BLOCK 10-3
SRB Bytes 10-5
SVC Functions 10-7
SVC FUNCTION CODES 10-11

SERVICE CALL DESCRIPTION

The system processor in the 8ØØ2 μPROCESSOR LAB has as one of its functions, the monitoring of the emulator processor. Providing all of the input and output for the emulator processor is one of the services included in the monitoring function. The emulator processor has no direct access to any of the system peripherals, including the console terminal and the flexible disc drive. All access to peripherals for the emulator processor are serviced by the system processor.

The emulator processor obtains service from the system processor by issuing a service call (SVC). The service call actually exists as an output instruction sequence in the user program. Some of the items in the instruction sequence include specification of the type of input or output (I/O) to be performed, channel assignments to I/O devices or files, and sizes of buffers for data transfer. The specification for service requested is stored in your program in a block of code called the service request block (SRB).

A total of six SVC's may be defined at any one time in a program. Each SVC refers to a predetermined location in memory. The referenced location and the memory location immediately following it, contain the address of the SRB (service request block). The SRB address stored in the SVC referenced location is called the "SRB pointer" because the stored SRB address points to the service request block. In summary, the SVC refers to the SRB pointer which points to the service request block. The following table shows each SVC and the address of the SRB pointer. These addresses are all hexadecimal.

Table 10-1

SVC REFERENCES (HEXADECIMAL)

SVC	I/O Address	SRB Pointer Location	
1	XXF7	ØØ4Ø	ØØ41
2	XXF6	ØØ42	ØØ43
3	XXF5	ØØ44	ØØ45
4	XXF4	ØØ46	ØØ47
5	XXF3	ØØ48	ØØ49
6	XXF2	ØØ4A	ØØ4B

The SVC is an address, 2 bytes in length. In Table 10-1, the "XX" portion of the I/O address, which is the high order byte of the address word, varies with the microprocessor. In some cases this high order byte may be ØØ. For example, SVC5 shown in Table 10-1 as XXF3, would be ØØF3 when the high-order byte specified for the emulator being used is ØØ.

To restate the concept above, each SRB pointer is stored in a fixed predetermined location and that location is referred to by issuing a corresponding SVC.

SERVICE REQUEST BLOCK (SRB)

A service request block must be included in your program for each service call that is issued. The SRB contains the information that is needed to perform the function requested by an SVC. Each SRB contains eight bytes of data. The following table contains a listing of the SRB contents in the order they must appear in your program.

Table 10-2

CONTENTS OF THE SERVICE REQUEST BLOCK

Byte	Contents
1	SVC function code
2	Channel number
3	SRB status
4	Single byte data
5	I/O byte count
6	I/O buffer length
7-8	I/O buffer pointer

In the following subsections you will find more complete descriptions of the SRB contents and the SVC function codes.

The SRB pointer is the address of the first byte in the service request block. The actual location of the SRB is determined by the assembler at the time of assembling the source code.

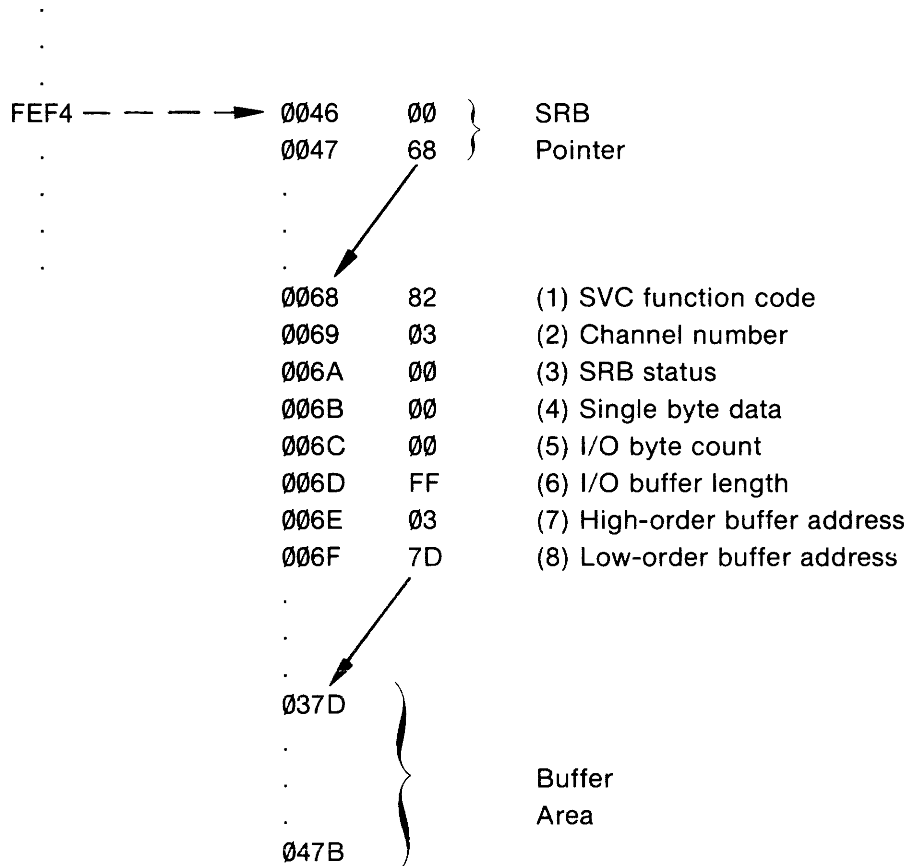


Fig. 10-1. A Typical SVC Instruction Sequence.

In Fig. 10-1 an SVC is issued as FEF4. This SVC refers to the SRB pointer at memory addresses ØØ46 and ØØ47. The SRB pointer contains the address ØØ68 which is the address of byte 1 of the SRB. The byte at this address is the hexadecimal value 82—the function code for Write ASCII and proceed. (The SVC function codes are listed in Table 10-6.) Byte 2 is the assigned channel number over which the action takes place. Bytes 3, 4 and 5 will be provided by the system processor as it services this request. Byte 6 specifies the maximum number of bytes of data to be output. Bytes 7 and 8 contain the starting address of the buffer which holds the data to be output.

Description of SRB Bytes

This subsection contains a description of each byte in the service request block. The SVC function codes, byte 1 of the SRB, are more completely described in the subsection titled "SVC Function Codes".

SVC FUNCTION CODE. The SVC function code, byte 1, specifies the I/O or service function to be performed. The functions are described in the subsection titled "SVC Function Codes", and listed in the table in that section.

CHANNEL NUMBER - Byte 2. A logical channel number must be assigned for each SVC function code that requests I/O service. (See the ASSIGN command in Section 4—TEKTRONIX DISC OPERATING SYSTEM.) The channel number must be in the range Ø to 7. When a channel is assigned to a physical device or file, the channel stays connected to that device or file until a CLOSE command is issued on the channel or the job is aborted.

The console devices CONO and CONI as well as the flexible disc are sharable devices which can be assigned to more than one channel. The other devices are non-sharable and can be assigned to only one channel at a time. A user program can have a maximum of seven channels assigned to files.

SRB STATUS - Byte 3. The system processor stores an SRB status code in this byte. When a "Read and Proceed" or a "Write and Proceed" SVC function is requested, the system processor will write 7F (I/O in progress) in this byte. When the I/O operation is completed, one of the other SRB status codes will be stored in this byte. The following table lists the SRB status codes in hexadecimal, and a short interpretation of each.

