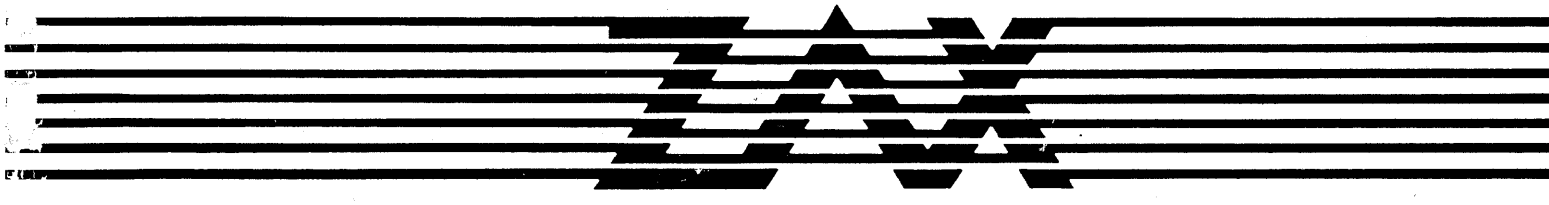
ERX
FOR
64180
USER'S MANUAL



Zax Corporation

Limitation on Warranties and Liability

ZAX Corporation warrants this equipment to be free from defects in materials and workmanship for a period of 1 (one) year from the original shipment date from ZAX. This warranty is limited to the repair and replacement of parts and the necessary labor and services required to repair this equipment. During the 1-year warranty period, ZAX will repair or replace, at its option, any defective equipment or parts at no additional charge, provided that the equipment is returned, shipping prepaid, to ZAX. The purchaser is responsible for insuring any equipment returned, and assumes the risk of loss during shipment.

Except as specified below, the ZAX Warranty covers all defects in material and workmanship. The following are not covered: Damaged as a result of accident, misuse, abuse, or as a result of installation, operation, modification, or service on the equipment; damage resulting from failure to follow instruction contained in the User's Manual; damage resulting from the performance of repairs by someone not authorized by ZAX; any ZAX equipment on which the serial number has been defaced, modified, or removed.

Limitation of Implied Warranties

ALL IMPLIED WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO THE LENGTH OF THIS WARRANTY. IN NO EVENT WILL ZAX BE LIABLE TO THE PURCHASER OR ANY USER FOR ANY DAMAGES, INCLUDING ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES, EXPENSES, LOST PROFITS, LOST SAVINGS, OR OTHER DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS EQUIPMENT. THIS EXCEPTION INCLUDES DAMAGES THAT RESULT FROM ANY DEFECT IN THE SOFTWARE OR MANUAL, EVEN IF THEY HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

Disclaimer

Although every effort has been made to make this User's Manual technically accurate, ZAX assumes no responsibility for any errors, omissions, inconsistencies, or misprints within this document.

Copyright

ZAX Corporation
Technical Publications Department
2572 White Road, Irvine, California 92714

Contents

Reserved Memory
Spaces

I/O Address	ROM Writer	8-bit ERX	16-bit ERX	32-bit ERX
100 to 17F	N/A	P	P	P
180 to 1DF	N/A	I	I	I
1E0 to 1EF	R	N/A	N/A	N/A

Key: N/A; unused Host I/O space
R; host I/O space reserved for ROM writer
P; host I/O space reserved for ERX Personality Pod
I; host I/O space reserved form ERX interface modules

ERX for 64180 Features

GENERAL CHARACTERISTICS

* Real-time Execution for Transparent Emulation
* PC-driven Control and Symbolic Debug
* 128K Bytes Emulation Memory
* Adjustable Real-time Trace Buffer
* 64,000 Breakpoints x 4 channels (up to 256,000 breakpoints)
* High-Speed Static RAM Ensures No Added Wait States
* All memory available to the target
* All I/O ports available
* On-line Help Facility
* One-Year Warranty

USER INTERFACE

* You control all functions from computer
* Symbolic debugging available through ER-ICE
* Mnemonic command names
* Setup emulation control from batch file on computer

EMULATION CONTROLS

* Software selectable internal or external clock
* Disable interrupt inputs
* Disable bus request input

MEMORY MAPPING

* 128K bytes emulation memory
* 1K mapping resolution
* Mapping options:
    Read/Write
    Read Only
    User/Target Memory
    No Memory (guarded access)

BREAKPOINTS

* Break on specific address or data
* Break on memory status
* Break on symbol
* Break on I/O port access
* Break after executing command
* Break on access to non-memory area
* Break on write to read-only area
* Sequential break (up to four levels deep)
* Break on Nth occurrence
* Break on range

SINGLE-STEP CAPABILITY

* Single step
* Step N steps
* Display jump instructions only

REAL-TIME TRACE

* Stores address, data and status
* 4K deep x 32 bits wide trace memory size
* Trigger on event or symbol
* Trace control modes include:
    - Begin Monitor
    - End Monitor
    - Begin Event
    - End Event
    - Center Event
    - Multiple Event
    - Inner Event
    - Outer Event
* Adjustable trace range

DISASSEMBLY CAPABILITIES

* Disassemble from program memory
* Disassemble trace memory from any selected area

SPECIAL FEATURES

* Assemble into memory
* Search program memory for pattern
* Instruction execution counter
* Built-in timing analysis
* Multiple event tracing
* Subroutine simulation
* On-line help facility

SYSTEM REQUIREMENTS

IBM PC-AT or any 100% compatible computer with minimum one diskette drive (360K or 1.2M) and 512K-byte RAM.

ERX EMULATION SPECIFICATIONS

Processors Supported (Max Clock Speed)
  ERX for 64180 (6MHz)
Resident Emulation Memory
  128K-bytes high-speed static RAM
Mapping Resolution
  1K-byte blocks
Real-time Trace Buffer
  4K bytes deep x 48 bits wide

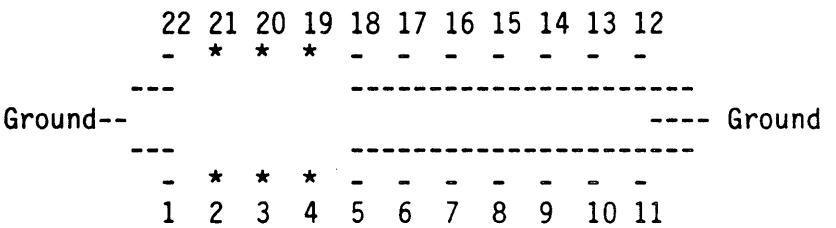
PHYSICAL SPECIFICATIONS

Personality Pod Dimensions
  75mm (3.0in) high
  160mm (6.3in) wide
  220mm (8.7in) deep
Auxiliary Module Dimensions
  330mm (13in) high
  125mm (5in) wide
Interface Cable Dimensions
  1000mm (39.3in) x 60mm (2.4in)
Emulation Probe Dimensions
  500mm (20in) x 85mm (3.2in)
Weight/Main Unit
  1.2kg (2.6lb)
Power Requirements
  +5VDC

ERX Clock Modification

The XTAL and EXTAL pins are used to connect a parallel resonant fundamental crystal, AT cut. For instance, because a divide-by-4 circuit in employed, in order to obtain the system clock of 1 MHz, a 4MHz resonant fundamental crystal is used.

To attach a crystal clock, find the clock module at location M1 on the TRU interface module (internal to the ERX Personality Pod). Connect one of the crystal leads to pin 2, 3 or 4. Connect the remaining lead to pin 19, 20 or 21. Solder the crystal case to any ground pin (5 through 18).



* - Location of crystal leads.

SECTION 1 ERX DESCRIPTION AND OPERATION

Introduction

In Section 1 you'll learn about the different components that make up your ERX, what they do, and how to use them. You'll also learn how to connect the ERX to your system (computer, printer, target system), and find out about how to use the accessories that come with the ERX. Your ERX has a few special features that you should know about too; you can find information about these features in this section as well.

A Word Of Caution!

You shouldn't try to attach the ERX to any external device before you finish reading this section. As long as power is absent to the emulator you can't hurt anything internally, but don't connect the ERX to your target system before you read "How To Connect Your ERX To Other Devices." Although it is difficult, it is possible to reverse the cables going to the target system, which could result in damage to the ERX's internal components as well as your target system.

Getting Acquainted With Your ERX

Your ZAX ERX-series emulator is a PC-driven microprocessor emulation device that can be used for developing and maintaining microprocessor-based (HD64180) systems. It does this by letting you direct and test activities in your prototype ("target") system. You carry out these operations by entering one or more of the emulator's debugger commands.

ZAX ERX-series emulation equipment require the presence of a computer (AT-class) for control. This arrangement has several advantages over dedicated microprocessor development systems and even stand-alone emulators. Instead of operating as an independent device, the ERX uses several of the computer's facilities and is therefore smaller and less intricate in its design than a typical emulator, yet just as powerful!

The ERX emulation system incorporates six "modules" (also called circuit boards or "cards") that permit emulation of the target processor, memory management, program tracing and performance evaluation. Two of the modules are common to all 8-bit ERX-series emulators and are installed within your computer on a semi-permanent basis. (Before you can use the ERX emulator, you'll need to install these two auxiliary modules within your computer.) The other four modules reside in the ERX Personality pod.

An emulation probe connects the ERX Personality pod to your target system for running hardware tests.

After you form the ERX system, you'll use simple mnemonic command statements to invoke the debugger functions. You can use the debugger commands by simply inputting the command statements from the computer keyboard.

A Few Features

Here are just a few things you can do using the debugger commands:

* Use the ERX's emulation memory to simulate or take the place of memory (or future memory) in your target system.

* Use a single-step trace operation to move through your program, one step at a time, and examine the registers' contents after each step.

* Set breakpoints to stop your program when: data is written or read into a specific address, an event point or symbol is passed, a non-existent memory access is attempted, or an interrupt is acknowledged by the MPU. Hardware breakpoints can also generate triggers for instruments such as logic analyzers and oscilloscopes.

* Substitute an emulator command to simulate a not-yet-written subroutine.

* Analyze the time spent executing subroutines.

* Record ("trace") a portion of your program (beginning and ending anywhere within the program) and store it in the ERX real-time trace buffer without affecting the emulation process. Later you can display the recorded memory contents in either machine code or in its disassembled format.

* Translate symbolic codes into machine instructions, item for item, using the in-line assembler.

* Selectively enable and disable the interrupt inputs - including non-maskable interrupts.

You can turn to Section 2 for a complete list of the ERX's debugger commands. To find out about other things your ERX can do, turn to "More About Your ERX."

Now turn the page to learn about how to connect your ERX to other devices.

A Word About Grounds and Power

About Grounds

Your ERX receives its power (+5 VDC) from the auxiliary modules located in your computer. The computer, in nearly all cases, receives its power through a three-wire polarized power cord. This cord connects to a power source and protective ground. Make sure that you plug the power cord into a properly grounded 115 VAC receptacle. Do not try to bypass the three-prong plug with an adapter (three-into-two-prong adapter).

WARNING: THE GROUND TERMINAL OF THE 3-PRONG PLUG IS USED TO PREVENT SHOCK HAZARDS - DO NOT BYPASS IT!

About Power

Your ERX runs on a voltage supply of +5 VDC provided by the computer. This +5 volt supply is provided for the ERX regardless of the computer's operating voltage. To determine the proper operating voltage of your computer for your application, consult your computer's operator manual.

NOTE: Whenever possible, use a multiple power outlet strip to provide voltage to the entire system (computer, printer, target system). Most power outlet strips are equipped with a circuit breaker in case of an overload, and all are properly grounded.

ERX Inventory Checklist

The ERX emulation system features the following:

- 1 ERX Personality Pod (contains 4 modules)
- 2 Auxiliary Modules (reside in computer)
- 3 Interface Cables

ERX Personality Module

Forming the ERX Personality pod itself are four modules:

- 2 MPU (Microprocessor Unit) Modules
- TRU (Transfer Unit) Module
- EMU (Emulator Control Unit) Module

One of the MPU modules contains the HD64180 processor that permits emulation of the prototype or target system. It also contains two receptacles for receiving the emulation probe. The TRU module manages the coverage and performance facilities. The EMU module controls the emulation mode or monitor mode of operation for the ERX. The TRU and EMU modules each contain a receptacle that accepts the interface cables running to the computer. The four modules are housed in the top and bottom case covers which form the ERX emulator.

Auxiliary Modules

The two auxiliary modules include:

- MMU (Memory Mapping Unit) Module
- RTS (Real-time Storage) Module

The two auxiliary modules are installed in your computer. The MMU module contains high-speed static RAM which can be used for downloading files, altering the memory contents and loading future memory into the target system. The RTS module includes the controller, memory and real-time counter for tracing and storing program execution.

The two auxiliary modules are designed for simple installation into two of the computer's 16-bit (long) expansion slots. The only restriction on positioning the interface modules is that they be located adjacent to one another (a short bus cable joins the two auxiliary modules together).

Interface Cables

The three interface cables are used to connect the modules to your computer and target system. The cable that connects the ERX Personality pod to your target system is called the "MPU emulation probe." Each cable is designed to connect to a specific component at a specific location; you cannot substitute one cable for another.

Connecting Your ERX To A Personal Computer

The following describes the steps that must be taken to prepare your computer and physically install the two auxiliary (RTS and MMU) modules, and how to connect the ERX Personality pod to your computer and target system.

NOTE: Instructions are given for the IBM PC-AT machine only; you should consult your reference manual if you are installing the auxiliary modules in an IBM compatible computer. In this case, we recommend using a 100% IBM PC-AT compatible computer.

Tool List

The following tools are needed to install the two auxiliary modules:

- Flat-blade or Phillips-type screwdriver
- 3/16-inch nutdriver or 3/16-inch wrench
- Small needle-nose pliers

Removing the PC Cover

In order to gain access to the expansion slots on the PC motherboard, you must disconnect the system unit (mainframe) from the other system components and remove the unit cover. The following steps explain how to prepare your computer for installing the auxiliary modules. For non-IBM units, refer to the appropriate installation manual for instructions on removing the system cover.

Step 1. Switch off the power to the system unit. Switch off the power on all peripheral equipment (printer, monitor, etc.). Unplug the system unit and other optional equipment from the wall outlet or power supply.

Step 2. Disconnect all peripheral cables (including the keyboard cable) from the rear panel of the system unit. Remove all peripherals from the work area.

Step 3. Position the system unit so that you have easy access to the rear panel.

Remove the cover mounting screws located on the rear panel, using a flat blade screwdriver.

The number of screws depends on the make of your PC. The screws are located in the upper and lower left and right corners and the upper center of the back panel.

Turn the screws counterclockwise. After removing the screws, place them in a safe location.

Step 4. Carefully slide the system unit cover forward from the rear as shown below. When the cover will go no farther, tilt the cover up and remove from the base. Set the cover in a safe place.

Installing the Auxiliary Modules

The following steps explain how to install the two auxiliary modules.

Step 1. Facing the disk drive(s), look at the inside left rear of your system unit. Several expansion slots (some long, some shorter) are available for additional modules. You can install the two auxiliary modules in any two adjacent full-length (16-bit) slots.

Step 2. Using a flat-blade screwdriver or a 3/16-inch nutdriver, remove the screws that hold the system expansion slot covers in place (see below).

Step 3. Insert the two auxiliary modules into the motherboard connectors which face up on the motherboard. Place the RTS module into the full-length slot that is closest to the power supply. Now, place the MMU module (the module with the second receptacle on top) into the full-length slot adjacent to the RTS module. Press down firmly on both auxiliary modules to be sure that they are securely seated in the connectors.

Connecting the Interface Cables

The following steps explain how to connect the two interface cables to the two auxiliary modules that are located in the computer.

Step 1. Note the identification marking on the RTS and MMU auxiliary modules, and then note the same markings on the interface cables.

Step 2. Connect the appropriate interface cable to the corresponding auxiliary module. Note the key on the end-connectors and then press the cable connectors firmly into the module receptacles. Now, snap the latching clips into place to lock the connectors.

CAUTION: DO NOT REVERSE CABLE POSITIONS TO THE AUXILIARY MODULES. MISMATCHING THE CABLE POSITIONS WILL CAUSE SEVERE DAMAGE TO THE ERX.

Step 3. Connect the RTS and MMU modules together with the short bus cable. Note the key on the end-connectors and then press the cable connectors firmly into the module receptacles. Now, snap the latching clips into place to lock the connectors.

The ERX is now ready for operation. Refer to "Installing ER-ICE" for an explanation on installing the communications program, or continue on to learn how to connect the ERX Personality pod to your target system.

Connecting Your ERX To A Target System

The MPU emulation probe is used to connect the ERX Personality pod to your target system when you are emulating the HD64180 MPU. The probe consists of a 20-inch multi-way cable and a 60-pin end connector. The 60-pin end connector of the probe plugs into the target system's microprocessor socket.

The following steps explain how to connect the MPU emulation probe to your target system.

Step 1. Remove the existing MPU (HD64180) from your target system.

Step 2. Carefully insert the ERX's MPU emulation probe (60-pin end) into the target system's MPU socket. Be careful not to damage the pins of the end connector.

NOTE: Pin #1 of the probe's end connector goes into pin #1 of the target system's MPU socket.

The ERX is now ready for operation. Refer to "Installing ER-ICE" for an explanation on installing the communications program.

Installing ER-ICE

ER-ICE is ZAX's communications/symbolic debug program for the ERX emulation system. The program is contained on a single diskette that is included with your ERX system. Before invoking ER-ICE, the program must be properly installed on your particular system.

NOTE: The ERX emulator is designed for interface to an IBM PC-AT or 100% compatible computer. Other systems are not recommended.

To install ER-ICE, complete the following steps:

Step 1. Bootup your computer, and then insert the ER-ICE diskette into any available drive. On the diskette reside four files:

```
ERX180V.EXE (executable program)
ERX180V.MAC (autoexecuteable macro)
ERX180V.HLP (help menu - contains
             command syntax format
             and examples)
CVT.18 (conversion program)
```

Step 2. Transfer the files to your system's fixed-disk drive by using the INSTALL.BAT file. To do so, enter the following:

```
INSTALL
```

When you do so, ER-ICE will place the three working files in a directory called, ZAXBIN.

Step 3. Now, enter the following:

```
[drive:][path]ERX180V
```

to invoke the communications/symbolic debug program.

The DOS prompt will vanish for approximately 15 seconds and then the general ERX identification message will appear followed by the following prompt:

```
ERX180V>
```

This prompt indicates that the system is working properly and that the ERX is ready to accept commands.

NOTE: When the ERX initializes itself, it defaults its entire memory space (0-FFFFH in ERX) to the "user" memory specification. In this mode, the ERX assumes that you'll be executing memory entirely out of the target and not out of the emulator. If you wish to examine ERX memory at this point, you must re-map the memory as read/write or read-only. To do so, enter the following command:

```
MA 0,0FFFF=RW <CR>
```

or refer to the MAP command for more information.

What Can You Do With Your MDS?

You should now have a fully operational Microprocessor Development System (MDS) capable of developing and debugging your hardware and software designs. If your MDS is functioning correctly, and the ERX's identification message appears on your screen, you can now:

* Turn to the "Master Command Guide" in Section 2, for a complete analysis of your ERX's debugger commands;

* Turn to Appendix B for a demonstration of the features and functions of your ERX;

* Use the fold-out "Command Reference Guide" (from the front of this manual) as a source for the various command formats.

NOTE: Appendix B and the Command Reference Guide are not included in this manual. They will be available in the first quarter of 1988.

What To Do If Your MDS Is Not Working

If your MDS is not functioning correctly or gives you problems during emulation, turn to "Trouble Shooting," on the next page. Start by reading "Checking Electrical Connections," and then proceed to "Diagnosing Other ERX Problems" if you encounter problems when you're emulating.

Introduction: The Problem ...

Because you must install the interface modules within the computer and then connect the various interface cables to each component of the system, there is always the possibility of misplacing a cable, misaligning a module, or bypassing a procedure. The result of this exercise is sometimes a system that works improperly, or worse, doesn't work at all.

... And The Solution!

"Trouble Shooting" is designed to get you through the problems you might have encountered in "How To Connect Your ERX To Other Devices," and begins with a typical example of what the ERX should do if the system is operating correctly. Then the ERX is disconnected and reconnected to ensure that the cables are properely positioned.

What Should Happen

After installing the interface modules in the computer and then connecting the interface cables to the ERX Personality pod and computer, you enter the following command:

```
ERX180V(X) <CR>
```

The DOS prompt will vanish for approximately 15 seconds and then the general ERX identification message will appear followed by the following prompt:

```
ERX180V(X)>
```

This prompt indicates that the system is working properly and that the ERX is ready to accept commands.

At this point, any of the "status commands" (command name followed by a RETURN) can be entered.

Enter one of the following:

B, CLO, ID, MA or R

If the response from the ERX is the command's status, then the system is probably functioning properly.

To correct this condition you will need to either attach a target system to the ERX or remap all or a portion of ERX memory to read/write or read-only, as shown in the example below:

```
>DI 100,2FF <--tries to disassemble range of memory
ER-ICE: Target memory access failure at 0100.
>
>MA 0,0FFFF=RW <--remaps memory as read/write
>DI 100,2FF <--successful memory disassembly
00100 0100     MAIN      NOP
00101 0101               LD B,H
00102 0102               NOP
00103 0103 ...
```

At this point, the next snag may involve the use of the ERX command syntax, specifically, using a non-existant symbol name to define an address, as in the following:

```
>DI 0,EFF
```

Here, the ERX sees the "EFF" as a symbol name rather than an address value. To convert the parameter to an address, simply insert a zero before the "E" as shown below:

```
>DI 0,0EFF
```

Another potential obstacle exists in the default clock speed of the ERX's MPU. With the ERX for 64180, there are two external and three internal clock speeds:

External: TTL
External: XTAL
Internal: 6MHz
Internal: 3MHz
Internal: 1.5MHz

The default internal clock speed is 3MHz, but this may be changed to 6MHz or 1.5MHz by using the CLOCK command. If your application requires a different clock speed, see the CLOCK command in Section 2.

What To Do If The ERX Still Doesn't Work

In most cases, the procedures just listed will solve all but the most stubborn problems. However, it is possible that your ERX is still not functioning correctly. If this is the case, you should consult directly with ZAX Corporation's Customer Service department at the number below:

1-800-421-0982 (outside California)

1-800-233-9817 (in California)

SECTION 2 ERX MASTER COMMAND GUIDE

Fast Start

If you are already familiar with the ERX command syntax, you may choose to begin debugging and/or developing with your ERX immediately. In this case, enter the following commands to initialize your ERX:

```
>MA 0,0FFFF=RW <--remaps ERX memory to internal
>R RESET <--resets registers
>R <--examines registers
>L[/source] filename[.ftype] <--downloads file from
                                    host computer
>G address <--begins program execution
```

Introduction

All ERX-series emulators respond to mnemonic commands (e.g., G for Go, B for Break) entered from the computer keyboard. By using a simple-to-understand and universally adopted command structure (as opposed to complicated logic statements), ZAX emulators are able to perform hundreds of testing and debugging tasks, immediately.

The following chart and listing defines and explains the principal debugger commands available with the ERX emulator.

Debug Mode	Edit Mode	Batch/Macro	Miscellaneous
BREAK	ABASE	BATCH	BEEP
COVERAGE	AMAP	CGET	CALCULATE
EVENT	ASSEMBLE	CLOSE	CLOCK
EDELETE	COMPARE	CPUT	EMSELECT
ESAVE	DISASSEMBLE	DEFM	ERX
ESHOW	DUMP	DISPLAY	HELP
GO	EXAMINE	ECHO	IDENTIFICATION
HISTORY	FILL	EOF	PROMPT
ICERESET	IF	EXECUTE	QUIT
PERFORMANCE	LOAD	FNKEY	
PIN	MAP	GET	
PORT	MOVE	GOTO	
RESET	SAVE	JOURNAL/	
SCOPE	SEARCH	NOJOURNAL	
STEP	REGISTER	KEY	
STOP	VERIFY	LET	
STUB		LOG/NOLOG	
TRIGGER		LOOPOUT	
WAIT		MACRO	
		MDELETE	
		MLOAD	
		MODLEN	
		MSAVE	
		MSHOW	
		OPEN	
		PAUSE	
		PUT	
		REM	
		REPEAT	
		SHELL	
		SYMLEN	
		WHILE	

Debug Mode

BREAK...Stops program execution on a variety of different parameters. When the conditions satisfying the parameters are met, program execution halts and control of the emulator is returned to the user.

COVERAGE...Flags each address passed during program execution, compares it to a range, and then reports (as a percentage) the number of address lines passed during program execution, or displays the passed or unpassed address lines themselves.

EVENT...Defines an event in the program. An event may be defined by a symbol name; an address location or range; a memory type; and a data value or range of values. Once an event is specified, it may be directed to act as a breakpoint or a trigger.

EDELETE (Event DELETE)...Deletes a pre-set event point either by its symbol name or number.

ESAVE (Event SAVE)...Stores the parameters of an event point or a series of event points that were previously defined with the EVENT command.

ESHOW (Event SHOW)...Displays the event point(s) and the parameters of the event point(s). The display can also include a range of previously defined events expressed as either numbers or symbols.

GO...Executes the user program from the current program counter or a defined address.

HISTORY...Records program execution in real time and then displays it in either machine or disassembled format.

ICERESET...Halts emulation and resets the MPU and I/O of the ERX Personality pod (causes the Reset pin to go low).

PERFORMANCE...Records and displays the total emulation time, the number of event points passed during program execution, the time duration between event points, the average time of each event duration and the the percentage of total event duration to total emulation time.

PIN...Masks or unmasks selected input signals, including the RESET, NMI (non-maskable interrupt), STBY (stand-by), and IRQ (interrupt request) signals, or all the interrupt signals.

COMPARE...Compares the contents of specified memory blocks within the ERX or target system and then displays the non-matching data along with their locations. The comparison can be made between different memory blocks as mapped to the ERX, or between one block of memory within the ERX and one within the target system.

DISASSEMBLE...Translates the memory contents from machine codes to assembly language mnemonics, and then displays the converted contents.

DUMP...Displays the memory contents in both hexadecimal and ASCII format.

EXAMINE...Inspects one or more memory locations and optionally modifies them. The locations can be displayed and changed with either ASCII or hexadecimal values.

FILL...Fills a block of memory with either hexadecimal or ASCII codes.

IF...Allows conditional execution of commands dependent upon specific register, memory or port contents; also allows specification of arithmetic and bit-wise operators.

LOAD...Downloads object files from the host computer in either Intel, Motorola or dump format. Also downloads a previously saved event file to the ERX.

MAP...Categorizes the ERX/target system memory functions as read/write memory, read-only memory, user (target system) memory, or nonexistent memory (guarded access).

MOVE...Moves the memory contents between different locations within the ERX, or between the ERX and the target system.

SAVE...Saves an Intel, Motorola or dump format file to the host computer.

SEARCH...Searches through the memory contents and displays the matching or unmatching data, if any.

REGISTER...Displays the status of a register or all the registers, and optionally modifies the register(s) contents.

VERIFY...Compares the contents of specified memory blocks within the ERX or target system and acknowledges the match, if any. If the match is exact, nothing is displayed; otherwise the non-matching data is displayed along with their locations. The comparison can be made between different memory blocks as mapped to the ERX, or between one block of memory within the ERX and one within the target system.

Batch/Macro Commands

BATCH...Executes a series of command grouped as a batch file.

CGET (Console GET)...Allows the entry of numerical values, or string values into a numerically specified "N" variable.

CLOSE...Closes a previously opened file and then informs you if the file was closed successfully.

CPUT (Console PUT)...Allows the entry of numerical values, or string values into a numerically specified "N" variable.

DEFM (DEFault Module)...Allows alteration of the default module name.

DISPLAY...Enables and disables the display of symbols during disassembly.

ECHO...Enables or disables the display of the command lines within a batch or macro file.

EOF (End-Of-File)...Informs you if the end-of-file has or has not been reached.

EXECUTE...Allows the execution of a single DOS command from within the ER-ICE environment.

FNKEY (Function KEY)...Allows the execution of a pre-defined function key from within a batch file. The function is specified by the KEY command.

GET...Fetches numeric values or ASCII characters from a file and assigns them to a variable.

GOTO...Allows you to branch to a label located within a batch file or macro.

JOURNAL, NOJOURNAL...Opens a file for storing all subsequent commands until a NOJOURNAL command is issued.

KEY...Program the function keys to a command or series of commands.

LET...Assigns values to variables.

LOG, NOLOG...Opens a file which is used to store all subsequent user commands and ERX output for later processing. The NOLOG terminates command logging.

LOOPOUT...Breaks out of a loop condition from within a macro or batch file.

MACRO...Creates an unlimited number of user-defined commands that can be locally created, and loaded/ saved from/to a disk file.

MDELETE (Macro DELETE)...Deletes a macro saved in the host computer.

MLOAD (Macro LOAD)...Loads a macro stored in the host computer.

MODLEN (MODule LENgth)...Sets the length of module names to be used on the screen display.

MSAVE (Macro SAVE)...Saves a macro to a file in the host computer.

MSHOW (Macro SHOW)...Displays the name and contents of all the macros.

OPEN...Opens a file in the host computer for reading or writing.

PAUSE...Suspends execution of a batch file until a key on the keyboard is depressed.

PUT...Displays ASCII text or variable values to a file.

REM (REMark)...Allows you to insert comments into batch files.

REPEAT...Executes a command or series of commands "x" number of times

SHELL...Allows another system command interpreter (shell) to be run while preserving all symbols and the ER-ICE environment.

SYMLEN (SYMbol LENgth)...Varies the number of symbol-name characters to be used for output.

WHILE...Repeats a command or a block of commands as long as a specified condition remains lexicographically true.

Miscellaneous

BEEP...Sounds the system bell to indicate the completion of a batch file.

CALCULATE...Performs addition, subtraction, multiplication and division of hexadecimal, decimal, octal and binary numbers. It also performs base conversions.

CLOCK...Sets the clock configuration for the MPU via the keyboard. Four internal speeds are selectable and two external inputs are available through the target system.

EMSELECT (Emulation Method Select)...Allows you to control signal I/O between the ERX and target system during emulation.

ERX...The executable program for the ERX emulator.

HELP...Summons the on-line help menu for the ERX and provides on-line syntax and examples for ERX commands.

IDENTIFICATION...Displays the ERX model, software and hardware versions.

PROMPT...Alters the ER_ICE prompt to any alphanumeric character string.

QUIT...Terminates ER-ICE and returns control to DOS.

Understanding ERX Commands

The key to using the ERX debugging commands effectively is understanding their individual characteristics as well as their relationship to each other. Some commands are "active" while other commands are "passive," and some commands are both active and passive depending on which variation of the command you specify. Active commands perform or execute actions which you can see, now! For example, the DISASSEMBLE command translates the memory contents from machine language to assembly language mnemonics and then displays the converted data - all in one step. A passive command, such as the EVENT command, allows you to set conditions that will be utilized by the emulator at a later time (e.g., during program execution). Some passive commands masquerade as active commands by allowing you to observe their status; it's all in how you state the command syntax.

You can use the ERX commands together, in various combinations, to effectively test, debug, modify and integrate both your hardware and software prototype designs. The following example shows how you might combine the ERX commands to test prototype code:

COMMAND	ACTION
LOAD	Downloads a prototype program from the host coding station or prototype hardware to the ERX.
DISASSEMBLE	Disassembles a section of the program residing in the ERX for inspection.
ASSEMBLE	Assembles a software patch into the program.
BREAK & EVENT	Instructs the ERX to stop program execution at a particular point in the program, such as when a memory write operation occurs to a certain address.
GO	Runs the program from a specified address. When the program stops at the breakpoint, shows the status of the registers and where the break occurred.

COMMAND	ACTION
HISTORY	Replays the latest series of machine cycles allowing you to examine address, data and control bus conditions.
STEP	Steps through the program display on a line-for-line basis.
REGISTER	Examines the register contents at a specific location.
COMPARE, EXAMINE, SEARCH, MOVE, GO	Locates or compares different memory locations, makes changes to the program, re-starts the program from the beginning and checks the performance again.

About the Command Language

All ZAX ERX-series emulators execute operations in response to "command statements" made up of the "command name" and "parameters." The command name refers to a character or group of characters that designate the basic emulation operation to be performed (e.g., G for GO, MA for MAP, etc.). Parameters refer to any additional information that complements the command name, such as a specific address or symbol, an address range, a memory type, or a base value. Together, the command name and the parameters can be combined to execute a variety of complex debugging operations.

The control firmware within the ERX requires that the command statements be entered in a concise and logical manner, and that all required elements of the command statement be used. The elements of the command statement are described in the following paragraphs. The elements shown here represent all possible items within a command statement. Of course, not all commands require the presence or absence of each element.

Elements Within A Command Statement

The Prompt Character. The prompt character lets you know that the ERX is ready to accept a command statement. The prompt character is supplied by the ERX - you do not enter it - and it is always displayed on the left side of the console's screen. The prompt may also be changed to any alphanumeric string of characters you desire (see the PROMPT command).

Example of prompt character: ERX180X>

The Emulation-in-Progress (EIP) Symbol. The emulation-in-progress symbol is displayed whenever you are executing code in real time (i.e., after you enter a GO command). The symbol is displayed immediately before the ">" character.

Example of the EIP symbol: ERX180X+>

The Command Name. Commands are represented by the first or first few letters of the command name. The commands are displayed in upper-case typeface but may be entered using any combination of upper- or lower-case letters.

Examples of command names: B (for BREAK), CO (for COMPARE), SA (for SAVE).

Command Qualifiers. The slash key (/) acts to signal a qualifier for the command whenever it appears immediately following the command mnemonic.

Examples of a qualifier: F/W H/M L/I

The Space Character. The space character is an invisible character that not only improves the readability of a sentence, but in the case of the command format, it is recognized as a delimiter for the command name. Spaces must be interpreted from the command format; there is no symbol used to indicate spacing.

Example of space character in use: HE CLO

Keywords are items which you must enter as shown. These items are displayed by upper-case typeface, but any combination of upper-case or lower-case letters may be used to enter them. NOTE: Some terminals must use upper-case letters only. If the ERX responds with an error message, try using upper-case letters.

Examples of keywords: EN LO ON OFF

User-Supplied Items. Lower-case letters in italic typeface show items which you may supply; these are called user-supplied items.

> Examples of user-supplied items include the name of your file (TEST.HEX), a beginning address, ending address and comparison address (100), a symbol name (demo.tst), and data (55).

Address And Data Parameters. The common numerical parameters for the ERX commands are described below:

addr, beg_addr, comp addr, mov_addr, end_addr, search addr, etc. = hexadecimal numbers in 16 bits (0-FFFF). Symbols may also be used.

data, mod_data, and search data = hexadecimal/binary number in 8/16 bits (0-FFFF).

The Equal Sign. The equal sign (=) causes the value or information on its right to assume a relationship with the value on its left.

> Example of the equal sign: MA 0,0FFFF=N0
>
> In this example, the ERX does not display anything in response to this entry, but the value entered on the right (which represents a non-memory area) is now assigned a relationship with the value on the left (an address range from 0 to 0FFFF.)

The Comma Character. The comma character (,) is used to separate parameters when more than one parameter is required to form a command statement. NOTE: A space and a comma may be be used interchangeably.

> Example of the comma character: DI 0,100

Brackets. Items in square brackets are optional. If you choose to include the information, you should not enter the brackets, only the information inside the brackets.

> Examples of brackets: [D=data] [,switch]

The Return Key. The return key is used to terminate statements and execute commands, and it must be entered after every statement. It is assumed that the return key must be pressed after the command statement is entered; there is no symbol used to indicate the return key in the command format.

Sequential Command Execution

Your ERX can execute a single command or a series of commands in a sequential manner. To execute a series of commands, simply enter a semi-colon (;) after each command operation. For example:

```
>F 100,1FF,00;DI 100,1FF;R RESET;G 100,17F
```

This command line tells the ERX to first fill memory, then disassemble it, reset the registers, and finally start program execution at address 100H and stop it at address 17FH - all in one step!

What To Do If You Make An Input Error

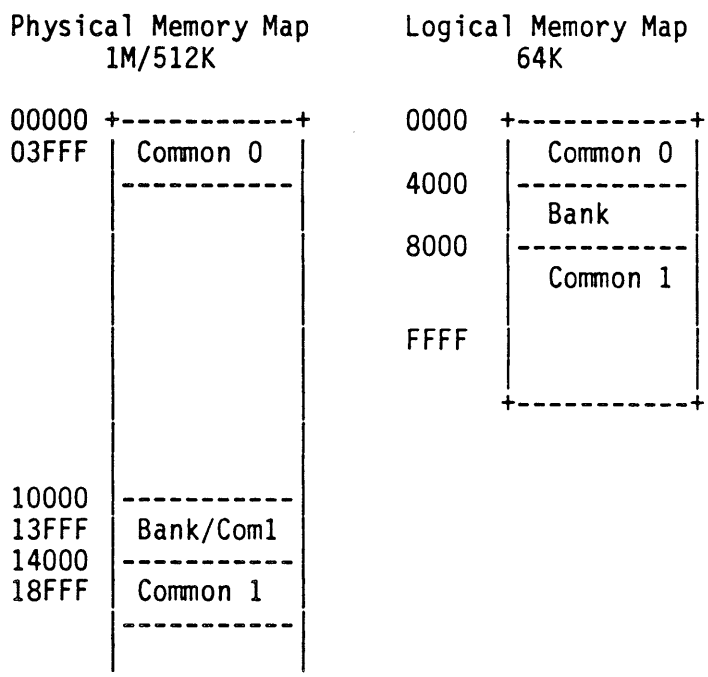
If you make an error while entering a command statement, merely backspace over the error (which cancels the character) and enter the new information. You can also press the Esc key to exit the current command line and bring back up the ERX prompt.