Table 10-3

SRB STATUS CODES FOR BYTE 3

Code		Description
ØØ	—	Function complete/no error
Ø1	—	Channel assigned to new file
Ø2	—	Illegal channel number
Ø3	—	Channel not assigned
Ø4	—	Channel busy
Ø5	—	Illegal function code
Ø6	—	No EOL on ASCII read
Ø7	—	No EOL on ASCII write
Ø8	—	Illegal drive number
Ø9	—	File in use
ØA	—	Device not operational
ØB	—	Device not available
ØC	—	Device not ready
ØD	—	Device in use
ØE	—	Directory read error
ØF	—	Directory write error
1Ø	—	Directory full
11	—	Device read error
12	—	Device write error
15	—	File name in use
16	—	Illegal file name
17	—	File in read/write progress
18	—	Channel already assigned
19	—	Incorrect flexible disc
7F	—	I/O in progress
FF	—	End of file or end of device

SINGLE BYTE DATA - Byte 4. This byte is used by the system processor to return single byte data requested by a non-I/O SVC function. For an I/O requested SVC function, the system processor will store the physical status of the device being accessed in this byte.

I/O BYTE COUNT - Byte 5. In this byte, the system processor stores the actual number of bytes of data input or output. For line oriented ASCII I/O operations, this count is the actual number of characters plus the end-of-line character. For binary I/O the count is the actual number of bytes. Byte 5, I/O Byte Count, is also used with Byte 4, Single Byte Data, to return double byte data requested by a non-I/O SVC function (e.g., GET TIME).

I/O BUFFER LENGTH - Byte 6. In this SRB byte, you store the maximum number of bytes for I/O that you will expect from the buffer.

I/O BUFFER POINTER - Bytes 7 and 8. These bytes point to the address of the I/O buffer. The location of this buffer must be in the first 16K page of program memory. This buffer is used for data transfer to or from your program.

SVC Functions

Each of the SVC functions that may be referenced in byte 1 of the service request block is described below.

ASSIGN CHANNEL - Code 10. A channel is assigned to a device or file by issuing an SVC in your program using code 10 in byte 1. Then in byte 2, you specify the channel number in the range of Ø - 7. Bytes 7 and 8 reference the starting address of the device or file name. If a file is named that does not exist, that file will be created and the system processor will store a Ø1 in byte 3 - SRB status.

READ OR WRITE ASCII - Codes Ø1, 81, Ø2, and 82. An ASCII read or write function is performed over the assigned channel specified in byte 2. The maximum number of characters that may be in a transaction is specified in byte 6 - I/O Buffer Length. Every ASCII line must be terminated by a carriage return - ØD hexadecimal. As a result, both byte 6 and the I/O buffer must be large enough to contain the ASCII line plus one character - the carriage return. The system processor will store the status of the I/O device in byte 3 and will store the actual number of characters handled in byte 5. Again, the count of characters in the ASCII line includes the carriage return.

READ OR WRITE BINARY - Codes 41, C1, 42, and C2. A binary read or write function is performed over the assigned channel specified in byte 2. The maximum number of characters that may be in a transaction is specified in byte 6 - I/O Buffer Length. The system processor will store the actual number of characters handled in byte 5. Binary read and write operations are performed according to byte count. Up to 256 bytes of data may be handled.

CLOSE CHANNEL - Code Ø3. The close function disconnects the given channel from the device or file to which it was assigned. When the channel is assigned to a file on a flexible disc, the data stored in the system memory buffer is output to the file and the disc directory is updated to indicate the length of the file.

REWIND FILE - Code Ø4. The rewind function applies only to files on a flexible disc. Rewind has the effect of positioning the file pointer to the beginning of the file. If a device other than a file is assigned to the channel, the rewind function will be treated as a NOP.

When the file is rewound, it is treated as if it had just been assigned. If the first operation for the rewound file is a read, the data is input from the file in the normal manner. If the first operation from the rewound file is a write, the file is treated as if it were a new file.

DELETE FILE - Code Ø5. The delete function causes the file assigned to the given channel to be deleted from the directory of the flexible disc. Also, the channel is disconnected from the file. If a device is assigned to the channel, the delete function will be treated the same as the close function.

RENAME FILE - Code Ø6. To rename a file which has been assigned to the channel specified by byte 2, the I/O buffer pointer (bytes 7 and 8) must contain the starting address of the new name given as an ASCII line. A file which is to be renamed must not be in the process of being read or written. The file must have just been assigned or rewound. If a device has been assigned to the channel specified in byte 2, the rename function will be treated as a NOP.

GET PARAMETER - Codes 13 and 1C. Parameters in the command line of a command invoking an application program are stored in either a procedure buffer or an emulation buffer in system memory. The parameters are identified by number according to the order in which they appear in the command line and are stored as strings of ASCII characters terminated by an EOL character. The desired parameter is requested as a number in byte 4 - single byte data. The actual parameter is returned to program memory and stored as an ASCII line starting at the location specified by bytes 7 and 8 - the I/O buffer location.

When a command line is originally entered from the console, parameters are delimited by a space, comma, or EOL character. A comma or space delimiter is replaced with an EOL character before the parameter is stored in the parameter buffer in system memory. A parameter may be omitted from an ordered sequence by two consecutive commas. When a parameter is omitted, the first character stored by the system processor in the I/O buffer for the SVC will be an EOL character.

When the parameter number requested in byte 4 is greater than the number of parameters included in the command line, the first byte in the I/O buffer for the SVC will be −1. The stored value, −1, will be followed by an EOL character and byte 3 - the SRB Status will contain status code Ø6.

LOAD OVERLAY - Code 17. Overlays stored on a flexible disc may be loaded by a resident user program. Each overlay must be stored on the disc as a load module complete with beginning and ending load addresses, and the address for starting execution. The resident user program loads an overlay by first presetting data in the SRB; then completes the load sequence by issuing the SVC.

The file name of the overlay is given as an ASCII string terminated by an EOL character. Bytes 7 and 8 of the SRB must point to the location of the string. The header information in the load module determines where the overlay is to be loaded in memory. The result of the load operation is stored in byte 4 - status, by the system processor. Execution of the overlay is not started and control remains with the requesting program.

EXECUTE OVERLAY - Code 18. This function is called and performed in the same way as the LOAD OVERLAY function. In addition to this, execution of the overlay is started after being loaded. The EXECUTE OVERLAY function also provides the capability of chaining separate programs.

GET OVERLAY ADDRESS - Code 12. The beginning and ending addresses along with the execution address of the overlay are stored in six consecutive bytes starting at the address given in bytes 7 and 8 of the specified I/O buffer. The first two bytes contain the beginning address, the next two bytes contain the ending address, and the last two bytes contain the address at which execution is to start.

SUSPEND EXECUTION - Code 19. This function will cause the requesting program to be suspended at the location after the one where the SVC is issued. The program can be restarted by entering the TEKDOS command CONT with the appropriate parameter.

EXIT - Code 1A. This function terminates program execution. The assigned channels will not be closed.

ABORT -Code 1F. This function terminates program execution. However, all channels assigned to the program will be closed.

GET TIME - Code 11. This function causes the accumulative time in milliseconds since start of execution, to be stored in bytes 4 and 5 of the SRB. The time will not accumulate if the clock has been disabled by the TEKDOS command CLOCK OFF.