Control Codes and their Functions

<ctrl_t> = abort batch process
<ctrl_e> = recall command function (backward)
<ctrl_x> = recall command function (forward)
<ctrl_a> = re-run last command
<ctrl_y> = same as SHELL command
<ctrl_z> = same as QUIT command
<ctrl_c> = abort ERX
<ctrl_s> = X-OFF
<ctrl_q> = X-ON

Command	ABASE
Operation	Allows you to program a specific configuration of logical and physical memory via the Common/Bank Area Register. It may be necessary to use this command only if Common Area 0, Common Area 1 or Bank Area overlap to a physical address space through HD64180 on-chip MMU programming.
Syntax	>AB [base_1,base_2]
Terms	base_1,base_2 = 0, 1 or B
Syntax Example	>AB 0,B
Remarks	0 refers to Common Area 0; 1 refers to Common Area 1; B refers to Bank Area. AB alone, displays the current status of the ABASE command. The Common/Bank Area Register (CBAR) is used to define the logical memory organization. CBAR specifies boundaries within the HD64180's 64K bytes logical address space for up to three areas: Common Area 0, Bank Area and Common Area 1. The CAR field of CBAR determines the start address of Common Area 1 (Upper Common) and by default, the end address of the Bank Area. The BAR field determines the start address of the Bank Area and by default, the end address of Common Area 0 (Lower Common). The CA and BA fields of CBAR may be freely programmed subject only to the restriction that CA may never be less than BA. Spacing: A space is required after AB. No spaces are permitted where commas act as separators.
Command Example	>AMAP Common 0 : 0000 - 03FFF (0000 - 3FFF) *1 Bank : 1000 - 03FFF (4000 - 7FFF) *2 Common 1 : 1000 - 07FFF (8000 - FFFF) *3 *1 = CBAR(I/O 3AH) may be programmed 84H (start Common 1 8000H, Bank 4000H) *2 = BBR(I/O 39H) may be programmed 0CH (Bank base 0C000H) *3 = CBR(I/O 38H) may be programmed 08H (Common 1 base 08000H).

```
>AB B,1 <--Bank area is first choice for translation
>D 10000+10
            0  1  2  3  4  5  6  7  8  9  A ...
10000 4000 00 00 00 00 00 00 00 00 00 00 00 ...
>AB 1,B <--Common 1 Area is first choice for
translation
>D 10000+10
            0  1  2  3  4  5  6  7  8  9  A ...
10000 8000 00 00 00 00 00 00 00 00 00 00 00 ...
```



Command	ASSEMBLE
Operation	The ASSEMBLE command invokes the in-line assembler to modify a program stored in memory. It allows you to use mnemonics to alter the memory contents instead of machine codes. Applications Note: The In-line assembler in ERX is a powerful software tool that can be used for writing patches into software code that has either been downloaded from a host computer or originated in the target system. This feature also allows you to quickly write your own routines, develop small programs, etc.
Syntax	>A mem_addr <CR> xxxxx xxxx {Assembly code} <CR> xxxxx xxxx <CR> >
Terms	mem_addr = The beginning memory address where assembled code is stored. Also, you may use a previously defined symbol name that has been directed at a starting address. If the symbol name has not been assigned a starting address, "0000" is assumed. xxxxx xxxx = The next storage location. Assembly code = The mnemonic instruction to be assembled and stored. (Operand may include number or symbol if previously defined.) <CR> = Exits the assemble mode.
Syntax Example	>A 100
Remarks	Spacing: A space is required between A and mem_addr. A space is required between opcode and operand of mnemonic instructions (no tab).
Command Example(s)	>A begin_prog <--begins assembling at symbol "begin_prog" >A 1000 <--begins assembling at address 1000H 01000 1000 RST 38H 01001 1001 SUB B 01002 1002 <CR> <--terminates input >

Command	BATCH
Operation	The BATCH command executes batch files. (A batch file contains a series of commands to be executed.)
Syntax	>BA file name[.cmd]
Terms	file_name = The name of the batch file to be executed. cmd = The default extension.
Syntax Example	>BA setup.1
Remarks	Spacing: A space is required between BA and filename.
Command Example	See Syntax Example, above.

Command	BEEP
Operation	The BEEP command sounds the system bell. Applications Note: You can use the BEEP command to indicate the completion of a batch file.
Syntax	>BEEP
Terms	none
Remarks	none
Command Example	>BA TEST >loop: >@counter = @counter + 1 >if @counter < 10 >goto loop >BEEP <-- sounds system bell at completion of batch command

Command BREAK

Introduction

The best way to safely stop a moving car is to use the brakes. In emulation, the best way to stop a program for examination is by using BREAKpoints. The BREAK command specifies a break condition that allows you to stop program execution on a variety of different parameters. When the conditions satisfying the parameters are met, program execution halts and control of the emulator is returned to the user. You can use the BREAK commands to set breakpoints anywhere within a program, and you can specify many different types of breaks to stop program execution.

Breakpoints can be created directly with the BREAK: Specification command, or converted to breakpoints from previously set event points (see the EVENT command). Breakpoints that were created from event points can also be deleted without canceling their designation as event points.

Breakpoints differ from event points in that they actually cause the program to stop execution, whereas event points are used to trigger various functions without necessarily affecting the emulation process.

With the ERX for 64180, there are 64,000 hardware breakpoints x 4 channels available, or 256,000 breakpoints available if only one channel is required. (Channels define operational parameters such as address locations, a memory status or data.) If one channel is used to define a particular operation, such as a memory write, a bank of 64,000 breakpoints will be reserved for that type of operation. A different operation, such as a memory read, opcode fetch or data value, requires the use of another bank of 64,000 breakpoints, and so on. However, by using a single channel (as is the case when only address locations are considered) you have access to 256,000 individual breakpoints.

The display below shows the four-channel restrictiveness as it applies to the ERX emulator:

```
>B 100,M <--breakpoint at addr 100; memory status
>B 200,MR <--breakpoint at addr 200; memory read
>B 300,MW <--breakpoint at addr 300; memory write
>B 400,P <--breakpoint at addr 400; port access
```

```
>B

 No.    Module   Symbol      Address  St  Data Count(EX) EX SEQ  B H P T
&1      $$       __1         0100     M  --,--     1    0 -- ---  * - - -
&2      $$       __2         0200     MR --,--     1    0 -- ---  * - - -
&3      $$       __3         0300     MW --,--     1    0 -- ---  * - - -
&4      $$       __4         0400     P  --,--     1    0 -- ---  * - - -
```

The display above shows that, while only four breakpoints have been set, in fact, all four channels have been exhausted. For each address location, a correspondingly different memory status attribute appears; in all, four different memory types. If you were to attempt to create another breakpoint with yet another memory status, such as a port read, the ERX would respond with the following error message:

```
>B 500,PR
ER-ICE : Reached maximum definitions on this event -
         create another event
```

At this point, you can't create another event, of course, because all four channels have been exhaused; however, you could still create other breakpoints at a differnt addresses but including one of the four memory types as shown below:

```
>B 600,M
>B 700,MR
```

The same restrictions hold true for data values.

If you choose only one memory type or data type, you have access to all four channels of 64,000 breakpoints for a total of 256,000 breakpoints - a generous supply.

Hardware breakpoints recognize machine cycles but do not disturb normal software execution. Hardware breakpoints can cause the ERX hardware to monitor the address and memory status signals for a specified condition. When the conditions are met, a break occurs.

You can also enable and disable multiple breakpoints within a range of the program, and you can break on all breakpoints within a range of the program. Hardware breakpoints can be activated (unmasked), and then temporarily deactivated (masked), without affecting their location addresses within the program or their parameter specifications.

NOTE: In addition to the individual examples for each BREAK command, there is a short demo at the end of the BREAK command section that further explains the theory and practice of setting, altering and using breakpoints.

Command — BREAK: Status

Operation — Displays the current state of the break command.

Applications Note: Use this command to check the condition of the breakpoint settings after they've been set or converted using the BREAK or EVENT commands.

Syntax — >B

Terms — none

Remarks — You can view the breakpoint settings in two ways: either by using the BREAK: Status (B) command, or by using the EVENT: Show (ES) command. The EVENT: Show command displays all the event points and breakpoints; the BREAK: Status command displays only breakpoints. You can distinguish a breakpoint by an asterisk ("*") located under the "B" in the display below (also in the EVENT: Show display).

Command Example — This command example shows what the break status might reveal after several break parameters have been defined.

No.	Module	Symbol	Address	St	Data	Count	(EX)	EX	SEQ	B	H	P	T
&1	$$	__1	0100	MR	--,--	1	0	--	---	*	-	-	-
&2	$$	__2	0200	MW	--,--	1	0	--	---	*	-	-	-
&3	$$	test	0300	--	55,55	1	0	--	---	*	-	-	-
&4	$$	__4	0400	OF	--,--	4	0	--	---	*	-	-	-

NOTE: "No." shows symbol line numbers for each breakpoint; "Module" shows the name of the main module ($$ is the default); "Symbol" shows the symbol name (__n is the default); "Address" shows the location where the breakpoint resides' "St" shows the memory status for the breakpoint; "Data" shows the data value (e.g., 55) or data range (e.g., 24-58) to match for the breakpoint; "Count" shows the passcount specification (the number of occurrences before a break); "(EX)" shows the number of passcounts actually executed; "EX" shows the level or edge-position of an external signal trigger (see EVENT command); "SEQ" shows the sequence of break execution (see EVENT command); "B" shows the presence of a breakpoint; "H," "P" and "T" relate to the HISTORY, PERFORMANCE and TRIGGER commands.

Command	BREAK: Specification
Operation	Sets hardware breakpoints within the user program. Setting a hardware breakpoint configures the ERX to monitor the address and status signals for the specified condition to occur.
Syntax	>B address[,status][,passcount]
Terms	address = The address or symbol name to break on. If symbol name is chosen, it must be predefined by the EVENT command.

status = The type of cycle to break on, including:

M (any memory access)	CM (CPU memory access)
DM (DMA memory access)	MR (memory read)
CMR (CPU memory read)	DMR (DMA read)
MW (memory write)	CMW (CPU memory write)
DMW (DMA memory write)	CPU (CPU access)
P (port access)	CP (CPU port access)
DP (DMA port access)	PR (port read)
CPR (CPU port read)	DPR (DMA port read)
PW (port write)	CPW (CPU port write)
DPW (DMA port write)	DMA (DMA access)
HA (halt acknowledge)	IA (interrupt acknowledge)
OF (opcode fetch)	ANY (any operation)

passcount = The number of occurrences before a break, from 1H to FFFFH (1 to 65535).

Syntax Example	>B 100,MR,3
Remarks	If status is omitted, ANY is assumed.

If passcount is specified, real-time operation is momentarily lost each time the condition occurs. If the passcount specification is omitted, 1 is assumed.

Spacing: A space is required between "B" and "address." No spaces are permitted where commas are used as separators.

Command Example	See "More BREAK Command Examples."

Command	BREAK: Event Conversion
Operation	Converts event points with symbol names to breakpoints. Also masks and unmasks the converted breakpoints (essentially, swapping them back to event points and visa-versa).
Syntax	>B event_symbol[=switch]
Terms	event_symbol = The name of the event to convert to a breakpoint. switch = ON or OFF
Syntax Example	>B fetch
Remarks	Wildcard ("*") characters are allowed in the event_symbol name. To convert the event point to a breakpoint directly, only the event symbol name is required; you do not need to switch on the event symbol name. OFF masks the breakpoint as an event point, and ON unmasks the breakpoint as an event point. Spacing: A space is required between B and event_symbol. No spaces are permitted thereafter.
Command Example	>B evnt1 <--converts event point to breakpoint >B evn*=OFF <--masks all "evn*" breakpoints >B evn*=ON <--unmasks all "evn*" breakpoints

Command	BREAK: Multi-breakpoint Specification
Operation	Enables or disables all pre-set breakpoints within a range according to their symbolic names or symbolic line numbers. With this command, it is possible to enable or disable up to 256,000 breakpoints at a time.
Syntax	>B/M beg sym_name/line,end_sym_name/line[=switch]
Terms	beg_sym name, beg_sym_line = The beginning symbol name or symbol line of the range. end_sym_name, end_sym_line = The ending symbol name or symbol line of the range. switch = ON or OFF
Syntax Example	>B/M rout1,rout2
Remarks	ON (the default) enables the break function from a masked state; OFF disables or masks the function but does not delete it (it may be recalled with ON). Spacing: A space is required between M and beg sym_name/line. Spaces are not permitted where commas are used to separate the parameters.
Command Example	>B/M 100,4FF <--enables a range of breakpoints >B/M tom,harry=OFF <--masks a range of breakpoints

More BREAK Command Examples

The BREAK command, along with its ally the EVENT command, work together as two of the most powerful and useful commands available with your ERX. Understanding their relationship to one another will help you to better understand when and how to use the BREAK command.

The following demonstration begins from an initialized state and then proceeds to show you how to set, alter and confirm breakpoints. If you would like to perform the demo yourself, merely initialize your ERX and then enter the item after "Command."

NOTE: Before beginning the demo, make sure that the ERX memory is mapped as read/write (use the command MA 0,0FFFF=RW).

It is assumed that a carriage return (CR) follows each command statement.

Command	Comment
>B	Normally, this command reveals the status of the breakpoints, however, because none have been set yet, nothing is displayed on the screen. The same holds true for the EVENT command:
>ES	Again, nothing is displayed on your screen - yet. Now, lets set a few event points and then convert them to breakpoints.
>EV	This command allows you to set an event point. When the command queries you for a symbol, enter: EVNT1 When the command queries you for an address, enter: 100/ (The slash exits the command.)
>EV	Let's set another event point. Enter the following symbol: EVNT2 and address: 200/

Now, lets examine the event points.

```
>ES
```

The Event Show command displays the two events that you just set. Next, look over to the far right corner. There you will see the letter "B" with two dashes under it. The "B" informs you if a breakpoint has been set for that symbol or address by displaying an asterisk (*).

Now we'll change the two event points into breakpoints with the following command:

```
>B EV*
```

This command told the ERX to convert all symbol names starting with "EV" to breakpoints. (We could have named them individually, but this is quicker.) You can verify this fact in two ways, either by using the BREAK Status or EVENT Show commands. First, use the EVENT Show command.

```
>ES
```

Look to the far right side of the display and you'll notice the two asterisks (*) under the "B." This indicates to you that both event points presently act as breakpoints. Now examine the BREAK Status command.

```
>B
```

You will see the exact same display. Why? The BREAK Status command now has something to reveal - but both display won't always appear alike.

```
>B EVNT1=OFF
```

We've now disabled or masked breakpoint "EVNT1," however, it still exists. Check both the BREAK Status and EVENT Show commands to see the changes.

```
>B
>ES
```

The first command displays only one breakpoint, "EVENT2." The second command shows that while both event points still exist, "EVNT1" is no longer a breakpoint. It can be converted back to a breakpoint by turning it back On.

```
>B EVNT1=ON
>B
```

See. You can't set a breakpoint directly using a symbol name, but you can define a breakpoint directly by an address or memory status. When you do, the ERX assigns its own symbol name (as in, "__3") and number (as in, "&3") to the value.

```
>B 300,MR
>B
```

If you choose to name a symbol as a breakpoint, you must do so by first creating it as an event, and then converting it to a breakpoint as just shown. This also holds true for breakpoints that contain specific data values. Hardware breakpoints that are defined by address and memory parameters can be created directly, as shown below.

```
>B 500,0F
```

This format creates a hardware breakpoint, directly.

Command	CALCULATE
Operation	Performs addition, subtraction, multiplication and division of hexadecimal, decimal, octal and binary numbers; also performs base conversions. The results are displayed in hexadecimal, decimal, octal and binary notation.
Syntax	>C exp#1 +,-,*,/ exp#n
Terms	exp#1 ... exp#n = The hexadecimal, decimal, octal or binary number.
Syntax Example	C 123+0FFH
Remarks	Addition(+), subtraction(-), multiplication(*) and division(/) operations may be performed on the same line, and more than one type of operation may be performed on the same line. Spacing: A space is required between C and exp#1. Spaces are not restricted thereafter.

Command Example

```
>C 123+65fh  <--adds decimal and hex values
Hex = 6DA
Dec = 1754(1754)
Oct = 3332
Bin = 0000,0000,0000,0000,0000,0110,1101,1010
>C 1  <--converts decimal value
Hex = 1
Dec = 1(1)
Oct = 1
Bin = 0000,0000,0000,0000,0000,0000,0000,0001
>C 1000H*2-4 <--multiple operations
Hex = 1FFC
Dec = 8188(8188)
Oct = 17774
Bin = 0000,0000,0000,0000,0001,1111,1111,1100
>C 2&3 <--bit-wise AND operation
Hex = 2
Dec = 2(2)
Oct = 2
Bin = 0000,0000,0000,0000,0000,0000,0000,0010
```

Command	CLOCK
Operation	Sets the clock configuration for the MPU via the keyboard. Four internal speeds are selectable and two external inputs are available to match the target system's unique clock configuration.
Syntax	>CLO [clock_mode]
Terms	clock mode = 0,1,2,3,4
Syntax Example	CLO 4
Remarks	0 means use an external TTL-level clock; 1 means use an external XTAL level clock; 2 means to use the ERX internal 6MHz clock; 3 means use the ERX internal 3MHz clock (the default); 4 means use the ERX internal 1.5MHz clock. The HD64180 contains a crystal oscillator and system clock generator. A crystal can be directly connected or an external clock input can be provided. In either case, the system clock is equal to one-half the input clock. For example, a crystal or external clock input of 8 MHz corresponds with a system clock rate of 4 MHz. If an external clock input is used instead of a crystal, the waveform should exhibit a 50% (+/-5%) duty cycle. Note that the minimum clock input HIGH voltage level is Vcc-0.6V. The external clock input is connected to the EXTAL pin, while the XTAL pin is left open. (For more information on clock circuitry, see Section 3, "Technical References.") Spacing: A space is required between CLO and clock_mode.
Command Example	>CLO 3 <--sets the clock speed to 1MHz >CLO <--displays and allows alteration of clock setting Clock Mode is 3 0. External (TTL) 1. External (XTAL) 2. Internal (6MHz) 3. Internal (3MHz) 4. Internal (1.5MHz) Select (0-4) ? 4

Command	CLOSE
Operation	The CLOSE command closes a previously opened file and then informs you if the file was closed successfully. (The variable "sts" will contain a non-zero value if the close was unsuccessful.)
Syntax	>CLOSE #file_no
Terms	#file_no = The named number of the file to close.
Syntax Example	>CLOSE #1
Remarks	Spacing: A space is required after CLOSE.
Command Example	>CLOSE #2 <--closes the file associated with file #2 >IF #sts <> 0 <--checks for successful closure >CPUT "File close failure" <--prints error if close failed

Command — COMPARE

Operation — Compares the contents of specified memory blocks within the ERX or target system and then displays the non-matching data along with their locations. The comparison can be made between different memory blocks as mapped to the ERX, or between one block of memory within the ERX and one within the target system.

Syntax — >CO beg_addr,end_addr,cmp_addr[,direction]

Terms — beg_addr = The beginning address for comparison.

end addr = The ending address for comparison. (May also be stated in number of bytes, e.g., beg_addr, +num bytes.)

cmp addr = The beginning memory address to be compared.

direction = UE or EU

Syntax Example — >CO 0,2FF,500,UE

Remarks — If UE is selected, beg addr is user memory and cmp_addr is ERX memory. If EU is selected, beg_addr is ERX memory and cmp_addr is user memory.

If "direction" is omitted, memory locations are selected according to the MAP command.

This command displays non-matching data on a line-for-line basis. To control the scrolling of the display, alternately press the space bar. To exit the display, press the Esc key.