GET DEVICE STATUS - Code 15. This function causes the status of the device assigned to the channel in byte 2, to be stored in byte 4 of the SRB. A zero will be stored in byte 4 if a status is not available.

GET DEVICE TYPE - Code 14. This function causes the device type to be stored in byte 5. This function also causes the identification number of the device assigned to the channel number in byte 2, to be stored in byte 4. The device name, the device identification number, and the device type are shown in the table below.

Table 10-4

DEVICE IDENTIFICATION AND TYPE

NAME	DESCRIPTION	I.D. NUMBER	TYPE CODE
CONI	Console Input	1	1
CONO	Console Output	2	2
LPT1	Line Printer	3	2
TTYR	TTY	6	1
PPTR	Paper Tape Reader	8	1
PPTP	Paper Tape Punch	9	2
REMI	Remote Input	1A	1
REMO	Remote Output	1B	1
Flexible Disc File Name		−1	43

The device type is a code signifying the type of I/O normallyperformed. The device types given in Table 10-4 show the usual way in which the devices are used. A user program can read from any input device in either ASCII or binary. Also, a user program can write to any output device in either ASCII or binary. The full list of codes and a brief description are shown in Table 10-5.

Table 10-5

DEVICE TYPE CODE AND DESCRIPTION

TYPE CODE	DESCRIPTION
1	ASCII read
41	Binary read
2	ASCII write
42	Binary write
3	ASCII read/write
43	Binary read/write

GET LAST CONSOLE INPUT CHARACTER - Code 16. This function causes the last character entered from the console terminal to be stored in byte 4 of the SRB. If sensed in a loop while performing extensive calculations or I/O, this function provides the user program with a way of responding to a request for attention or other action by you.

Table 10-6

SVC FUNCTION CODES (HEXADECIMAL)

CODE	FUNCTION
1Ø	Assign channel to device or channel
Ø1	Read ASCII and wait
81	Read ASCII and proceed
Ø2	Write ASCII and wait
82	Write ASCII and proceed
41	Read binary and wait
C1	Read binary and proceed
42	Write binary and wait
C2	Write binary and proceed
Ø3	Close device or file on channel
Ø4	Rewind file on channel
Ø5	Delete file on channel
Ø6	Rename file on channel
13	Get parameter (procedure parameter buffer)
1C	Get parameter (emulation parameter buffer)
17	Load overlay
18	Execute overlay
12	Get overlay addresses
19	Suspend execution
1A	Exit
1F	Abort
11	Get time (milliseconds)
15	Get device status
14	Get device type
16	Get last console input character

Section 11

REAL-TIME PROTOTYPE ANALYZER

INTRODUCTION

The Real-Time Prototype Analyzer, Option 46, readily isolates prototype problems by providing real-time tracing, event comparison, and expanded breakpoint capability. The prototype address bus, the data bus, and eight selected locations on the prototype circuit board can be dynamically monitored. During the final stages of system integration and debugging, the analyzer can locate timing problems and hardware/software sequence discrepancies.

The Real-Time Prototype Analyzer functions in all emulation modes, and its operation is enhanced by the emulation and debug system modules.

CONTENTS

SECTION 11 REAL-TIME PROTOTYPE ANALYZER

INTRODUCTION 11-1
DESCRIPTION 11-2
COMMANDS 11-3
- EVT 11-4
- BIF 11-7
- RTT 11-8
- DRT 11-9
- CNT 11-10

DESCRIPTION

Component parts of the Real-Time Prototype Analyzer are:

1. The Real-Time Protoype Analyzer module;
2. A Data Acquisition Interface; and
3. An 8-channel Data Acquisition Probe.

The interface and probe permit data transmission between the instrument being tested and the analyzer. Data from the prototype is buffered and driven by the probe to the interface; then to the analyzer module.

The real-time trace buffer, located within the module, can retain up to 128 data words. This dynamic storage ability allows the analyzer to continuously store the last 128 program bus transactions. Each 48-bit word contains a 16-bit address, 8-bit or 16-bit data from the system bus, and 8 data bits from the test probes. The other 8 bits identify cycle type; read, write, I/O memory, or instruction fetch. An identification number assigned to each program starting point permits displays to begin at the most recent start.

Two comparators within the analyzer module can halt program execution and stop or start real-time trace. The comparators can be set to trigger on internal data, address, instruction cycle type, input from the probe, or on any combination of these factors. Triggering can be immediate, or can be delayed until n repetitions of the event combination have occurred. Delay of n clocks is also permitted, where clocks can be trace stores, microseconds, milliseconds, instruction fetches, or I/O operations.

A breakpoint can be enabled so program execution can be halted at any time. Stored transactions can then be displayed or printed, and register contents can be examined. Two comparators (triggers) may be used as independent breakpoints, or they may be combined in the ARM, LIM, IND, or FRZ modes. These modes are defined in the command descriptions contained in this section. The program can be instructed to break when a Read or Write is performed anywhere within a designated address range.

From the time the emulator processor is initiated, transactions are stored continuously until the processor stops for any reason. Whether or not the halt is at a designated breakpoint, contents of the real time trace buffer can be displayed in whole or in part.

Memory mapping, while not part of the Real Time Prototype Analyzer, is used to enhance the analyzer's capabilities. The memory mapping feature divides program memory into 128-byte blocks. One data bit in the buffer is then assigned to represent each block. The state of each bit determines whether the represented transaction is routed to program memory or to user prototype memory.

COMMANDS

The Real-Time Prototype Analyzer commands are described in the sequence shown in this summary.

COMMAND	FUNCTION SUMMARY	PAGE
EVT	Set or display event comparator trigger options.	11-4
BIF	Set or clear break options.	11-7
RTT	Select transaction type to be stored.	11-8
DRT	Display real-time trace buffer contents.	11-9
CNT	Set or display general purpose delay counter units count.	11-10

SYNTAX

EVT [n-trigger] [mode] [option list]

PURPOSE

The EVT command sets or displays event comparator trigger options.

EVT COMMAND PARAMETERS

The n-trigger parameter specifies which trigger is addressed: 1, 2, or both. Both triggers are addressed when this parameter is not entered.

CLR clears all previously designated EVT command options.

Selected options (the option list) must be entered as parameter pairs consisting of an option identifier and an option value. Pair members are separated by relational operators defined as follows:

=	16 bit equality
—	16 bit equality
:	8 bit equality
>	16 bit greater than or equal
<	16 bit less than or equal

The EVT command is not effective while the TRACE command is active.

Available options are shown in the following table.

OPTION IDENTIFIER	OPERATORS	OPTION VALUE	FUNCTION
A	- = : < >	ØØØØ to FFFF	Hexadecimal bus address.
D	- = : < >	ØØØØ to FFFF	Hexadecimal bus data.
T	- =	ØØØØØØØØ to 11111111	Test clip bits. "X" may be used to indicate "don't care".
B	- =		Bus options.
		F	Instruction fetches only.
		I	I/O address only.
		M	Memory accesses only.
		R	Read operations only.
		W	Write operations only.
		IR	I/O reads only.
		IW	I/O writes only.
		MR	Memory reads only.
		MW	Memory writes only.
		ALL	All bus activity.
P	- =	Ø to 65535	Decimal pass count.
C	- =	Ø to 65535	Decimal counter delay units specified by CNT command.

Until the CLR option is entered, the EVT command can be used to modify previously entered options.

The following options may be used in the mode parameter:

ARM — Trigger 1 arms EVT comparator 2. Trigger 2 subsequently arms EVT Comparator 1.

LIM — EVT 1 and EVT 2 must both be satisfied to generate trigger 1.

IND — Trigger 1 and trigger 2 are independent.

CLR — All options are cleared from both comparators.

The EVT mode overrides the current BIF mode.

When entered without parameters, EVT displays the current status of event comparators 1 and 2.

EVT COMMAND EXAMPLES

EVT 1 CLR	Clear EVT 1
EVT	Display status of EVT 1 & 2
EVT 2 CLR A=2DCO	Set EVT 2 to trigger on bus address = 2DCO base 16
EVT 2 B=MR	Modify EVT 2 to respond to memory reads only
EVT CLR	Clear EVT 1 & 2

SYNTAX

BIF [n-trigger] [mode] [return-option]

PURPOSE

The BIF command sets or clears break options for the two event comparators.

BIF COMMAND PARAMETERS

The n-trigger parameter specifies which trigger is addressed: 1, 2, or both. Both triggers are addressed when this parameter is not entered.

CLR clears all previously selected breaks.

Two return-options are available to control execution after breaks. The default value, "S" (step), returns control to TEKDOS at the console. "C" (cont) causes control to revert to the program.

If the "mode" option is selected, one of the following parameters must be entered:

ARM — Trigger 1 arms EVT comparator 2. Break on trigger 2.

LIM — EVT 1 further includes EVT 2 address and control requirements.

IND — Trigger 1 and trigger 2 are independent and either or both may generate a break.

CLR — Clear the selected breaks.

FRZ — Same as ARM. Additionally, the real time trace buffer is frozen at trigger 1.

The BIF modes override previously entered EVT modes. If no parameters are entered, current break options will be displayed.

At a break, a line of text will be displayed at the console. Information in the text line will be contingent upon emulator type, but will contain the program counter value, program memory register contents, and the instruction mnemonic.

SYNTAX

RTT [option]

PURPOSE

The RTT command selects or displays the transaction type stored in the real-time trace buffer.

RTT COMMAND PARAMETERS

The options available are:

F	Instruction fetches only.
I	I/O accesses only.
M	Memory accesses only.
R	Read operations only.
W	Write operations only.
IR	I/O reads only.
IW	I/O writes only.
MR	Memory reads only.
MW	Memory writes only.
ALL	All bus transactions.

If no option is entered, the existing RTT option will be displayed.

SYNTAX

DRT [option]

PURPOSE

The DRT command displays real-time trace buffer content.

DRT COMMAND PARAMETERS

If an asterisk (*) is entered as the option parameter, all buffered bus transactions will be displayed.

If a number, n, is entered, n transactions will be displayed. DRT, without an option parameter, displays only those transactions stored since the most recent program started. If none were stored, none will be printed.

Transactions are displayed sequentially, from oldest to most recent. Blank lines will separate items associated with each start.

Display format is as follows:

>DRT 1Ø

LOC	DATA	MNEMONIC	CLIPS	BUSS
F8BF	FE	OPI	ØØØØØØØØ	M R F
F8CØ	DA		ØØØØØØØØ	M R
F8Ø1	Ø8	RZ	ØØØØØØØØ	M R F
IE9Ø	BA		ØØØØØØØØ	M R
IE9D	3Ø		ØØØØØØØØ	M R
3ØBA	D3	OUT	ØØØØØØØØ	M R F
3ØBB	F3		ØØØØØØØØ	M R
F3F3	ØA		ØØØØØØØØ	I W
3ØBC	ØØ	NOP	ØØØØØØØØ	M R F
3ØBD	76	HLT	ØØØØØØØØ	M R F

>

SYNTAX

CNT [option]

PURPOSE

The CNT command sets the general purpose delay counter count units, or displays the current count value.

CNT COMMAND PARAMETERS

The available options are:

F	Bus instruction fetches.
C	Bus cycles.
S	Emulator clocks.
T	Real-time trace stores.
U	Microseconds.
M	Milliseconds.
1	EVT 1 Compares.
2	EVT 2 Compares.

The delay counter will be reset whenever the program is started. The counter will "freeze" at 65,534 if the count reaches that value.

When EVT or BIF are in ARM mode, the counter will not start until EVT 1 occurs. The count will stop at the next occurrence of EVT 2.

Section 12

INTER-SYSTEM COMMUNICATION

This section will contain a description of the methods, commands and parameters required to effect communications between the 8002 μPROCESSOR LAB and another computer system. One purpose for inter-system communications is to interchange developmental code and data.

Full information for the writing of this section is not available at the time of this printing.

Appendix A

TEKDOS ERROR CODES

1 — DIRECTORY READ ERROR
2 — DIRECTORY WRITE ERROR
3 — COMMAND FILE NOT FOUND
4 — COMMAND FILE INPUT ERROR
5 — PROCEDURE BUSY
6 — DEVICE READ ERROR
7 — DEVICE WRITE ERROR OR END-OF-DEVICE
8 — DRIVE NOT SPECIFIED
9 — INVALID DRIVE
10 — COMMAND LOAD FAILURE
11 — MEMORY AREA IN USE
12 — INVALID FILE NAME
13 — INPUT FILE NOT FOUND
14 — INVALID INPUT DEVICE
15 — INVALID OUTPUT DEVICE
16 — INPUT DEVICE ASSIGN FAILURE
17 — OUTPUT DEVICE ASSIGN FAILURE
18 — DEVICE IN USE
19 — INVALID CHANNEL NUMBER
20 — CHANNEL IS USE
21 — CHANNEL ASSIGN FAILURE
22 — COMMAND LINE BUFFER OVERFLOW
23 — INVALID COMMAND
24 — JOB NOT ACTIVE
25 — JOB NOT SUSPENDED
26 — JOB ALREADY SUSPENDED
27 — JOB EXECUTING
28 — JOB UNDER DEBUG CONTROL
29 — PROM POWER FAILURE
30 — INVALID PARAMETER
31 — PARAMETER REQUIRED
32 — TOO MANY PARAMETERS
33 — BIAS PARAMETER ERROR
34 — INVALID ADDRESS
35 — INVALID START ADDRESS
36 — INVALID END ADDRESS
37 — INVALID GO ADDRESS
38 — INVALID DEBUG USER PROGRAM ADDRESS
39 — INVALID HEX CHARACTER
40 — INVALID RHEX INPUT FORMAT
41 — INVALID BREAKPOINT ACCESS MODE
42 — INVALID REGISTER PARAMETER
43 — INVALID DATA PARAMETER
44 — INVALID TRACE MODE PARAMETER
45 — INVALID EMULATOR SRB ADDRESS
46 — EMULATOR HALTED
47 — SYSTEM AREA BAD
48 — FETCH FILE NOT FOUND
49 — FETCH FILE ASSIGN FAILURE
50 — FILE NOT A FETCH MODULE
51 — INVALID FETCH REQUEST
52 — INVALID DEVICE
53 — INVALID EMULATOR PROCESSOR
54 — INVALID MODE
55 — INVALID MEMORY
56 — INVALID DEVICE ADDRESS
57 — FILE NAME IN USE
58 — DEVICE ASSIGN FAILURE
59 — MEMORY WRITE ERROR
60 — END OF MEDIA
61 — FILE IN USE
62 — DEVICE NOT OPERATIONAL
63 — DIRECTORY FULL
64 — INVALID DISC
65 — SYSTEM MEMORY PARITY ERROR
66 — PROGRAM MEMORY PARITY ERROR

EDITOR ERROR MESSAGES

This section provides a list of all Editor messages and an explanation of their meaning.