Spacing: A space is required between CO and beg_addr. No spaces are permitted where commas are used to separate the parameters.

Command Example —

```
>CO 0,0FF,1000  <--compares memory to addr 1000H
Address Data  Address Data
 0000    01    1000    00
 0001    FF    1001    AA
 0002    AC    1002    0A
 0003 ...

>CO 0,+100,2000  <--compares memory to addr 2000H
Address Data  Address Data
 0000    30    2000    37
 0001    FE    2001    A1
 0002    25    2002    16
 0003 ...
```

Command	COVERAGE: Specification
Operation	Initializes and defines the range of the Coverage function. With this command you can define the boundaries of the Coverage function to either a select range or the entire memory range (0-1FFFF).
Syntax	>COV/CL [beg_addr[,end_addr]]
Terms	CL = Initializes (clears) the entire memory area or a range of memory (if beg_addr and end_addr are supplied). beg_addr = The beginning address of the range. end_addr = The ending address of the range.
Syntax Example	>COV/CL 0,1FF
Remarks	If beg_addr and end_addr are omitted, the entire memory range will be covered. Spacing: A space is required after CL. No spaces are permitted where commas are used as separators.
Command Example	>COV/CL <--initializes and specifies to cover the entire memory range >COV/CL 1000 <--initializes and specifies to cover from addr 1000 to 1FFFF >COV/CL 2000,2FFF <--initializes and specifies the coverage of a range Also, see "More COVERAGE Command Examples," on the following pages.

Command	COVERAGE: Display
Operation	Displays the results of the COVERAGE command as a percentage of the selected coverage range compared to the total number of of instructions passed during program execution.
Syntax	>COV[/disp] [beg_addr[,end_addr]]
Terms	disp = CA, U or P beg_addr = The beginning address or symbol name to display (default is 0). end_addr = The ending address or symbol name to display (default is 1FFFF).
Syntax Example	COV/U 100,7FF
Remarks	CA displays the percentage of address lines passed; U displays all unpassed address lines; P displays all passed address lines. Spacing: A space is required between disp and beg_address. Spaces cannot be used where commas are used to separate the parameters.
Command Example	>COV/CL 0,3FFF <--initializes area for coverage >COV/S MR <--sets coverage area for memory read >COV 0,1FFF <--display coverage range >COV/U 0,0FFF <--display all unpassed symbols from range Also see, "More COVERAGE Command Examples," on the following pages.

Command	CPUT (Console Put)
Operation	Allows the display of numerical (@), or string (!) values into a numericaly specified "N" variable.
Syntax	>CPUT @num >CPUT !num
Terms	@ = Numerical data entry modifier ! = String data entry modifier num = Variable name
Syntax Example	>CPUT @2 (Numerical display) >345235H >CPUT !4 (String display) >This is just a test... >CPUT !2 (String with "\n") >1 >2 >3 >Testing...
Remarks	The CPUT Command allows the use of the "\n" command ("\n" is the C Language command for carriage return, linefeed). Spacing: A space is required after CPUT. No spaces are permitted thereafter.
Command Example	>CPUT !2 :for !2 = 1...\n2...\n3...\nTesting...\n >1... >2... >3... >Testing...

Command	DEFM (Default Module)
Operation	Allows you to change the default module. After the change, all memory disassembly, inspection, etc. commences from the new module.
Syntax	>DEFM [def_mod_name]
Terms	def_mod_name = The name of the new default module.
Syntax Example	>DEFM MAIN
Remarks	DEFM alone, displays the current default module name ($$ is the default). Spacing: A space is required after DEFM.
Command Example	>DEFM <--displays current module name DEFAULT MODULE_NAME: $$ >DEFM MAIN <--changes module name to MAIN >DEFM DEFAULT MODULE_NAME: MAIN <--shows new name >

Command	DISASSEMBLE
Operation	Translates the memory contents from machine codes to assembly language mnemonics, and then displays the converted contents.
Syntax	>DI [beg_addr][,end_addr]
Terms	beg_addr = The beginning memory address in the program. end_addr = The ending memory address in the program, or the number of bytes from the beginning address.
Syntax Example	>DI 50,8F >DI 300 (need displays) >DI 100,+30 >DI START,START+20H
Remarks	If beg_addr is omitted, disassembly begins at the current program counter. If end addr is omitted, 10 lines of instructions are automatically displayed. This command displays items on a line-for-line basis. To control the scrolling of the display, alternately press the space bar. To exit the display, press the Esc key. Spacing: A space is required between DI and beg_addr (if beg_addr is selected). Space are not permitted where commas are used to separate the parameters.
Command Example	See Syntax Example. The first example shows that the memory contents in the ERX are disassembled beginning from address 50 to address 8F. In the second example, the ending address is omitted, which causes 10 lines of the memory contents to be disassembled starting from address 300. The third example illustrates how 30 bytes are disassembled from a starting address (100). The fourth example disassembles from the symbol "START" to "START" plus 20 bytes.

Command	DUMP
Operation	Displays the memory contents in both hexadecimal and ASCII code.
Syntax	>D[/length] beg_addr[,end_addr]
Terms	length = B, W or S beg_addr = Beginning address of display. end_addr = Ending address of display. (May also be stated in byte format, e.g., beg_addr+,num_bytes.)
Syntax Example	D/W 100,1FF D 2000 D 1000,+30 D/S START
Remarks	B means byte length display; W means word length display; S means symbol display. The end_add is an optional parameter. If it is omitted, 16 bytes are displayed starting with beg_addr. This command displays items on a line-for-line basis. To control the scrolling of the display, alternately press the space bar. To exit the display, press the Esc key. Spacing: A space is required between D and beg addr or length and beg_addr. Spaces are not permitted where commas are used to separate the parameters.
Command Example	See Syntax Example. The first example shows that the memory contents are displayed in word units, beginning with address 100 and ending with address 1FF. The second example shows that the last 16 bytes are displayed beginning at address 2000. The third example shows how 30 bytes of memory are displayed from address 1000. The fourth example displays 16 bytes of memory from the symbol "START."

Command	ECHO
Operation	Enables or disables the console display of the command lines within a batch or macro file.
Syntax	>EC switch
Terms	switch = ON or OFF
Syntax Example	>EC OFF
Remarks	ON (the default) enables the display of command lines with a batch or macro file, and OFF disables the display. Spacing: A space is required between EC and switch.
Command Example	>ECHO ON >TEST CPUT "TEST COMPLETED!!\n" TEST COMPLETED !! Macro TEST completed. > >ECHO OFF >TEST TEST COMPLETED !! >

Command	EMSELECT (Emulation Method Select)
Operation	Allows you to control signal I/O between the ERX and the target system during emulation.
Syntax	>EMS [select][=switch]
Terms	select = A, B, C, D or E switch = EN, DI, 1, 2 or 3
Syntax Example	EMS C=EN EMS E=3
Remarks	A: Allows you to control the generation of a user wait state during an emulation; B: Allows you to control the generation of a timeout break for all memory access conditions; C: Allows you control the generation of an automatic wait state for all memory access conditions; D: Allows you to control the generation of an automatic wait state for all I/O port access conditions; E: Allows you to control the insertion of up to three wait states into each machine cycle. EN unmasks and allows the implementation of the associated function; DI masks or suppresses the implementation of the associated function. 1, 2 and 3 refer to the number of wait states associated with the E function. EMS alone, displays the status of the EMSELECT command. Spacing: A space is required after EMS. No spaces are permitted thereafter.
Command Example	See Syntax Example, above. The first example shows how to generate an auto wait state for all memory access conditions, and the second example shows how to insert three wait states into each machine cycle.

Command	EOF (End Of File)
Operation	The EOF (End Of File) command informs you if the end-of-file has or has not been reached. (The user-defined variable will contain zero if end-of-file is reached.)
Syntax	>EOF #file_no,@ret_value
Terms	file no = The number of the file. ret_value = 0 if end of file, non-zero if not end of file.
Syntax Example	>EOF #1,@1
Remarks	Spacing: A space is required between EOF and #. Spaces are not permitted where commas are used to separate the parameters.
Command Example	>EOF #4,@1 >IF @1==0 >GOTO GET_END

Command	ERX
Operation	The ERX command is the executable program for the ERX emulator. With it, you can invoke the communications program or invoke the setup program for the ERX emulator.
Syntax	DOS>ERX [batch_file[.cmd]]
Terms	batch_file.cmd = The name of the batch file to be executed.
Syntax Example	DOS>ERX SETUP
Remarks	Spacing: A space is required after ERX.

Sequential = The order of break implementation. Two different sets of four priorities can be named, including:

#1,#2,#3,#4
$1,$2,$3,$4

The Sequential function is set by the EVENT command and used by the BREAK command to prioritize breaking.

Syntax Example

```
>EV
Symbol     :             = Check1
Address    : 0000,FFFF   = 2000
Status     : --          = MR
Data       : --,--       = 34
Passcount  : 00001       = 5
External   : --          = HI
Sequential : --          = #2
>
```

Remarks

Setting an event command requires that you supply parameters in response to the ERX. These parameters allow you to create and identify an event by its 1) symbol name; 2) address; 3) memory type; 4) data type; 5) passcount number; 6) external trigger specification; or 7) sequential designator. You can supply all seven parameters or any combination thereof, including only a single parameter.

A slash terminates the entry and exits the input mode. Example:

```
>EV
Symbol     :             = fred
Address    : 0000,FFFF   = 234/ <--slash to terminate
>
```

Command Example

```
#1
>EV
Symbol     :             =
Address    : 0000,FFFF   = 1234
Status     : --          =
Data       : --,--       =
Passcount  : 00001       =
External   : --          = HI/
>

#2
>EV
Symbol     :             = bob
Address    : 0000,FFFF   = 4FF
Status     : --          = MR/
>

#3
>EV
Symbol     :             = tom
Address    : 0000,FFFF   = 0FEE,3FF0
Status     : --          =
Data       : --,--       = 12/
>
```

```
#4
>EV
Symbol      :             = allrange
Address     : 0000,FFFF   =
Status      : --          = MW/
>
```

In example #1, an address and external trigger specification is defined. The ERX then assigns its own symbol name to the event point. In example #2, a symbol name, address and memory type is defined. In example #3, a symbol name and address range is defined for a data value. In example #4, a symbol is defined for a memory type.

Command	ESAVE (Event Save)
Operation	Stores in host memory the parameters of an event point or a series of event points that were previously defined with the EVENT command.
Syntax	>ESA file_name[.evt]
Terms	file_name = The name assigned to the event point map. (If the extension is omitted, EVT is automatically supplied as the extension.)
Syntax Example	>ESA EVENT1
Remarks	The ESAVE command stores all parameters for all previously defined event points, including the number, module designation, symbol name, address, etc. The LOAD command downloads the event file(s) back to the ERX (see the LOAD command). Spacing: A space is required between ESA and file_name.
Command Example	See Syntax Example, above. This example saves an event file named "EVENT1."

Command	EDELETE (Event Delete)
Operation	Deletes a pre-set event point or series of event points either by its/their symbol name(s) or number(s).
Syntax	>ED beg_addr,end_addr >ED sym_name >ED sym_line >ED beg_sym_name,end_sym_name >ED beg_sym_line,end_sym_line
Terms	beg_addr, beg sym_name, beg_sym_line = The beginning address, symbol name or symbol line that marks the range of the pre-set event points. end_addr, end_sym_name, end_sym_line = The ending address, symbol name or symbol line that marks the range of the pre-set event points. sym_name, sym_line = The symbol name or symbol line that identifies a particular event point.
Syntax Example	>ED start
Remarks	Wildcard characters (*) may be used in place of specific names or addresses (e.g., ED module*) Because event points have a higher priority than breakpoints, deleting an event point also causes the deletion of a breakpoint if they occupy the same address, symbol line or symbol name. The same holds true for a history trigger, performance trigger, or external trigger. Spacing: A space is required between ED and the next parameter. Spaces are not permitted where commas are used to separate the parameters.
Command Example	>ED 1000,3000 <--deletes by address range >ED start <--deletes by symbol name >ED module_* <--deletes by range of symbol names >ED &20 <--deletes by symbol line >ED start,end <--deletes by range of symbol names >ED &34,&51 <--deletes by range of symbol lines

Command	ESHOW (Event Show)
Operation	Displays the event point(s) and the parameters of the event point(s).
Syntax	>ES >ES beg_addr,end_addr >ES sym_name >ES sym_line >ES beg_sym_name,end_sym_name >ES beg_sym_line,end_sym_line
Terms	beg_addr, beg sym_name, beg_sym_line = The beginning address, symbol name or symbol line that marks the range of the event points to be displayed. end_addr, end_sym name, end sym_line = The ending address, symbol name or symbol line that marks the range of the event points to be displayed. sym_name, sym_line = The symbol name or symbol line that identifies a particular event point.
Syntax Example	ES 100,300
Remarks	Wildcard characters (*) may be used in place of specific names or addresses (e.g., ED module*) Spacing: A space is required between ED and the next parameter. Spaces are not permitted where commas are used to separate the parameters.
Command Example	>ES <--displays event status >ES 1000,3000 <--displays event status by range >ES start <--displays event named "start" >ES module_* <--displays range of event names >ES &20 <--displays event by symbol line >ES start,end <--displays event symbol names >ES &34,&51 <--displays event by symbol line range

Now examine the sample status display below:

No.	Module	Symbol	Address	St	Data	Count	(EX)	EX	SEQ	B	H	P	T
&1	$$	event1	0100	--	--,--	1	0	--	---	-	S	-	-
&2	$$	event2	02FF	--	--,--	1	0	--	---	-	E	-	-
&3	$$	test	0300	MR	--,--	1	0	--	---	*	-	-	-
&4	$$	__4	0400	OF	55,--	1	0	--	---	-	-	-	*
&5	$$	newrt1	0600	OF	--,--	1	0	--	---	-	-	S	-
&6	$$	newrt2	07FF	OF	--,--	1	0	--	---	-	-	E	-

The first (&1) and second (&2) lines show the parameters for a beginning and ending trigger for the real-time trace feature; the third (&3) line shows the parameters that define a typical breakpoint; the fourth (&4) line shows the parameters that define a typical external trigger; and the fifth (&5) and sixth (&6) lines show the parameters that define a typical performance function.

Command	EXAMINE
Operation	Inspects one or more memory locations and optionally modifies them. The locations can be displayed and changed with either ASCII or hexadecimal values.
Syntax	>E[/length][/N] beg_addr[=mod_data]
Terms	length = B or W N = No-verify (the default is to read-verify after writing). beg_addr = Beginning address of display. mod_data = Modified (new) data for this location.
Syntax Example	E/W 100=5555 E OFFE
Remarks	B means byte length display, and W means word length display. If mod_data is omitted, the command enters a repeat mode, which allows several locations to be changed. The repeat mode includes: return (cr) to display the next byte (word) of data comma (,) to display the same byte (word) of data caret (^) to display previous byte (word) of data slash (/) to exit the EXAMINE command. Spacing: A space is required before beg_addr. No spaces are permitted between beg_addr and mod data; the equal sign acts as the separator.
Command Example	See Syntax Example, above. This example shows that the memory address to convert is 100, and that the data to convert to is 5555. Now examine the following examples: >E 0 0020 54=74, <--change value to 74H/re-examine 0020 74= <--leave value unchanged/go to next address 0021 68= <--leave value unchanged/go to next address 0022 69='a' <--change value/go to next address 0023 73=74^ <--change value/go to previous address 0022 61=^ <--leave value unchanged/go to previous address 0021 68, <--leave value unchanged/re-examine address 0021 68=^ <--leave value unchanged/go to previous address 0020 74=/ <--leave value unchanged/exit command >

```
>E/W 30
0030 A9BF=4455, <--change word value/re-examine
0030 4455=  <--leave value unchanged/ go to next
              address
0032 FDB2='HI', <--change value (ASCII)/re-examine
0032 494B= <--leave value unchanged/go to next
             address
0034 CFED=0/ <--change value/exit command
>

>E 40
0040 06= <--examine only
0041 A0=
0042 00=
0043 64=
0044 0C=
0045 0E=/ <--exits command
```

Command	FILL
Operation	Fills a block of memory with either hexadecimal or ASCII codes.
Syntax	>F[/length][/N] beg_addr,[end_addr],data
Terms	length = B or W N = No-verify (the default is to read-verify after writing). beg_addr = The block beginning address to be filled (from 0 to 1FFFEH). end_addr = The block ending address to be filled (from 0 to 1FFFFH). data = Data that fills the block.
Syntax Example	F 100,0FEF,55
Remarks	B means byte length display, and W means word length display. When W is selected, the word will be displayed or entered in LSB/MSB (Least Significant Bit/Most Significant Bit) order. Spacing: A space is required before beg_addr. No spaces are permitted where commas act as separators.
Command Example	>F/N 4000,4FFF,0 <--fills memory without verifying >F/W 20,0FF,3412 <--fills on word basis >F 4000,+500,'ABCDEFGH' <--fills with ASCII codes

Command	GO
Operation	Executes the user program.
Syntax	>G [beg_addr][,end addr]
Terms	beg_addr = The address to begin execution. end_addr = the last address to execute.
Syntax Example	>G >G 100 >G 200,2FFF
Remarks	All parameters for this command are optional. If beg_addr is omitted, the program continues from the current program counter. If end_addr is omitted, the program continues until a breakpoint or until a STOP command is issued. Spacing: A space is required between G and any additional parameters. Spaces are not permitted where commas are used to separate the parameters.
Command Example	See Syntax Example, above. The first example starts the program from the current program counter. The second example starts the program from address 100H. The third example starts the program from 200H and stops it at address 2FFFH.

Command	GOTO
Operation	Allows you to branch to a label located within a batch file or macro.
Syntax	>GOTO label
Terms	label = The name of the label located within a batch file or macro.
Syntax Example	>GOTO LABEL_1
Remarks	Spacing: A space is required between GOTO and label.
Command Example	See Syntax Example, above.

Command	HELP
Operation	Summons the on-line help menu for the ERX and provides on-line syntax and examples for ERX commands.
Syntax	>HE [command]
Terms	command = The identifier for the particular command (B for BREAK, EX for EXAMINE, etc.).
Syntax Example	>HE >HE FILL
Remarks	Entering HE only, will display the command names. Explanations and syntax help is available for each command but each must be entered individually. Spacing: A space is required between HE and command.
Command Example	See Syntax Example, above.

Command

HISTORY

Introduction

The real-time trace is one of the most powerful and useful features of your ERX. It allows you to record (hence the name "History" command) and then analyze a specific section of program execution rather than sifting through the entire program looking for an isolated problem.

As with the BREAK command, you must first set event points and then specify them to act as triggers to start and stop the data storage process to the real-time trace buffer. By specifying different storage modes, you can control how the real-time trace captures data. After program execution stops, you can view the address, data, and control bus of the latest series of machine cycles (in either machine cycle or disassembled format) on the console screen, or dump the information to a printer. Thus, if a problem develops during program execution, the real-time trace provides a record that can be reviewed to determine what the problem is and where it resides.

Trace Width and Depth

An emulator's trace memory needs to be wide enough to accommodate the processor's address and data lines. With the ERX for 64180, the trace memory is 32 bits wide (8 bits data/16 bits address/8 bits status).