** WSP FULL **
The buffer is full.

** NOT FOUND **
The given string could not be found.

** DISC FULL **
Output disc is full.

** NUMBER **
The parameter n is in error.

** RANGE **
The parameter N is an error or an attempt was made to reference lines which are not in the workspace.

** MODE **
An attempt was made to execute a macro string from within a macro string; this is not allowed.

** NEST **
The nesting brackets < and > do not balance.

** COMMAND? **
An unknown command was encountered in the command line.

** BREAK **
The ESCAPE Console Key was depressed to terminate execution of a file I/O function.

** PROCEDURE ERROR **
Editor usage is in error.

** TEKDOS STAT=XX **
XX is the TEKDOS SRB status byte returned to the Editor when an unusual request or event has occurred. The meaning of the status byte can be found in Chapter 10.

** NO PI **
For this editing session there is no PRIMARY INPUT file; the user may not do "GET's" without specifying an Alternate Input file.

** NO PO **
For this editing session there is no PRIMARY OUTPUT file; the user may not do "PUT's" without specifying an Alternate Output file.

** READ FILE? **
An attempt was made to read from a non-existent file or an illegal input device.

** (INPUT) **
The Editor response is in reference to an input attempt.

** (OUTPUT) **
The Editor response is in reference to an output attempt.

** PI **
** PO **
** AI **
** AO **
The Editor response occurred in reference to the Primary or Alternate Input or Output, as applicable.

** NEW FILE **
A new file was created.

** (LPT1) **
The Editor response occurred in reference to the line printer.

** ASSIGN PROBLEM **
The Editor was unable to assign a channel to a given device.

** P1=NEW FILE? **
An attempt was made to "EDIT INFILENAME OUTFILENAME" where INFILENAME and OUTFILENAME were not the same file and INFILENAME was non-existent.

** EOF **
An end-of-file was reached on input or output or the end of workspace text was reached.

** NO FILES SPECIFIED **
The user initiated the Editor without specifying any primary files; for this editing session the user may not do "GET's" or "PUT's" without specifying an Alternate file.

** ABORTED **
A command line exceeded 128 characters and was rejected.

** TRUNCATED **
An INPUT line exceeded 128 characters and was truncated to the first 128 characters entered.

A SUBSTITUTE caused the line to exceed 128 characters and the line was truncated to 128 characters (see example in paragraph 6.5.5).

Appendix B

TABLES

CONTENTS

APPENDIX B **TABLES**

HEXADECIMAL-DECIMAL CONVERSION B-3
HEXADECIMAL ADDITION B-5
HEXADECIMAL MULTIPLICATION B-7
POWERS OF 2 .... B-9
ASCII CODE CONVERSION B-11

HEXADECIMAL TO DECIMAL CONVERSION TABLE

HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0000	0	000	0	00	0	0	0
1000	4,096	100	256	10	16	1	1
2000	8,192	200	512	20	32	2	2
3000	12,288	300	768	30	48	3	3
4000	16,384	400	1,024	40	64	4	4
5000	20,480	500	1,280	50	80	5	5
6000	24,576	600	1,536	60	96	6	6
7000	28,672	700	1,792	70	112	7	7
8000	32,768	800	2,048	80	128	8	8
9000	36,864	900	2,304	90	144	9	9
A000	40,960	A00	2,560	A0	160	A	10
B000	45,056	B00	2,816	B0	176	B	11
C000	49,152	C00	3,072	C0	192	C	12
D000	53,248	D00	3,328	D0	208	D	13
E000	57,344	E00	3,584	E0	224	E	14
F000	61,440	F00	3,840	F0	240	F	15

HEX	F000 + B00 + 70 + 3 = FB73
DEC	61440 + 2816 + 112 + 3 = 64371

HEXADECIMAL ADDITION TABLE

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

HEX	F + 8	=	17	
HEX	10	=	16	DEC
HEX	7	=	7	DEC
HEX	17	=	23	DEC

HEXADECIMAL MULTIPLICATION TABLE

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

HEX 9 x 8	=	48
HEX 40	=	64 DEC
HEX 8	=	8 DEC
HEX 48	=	72 DEC

TABLE OF POWERS OF TWO

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1 024
11	2 048
12	4 096
13	8 192
14	16 384
15	32 768

DEC. $2^8 = 256$

ASCII CODE CONVERSION TABLE

HEXADECIMAL

MOST SIGNIFICANT CHARACTER

	—	Ø	1	2	3	4	5	6	7
	Ø	NUL	DLE	SP	Ø	@	P	`	p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	$	4	D	T	d	t
LEAST	5	ENQ	NAK	%	5	E	U	e	u
SIGNIFICANT	6	ACK	SYN	&	6	F	V	f	v
CHARACTER	7	BEL	ETB	'	7	G	W	g	w
	8	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	¦
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	Λ	n	≈
	F	SI	US	/	?	O	—	o	DEL

EXAMPLES

W = 57
H = 48
a = 61
t = 74
@ = 4Ø
NUL = ØØ
DEL = 7F

Appendix C

SYSTEM INSTALLATION

INTRODUCTION

This appendix describes unpacking, installation and interconnection instructions for the 8002 μPROCESSOR LAB and Flexible Disc Unit. Refer to the individual peripheral manuals for specific unpacking and installation procedures for these peripheral devices.

UNPACKING

The 8002 μPROCESSOR LAB and Flexible Disc Unit are shipped in separate cartons. Before unpacking these units, inspect each carton for signs of external damage. If any damage is detected, contact your Tektronix representative.

Unpacking the 8002 μPROCESSOR LAB

To unpack the 8002 μPROCESSOR LAB, open the outer carton and remove the corner packing supports. Lift the inner carton out and place on a flat surface. Remove the coiled power cord taped to the top of the inner carton and set it aside until the unit is to be connected to the primary power source. Open both ends of the inner carton and slide the 8002 μPROCESSOR LAB out of the carton. Remove the plastic bag, cardboard packing and front panel protective foam cushion from around the unit. The top cover is secured to the 8002 μPROCESSOR LAB with three screws located along each side of the unit. Remove the screws and lift the top cover straight up. Remove the protective foam cushion on top of the modules (printed circuit cards).

The recommended arrangement for the modules (printed circuit cards) is shown in Figure C-1. The modules may be arranged in other configurations if the following guidelines are not violated:

1. Positions 1 and 2	PROM programmer modules may be located in either of these positions. (2 positions)
2. Positions 3 thru 8	The system modules may be located in any position within this section. (6 positions)
3. Position 9	Debug/front panel module. (1 position)
4. Positions 10 thru 20	The program modules may be located in any position within this section. (11 positions)

Due to power supply limitations, no more than two emulator processor modules should be installed in a 8002 μPROCESSOR LAB configured for 60 Hz operation. No more than one emulator processor module should be installed in a 8002 μPROCESSOR LAB configured for 50 Hz operation (special order).