When it comes to the trace memory's depth, more is not always better. If too much depth is specified, it may be difficult to sift through all the data; if the trace memory depth is insufficient, the chances of recording the trace section where the problem exists are diminished. Your ERX has a maximum trace memory depth of 4K (4095) machine cycles, but this may be reduced by specifying the "length" in the HISTORY command. The ability to alter the size of the trace storage size permits very specific tracing.

Real-time Trace Buffer

The data that is recorded during program execution is stored in the real-time trace buffer. The real-time trace buffer can be thought of as a data storage facility that moves along parallel to the user program, storing the same data that is being executed by the user program.

The maximum storage capacity of the real-time trace buffer is 4K machine cycles, but by using a "First-In/First-Out" (FIFO) recording technique, the buffer captures the latest program execution by discarding old data and replacing it with new data. By using this technique, the display always reveals the latest data that the buffer has stored.

Using The Real-time Trace

The ERX's real-time trace is always active, that is, it records the program execution even if the HISTORY command parameters are omitted. There are, however, eight different storage modes to choose from. The storage modes determine where and when the real-time trace begins and ends, and how much information it stores. After the information has been stored in the real-time trace buffer, it can be displayed in either machine cycle or disassembled format.

The options, then, for the HISTORY command involve first creating event points, then converting them to triggers, and finally, selecting the proper storage mode to trigger or halt the real-time trace. A discussion of each storage mode follows.

Simplest Case: Begin Monitor Mode

An easy way to understand how the real-time trace works is to examine the Begin Monitor mode. In this mode, the GO command (which begins emulation) also triggers the start of real-time tracing so that all activity executed from the program memory area is simultaneously transferred to the real-time trace buffer.

After the user program executes (and the buffer stores) the activity equivalent of the length, the trace buffer fills to that point and then stops. The activity that is now stored in the buffer is the "captured" trace section (the section that the ERX displays). The real-time trace then enters a non-trace mode and stops when a monitor break (accomplished by entering the STOP command) or breakpoint is encountered.

Begin Event Mode

The Begin Event mode works in the same way as the Begin Monitor mode except that an event point triggers the real-time trace instead of the GO command. The buffer stores the amount specified by the length (up to 4K) and then stops.

NOTE: The event itself is not stored in the buffer, but triggers the buffer to begin storing.

End Monitor Mode

The End Monitor mode begins storing all activity upon program execution and then terminates the tracing process when a breakpoint is encountered or when the STOP command is issued. The captured trace section is the last 4K before the breakpoint or STOP command.

The ERX accomplishes this type of tracing by storing data on a First-In/First-Out (FIFO) basis after the buffer is filled. By using this storage technique, the ERX displays the latest data in the trace buffer.

The End and Center Event modes use this same FIFO recording technique in their operation.

End Event Mode

The End Event mode works in the same way as the End Monitor mode except that an event point (instead of a breakpoint) triggers the buffer to suspend tracing. The captured trace section is the last 4K before and including the event point.

Center Event Mode

The Center Event mode is used when you desire the trace to surround a single event point in the program. It performs this task by reading the length specification and recording that number of cycles after the event point occurs. The remainder of the 4K buffer then contains cycles just prior to and including the event point. For example, if 1K is specified as the range, 1K cycles would be captured after the event point, and the remaining 3K cycles would be captured before the event point. If the specified length is 4K, 4K cycles would be captured after the event, and the remaining 95 cycles would be captured before the event point. (4K = 4095 cycles.)

Just like the End Monitor and End Event modes, the Center Event mode causes the real-time trace to start recording activity immediately after the GO command.

Command — HISTORY: Real-time Trace Status

Operation — Displays the current status of the real-time trace.

Applications Note: Use the real-time trace status to analyze the condition of the real-time trace buffer, i.e., the address or symbol to trigger on, the parameters defining the trigger, the length of the trace range, etc.

Syntax — >H/S

Terms — none

Remarks — The status of the HISTORY command can only be examined after the trigger parameters have been specified. See the HISTORY: Specification command for details on how to set the trigger parameters.

Command Example — >H/S

```
 No.  Module  Symbol       Address    St  Data Count(EX) EX SEQ  B H P T
&1    $$      trig1         0100      -- --,--    1    0 --  --  - S - -
&2    $$      trig2         0200      -- --,--    1    0 --  --  - E - -
Auto Start : ON
Length     : $
Multi      : ON
Freeze     : OFF
>
```

This command shows what the HISTORY: Status command might display after two trigger points have been set. Note the "B H P T" display on the far right. The "H" character refers to the trigger status of the HISTORY command. The "S" indicates the starting point to trigger the real-time trace, and the "E" indicates the ending point to trigger the real-time trace. Now look to the far left of the display. "Auto Start : ON" tells you that the storage process will commence when emulation begins; "Length : $" tells you that the storage range is set to maximum (4K); "Multi : ON" tells you that the ERX will recognize several trigger points; and "Freeze : OFF" tells you that the ERX will trace without interruption.

See the other HISTORY commands for more information and then refer to "More HISTORY Command Examples."

Command	HISTORY: Real-time Trace Storage Display
Operation	Displays the number of cycles passed since the program was started, from 1 to 4096.
Syntax	>H/SI
Terms	none
Syntax Example	see above
Remarks	If the Storage Size displays "Full," it indicates a full buffer or 4095 cycles.
Command Example	>H/SI Storage Size = 0256 This example shows that 256 machine cycles have been passed since the program was started.

Command — HISTORY: Real-time Trace Format Display

Operation — Displays the contents of the real-time trace buffer in either machine cycle format or disassembled format. This command may be used after or during program execution.

Syntax — H/mode[,int_point][,term_point][,A=address][,ST=status][,D=data]

Terms — mode = M or D

int_point = Initial point of display, from 1 to 4095.

term_point = Point at which display terminates, from 1 to 4095.

address = The address to begin display.

status = The type of memory cycle to display, including:

M (any memory access)	CM (CPU memory access)
DM (DMA memory access)	MR (memory read)
CMR (CPU memory read)	DMR (DMA read)
MW (memory write)	CMW (CPU memory write)
DMW (DMA memory write)	CPU (CPU access)
P (port access)	CP (CPU port access)
DP (DMA port access)	PR (port read)
CPR (CPU port read)	DPR (DMA port read)
PW (port write)	CPW (CPU port write)
DPW (DMA port write)	DMA (DMA access)
HA (halt acknowledge)	IA (interrupt acknowledge)
OF (opcode fetch)	ANY (any operation)

data = The type of data to display.

Syntax Example —
```
H/M,200,100,A=100,D=00
H/D
```

Remarks — M specifies to display the program execution in machine cycle format; D displays the program execution in disassembled format.

With this command, int_point must be greater than or equal to term_point. The storage pointer is numbered by bus cycles - displayed from high to low - where "1" is the most recent bus cycle.

This command displays items on a line-for-line basis. To control the scrolling of the display, alternately press the space bar. To exit the display, press the Escape (Esc) key.

Spacing: No spaces are permitted where commas are used as separators.

Command Example

```
>H/D
Point E  Phy    Log   Dt St          Opcode Operand
 4096 * 082AC  82AC         trig1   NOP
 4095   082AD  82AD                 INC    L
 4094   082AE  82AE                 NOP
 4093   082AF  82AF                 DEC    C
 4092 ...
```

This example shows what the real-time trace format display might reveal after program execution has been terminated. This command and the resulting display could also have been entered during program execution.

Note the event point counter (under "Point"), the event point trigger indicator (the asterisk under "E"), the address location (expressed both physically and logically), the type of data to display (under "Dt"), the type of memory cycle displayed (under "St"), the Opcode name and the Operand. The "trig1" indicates the name of the symbol assigned to the event point trigger at that particular address.

Command

HISTORY: Real-time Trace Storage Mode

Operation

Specifies the parameters that define the eight different trace modes. These modes include Begin Monitor, End Monitor, Begin Event, Center Event, End Event, Multiple Event, Inner Event and Outer Event. Upon invocation, this command will query you to supply information such as the location to begin tracing, the location to end tracing, the length of the trace section, etc.

Syntax

```
>H
Start Event          = event_symbol[,switch]
End   Event          = event_symbol[,switch]
Auto Start  :  ON    = switch
Length      :  $     = length
Multi       :  ON    = switch
Freeze      :  OFF   = switch
>
```

Terms

event_symbol = The symbol name that identifies the previously defined event point. If a symbol name is omitted, the ERX supplies its own symbol name (as in "__1,"" __3," etc.)

switch = ON or OFF

length = 1 to 4096 or $

Syntax Example

see above

Remarks

ON enables the particular function, and OFF disables the particular function.

The length defines the size of the trace buffer, from 1 to 4096 machine cycles. The "$" instructs the ERX to use the maximum allowable trace buffer size for the particular condition.

Creating trigger points for real-time tracing begins by setting event points and then converting them to triggers. (See HISTORY Command Examples, a few pages from here.)

The eight different trace modes were defined at the beginning of the HISTORY command. Refer to this information to determine which mode is correct for your application and then supply the proper information as shown on the next few pages.

NOTE: You can also invoke the eight different trace modes through a macro, just by entering the mode mnemonic (e.g., EM for End Monitor, BE for Begin Event, and so on).

End Monitor Mode

To select the End Monitor Mode, enter the following information:

```
>H
Start Event          = *,OFF
End   Event          = *,OFF
Auto Start  :  ON    = ON
Length      :  $     = $ (4095)
Multi       :  ON    = OFF
Freeze      :  OFF   = OFF
```

Or, to invoke the macro for this mode, simply enter EM <CR>. This function duplicates the attributes listed in the display above.

Begin Event Mode

To select the Begin Event Mode, enter the following information:

```
>H
Start Event          = MAIN.MAIN <-example only
End   Event          = *,OFF
Auto Start  :  ON    = OFF
Length      :  $     = $
Multi       :  ON    = OFF
Freeze      :  OFF   = ON
```

Or, to invoke the macro for this mode, simply enter BE <CR>. This function duplicates the attributes listed in the display above, however, you must supply the starting event point after BE (e.g., BE evnt1), and the event point must have been previously defined with the EVENT command.

Multiple Event Mode

To select the Multiple Event Mode, enter the following information:

```
>H
Start Event          = MAIN.MAIN,ON <-example only
End   Event          = *,OFF
Auto Start  :  ON    = OFF
Length      :  $     = 256 <--example only
Multi       :  ON    = ON
Freeze      :  OFF   = OFF
```

Or, to invoke the macro for this mode, simply enter ME <CR>. This function duplicates the attributes listed in the display above, however, you must supply the starting event point after ME (e.g., ME evnt1), and the event point must have been previously defined with the EVENT command.

Inner Event Mode

To select the Inner Event Mode, enter the following information:

```
>H
Start Event          = MAIN.MEM_LOP,ON <-example only
End   Event          = MAIN.MEM_END,ON <-example only
Auto Start  :  ON    = OFF
Length      :  $     = 256 <--example only
Multi       :  ON    = OFF
Freeze      :  OFF   = OFF
```

Or, to invoke the macro for this mode, simply enter IE <CR>. This function duplicates the attributes listed in the display above, however, you must supply the event points after IE (e.g., IE evnt1,evnt2), and the event points must have been previously defined with the EVENT command.

Outer Event Mode

To select the Outer Event Mode, enter the following information:

```
>H
Start Event          = MAIN.MEM_END,ON <-example only
End   Event          = MAIN.MEM_LOP,ON <-example only
Auto Start  :  ON    = ON
Length      :  $     = $
Multi       :  ON    = OFF
Freeze      :  OFF   = OFF
```

Or, to invoke the macro for this mode, simply enter OE <CR>. This function duplicates the attributes listed in the display above, however, you must supply the event points after OE (e.g., OE evnt1,evnt2), and the event points must have been previously defined with the EVENT command.

Command Example

See "More HISTORY Command Example."

```
>EV
Symbol = T2
Address = 1FF/
```

Here, you are creating a second event point named "T2" at address 1FFH.

```
>ES
```

Take a look at the two event points you just created. They'll now be used to mark the trigger points for the real-time trace.

```
>H
Start Event = T1
End Event = T2
Auto Start = OFF
Length = 256
Multi = OFF
Freeze = OFF
```

Enter this data in response to the ERX. Here, you are establishing event points as trigger points, and defining how much activity is stored in the trace buffer. Once these parameters have been set, you can examine the status of the HISTORY command.

```
>H/S
```

Look to the far right under the "H" and you'll see that the ERX has now assigned a starting ("S") and ending ("E") trigger to the event points.

Now, we need some code to execute.

```
>F 100,1FF,00
```

First, fill a memory range with NOPs.

```
>A 200
200 JP 100
203 <cr>
```

Now, assemble a Jump to 100 instruction at the end of the NOP range.

```
>R RESET
>R PC=100
>G
```

Reset the registers, initialize the program counter to 100, and Go!

```
>H/D
```

You can now examine the program even though it's running. To control the scrolling, press the space bar. To resume emulation, press the Esc key.

```
>STOP
>H/D
```

You can also stop program execution and examine the real-time trace buffer, directly. Press the space bar to scroll through the display; press the Esc key to exit.

Command	ICERESET
Operation	Halts emulation and resets the I/O of the ERX.
Syntax	>ICERES
Terms	none
Syntax Example	see above
Remarks	This command is different from the STOP command in that it only halts emulation but does not display the current status of the registers at the break, the address of the break, or the instruction just executed.
Command Example	>G 100 <--starts program execution +> <--emulation-in-progress indicator +>ICERES <--halts emulation >

Command	IDENTIFICATION
Operation	Displays the ERX model and software version.
Syntax	>ID
Terms	none
Syntax Example	see above
Remarks	none
Command Example	>ID THE BOX-ER.ICE HD64180 Software Version 1.00D This example shows the ERX emulates the HD64180 processor and that the software version is 1.00D. (Your software version may be different, depending on your purchase date.)

Command	IF
Operation	Allows conditional execution of commands dependent upon specific register, memory or port contents. You may also specify arithmetic and bit-wise operators.
Syntax	>IF condition
Terms	condition = The condition to execute the command, including: symbol name MB (memory address; byte value) MW (memory address; word value) PB (port address; byte value) LOG (logical address) PHY (physical address) register name @n (variable) !n (character variable) #STS (status variable) +,-,*,/,&,|,^,~,==,>,>=,<,<=,<>
Syntax Example	>IF @10 == 0
Remarks	Spacing: A space is required between IF and condition.
Command Example	MACRO>IF #STS==0 MACRO> GET #1,!1 MACRO>ELSE MACRO> LET @1=0 MACRO>END1

Command	JOURNAL
Operation	Opens a file for storing all subsequent commands until a NOJOURNAL command is issued. The command sequences can be recorded to a disk file for later re-execution as a BATCH file.
Syntax	>J file_name[.cmd]
Terms	file_name = The name of the file to store all subsequent commands.
Syntax Example	>J TEST
Remarks	Spacing: A space is required after J.
Command Example	See Syntax Example, above.

Command	KEY
Operation	Programs the function keys to carry out a command or series of commands. Applications Note: The KEY command can be used to conveniently execute a single command or string of commands by simply pressing a single function key (F1, F2, etc.) If your function keys have be previously defined for other operations, you can still use this command to program functions. (See the FNKEY command for more information.)
Syntax	>K [number="command_string"]
Terms	number = 1,2,3,4,5,6,7,8,9 or 10 command_string = The single command or command string to program for execution.
Syntax Example	K 3=DI 100
Remarks	A single command may be programmed to a particular function key, or a string of commands. A string of commands may be programmed to a function key by entering a semi-colon (;) after each command (e.g., R RESET;R;G 100). A "K" followed by a return, displays the command(s) associated with the function key(s). Spacing: A space is required between K and "number" (if a number is used). No spaces are permitted thereafter.
Command Example	>K <--displays all previous defined keys >K 3="H/S" <--programs function key "3" to display History status >K 7="R;DI 0 10;G MAIN.LOOP" <--programs function key "7" to execute series of commands

Command	LET
Operation	The LET command assigns values to variables.
Syntax	>LET var=val
Terms	var/val = symbol name = MB (memory address; byte value) = MW (memory address; word value) = PB (port address; byte value) = LOG (logical address) = PHY (physical address) = register name = @n (variable) = !n (character variable) = #STS (status variable) +,-,*,/,&,|,^,~,==,>,>=,<,<=,<>
Syntax Example	>LET @2=1000H-@2
Remarks	Spacing: A space is required between LET and "var." No spaces are permitted thereafter.
Command Example	MACRO>LET @1=1000 MACRO>LET @2=MB(1000) MACRO>LET !1="This is a message.\n"

Command	LOAD
Operation	The LOAD command downloads object files from the host computer in either Intel, Motorola or dump format. The LOAD command also loads an event file.
Syntax	>L[/format] [object_file_name[.abs]] [,event_file_name[.evt]]
Terms	format = The format of the file, including: I (Intel-hex) M (Motorola) D (dump format) object_file_name = The name of the file to download to the ERX. event_file_name = The name of the event file to download to the ERX. (See the EVENT command for more information on setting and saving events.) Note: An event file may be loaded without specifying the object filename. In this case, a comma must be substituted for the object filename (e.g., "L ,EVENT.EVT").
Syntax Example	>L/M TEST >L TEST1,EVENT1.EVT
Remarks	If the format specification is omitted, command defaults to dump format. The "format" is not needed if the command is used to load an event file. (See the EVENT: Save command for more information.) Spacing: A space is required after L/format or L and the next parameter if "format" is omitted.
Command Example	See Syntax Example, above. The first example shows how to load a Motorola-format file named "TEST." The second example shows how to load an event file.

Command	LOG
Operation	Opens and records your emulation session to a file.
Syntax	>LOG filename[.log]
Terms	filename = The name of the file to store your emulation session.
Syntax Example	>LOG DEBUG1
Remarks	Spacing: A space is required between LOG and filename.
Command Example	See Syntax Example, above.

Command	LOOPOUT
Operation	Breaks out of a loop condition from within a macro or batch file.
Syntax	>LOOPOUT
Terms	none
Syntax Example	see above
Remarks	none
Command Example	>LET @FRED=10 >:LOOP >D @FRED >LET @FRED=@FRED-1 >IF @FRED==0 >LOOPOUT >GOTO LOOP

Command	MACRO
Operation	Creates an unlimited number of user-defined commands that can be locally created and loaded/saved from/to a disk file.
Syntax	>MAC macro_name
Terms	macro_name = The name of the disk file to store the user-defined commands.
Syntax Example	>MAC THEKNIFE
Remarks	The MACRO command prompts for command lines until a blank line is read. Spacing: A space is required between MAC and macro_name.
Command Example	>MAC INITL <--creates a macro named "initl" MACRO>R RESET <--resets the registers MACRO>R <--examines the registers MACRO>R PC=1 <--changes the pc register to 1 MACRO> <--terminates macro > >INITL 1000

Command	MAP
Operation	Categorizes the ERX/target system memory functions as either read/write, read-only, user, or no memory (guarded access). Applications Note: This command can be used to develop your target system's firmware (ROM) by allowing code in a mainframe to be downloaded to the ERX, mapped as RO, and tested before being burned into the target's ROM.
Syntax	>MA [beg_addr[,end_addr]=area]
Terms	beg_addr = The beginning address of mapping. end_addr = The ending address of mapping. (The ending address may also be expressed in absolute number of bytes from the beginning address; e.g., MA 100,+30=RO.) area = RO, RW, US or NO
Syntax Example	>MA >MA 1000,1FFF=RW >MA 150=RO
Remarks	The mapping granularity is 1K-byte blocks. The MA specification displays the current status of the memory map. If the beg_addr or end_addr does not coincide with the beginning or ending of a 1K-block location, the beginning or ending area is assigned a location that includes beg_addr or end_addr. Two of the areas, RO and RW, refer to ERX user memory, and RW gives the user program free access to this memory. RO enables the user program to read this memory, but any attempt to write to this area will be blocked. US acts as target system memory area (US being RAM, ROM, I/O, etc.). NO memory assignment is useful in debugging by allowing you to create a break in program execution if an attempt is made to access this non-existent memory area.

Special Note: When the ERX boots up, the entire memory range is mapped as US (user or target system memory). If you are working without a target (developing software only), you will need to re-map the area as read/write (e.g., MA 0,1FFFF=RW).

Spacing: A space is required between MA and beg_addr. No spaces are permitted where commas act as separators.

Command Example

```
>MA <--displays the status of the memory map
>MA 0,1234=RO <--maps from 0 to 1234H as read-only
>MA 4000=NO <--maps addr 4000H as no memory
```

Command	MDELETE (Macro DELETE)
Operation	Deletes a macro loaded within the ER-ICE environment.
Syntax	>MD macro_name
Terms	macro_name = The name of the macro to be deleted.
Syntax Example	>MD PRO_START
Remarks	Spacing: A space is required between MD and macro_name.
Command Example	See Syntax Example, above.

Command	MLOAD (Macro LOAD)
Operation	Loads a macro (or macros) stored in the host computer.
Syntax	>ML file_name[.mac]
Terms	file_name = The name of the macro to load to the ERX.
Syntax Example	>ML USERMACRO
Remarks	Spacing: A space is required between ML and file_name.
Command Example	See Syntax Example, above.

Command	MODLEN (MODule LENgth)
Operation	Displays or sets the length of module names to be used on the screen display.
Syntax	>MOD [length]
Terms	length = 0 (default) to 20
Syntax Example	>MOD 12
Remarks	MOD alone displays all module lengths. Spacing: A space is required between MOD and length (if length is used).
Command Example	See Syntax Example, above.

Command	MOVE
Operation	Moves the memory contents between different locations within the ERX, or between the ERX and the target system.
Syntax	>M[/N] beg_addr,end_addr,mov_addr[,direction]
Terms	N = No verify mode. beg_addr = The beginning address of data source. end_addr = The ending address of data source. (The ending address may also be expressed in absolute number of bytes from the beginning address; e.g., M 100,+30,1000.) mov_addr = Beginning address for destination. direction = UE or EU
Syntax Example	M 1000,1FFF,3000,UE
Remarks	UE means that the source is user (target system) memory and the destination is ERX program memory. EU means that the source is ERX program memory and the destination is user memory. If direction is omitted, data is relocated within the memory area as specified by the MAP command. Spacing: A space is required between M and beg_addr. No spaces are permitted where commas are used as separators.
Command Example	>M 0,100,1000 <--moves range of memory to addr 1000H >M 1000,2000,1000,UE <--moves range in target to ERX addr 1000H >M/N 1000,2000,1000,EU <--moves range in ERX to target at addr 1000 but does not verify the move

Command	MSAVE (Macro SAVE)
Operation	Saves a macro to a file in the host computer.
Syntax	>MSA file_name[.mac]
Terms	file_name = The name of the macro file.
Syntax Example	>MSA USER_MACRO
Remarks	Spacing: A space if required between MSA and file_name.
Command Example	See Syntax Example, above.

Command	NOJOURNAL
Operation	Terminates command journaling.
Syntax	>NOJ
Terms	none
Remarks	none
Command Example	See Syntax Example, above.

Command	NOLOG
Operation	Terminates emulation logging.
Syntax	>NOL
Terms	none
Remarks	none
Command Example	See Syntax Example, above.

Command	OPEN
Operation	Opens a file in the host computer for reading or writing.
Syntax	>OPEN #file_no,file_name,mode
Terms	#file_no = The number of the file to open. file_name = The name of the file to open. mode = R or W
Syntax Example	>OPEN #1,TEXT1.DOC,R
Remarks	R means open the file for reading; W means open the file for writing. Spacing: A space is required between OPEN and #file_no. No space are permitted where commas are used as separators.
Command Example	See Syntax Example, above.

Command PERFORMANCE

Operation Records and displays the total emulation time, the number of event points passed during program execution, the time duration between event points, the average time of each event duration and the percentage of total event duration as compared to total emulation time.

Applications Note: The Performance function is useful for determining how much time was spent within a particular routine. The procedure would be to set event points at the addresses that mark the beginning and ending points of a particular subroutine or all the subroutines, run the program, and then analyze the total and average time spent within these sections.

NOTE: The PERFORMANCE command works alongside the EVENT command. As with the COVERAGE, HISTORY and TRIGGER commands, you must first create an event point (or event points), along with their attributes (location, data value, memory type, etc.), and then direct the event as a trigger using the PERFORMANCE: Specification command.

The following shows the order of command implementation for using the Performance function:

1. EVENT (Sets two, or more, events according to symbol name and address parameters. Optionally includes data or memory type, etc.)

2. BREAKPOINT (Optional. Terminates program execution after passing the event points. You can also issue a STOP command to terminate program execution.)

3. PERFORMANCE: Specification (Directs the event points to act as triggers for the Performance function.)

4. PERFORMANCE: Status (Allows you to view the event points, along with their attributes, that will be used to trigger the Performance function.)

5. GO (Commences program execution.)

6. PERFORMANCE: Display (Allows you to view the results of the Performance function.)

There are three PERFORMANCE commands available: Status, Specification and Display. The PERFORMANCE: Status command shows which event points have been designated as triggers for use by the Performance function. The PERFORMANCE: Specification command specifies the event points (by their symbolic names) that mark the beginning and ending location for the Performance function. The PERFORMANCE: Display command shows the results (in percentage format) of the Performance function.

The PERFORMANCE: Display command reveals the following information:

```
Total Emulation Time          =    0H:     0M:     0S:  000000US
Number of Event Occurrence     =                              0
Time of Event Duration        =    0H:     0M:     0S:  000000US
Average Time                  = ----H:    --M:    --S:  ------US
Time Percentage               =                          ---.--%
```

NOTE: All time values are expressed in hours, minutes, seconds and microseconds. Events are expressed in positive integers. Time percentage expresses total event duration compared to emulation time.

Command	PERFORMANCE: Status
Operation	Displays the status of the Performance function. This command allows you to view the event points (along with their attributes) that have been designated as triggers for the Performance function.
Syntax	>PE
Terms	none
Syntax Example	see above
Remarks	To see the contents of the Status command, you must first create event points and then direct them to act as triggers through the PERFORMANCE: Specification command.
Command Example	See "More PERFORMANCE Command Examples."

Command	PERFORMANCE: Specification
Operation	Directs previously defined event points to act as triggers for the Performance function. Only event symbol names can be used.
Syntax	>PE S=beg_event[,switch],E=end_event[,switch]
Terms	beg_event = The name of the event point to initiate the Performance function. end_event = The name of the event point to terminate the Performance function. switch = ON or OFF
Syntax Example	>PE S=trig1,trig2
Remarks	ON unmasks the associated function, and OFF masks the associated function. You do not need to enter ON to enable the function, only to unmask it from a masked state. Spacing: A space is required between PE and S. No spaces are permitted where commas are used as separators.
Command Example	>PE S=&1,E=&2 >PE S=&1,OFF,E=&2,OFF >PE S=&1,ON,E=&2,ON The three examples above show how two event points are directed as triggers, temporarily masked, and finally unmasked. Also see, "More PERFORMANCE Command Example."

Command	PERFORMANCE: Display
Operation	Displays the total emulation time, the number of event points passed during program execution, the time duration between event points, the average time of each event duration and the percentage of total event duration as compared to total emulation time.
Syntax	>PE/D
Terms	none
Syntax Example	see above
Remarks	This command is viewed after first defining the trigger points for the Performance function, running the program, and then terminating its execution.
Command Example	See, "More PERFORMANCE Command Example."

>ES	Now, examine the display to see the event symbol names and their locations.
>PE S=TR1,E=TR2	This command allows the two event points to serve as triggers for the Performance function.
>PE >ES	Now, check these two displays to confirm the settings you just made. Notice the "S" and "E" under the "P" in the second display. This indicates the starting and ending points of the Performance function.
>G 100	Begins program execution. Once the program breaks, you can then view the information contained in the Performance Display.
>PE/D	Displays the total emulation time, the number of event points passed during program execution, the time duration between event points, the average time of each event duration and the percentage of total event duration as compared to total emulation time.

Command	PIN
Operation	Masks or unmasks selected input signals. Also allows you to view the status of a single pin or all the pins.
Syntax	>PI [[signal]=switch]
Terms	signal = RESET, NMI, INT0, INT1, INT2, BUSRQ, DRQ0, DRQ1, ALL switch = EN or DI
Syntax Example	PI NMI=DI
Remarks	EN is used to unmask the associated signal, and DI is used to mask the associated signal. PI alone, allows you to examine the status of all pins. PI_signal allows you to examine a specific signal. Spacing: A space is required between PI and the next parameter.
Command Example	>PI <--displays Pin command status >PI NMI <--displays status of NMI pin >PI NMI=DI <--masks NMI pin >PI INT0=EN <--unmasks INT0 pin >PI ALL=DI <--masks all pins

Command	PORT
Operation	Examines one or more I/O port locations and optionally modifies them. The locations can be displayed and replaced with either hexadecimal or ASCII values.
Syntax	>P[/NOR] port_addr[=mod_data]
Terms	NOR = Write to port only, do not read back. port_addr = Starting address for display. mod_data = New data for this location.
Syntax Example	>P 1000=20
Remarks	If mod_data is omitted, the command enters a repeat mode that allows several locations to be changed. The repeat mode includes: return (cr) to display the next byte of data; comma (,) to display the same byte of data; caret (^) to display previous byte of data; slash (/) to exit the PORT command. Spacing: A space is required between P and port_addr (if /NOR is omitted) or between P/NOR and port_addr. No spaces are permitted thereafter.
Command Example	>P 12 <--start by examining port #12 0012 12=23, <--change value to 23; re-examine 0012 12= <--leave value unchanged; go to next addr 0013 00= <--leave value unchanged; go to next addr 0014 14='21' <--change value; go to next addr 0015 00=34^ <--change value; go to previous addr 0014 14=00/ <--change value; exit command

Command	PROMPT
Operation	Alters the ER-ICE prompt to any alphanumeric character string.
Syntax	>PRO [string]
Terms	string = The name of the new ER-ICE prompt.
Syntax Example	>PRO DEMO
Remarks	Spacing: A space is required between PRO and string.
Command Example	> >PRO NEWPROMPT NEWPROMPT>

Command	PUT
Operation	Writes the variable contents to a file.
Syntax	>PUT #file_no[,expression]
Terms	#file_no = The number of the file to store the variable contents. expression = Literal expression or a variable.
Syntax Example	PUT #5,!2
Remarks	A space is required after PUT. No spaces are permitted where commas are used as separators. Spacing: A space is required between PUT and #file_no. No spaces are permitted where commas are used as separators.
Command Example	>LET @2,"abcdefghi\n" >PUT #1,@2 >GOTO FILE_CLOSE

Command	QUIT
Operation	Terminates ER-ICE and returns control to DOS.
Syntax	Q
Terms	none
Syntax Example	see above
Remarks	none
Command Example	>Q Exiting ERX DOS>

Command

REGISTER

Operation

Displays the status of a register or all the registers, and optionally modifies the register(s) contents.

Syntax

```
>R [reg[=data]]
>R RESET
```

Terms

reg = Any one of the following registers:

A	B	C	D	E	H	L
BC	DE	HL	IY	IY	SP	F
A'	B'	C'	D'	E'	H'	L'
AF'	BC'	DE'	HL'	F'		
PC	S	Z	HC	PV	N	CY
I	IFF	IO	CL	BL	CB	BB

data = New value for register contents.

RESET = Resets the registers to their initialized values.

Syntax Example

R PC=1000

Remarks

R alone, examines the values of all the registers.
R reg examines the value of a specific register.
R RESET initializes all the registers.

If data is omitted, the command enters a repeat mode that allows several locations to be changed. The repeat mode includes:

return (cr) to display the next byte of data;
comma (,) to display the same byte of data;
caret (^) to display previous byte of data;
slash (/) to exit the PORT command.

Spacing: A space is required between R and the next parameter; no spaces are permitted thereafter.

Command Example

```
>R <--displays registers' contents
>R RESET <--initializes all the registers
>R BB=1111 <--changes BB to 1111
>R PC <--examines/allows changes from register "PC"
PC 1000=1FFF
A  FF=55
B  00=^ <--display previous register
A  55=^ <--display previous register
PC 1FFF=/ <--"slash" terminates entry
```

Command	REM
Operation	Allows you to insert comments into batch files.
Syntax	>REM [comment]
Terms	comment = The comment to insert into the batch file.
Syntax Example	>REM THIS IS A COMMENT
Remarks	Spacing: A space is required between REM and the comment.

Command Example

```
>MAC TEST
MACRO>REM ****TEST MACRO PROGRAM ****
MACRO>CPUT "Test Ok!! \n"
MACRO>
>ECHO ON
>TEST
REM ****TEST MACRO PROGRAM ****
CPUT "Test Ok!! \n"
Test Ok!!
Macro TEST completed.
>ECHO OFF
>TEST
Test Ok!!
>
```

Command	REPEAT
Operation	Repeats a command or series of commands "x" number of times.
Syntax	>REPEAT [num]
Terms	num = The number of time to repeat.
Syntax Example	>REPEAT 5
Remarks	Spacing: A space is required between REPEAT and num.
Command Example	>MACRO TEST MACRO>REPEAT 5 MACRO> CPUT "*" MACRO>ENDR MACRO>REPEAT 5 MACRO> REPEAT 10 MACRO> ENDR MACRO> CPUT "*" MACRO>ENDR MACRO>CPUT "\N" MACRO> >ECHO OFF >TEST ********** >

Command	RESET
Operation	Resets the I/O of the target system via the ERX.
Syntax	>RES
Terms	none
Syntax Example	see above
Remarks	When used singularly, the RESET command initializes the target system I/O, only. When the RESET command is used with the REGISTER command, it can initialize the processor's registers as well.
Command Example	See Syntax Example, above.

Command	SAVE
Operation	Saves an Intel, Motorola or dump format file to the host computer.
Syntax	>SA[/format] filename[.abs],beg_addr,end_addr [,start_addr]
Terms	format = The format of the file, including: I (Intel-hex) M (Motorola) D (dump format) beg_addr = First address to save. end_addr = Last address to save. (The ending address may also be expressed in absolute number of bytes from the beginning address; e.g., SA TEST,0,+30.) start_addr = Starting address of the user program.
Syntax Example	>SA TEST,0,1000,0
Remarks	Spacing: A space is required between format and filename (if format is included). No spaces are permitted where commas are used as separators.
Command Example	>SA FRED,100,+40,800 <--saves file named "fred" to address 800H >SA/M SAM,0FF,3FF <--saves a Motorola-format file named "sam"

Command	SHELL
Operation	Allows a DOS shell to be run while preserving all symbols and the ER-ICE environment.
Syntax	>SHE
Terms	none
Syntax Example	see above
Remarks	none
Command Example	>SHE DOS>dir a: : DOS>exit (returns to ER-ICE) >

Command	STEP
Operation	Allows you to step through program execution in non-real time either by examining every line from the current pc or examining only Jump instructions.
Syntax	>S[/J] [step]
Terms	J = Display only Jump instructions. step = 1 (default) to 65535 or $ (continuous)
Syntax Example	>S 10
Remarks	When the registers' contents are displayed as a series of periods (....), it indicates that the contents of the registers are unchanged. The registers' contents are displayed fully, however, at least once every 11 lines. This command displays items on a line-for-line basis. To control the scrolling of the display, alternately press the space bar. To exit the display, press the Esc key. Spacing: A space is required between S/J and step or S and step.
Command Example	>R RESET >STEP <--displays a single instruction line >STEP/J 2 <--displays two Jump instruction lines >R RESET >STEP 4 <--displays four instruction lines

Command	STOP
Operation	Breaks program execution that was previously initiated with the GO command.
Syntax	>STO
Terms	none
Syntax Example	see above
Remarks	When STOP is entered, the ERX breaks program execution and also displays the contents of the registers and the address where the program broke.
Command Example	>GO +> +>STO registers' contents ... <Break> >

Command	STUB (SubsTitute sUBroutine)
Operation	Substitutes an ERX command for a subroutine that has yet to be developed. In this way, a program that calls a subroutine will be supplied an ERX command in order to verify that the subroutine call occurred.
Syntax	>STUB [address,"commands"] >STUB address=switch
Terms	commands = The name(s) of the ERX command(s). address = The address where the STUB is located. switch = ON, OFF or CLR
Syntax Example	>STUB SIO_PORT,"EX BUFF_1=41;EX BUFF_2=42"
Remarks	STUB alone, displays the status of the STUB command. ON enables the function at the address specified, and OFF disables or masks the function at the specified address. Spacing: A space is required after STUB and before the next parameter. No spaces are permitted where commas act as separators.
Command Example	>STUB <--displays the status of the Stub command >STUB PIO_PORT,"BAT CLR" <--executes batch command >STUB 1000,"DI;R;PIN;G" <--executes command series >STUB 1000=OFF <--disables Stub at addr 1000H

Command	TRIGGER
Operation	Enables the output of an external trigger from the ERX when a designated symbol is encountered during program execution. The EXT.TRG. probe line on the ERX carries the signal (see "More About Your ERX" in Section 1).
Syntax	>TRI [event_symbol[=switch]]
Terms	event_symbol = The name of event point to trigger on. switch = ON or OFF
Syntax Example	>TRI EXTTRIG=ON
Remarks	The event point must first be created by the EVENT command and then converted to a trigger point by the TRIGGER command. (See the EVENT command for more information.) Entering TRI alone, displays the status of the TRIGGER command. (You can also view the trigger points by using the EVENT: Show command.) Wildcard characters (*) may be used in place of specific names or addresses (e.g., TRI T1.*=ON). Spacing: A space is required after TRI. No spaces are permitted thereafter.
Command Example	>TRI <--displays status of Trigger command >TRI MAIN <--specifies "main" symbol as external trigger >TRI &34=ON <--enables "&34" as an external trigger >TRI PORT.*=OFF <--masks all "port" symbols from external trigger recognition

Command	VERIFY
Operation	Compares an Intel, Motorola or dump format file in the host computer to a file in ERX memory and acknowledges the match, if any. If the match is exact, only the heading is displayed. If the data is a mismatch, the file data and memory data for the mismatching address locations is displayed. If the file doesn't exist, an error message is displayed.
Syntax	>V[/format] filename[.abs]
Terms	format = The format of the file, including: I (Intel-hex) M (Motorola) D (dump format) filename = The name of the file to compare.
Syntax Example	>V/I TEST.HEX
Remarks	The default format is Dump. The parameters for the VERIFY command are similar to the LOAD command, with the exception that the VERIFY command does not alter memory but only compares the memory contents. Spacing: A space is required before filename.
Command Example	>V/D LOOP.ABS Address File Memory <--acknowledges exact match >V/D MEMREG.ABS Address File Memory <--shows mismatching data and 0201 00 10 the location(s) where the 0202 00 FF mismatch occurs 0203 C3 01 >V/D BOGUS.FIL ER-ICE : No object record in object file. <--could not file

Command	WAIT
Operation	Used within a batch file, this command suspends the execution of batch commands during emulation until a breakpoint is encountered. When a break occurs, emulation stops and the next batch command is excuted.
Syntax	>WAIT
Terms	none
Syntax Example	see above
Remarks	none
Command Example	>G <--runs sample program +> +>D 2000,20FF <--program attempts to invoke a memory dump but can't because emulation is still in progress 0 2 4 6 8 A C E ASCII CODE 2000 ER-ICE: Emulation Busy. +>STOP <--emulation manually halted >B 100,,3 <--breakpoint inserted to halt emulation >R RES >G/W (or G;W) <--GO with WAIT inserted The display above shows the usefulness of the WAIT command. Without it, the DUMP command (the next commannd to implement in the batch file) could not have been executed while the ERX is emulating. To use the Wait feature, a breakpoint was set at address 100, the registers were reset, and the program was rerun with a GO/WAIT command. The program would now wait until the breakpoint was executed before invoking the memory dump.