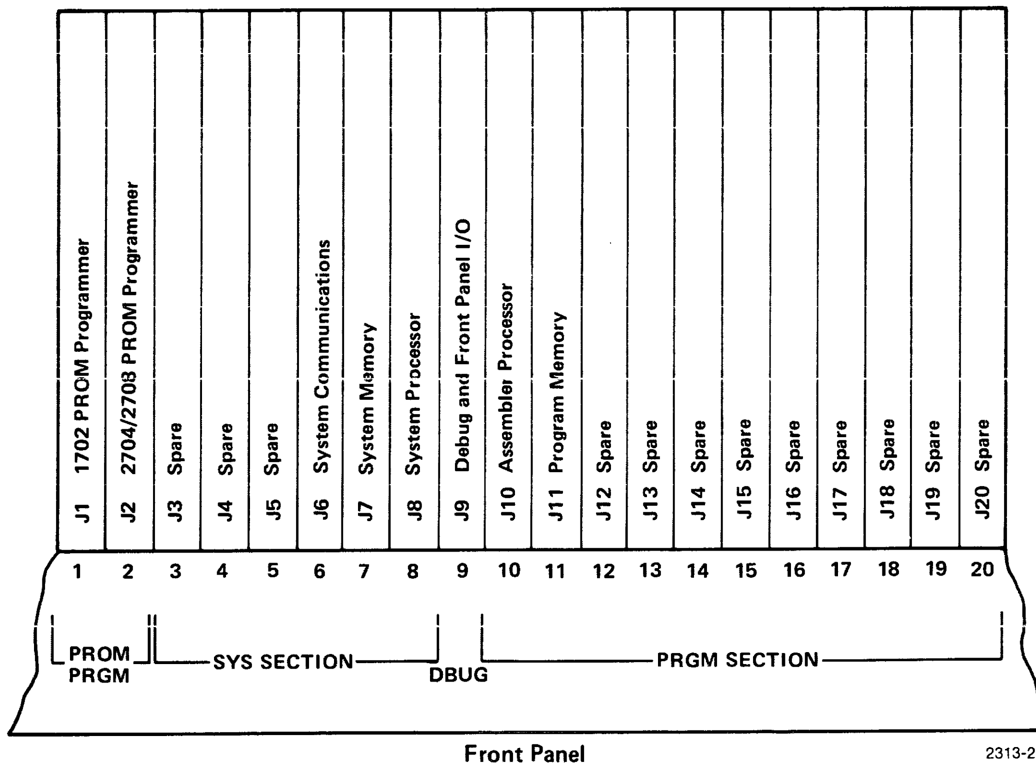


Fig. C-1. Typical Module Arrangement for the 8002 μPROCESSOR LAB.

NOTE

The location of a module within a section may be limited by the length of interconnecting cables.

After modules have been correctly positioned, make sure they are properly seated by pushing them firmly into the sockets. The module's 1ØØ pin edge connector is offset to prevent the modules from being installed backwards. The ribbon cables and connectors within the 8ØØ2 μPROCESSOR LAB are installed prior to shipping. Make sure each connector is properly seated. Figure C-2 illustrates the correct location for each connector on the modules. Visually inspect inside the unit for physical damage that might have been incurred during shipping. Do not replace the top cover on the 8ØØ2 μPROCESSOR LAB at this time.

NOTE

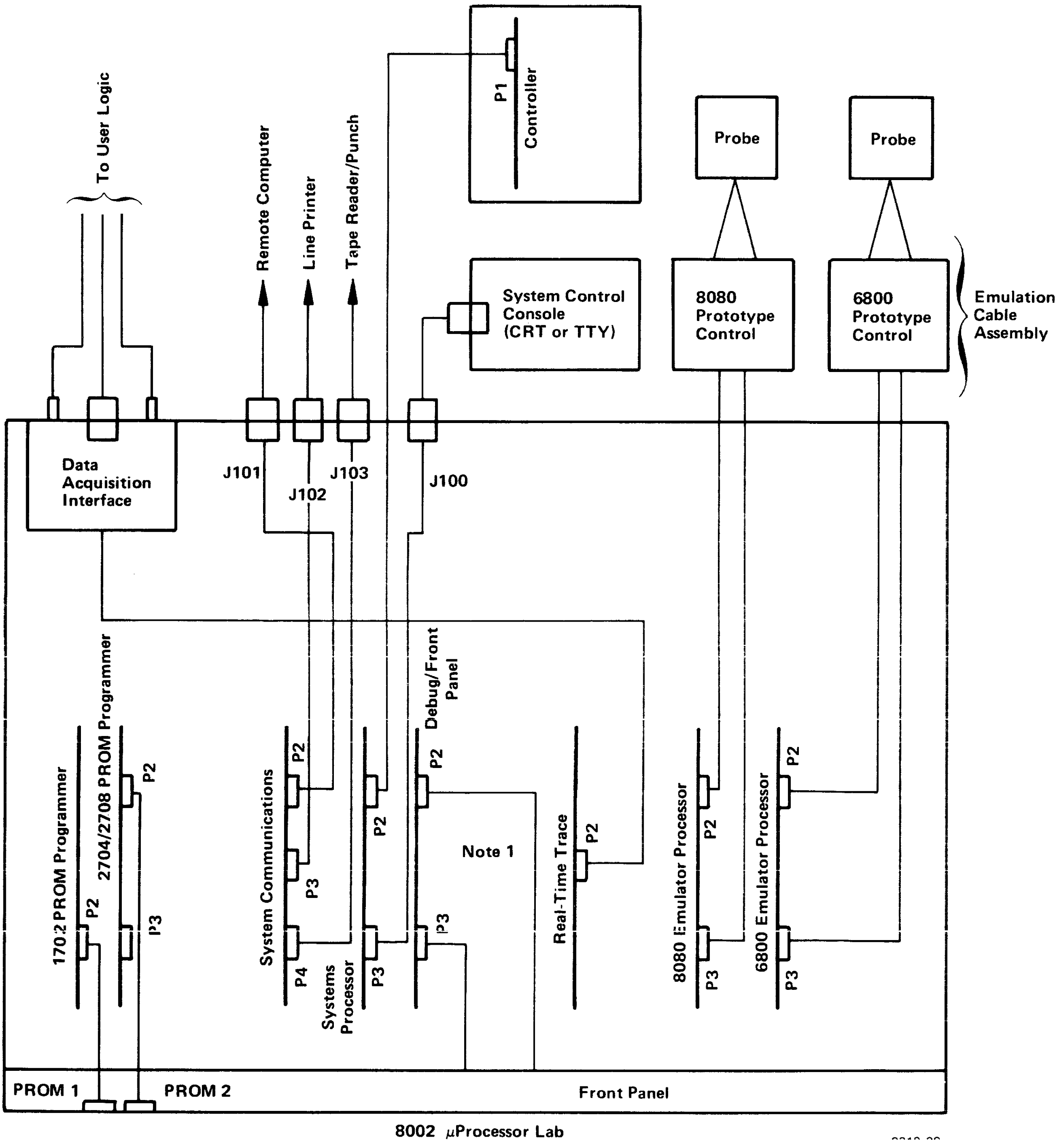
The red wire on each cable designates the side of the cable plug that is connected to pin 1 in the mating connector.