Command	WHILE
Operation	Repeats a command or a block of commands as long as a specified condition remains logically true.
Syntax	>WHILE exp
Terms	exp = symbol name = MB (memory address; byte value) = MW (memory address; word value) = PB (port address; byte value) = LOG (logical address) = PHY (physical address) = register name = @n (variable) = !n (character variable) = #STS (status variable) +,-,*,/,&,|,^,~,==,>,>=,<,<=,<>
Syntax Example	>WHILE mb==11
Remarks	Spacing: A space is required between WHILE and exp.
Command Example	>WHILE @2==0 : : >ENDW

```
ABASE
>AB [base_1,base_2]
*>AB 0,B

AMAP
>AM

ASSEMBLE
>A mem_addr <CR>
xxxxx xxxx  {Assembly code} <CR>
xxxxx xxxx  <CR> >
*>A 1000
  01000 1000 beg_prog <CR>
  01001 1001  <CR>

BATCH
>BA file_name[.cmd]
*>BA setup.1

BEEP
>BEEP

BREAK
>B
>B address[,status][,passcount]
>B/R beg_addr,end_addr[,status][,passcount]
>B event_symbol[=switch]
>B/M beg_sym_name/line,end_sym_name/line[=switch]
*>B 100,MR,3
*>B/R 100,0BFF,MR,3
*>B evnt1
*>B/M rout1,rout2

CALCULATE
>C exp#1 +,-,*,/ exp#n
*>C 123+0FFH

CGET (Console Get)
>CGET [value]name
*>CGET !FRED

CLOCK
>CLO [clock_mode]
*>CLO 4

CLOSE
>CLOSE #file_no
*>CLOSE #1

COMPARE
>CO beg_addr,end_addr,cmp_addr[,direction]
*>CO 0,2FF,500,UE
```

```
COVERAGE
>COV/CL [beg_addr[,end_addr]]
>COV/S [status]
>COV[/disp] [beg_addr[,end_addr]]
*>COV/CL 0,1FF
*>COV/S MW
*>COV/U 100,7FF

CPUT (Console Put)
>CPUT [value]name
*>CPUT !store

DEFM (Default Module)
>DEFM [def_mod_name]
*>DEFM MAIN

DISASSEMBLE
>DI [beg_addr][,end_addr]
*>DI START,START+20H

DISPLAY
>DISP [switch]
*>DISP ON

DUMP
>D[/length] beg_addr[,end_addr]
*>D/W 100,1FF

ECHO
>EC switch
*>EC OFF

EMSELECT (Emulation Method Select)
>EMS [select][=switch]
*>EMS C=EN

EOF (End Of File)
>EOF #file_no,@ret_value
*>EOF #1,@I

ERX
DOS>ERX [batch_file[.cmd]]
*DOS>ERX SETUP

EVENT
>EV

ESAVE (Event Save)
>ESA file_name[.evt]
*>ESA EVENT1
```

```
EDELETE  (Event Delete)
>ED [beg_addr/beg_sym_name/line][end_addr/end_sym_name/line]
*>ED start,end*

ESHOW (Event Show)
>ES [beg_addr/beg_sym_name/line][end_addr/end_sym_name/line]
*>ES 1000,3000

EXAMINE
>E[/length][/N] beg_addr[=mod_data]
*>E/W 100=5555

EXECUTE
>EXEC dos_command
*>EXEC TYPE DEMO.1ST

FILL
>F[/length][/N] beg_addr,[end_addr],data
*>F 100,0FEF,55

FNKEY (Function Key)
>FN number
*>FN 4

GET
>GET #file_no,[value]variable
*>GET #1,!variable

GO
>G [beg_addr][,end_addr]
*>G 100

GOTO
>GOTO label
*>GOTO LABEL_1

HELP
>HE [command]
*>HE FILL

HISTORY
>H
>H/S
>H/SI
>H/mode[,int_point][,term_point][,A=address][,ST=status][,D=data]
*>H/M,200,100,A=100,D=00

ICERESET
>ICERES

IDENTIFICATION
>ID
```

```
MSAVE (Macro SAVE)
>MSA file_name[.mac]
*>MSA USER_MACRO

MSHOW (Macro SHOW)
>MS

NOJOURNAL
>NOJ

NOLOG
>NOL

OPEN
>OPEN #file_no,file_name,mode
*>OPEN #1,TEXT1.DOC,R

PAUSE
>PAUSE

PERFORMANCE
>PE
>PE S=beg_event[,switch],E=end_event[,switch]
>PE/D
*>PE S=&1,E=&5
*>PE S=&1,OFF,E=&5,OFF

PIN
>PI [[signal]=switch]
*>PI NMI=DI

PORT
>P[/NOR] port_addr[=mod_addr]
*>P 1000=20

PROMPT
>PRO [string]
*>PRO DEMO

PUT
>PUT #file_no[,expression]
*>PUT #5,"

QUIT
>Q

REGISTER
>R [reg[=data]]
>R RESET
*>R PC=1000

REM
>REM [comment]
*>REM THIS IS A COMMENT
```

```
REPEAT
>REPEAT [num]
*>REPEAT 5

RESET
>RES

SAVE
>SA[/format] filename[.abs],beg_addr,end_addr[,start_addr]
*>SA TEST,0,1000,0

SCOPE
>SCO [number]
*>SCO 5

SEARCH
>SE[/length][/D] beg_addr,[end_addr],search_data
*>SE/B 100,5FF,

SHELL
>SHE

STEP
>S[/J] [step]
*>S 10

STOP
>STO

STUB (SubsTitute sUBroutine)
>STUB [address,"commands"]
>STUB address=switch
*>STUB SIO_PORT,"EX BUFF_1=41;EX BUFF_2=42"

SYMLEN (SYMbol LENgth)
>SYM [number]
*>SYM 12

TRIGGER
>TRI [event_symbol[=switch]]
*>TRI EXTTRIG=ON

VERIFY
>V[/format] filename[.abs]
*>V/I TEST.HEX

WAIT
>WAIT

WHILE
>WHILE exp
*>WHILE mb==11
```

Note: "*" denotes command example.

SECTION 3 TECHNICAL REFERENCES

NOTE: THIS SECTION IS STILL UNDER DEVELOPMENT. IT WILL BE COMPLETED THE FIRST QUARTER OF 1988.

APPENDIX A

IN-CIRCUIT EMULATORS SPEARHEAD THE NEW MICROPROCESSOR DEVELOPMENT SYSTEMS

by Mark D. Johnson

Although turnkey microprocessor development systems have been available for about a decade now, the new integrated microprocessor development systems (MDS) used for testing, debugging and integrating both hardware and software designs have only recently become a common sight in the design engineer's lab.

The new MDS, with its unbundled structure, is a multi-component, multi-vendor system pieced together by the user. It is the latest, and least expensive, approach to implementing and automating the design of microprocessor-based systems. These open-architecture development systems - designed to be compatible with existing development equipment - are the newest and fastest growing design aids that are more than adequately competing with the turnkey development systems supplied by the chip manufacturers themselves.

The latest tri-based systems incorporate a host computer, development software and diagnostic tools such as logic state analyzers and emulators. Of these components, it is the emulator that transposes the run-of-mill computer system into a full-fledged microprocessor-based development system for managing a project from the software coding phase right through to hardware/software integration and final testing.

What makes the new MDS so popular? Primarily, its low cost. By utilizing existing resources in their labs, engineers can now integrate the development software and diagnostic tools needed to form a complete MDS, while saving thousands of dollars over the price of dedicated systems.

PC Makes The Ideal Manager

The heart of the MDS is the management unit - the host computer. The host computer provides control for the system, acts as a pipeline to development software and provides mass memory storage. In the past, mainframe micro computers were the accepted standard for project control, but with the new MDS, this expense far outweighs the costs of

the remaining components. The most attractive alternative is the IBM personal computer - indisputably the design engineer's favorite choice in a personal computer. Other PCs can be utilized, but the IBM PC is really the key to the flexible MDS as other market leaders continue to tailor their software products and diagnostic tools to IBM's architecture.

Using the IBM PC, design engineers can now tap an extensive line of development software products for generating source code and compiling complete programs. This development parallels the longstanding ability of programmers to incorporate the use of assemblers, compilers, linkers, loaders and editors as software development tools.

The remaining members in the MDS structure are diagnostic tools; specifically, logic-state analyzers and in-circuit emulators. When the hardware prototype is constructed, it must be tested to determine whether the actual operation in real life is within the hardware specifications. This is where specialty diagnostic tools become a necessity, as they have the capability to monitor signals from the microprocessor and peripheral circuits. Since there are numerous microprocessor signals which require simultaneous monitoring, the use of an oscilloscope (capable of monitoring only a few signal lines) would be impractical for keeping track of all the necessary signals. This is where the logic state analyzer and emulator play such a vital role.

Emulation vs Simulation

A logic state analyzer (LSA) can vary from a simple (where some indication of a single multi-signal state is needed) to a very sophisticated device which can store several multi-signal states and display the results in various forms - even showing the instruction mnemonics. LSAs, however, lack the one feature available on virtually every emulator - the ability to "loan" resources. Thus, the emulator is an advance over the LSA because it permits the use of the more sophisticated facilities of the LSA, combined with the ability to transfer blocks of memory between the prototype and the emulator, and then modify the memory by categorizing it as read-only, read/write, and so on.

Another popular configuration ties an LSA directly to an emulator. The combination merges a logic analyzer's sophisticated trigger, qualification, data-acquisition, and measurement capabilities with an emulator's debugging and loan facilities. The emulator/LSA interface permits simultaneous control and monitoring of the prototype system under development.

For all their testing, debugging and management capabilities, emulators remain the most misunderstood and often intimidating devices associated with the MDS. Part of the mystique arises from their functionality (the ability to monitor as well as execute), part from the lack of application information (often vague and incomplete), and part from their control mechanism (emulators need to be told what to do and how to do it by a set or series of complicated and often confusing instructions.) Couple these problems with the fact that there are several different system configurations, all requiring a particular compatibility formula, and it's no wonder designers turn to the single-vendor development systems. Fortunately, however, the situation is changing for the better, since vendors are beginning to coordinate various manufacturers' equipment and development packages for their users.

Mimicking The Prototype

Emulation is the result of replacing the microprocessor in a prototype design with a piece of test equipment which incorporates an identical microprocessor to that found in the prototype. The strategy behind this exercise is to provide the test equipment with all the functions of a particular microprocessor, along with capabilities which assist in the integration of the prototype's hardware and software components. In a real-world testing environment, the emulator is attached to prototype circuitry by replacing the microprocessor with a multi-pinned header plug with the same pin configuration as the processor's. Thus, in its simplest operating mode, the emulator "tricks" the prototype into thinking nothing has changed with its relationship to the microprocessor. The prototype can then execute its functions while being monitored by the emulator.

The basic components of all emulators are very similar, usually consisting of a mainframe chassis, individual control and memory circuit boards, intermediary circuitry, and an isolated power supply. The controls and components on an emulator's external casing can range from virtually non-existent - where control and indication is regulated to an external terminal - to those possessing complete keyboard facilities, control switches and even an EPROM socket. The physical size of an emulator is usually dictated by the amount of resident memory it contains, the control and indication mechanisms, its emulation probe arrangement and the sophistication level of its debugging facilities.

The emulator's microprocessor, which permits the actual emulation of the prototype hardware, is located on the emulation board or emulation pod. The board typically contains an emulation probe (or the means of attaching one), which connects the emulator's microprocessor to the prototype. The connection between the prototype and the emulator is necessary for hardware development but is not needed for software debugging and testing.

Although apparently simple in its construction, the design of the probe is critical if transparent emulation is to occur. (Transparency is the ideal emulation condition in which the operation of the prototype is unaffected when the emulator substitutes the microprocessor. An emulator should make no demands on any part of the prototype's resources such as interrupts and memory allocation, and the emulator should resemble as closely as possible the microprocessor's characteristics such as timing and clock speed.) Ideally, the emulation cable represents a compromise between user convenience and minimizing the propagation delays and capacitive loading that the length of cable introduces. Some emulator vendors try to eliminate the latter by locating the emulated processor in a pod positioned both physically and electrically as close to the prototype's socket as possible - sometimes a mere inch away. While this arrangement minimizes propagation delays, there are some disadvantages such as a bulkier probe design (especially when interfacing to pre-constructed designs) and the necessity for a larger buffer board within the emulator. If clock speeds are kept under 10 MHz, this type of cable provides virtually transparent emulation. If speeds are faster, the microprocessor pod arrangement is preferable.

Emulator Supplies The Speed

An emulator's resident memory is composed of high-speed static RAM, which is fast enough (i.e., sufficient access time) such that the program can run at the same speed as if the prototype were supplying the memory. High-speed static memory also eliminates the need for the emulator to insert wait states when accessing the memory. Emulators typically contain between 32 and 128 kbytes of internal RAM, and nearly all emulators allow expansion to larger memory resources (some to 16 Mbytes via off-board memory modules or by accessing a portion of prototype memory.

Emulators can also allocate their memory to the prototype (or any location within the addressable memory space) in specified byte blocks. For an emulator featuring 64 kbytes of memory, the block size (resolution) is typically 1 kbyte. The resolution for a given emulator usually varies with the amount of memory available, with 2- or 4-kbyte sizes popular with emulators that have a memory capacity of 128 kbytes to 1 Mbyte. When resolution falls below the 1 k-byte block size, it often becomes too "fine," which necessitates several allocating operations. The opposite condition, "coarse" resolution, makes it impossible to accurately emulate prototype systems incorporating limited memory chips. Most vendors seem to agree on confining resolution capacities to between 1K and 4K bytes.

Since the allocated memory should have the same design characteristics of memory that will eventually exist in the final design, mapping is used to categorize the type of emulation memory that will be allocated to the prototype. Most emulators allow their resident memory to be specified as read-only and read/write, and some add a "user" memory specification (user being RAM, ROM, I/O, etc. - whatever resides at those memory locations in the prototype). A few emulators provide a "no memory" mapping option, where any memory not mapped as read/write or read-only is assumed to be non-existent prototype memory. This type of mapping turns out to be a very useful provision in software debugging since a common result of programming error is an attempted access to an area where no memory is present. During emulation, the recognition of an attempted access to a protected memory area results in a program halt.

The ability to loan memory to the prototype design is what sets the emulator apart from an ordinary computer system, but emulators can also manage several other memory functions. Most emulators feature an in-line assembler which converts the mnemonics entered from the keyboard to machine language in memory. The emulator's assembler is useful for writing software patches into program code that has either been downloaded from a mainframe or developed in the prototype. Programmers can also use the assembler for writing their own routines or developing small programs.

Another memory control feature is the ability to display the emulator's memory contents (starting and ending anywhere that the user chooses). The ability to change the memory contents at individual locations and fill entire sections of memory with specified data is also available. Another memory control involves comparing specified blocks of memory with other blocks in the emulator's memory or with blocks residing in the prototype. The comparison can show all the matching or non-matching data and their locations. Instead of a complete block of memory, the emulator may also locate specific data in its memory or the prototype memory.

Breakpoints To Stop, Trigger To Flag

The key to effective program control is installing breakpoints and triggers within the user program so the emulator may show the condition on the processor's address, data, and control lines. Both software and hardware breakpoints are available within a given emulator, and each have their own functioning scheme. Triggers provide the emulator with "sensors" that can be used to initiate program recording (via the trace mechanism), terminate the program recording, or simply detect both operations.

Software breakpoints replace program instructions with monitor calls in order to stop program execution at a pre-determined location. Setting a software breakpoint causes the emulator to automatically replace the op code at the specified address with the processor's software interrupt instruction. When the code is encountered during program execution, a temporary break occurs while the original contents are replaced by the interrupt instruction; then the execution restarts at the same location (this causes the program to only run in real time up to the breakpoint). Software breakpoints are limited to address recognition only - there is no way for a software breakpoint to decipher between memory types.

Hardware breakpoints, which recognize machine cycles but do not disturb normal software execution, can monitor the address and status signals for a specified condition (for example, a memory read operation at address 1000H) and halt program execution when those conditions are met. Emulators allow several bus transactions to be monitored for program breaks including accessing any memory operation, accessing any I/O operation, memory reads, memory writes, I/O reads, I/O writes, operation instruction fetches and interrupt

acknowledgment. When a break condition occurs, most emulators ca stop the program being executed and display the current contents of the microprocessor's registers, the instruction just executed, the location of the instruction and the reason the program was halted.

Triggers can be used to initiate actions during emulation or simply observing them and relay the pertinent information to other devices. For example, a trigger which is set in the program (according to address location, data type or bus transaction) and encountered during emulation may be used to halt program execution in a manner similar to a normal hardware breakpoint. But triggers are more flexible than breakpoints in that they may be specified to merely observe the pre-defined condition and act without disrupting the program being executed.

Real-Time Trace Stores Program Execution

Breakpoints and triggers come together under the debugging capability called real-time tracing. The real-time trace feature allows a user to record program execution in real time and then analyze (by replaying) a section of the program in non-real-time. The entire program can be traced (and reviewed), or just a portion - depending on your strategy. By using various combinations of triggers and breakpoints, the desired section where a potential problem exists can be recorded. The program can then be stopped, and the address, data and control lines of the latest series of machine cycles can be displayed or dumped to a printer for examination.

The trace buffer size of an emulator is determined by its width and depth. An emulator's trace buffer should be wide enough to accommodate the processor's address and data lines and possibly a few external lines. Typically, emulators offer between 32 and 64 bits of width. Some universal emulators, designed to work with several different processors, feature extremely wide trace buffers for allocating the various address, data, status and external lines for the entire line of processors supported.

When it comes to trace buffer depth (i.e., how many cycles of machine cycle execution the buffer can hold), larger is not always better. If the size is fixed at a large depth, problems may exist when sifting through all the information accumulated in the buffer. Conversely, if the buffer is too small,

the chances of recording the section containing the error is reduced to a hit-and-miss situation. A good working buffer depth falls between 2k and 4k machine cycles. A large buffer with user-variable depth makes an ideal combination.

Emulator Enhancements

Functions in emulators which do not manipulate memory or affect emulation processes are simply debugging enhancements. Since no two vendors offer the same set of debugging enhancements, none have emerged as standards (as with breakpoints, for example). Among the more useful features are built-in test programs, which include memory tests, scope loop tests and signature analysis tests. Memory tests are useful in determining whether the system RAM is functioning properly. Scope loops provide repeated read or write operations to memory or I/O ports, which are useful when debugging these circuits with an oscilloscope, while signal analysis can be a useful test environment strategy.

Other feature include built-in PROM programmers, offset registers and calculation and conversion facilities. PROM programmers are used in microprocessor applications to place programs and data in ROM. Offset registers. Offset registers are helpful when debugging programs consisting of several modules. Each module listing typically shows the first address in the module as zero, with the linker/loader then relocating each module to the appropriate address. To determine the actual address for a given instruction, the load address must be added to the address shown in the listing. By setting the offset register to the load address, this procedure is handled by the emulator.

When debugging software, the use of a decimal and hex calculator/converter can be helpful. Some emulators have the ability to perform subtraction and addition of hex and/or decimal numbers, and perform hex-to-decimal or decimal-to-hex conversions. The results of a particular operation are displayed in both hex and decimal notation.

Command Formats Yet To Be Standardized

Before any operation can be executed or any function monitored, the emulator must be told what to do, and told in its own precise language - a language that's different with almost every emulator. While many emulators use their own simple

mnemonic command structure, others rely on complicated non-mnemonic logical statements. While one emulator contains a built-in control keyboard, another can only be activated from the remote keyboard of a terminal or computer. The two different command format styles and entry mechanisms influence the way debugging operations are both specified and executed. For example, assume a user wishes to move a block of memory from a location in the prototype memory space to a space in the emulator's memory. One emulator, with its built-in keyboard, would require the user to press a control keyswitch, punch in a start address, press another control keyswitch, punch in the end address, then press another control to initiate the transfer. With a mnemonic command-controlled emulator, the same operation can be executed with a single-line entry that specifies both the Move operation and the beginning and ending addresses that mark the block of memory in the prototype.

The Emulation Session

Assuming the user has an understanding of emulation principles and can interpret the tutorial material in the emulator's operation manual, he is now ready to begin the art of debugging and testing the prototype hardware and software.

Once the hardware prototype is available, the system may be configured for emulation by first removing the prototype's processor and then electronically replacing it with the emulator's processor via the emulation probe. The emulator may now attempt to emulate the prototype. (At this point, the prototype design takes on the role of a "target" for the emulator.)

Initially, the emulator is instructed to analyze certain conditions and act accordingly, for example, it may stop program execution when a read-only memory area is written to. The emulator then begins executing the prototype program either from an initial or reset state, a user-specified starting address or the location where the emulator was last stopped. During this time, the emulator is operating as if the prototype's processor were controlling the system as well as monitoring the address, data and control lines for a breakpoint, trigger or fault condition.

After a period of time, the emulator's circuitry stops the execution of the prototype program. It may have detected a breakpoint, an illegal instruction execution, a memory protection violation, a trigger acting as a breakpoint, or the

user may have manually stopped program execution. If a breakpoint stopped program execution, the emulator can display the instruction just executed, the instruction location, and the status of the registers. If the emulator's real-time trace mechanism has been activated, it will have stored the latest series of machine cycles and is now available for viewing. The buffer will show all bus activity from the breakpoint or trigger.

Once the current status of the processor is examined, the user may need to alter the prototype program. This may be as simple as changing the contents of specified registers or more involved by writing a software patch using the emulator's in-line assembler or transferring complete sections of prototype memory to the emulator for testing. After the section is tested, it is moved back into the prototype at the same location. When the user is satisfied with the changes, the emulator starts executing the program again, looping through the above process until the prototype functions correctly.

Economic Considerations

In the MDS engineering environment, emulators play a special role by providing the unique ability to not only monitor the prototype under actual operating conditions, but also to loan facilities to the prototype system itself. During prototype execution, the emulator's testing abilities permit the address, data and control lines of the emulator's microprocessor to appear transparent to the prototype until a problem occurs or a breakpoint or trigger is encountered. The emulator then possesses the facilities to track down and fix the problem, then test and recheck the program.

From a monetary standpoint, emulators are cost-effective diagnostic tools in comparison to single-vendor turnkey development systems. Their flexible design allows them to interface to affordable and increasingly popular personal computers, which, in turn, can be used to develop software code as well as handle the editorial chores associated with a wordprocessing system. They are also completely compatible with existing mini and mainframe computers - making them ideal design aids for both large and small design departments.

APPENDIX B ERX DEMONSTRATION

NOTE: THIS APPENDIX IS STILL UNDER DEVELOPMENT. IT WILL BE AVAILABLE THE FIRST QUARTER OF 1988.

APPENDIX C TECHNICAL BULLETINS & APPLICATION NOTES

Introduction

Things are constantly changing in the microprocessor industry, and ZAX wants to help you stay on top of these changes. New products, emulation methods, and applications are always being devised and tested by us in an effort to provide you with the latest and most effective equipment possible. In the same manner, revising your existing equipment keeps it current with the latest ERX designs from ZAX.

One of the best ways we have of keeping you up-to-date is by issuing Technical Bulletins and Application Notes.

Technical Bulletins

Technical Bulletins inform you of major changes or revisions to the equipment's hardware or firmware. Usually they are the result of a problem that's recently been solved, or they could be a feature that's been revised to improve the performance of the emulator.

Application Notes

Application Notes are the result of new methods or procedures derived from emulation practices. They may also caution you against doing something a certain way, or they may show you a new way of accomplishing an old task.

Both Technical Bulletins and Application Notes are sent to you as soon as they become available - you should never need to request them. When you receive your documents, insert them into this appendix for easy reference. That's all there is to it!

bit	A binary digit.
branch	To depart from the normal sequence of executing instructions in the computer. (Synonymous with a jump.)
breakpoint	A point in a program as specified by an instruction where the program may be interrupted by some external intervention or by a monitor routine. This program break permits a visual check, print out, or other analysis of the program before resuming with the normal sequence. Used extensively in debugging operations.
buffer	A storage device in which data is assembled temporarily during data transfers. It is used to compensate for the differences in the rate of flow of information when transferring information from one device to another.
byte	A sequence of adjacent binary digits operated upon as a unit and usually shorter than a computer word.
C	A high-level programming language designed to optimize run time, size, and efficiency. It was developed as the systems programming language of the UNIX operating system on the PDP 11/70 minicomputer from Digital Equipment Corporation.
CLK	clock
clock	Devices or units which control the timing of bits sent in a data stream and the timing of the sampling of bits received in a data stream. One such clock device is a real-time clock, which measures the past or used time on the same scale as the external events it will be used to describe. Most microprocessor clocks operate in the range of 1 to 12 MHz.
code	A group of symbols that represent data or instructions in a computer. Digital codes may represent numbers, letters of the alphabet, control signals, etc. as a group of separate bits rather than continuous signals. (See microcode.)
compiler	A computer program, more powerful than an assembler, that will convert a higher level language into machine language.

default value	The choice among exclusive alternatives made by the system when no choice is made by the user.
development system	A system of devices, usually consisting of a diagnostic tool (such as an emulator), a computer, a printer, etc., that can be used together to develop and debug hardware and software for a given microprocessor.
development tools	Hardware and software devices that are used to develop and debug programs and/or microprocessor systems.
DIP	Dual In-line Package. A standard IC package with two rows of pins at 0.1" intervals.
DIP switches	A collection of small switches on a DIP that are used to select options on circuit boards without having to modify the hardware.
disassembly (disassembler)	Refers to a program that translates from machine language to assembly language. Usually used to decipher existing machine language programs by generating symbolic code listings of a program.
don't care	A term applied to an operation which can be changed or interrupted upon receipt of a control signal. The output of the operation is independent of the input.
downloading	A process whereby a file is loaded "down" to a device from a computer system.
DTE	Data Terminal Equipment. Equipment comprising of a data source (transmitter) or data sink (receiver) that provides for the communication control functions (protocol).
dump	The process of transferring the contents of memory at a given instant of time onto a screen for viewing, or outputting the memory contents to a printer.
duplex	A simultaneous two-way independent transmission.
dynamic RAM	Memory that requires constant refreshing in order to store memory.
EAROM	Electronically Alterable Read Only Memory. A specialized random access read/write memory with a special slow write cycle and a much faster read cycle.

echo check	An accuracy check of a transmission in which the transmitted information received by an output device is returned to the information source and compared with the original information.
editor	A general-purpose text-editing program that allows entry and maintenance of text in a computer system. The original text is entered and held in memory where it can then be changed and corrected by inserting, deleting, or changing lines of text or characters within a line.
EEPROM	Electronically Eraseable Programmable Read-Only Memory. An EEPROM is a device that can be erased electrically in one second and reprogrammed up to a million times.
EEPROM programmer	A unit that provides a means of programming a single EEPROM or an EEPROM module from a terminal.
EIA-RS-232C	A standard method adopted by the Electronic Industries Association to ensure uniform interface between data communications equipment and data processing terminal equipment.
emulation	Techniques using software or microprogramming, in which one system is made to behave exactly like another system, i.e., the emulating system executes programs in the native machine language code of the emulated system.
emulation mode	The mode that the ERX assumes in order to execute instructions.
emulator	An instrument that imitates the control memory of future hardware. Also a device that causes a system (such as the target hardware) to accept certain software programs and routines and appear as if it were the other system.
emulator, stand-alone	An emulator whose execution is not controlled by a control program. It also does not share system resources with other programs and excludes all other jobs from the computing system when it is being executed.

EPROM	Eraseable Programmable Read-Only Memory. A specific type of ROM that can be programmed electrically. It can retain data even with the power disconnected but can be erased by exposure to short wavelength ultraviolet light, and may be reprogrammed many times thereafter. Other types of EPROMs may be electrically erased. (See EEPROM.)
field	A set of one or more characters which is treated as a whole.
firmware	Programs that are stored in a physical device (e.g. ROM) that can form part of a system or machine.
FIFO	First In, First Out.
First In, First Out	A method or technique of storing and retrieving items from a storage source. With this technique, the "oldest" data transmitted to the storage source is thrown out and replaced with the "newest" data. The technique is useful when the storage source capacity is smaller or less than the item quantity.
FORTRAN	FORmula TRANslator. A high-level language developed by the IBM Corporation, originally conceived for use on scientific problems but now used for many commercial applications as well. It requires the use of a compiler.
full duplex	A mode of communication in which data can be transmitted and received simultaneously.
gate	A device that has one output channel and one or more input channels such that the condition of the output state is determined by the state of the input channel. The NAND, NOR, AND, OR, XOR, and NOT functions are examples of gates.
GND	Ground
half-duplex	A mode of communication in which data may be transmitted in only one direction at a time.
halt	A condition which occurs when an operation in a program stops.
handshaking	A sequence of signals that are required for communication between different systems.

hardware	Physical (electric, electronic, or mechanical) equipment - as opposed to a computer program - used for processing data. Contrast with software.
hex, hexadecimal	Pertaining to a number system with a base of 16. The digits 0 through 9 are used, then A through F, to represent decimal numbers 0 through 15, e.g., "FF" represents "11111111" binary, and "0A" is "00001010" binary.
high-level language	Any group of computer languages which use symbols and command statements an operator can read. High-level languages allow a user to write in a familiar notation rather than the machine code language of a computer. BASIC, FORTRAN, FOCAL, and COBOL are all examples of high-level languages.
host computer	The primary or controlling computer in a system operation. A host computer can also be reduced to a simple memory storage facility.
hysteresis	The lagging in the response of a unit of a system behind an increase or a decrease in the strength of a signal.
ICE	In-Circuit Emulation.
in-circuit emulation	Hardware/software facilities for real-time I/O debugging of chips. With in-circuit emulation, the actual microprocessor is replaced by a connector whose signals are generated by an emulation program. The emulated microprocessor can be stopped, its registers examined or modified, etc.
instruction	A coded program step that tells the computer what to do for a single operation in a program.
instruction set	The basic set of instructions that a particular computer can perform.
interface	The physical connection between two systems or two devices.
interrupt	A break in the normal flow of a system or program that occurs in such a way that the flow can be resumed from that point at a later time.
journaling	Refers to a process where all information generated in an emulation session on a host computer is output to a storage file. The entire session can then be reviewed, line for line, just as it was initially entered.

kilobaud	Refers to the number of one thousand bits per second.
linking loader	A loader used to link compiled/assembled programs, routines, and subroutines and turn the results in operations.
loader	A program required on practically all systems that load the user's program along with system routines into the central processor for execution. Loaders transfer the object code from some external medium (tape or disk) into RAM.
logic state analyzer (LSA)	A device that monitors a system or component board and displays the resulting information.
machine cycle	The time interval in which a computer (or similar device) can perform a given number of operations.
machine language	A set of symbols, characters, or signs, and the rules for combining them, that conveys instructions or information to a computer.
macro	Pertains to a specific type of instruction in assembly language that is implemented in machine language by more than one machine-language instruction, e.g., a group of instructions often designed to serve as an additive command or group of commands.
macro assembler	An assembler that is capable of assembling object programs from source programs written in symbolic language.
macro instruction	An instruction which stands for a predefined sequence of other instructions, called the "body" of the macro. Whenever a macro instruction is encountered in program text, it is "expanded," i.e., replaced by its body.
mainframe, main frame	Usually refers to large-scale computers (as opposed to microcomputers, microprocessors, and minicomputers). May also mean the fundamental portion of a computer, i.e., the portion that contains the CPU and controller units within the computer system.

microcode	A set of control functions performed by the instruction decoding and execution logic of a computer system. The microcode defines the instruction set of a specific computer.
mnemonic code	Refers to techniques used to assist human memory. A mnemonic code resembles the original word and is usually easy to remember, e.g., mpy for multiply, acc for accumulator.
monitor mode	Refers to a process where monitor commands from the ERX are executed. Dump, Fill, Disassemble, and Examine are all examples of commands used in the monitor mode.
MOS	Metal-Oxide Semiconductor. A technology used for fabricating high-density ICs. The name refers to the three layers used in forming the gate structure of a field-effect transistor.
NOP, NOOP	NO-OPeration. An instruction used to force a delay of one instruction cycle without changing the status flags or the contents of the registers.
object code	The code produced by a compiler or special assembler which can be executed by the processor when it is loaded, as with most microcode, or it may require a linkage phase prior to loading and execution.
object program library	An organized set of computer programs, routines, or common or specifically designed software, containing various programs or routines, source or object programs classified for intelligence or retrieval.
operating system	Software that is required to manage the hardware and logical resources of a computer system. Also a part of a software package (program or routine) dedicated to simplifying input/output procedures, sort-merge generators, data-conversion routines, or tests.
operation code	The symbols that designate a basic computer operation to be performed. This can be a combination of bits specifying an absolute machine-language operator, or the symbolic representation of the machine-language operator.
operator	A symbol in programming language that represents an operation to be performed on one or more commands (e.g., "add x").

parameter	A constant or variable in an equation or statement that may be assigned an arbitrary value.
parity bit	A redundant bit added to a group of bits so that an inaccurate retrieval of that group of bits is detected.
Pascal	A language designed to teach programming and the elements of computer science. Based on the language ALGOL, it emphasizes aspects of structured programming.
peripheral devices	Various kinds of machines or devices that operate in combination with a computer but are not physically part of the computer. Peripheral devices typically display computer data, store data from the computer and return it to the computer on demand, prepare data for human use, or acquire data from a source and convert it to a form usable by a computer.
phantom ROM	A type of ROM that operates for the system bootstrap only and then "hides" behind the main memory.
PIO interface	Abbreviation for Parallel Input-Output interface. PIO interfaces allow the computer to input and output parallel data to and from an external parallel device such as a keyboard or printer. Parallel means that all the data bits output at the same time.
PROM	Programmable Read-Only Memory. A ROM that may be altered by the user. Some PROMs can be erased and reprogrammed through special physical processes.
PROM programmer	A module or external device used to program programmable read-only memories.
PROM programming	The process of altering PROMs (called "burning"), either by blowing (melting or vaporizing) fusible links in bipolar PROMs or by storing a charge on the floating gates of UVEPROMs.
RAM	Random Access Memory. This type of memory is random because it provides access to any storage location point in the memory by means of horizontal and vertical coordinates. Information can then be "written" in or "read" out very quickly.
read-only (RO)	Refers to a process where information can be read from memory only.

read-write (RW)	Refers to a process where information can be read from and written into memory.
real time	Pertains to the actual time during which a physical process transpires. In emulation, real-time operation is very important because of the necessity for the emulator to maintain a "transparent" condition with regard to the device being emulated.
register	A memory device capable of containing one or more computer bits or words to facilitate arithmetical, logical, or transferral operations.
ROM	Read Only Memory. A special memory that can be read into but not written into.
RS-232C	See EIA-RS-232C.
semantics	The relationship between symbols and their intended meanings independent of their interpretation.
source program	A program coded in something other than machine language that must be translated into machine language before use.
stand-alone	Pertains to a device that requires no other piece of equipment to execute and complete its own operation.
stand-alone system	Usually, a microprocessor development system (MDS) that runs independent of the control of a computer. The MDS may contain a terminal and built-in display facility, which in effect makes it a full microcomputer with debugging capabilities.
statement	An instruction (macro) to the computer or other related device, to perform some sequence of operations.
static RAM	RAM that does not need to be refreshed or receive any further attention as long as power is applied.
step	One instruction in a computer routine.
stop bit	The last element of a character that defines the character space immediately to the left of the most significant character in accumulator storage.

symbolic debugging	Symbolic commands that are used to assist in the debugging procedure. Symbolic refers to codes which express programs in source language, i.e., by referring to storage locations and machine operations by symbolic names and addresses that are independent of their hardware-determined names and addresses.
symbolic trace	A process where addresses in a program trace are replaced with symbols. The symbol conversion process is performed in the host system using the appropriate software program.
syntax	Rules that govern sentence structure in a language, or statement structure in a language such as that of a compiler program.
target system	Refers to the processor under development.
trace	Refers to the operation of the real-time trace buffer (storage facility) and its ability to capture and store a portion of the program memory area.
transparency	The ideal emulation condition in which the operation of the target system is unaffected when the emulator is substituted for the microprocessor. Transparency can be broken down into two categories: functional and electrical. To be functionally transparent, the emulator should make no demands on any part of the target system's resources such as interrupts and memory allocation. To be electrically transparent, the emulator should duplicate as closely as possible the microprocessor's characteristics, such as timing and clock speed.
trigger	Refers to a user-specified reference point (external to the user program) which determines where and when to initiate or terminate a program trace.
TTL	Transistor Transistor Logic. A family of integrated circuit logic elements with a specific output structure, usually +5 volt "ones" and 0 volt "zeros."
universal emulator	A single emulator that is able to support several different processors, even from different manufacturers.
uploading	A process whereby a file is transferred from an device to the computer system.

virtual memory	Refers to a technique that permits users to treat secondary memory (disk) storage as an extension of main memory and thus give the virtual appearance of a larger main memory.
XON/XOFF	Transmitter ON/OFF.

Notes

Notes

Notes