Unpacking the Flexible Disc Unit

To unpack the Flexible Disc Unit, follow the same procedure as for the 8ØØ2 μPROCESSOR LAB. Remove the three screws along each side and lift the top cover straight up. Visually inspect inside the unit for physical damage that might have been incurred during shipping. Uncoil the Flexible Disc Unit interconnect cable (Part No. 9ØØ14Ø21) and feed it through the channel opening provided in the rear panel. Replace the top cover on the Flexible Disc Unit.

INSTALLATION

The 8ØØ2 μPROCESSOR LAB and Flexible Disc Unit should be installed and operated on a flat surface. The units should be close enough to each other for the interconnecting cables to reach. (Figure C-3 shows the envelope dimensions for each unit.) Both units draw cooling air through openings in the bottom of the cabinets. The air is expelled out the back by two fans. These units should not be located where paper, plastic, carpeting or other materials might block the air intakes and cause overheating. Allow at least two inches clearance behind the units so that the air flow is not restricted.



NOTES:

1. P2 to front panel cable used only when the Maintenance Front Panel is used on the μProcessor Lab.
2. All units shown (except Emulation Cable Assemblies) are connected to primary power source (115 or 230 Vac). Power cables not shown.
3. PROM 1 may be connected to either 1702A or 2704/2708 PROM Programmers. PROM 2 is connected only to 2704/2708 PROM Programmer.

Fig. C-2. Typical Cabling Diagram of the 8002 μPROCESSOR LAB System Installation.

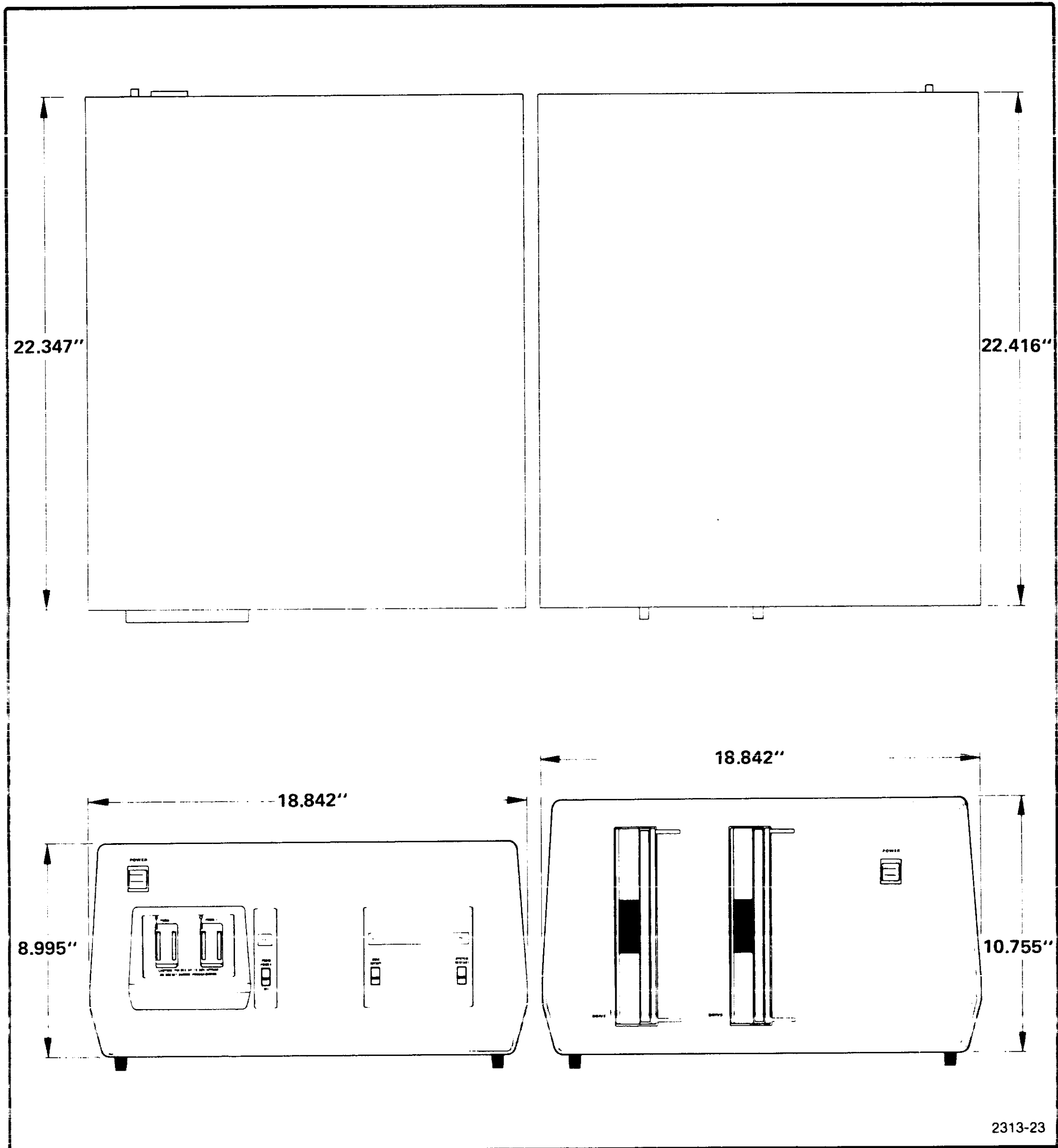


Fig. C-3. 8002 μPROCESSOR LAB System Envelope Dimensions.

CABLE INTERCONNECTIONS

Refer to Figure C-2 for the system interconnect diagram. Connect the cables as follows:

1. Connect the Flexible Disc Unit to the 8002 μPROCESSOR LAB by routing the 40 conductor ribbon cable (Part No. 90014021) from the rear of the Flexible Disc Unit through the center cableway on the rear panel of the 8002 μPROCESSOR LAB to P2 on the system processor module. Make sure that pin 1 of the cable plug (red strip) is mated to pin 1 on P2. Replace the top cover on the 8002 μPROCESSOR LAB.

2. Other interconnecting cables from optional or peripheral equipment must be accomplished in accordance with Figure C-2 and the individual equipment manuals.

3. Auxilliary bus and ground connections are provided to the user from terminal block TB2 located on the back panel of the 8002 μPROCESSOR LAB. The following connections are available on TB2 from within the 8002 μPROCESSOR LAB.

TB2-1 to auxilliary power bus lines
TB2-2 to logic ground bus lines
TB2-3 to chassis ground

NOTE

A shorting strap is installed between TB2-2 and TB2-3 which provides a single common tie point between logic ground and chassis ground. In the event the shorting strap is removed a one megohm resistor is provided between logic and chassis ground.

POWER SOURCE

The 8002 μPROCESSOR LAB and Flexible Disc Unit are both designed to be operated from a single-phase power source that has one of its current-carrying conductors (neutral) at ground (earth) potential. Operating from other power sources where both current-carrying conductors are live with respect to ground (such as phase-to-phase on a multiphase system, or across the legs of a 110-220 volt single-phase, three wire system) is not recommended, since only the line conductor has over-current (fuse) protection within the unit.

The ac power connector is a three-wire polarized plug with one lead connected directly to the instrument frame to provide electric shock protection. Connect this plug only to a three-wire outlet which has a safety ground. If the unit is connected to any other power source, the unit frames must be connected to a safety ground system.

The 8002 μPROCESSOR LAB and Flexible Disc Unit are both designed to operate from a 115-230 volt nominal line voltage that has a frequency of 50-60 Hz. The Flexible Disc Unit is factory wired for 115 volts, 60 Hz operation; however, 230 volts, 50 Hz configuration is available by special order. Each unit has a separate power cord and requires a separate outlet for the primary power. The system power requirements are listed below:

8002 μPROCESSOR LAB	3.5 amps at 115 V, ±10%, 60 Hz
	2.0 amps at 230 V, ±10%, 50 Hz
Flexible Disc Unit	3.0 amps at 115 V, ±10%, 60 Hz
	1.5 amps at 230 V, ±10%, 50 Hz

NOTE

The Flexible Disc Unit has a warning sign above the primary input power jack stating the unit is wired for a specific voltage and frequency at the factory.

The system fusing requirements for 115 volt or 230 volt primary power source are listed below:

8002 μPROCESSOR LAB	AMP	VOLTS
Primary (F4)	6 A	115 V
	3 A	230 V
±12 V Supply (F3)	2 A	115 V
	1 A	230 V
Flexible Disc Unit		
Primary (F1)	4 A	115 V
	2 A	230 V
Disc Drive (F3)	2.5 A	115 V
	1.5 A	230 V

WARNING

Dangerous voltages exist at several places inside the units. Disconnect the 8002 μPROCESSOR LAB and Flexible Disc Unit from the power source before removing or replacing the top cover. Only qualified technical personnel should attempt to change the power supply jumper arrangement. Unfamiliarity with electronic equipment and safety procedures can result in personal injury.

REPACKAGING FOR SHIPMENT

If either the 8ØØ2 μPROCESSOR LAB or Flexible Disc Unit is to be shipped to a Tektronix Service Center for service or repair, attach a tag showing: owner (with address), name of individual at your firm that can be contacted, complete serial number, and a description of the service required. If the original packaging is unfit for use or not available, repackage the equipment as follows:

1. Obtain a carton of corrugated cardboard having inside dimensions of no less that six inches more than the equipment dimensions; this allows for cushioning.

Refer to Table C-1 for carton strength requirements.

2. Surround the equipment with polyethylene sheeting to protect the finish.

3. Cushion the equipment on all sides with packing material or urethane foam between the carton and the sides of the equipment.

4. Seal with shipping tape or industrial stapler.

TABLE C-1
Shipping Carton Test Strength

Gross Weight		Carton Test Strength	
Pounds	Kilograms	Pounds	Kilograms
0-10	0-3.73	200	74.6
10-30	3.73-11.19	275	102.5
30-120	11.19-44.76	375	140.0
120-140	44.76-52.22	500	186.5
140-160	52.22-59.68	600	223.8

APPENDIX D
COMMAND INDEX

The minimum set of characters for each command is underlined.

COMMAND	PAGE
ABORT	4-33
AGAIN	5-62
ASM	6-2
ASSIGN	4-38
BEGIN	5-40
BIF	11-7
BKPT	8-20
BRIEF	5-60
CLBP	8-23
CLOCK	4-37
CLOSE	4-39
CMPF	4-20
CNT	11-10
CONT	4-32
COPY (TEKDOS)	4-21
COPY (EDITOR)	5-26
COPYSYS	4-17
CPROM	9-7
CSMS	9-9
CTRL-Z	4-27
DEBUG	8-11
DELETE	4-19
DEVICE	4-36
DOWN	5-38
DRT	11-9
DSTAT	8-18
DUP	4-15
DUMP	7-14
EDIT	5-4
END	5-41
EMULATE (TEKDOS)	4-40
EMULATE (EMULATOR ENVIRONMENT)	7-4
ESC (TEKDOS)	4-29
ESC (EDITOR)	5-57
EVT	11-4
EXAM	7-16
FETCH	7-12
FILE	5-14
FILL	7-23
FIND	5-42
FORMAT	4-10
GET	5-17
GO	7-26
INPUT	5-11
INSERT	5-13
KILL (TEKDOS)	4-47
KILL (EDITOR)	5-48
LDIR	4-18
LINK	6-5
LIST	5-33
LOAD	7-10
MAP	7-19
MACRO	5-54
MODULE	7-11
MOVE	7-22
N	5-35
PATCH	7-18
PRINT	4-24
PRINTL	4-24
PUT	5-21
PUTK	5-23
QUIT	5-59
RENAME	4-13
REPLACE	5-46
RESET	8-26
RHEX	7-9
RPROM	9-5
RSMS	9-10
RTT	11-8
RUB OUT	4-28

COMMAND	PAGE
SET	8-24
Space Bar (TEKDOS)	4-26
Space Bar (EDITOR)	5-56
STATUS	7-28
SUBSTITUTE	5-44
SUSPEND	4-31
SYSTEM	4-35
TAB	5-51
TABS	5-53
TRACE	8-12
TYPE (TEKDOS)	4-48
TYPE (EDITOR)	5-32
UP	5-36
VERIFY	4-12
WHEX	7-8
WPROM	9-6
WSMS	9-11
XEQ	7-27
*	4-46
?	5-61

APPENDIX E

SOFTWARE ERROR REPORT FORMS

You may feel that you have discovered an error in the software for the 8ØØ2 μPROCESSOR LAB. Please fill out one of the following Software Performance Reports, describing the error in as much detail as possible. Also indicate whether the error is predictably repetitive or only occurs randomly.

When you have completed the Software Performance Report, please mail the report to Tektronix, Inc., LDP, Group 132, P.O. Box 500, Beaverton, Oregon 97005. Postage is prepaid by Tektronix, Inc. A reply will be forthcoming.

Name ______________________

Address ______________________

Phone ______________ Ext ________

Date ______________________

Tektronix
COMMITTED TO EXCELLENCE

LDP SOFTWARE PERFORMANCE REPORT

Software Title ______________

Version ______________

Release ______________

Date ______________

Type (for 8080, 6800, etc) ______________

REASON FOR REPORT

O Design Failure
O Software Error
O Documentation Error
O Suggestion

PRIORITY

O Low
O Standard
O High

Is the error reproducible? O Yes O No O Intermittent

SYSTEM CONFIGURATION

O

	O 8002				
EMULATORS	O 8080	O 6800	O Z80	O 9900	O Others ________
OPTIONS	O R.T.P. Analyzer	PROM PGMER:	O 1702	O 2708	O Others ________
PROGRAM MEMORY	O 16K	O 32K	O 48K	O 64K	
PERIPHERALS	O CT8100	O CT8101	O LP8200	O Others ________	

DESCRIPTION OF PROBLEM (Please attach any listings if available).

____
____
____
____
____
____
____
____
____
____

Please send to: **TEKTRONIX, INC, LDP,** GROUP 132, P.O. Box 500, Beaverton, OR 97005

REPLY

Date ______________

Error # ______________

Signed ______________

FOLD FOLD

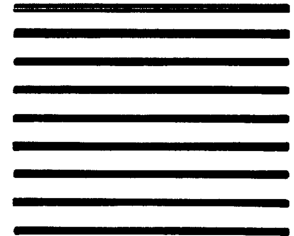
FIRST CLASS
PERMIT NO. 61
BEAVERTON, OREGON

BUSINESS REPLY MAIL

No postage necessary if mailed in the United States

Postage will be paid by

TEKTRONIX, INC.
LOGIC DEVELOPMENT PRODUCTS
GROUP 132
P. O. Box 500
Beaverton, Oregon 97005



FOLD FOLD